Diversify Sensor Nodes to Improve Resilience Against Node Compromise [*]

Abdulrahman Alarifi and Wenliang Du
Department of Electrical Engineering and
Computer Science
Syracuse University
Syracuse, NY 13244-1240 USA
{aalarifi,wedu}@ecs.syr.edu

ABSTRACT

A great challenge in securing sensor networks is that sensor nodes can be physically compromised. Once a node is compromised, attackers can retrieve secret information (e.g. keys) from the node. In most of the key pre-distribution schemes, the compromise of secret information on one node can have substantial impact on other nodes because secrets are shared by more than one node in those schemes. Although tamper-resistant hardware can help protect those secrets, it is still impractical for sensor networks.

Having observed that most sensor network applications and key pre-distribution schemes can tolerate the compromise of a small number of sensors, we propose to use diversity to protect the secret keys in sensor networks. Our scheme consists of two steps. First, we obfuscate the data and the code for each sensor, such that, when attackers have compromised a sensor node, they need to spend a substantial amount of time to find the secrets from the obfuscated code (e.g., by reverse engineering or code analysis). This first line of defense raises the bar of difficulty for a successful attack on one single node. Second, for different nodes, we make sure that the data and code obfuscation methods are different. This way, even if the attacks have successfully derived the location of the secrets, they cannot use the same location for another node, because for different nodes, their secrets are stored in different ways and in different places. Such diversity makes it a daunting job to derive the secret information from a large number of compromised nodes. We have implemented our scheme for Mica2 motes, and we present the results in this paper.

Categories and Subject Descriptors:
C.2.0 [Computer-Communication Networks]: Network Protocols, Wireless Communications

[*]This work was supported by Grant CNS-0430252 from the US National Science Foundation and also by Grant W911NF-05-1-0247 from the US Army Research Office.

General Terms:
Algorithms, Design, Security, Performance.

Keywords:
Diversity, wireless sensor networks, obfuscation, reverse engineering.

1. INTRODUCTION

Recent technological advances have made it possible to develop distributed sensor networks consisting of a large number of low-cost, low-power, and multi-functional sensor nodes that communicate over short distances through wireless links. Such sensor networks are ideal candidates for a wide range of applications such as monitoring of critical infrastructures, data acquisition in hazardous environments, and military operations [2]. When sensor networks are deployed in a hostile environment, security becomes extremely important, as they are prone to different types of malicious attacks. For example, an adversary can easily listen to the traffic, impersonate one of the network nodes, or intentionally inject misleading information into the networks. Therefore, because of their importance, it is necessary to guarantee the trustworthiness and resilience of sensor networks.

A great challenge in securing sensor networks is that sensor nodes can be physically compromised. Once a node is compromised, attackers can retrieve any secret information (e.g. keys) from the node. If they also understand the behavior of the software running on the node, they can totally control the behavior of the compromised node without violating the communication protocols used by the sensor networks. Since it is very likely for sensor-network applications to use commercial-off-the-shelf software, some of which might be open-source software (i.e., the source code is widely available), understanding the software behavior is not difficult.

This problem is aggravated in the key predistribution scenario. Key predistribution enables sensors to set up secret keys with their neighbors. Due to the energy constraint on sensor nodes, public key cryptography (which can provide an easy solution) is too expensive to use; therefore, studies of this security bootstrapping problem have been primarily focusing on symmetric key cryptography. Among the proposed schemes, *key predistribution* is a promising approach and has been studied extensively [13, 5, 11, 10, 21]. It is achieved by letting each sensor pre-load a set of keys (or key materials in some schemes) from a common key pool prior to deployment. After they are deployed to the sensing field, they use these key materials to establish secure communication with their neighbors. Because these pre-loaded keys can be used by multiple sensors, the damage of the compromise of one sensor can be magnified, and can thus compromise the security of the communication among non-compromised sensors. The research in key predistribution has been trying to reducing such undesirable impact. However, with limited resources in computation power, memory, and battery, the problem is difficult to be solved completely.

All the above problems boils down to one fundamental problem: *how can we protect secret information within sensor nodes while assuming that nodes can be physically captured by attackers?* One common solution is to use tamper-resistant hardware, such as Smartcards. Namely, keys are stored in this special hardware, and any computation that needs to use these keys is conducted within the hardware. While this can be a viable solution, it increases the cost and energy consumption of sensor nodes. Another solution is to use code obfuscation. Namely, obfuscation is a transformation of program code (source code or binary code) into another program with the same observable behaviors in order to prevent reverse engineering.

It is widely believed that code obfuscation can only raise the bar for attackers; it cannot totally hide secrets. Sooner or later, by reverse engineering the obfuscated code, dedicated attackers can eventually retrieve the secrets from a compromised node. However, it is also widely believed that compromising one or a few node does not always achieve an effective attack. Designs of sensor-network applications for hostile environments often assume that sensor nodes can be captured. Therefore, resilience against node capture is built into designs. For example, in key predistribution, various schemes have been proposed to increase resilience against node capture [10, 11, 4, 28]; in data aggregation, order statistics has been used to tolerate some number of incorrect readings from malicious nodes [30, 26, 25]. *Therefore, to achieve an effective attack, attackers need to capture a non-trivial number of sensors, and successfully retrieve secrets from these sensors.* This is the main observation that motivates this research.

Based on the above observation, if we can raise the bar of difficulty for a successful attack on one single node to a certain degree, the bar for a successful attack on a non-trivial number of nodes can be prohibitively high. It should be noted that this accumulation of difficulty is based on a very important condition: attackers' effort on compromising one node cannot save their efforts on compromising another node. In other words, if an attacker has spent 10 hours getting the secrets from one node, he/she might have to spend another 10 hours to do the same thing for another node. Therefore, achieving *non-repeatability* is a unique objective for code obfuscation in sensor networks because of the large number of sensor nodes in typical sensor-network applications.

1.1 Outline of Our Scheme

In this paper, we combine code obfuscation and randomization to protect the secret keys of sensor nodes. Our scheme diversifies data and code segments by creating different and obfuscated data and code segments for each node. Our scheme consists of three major steps: First, the data structure for storing secret keys is obfuscated. In our scheme, the k secret keys (each of length L) carried by each sensor are not stored in memory in a well-recognized data structure. Instead, they are scrambled using L hash functions; one has to know all these L hash functions in order to know all the L bits of a secret key. It is important to note that we choose different sets of hash functions for each node during obfuscation.

Second, code is obfuscated. Since these hash functions can be found in the code segment by using reverse engineering attacks, we need to obfuscate code to make the attacks difficult. However, code obfuscation only makes breaking one single node difficult; breaking the subsequently compromised nodes are quite simple because all the code share the same control flow. Therefore, the next step is necessary. In the third step, we randomize control flow of code, such that the control flow of one node is different from the control flow of the other nodes. In other words, after the third step, we have created a different version of the same code for different node.

Since each node has different hash functions and different obfuscated code, compromising a node will not reveal much useful information that can help simplify the process of compromising another node.

The contributions of our paper are summarized in the following:

- We propose a random data and code obfuscation scheme which is able to generate different version of sensor software for each sensor node. Although the versions are not completely different, data and code analysis of one version cannot help much to simplify the same process on another version.
- Our proposed scheme can be applied to many applications, where critical information needs to be protected. For the sake of substantiation, we present our scheme in the context of key pre-distribution schemes. Our scheme can make compromising a large number of secret keys much more difficult, a desirable goal for key pre-distribution.
- We have implemented our scheme for Mica2 motes, and have evaluated the performance of our scheme.

1.2 Organization

Our paper is organized as the following: In the next section, we present the related work. In Section 3, we present our main schemes, which include data disguising, code obfuscation, and random code generation. The performance results are presented in Section 4. Finally, we conclude the paper in Section 5.

2. RELATED WORK

Diversity. Diversity has been applied at the system level. It is mostly used to increase resistance to intrusion by increasing software complexity while preserving functionalities and performance. Diversification makes it hard to analyze and exploit vulnerabilities in software [19]. A variety of randomization techniques have been proposed, such as stack randomization [14], instruction set randomization [3, 18], library randomization [6], system call randomization [6], etc.

Diversification has also been applied at the network level to improve the diversity of networks. This is achieved by using different applications, operating systems, and communication protocols [16, 34] within a networked system.

Roux et al. developed cost adaptive mechanism (CAM) to provide network diversity for MANET reactive routing protocols [29]. Mont et al. introduced a new approach to ensure diversity for common, widespread software applications in which diversity is enforced at the installation time by a random selection and deployment of critical software components [24]. Hiltunen et al. have proposed the use of fine-grain customization and dynamic adaptation as key enabling technologies [17].

Wang et al. [32] have presented an intrusion tolerant architecture for distributed services, especially for COTS servers, which utilizes the techniques of both redundancy and diversity as building blocks. Ellison et al. [12] have described the survivability approach to help assure that a system that must operate in an unbounded network is robust in the presence of attack and will survive attacks that result in successful intrusions. Dasgupta et al. [8] have focused on investigating immunological principles in designing a multi-agent system for intrusion/anomaly detection and response in the network. Deswarte et al. [9] have focused on diversity, as a desirable approach for addressing the classes of faults that underlay all these topics, i.e., design faults and intrusion faults. O'Donnell and Sethu [27] have suggested using different software packages among neighbor nodes in sensor networks to improve resilience.

Obfuscation. The first classification of obfuscation was shown in [7], in which obfuscation is categorized as layout obfuscation, data obfuscation, control obfuscation, and preventive obfuscation. Layout obfuscation is the changing of structure for source code or binary code, such as scramble identifiers, change formatting, and remove comments. Data obfuscation is the changing of data structure, which is classified into storage and encoding, aggregation, and ordering. Control obfuscation is the changing of main skeleton of the program, which is classified into aggregation, ordering, and computations. Preventive obfuscation is intended to protect from decompilers and debuggers.

Wang [31] has proposed a new algorithm for obfuscation which is mainly consist of three parts: dismantling of control flow graph, flattening of control flow graph, and addition of structures with data aliasing. Wroblewski [33] has developed an obfuscation method that work on the low level of programming. Linn and Debray [20] have developed obfuscator of executable code to improve resistance to static disassembly.

Key Pre-distribution. Eschenauer and Gligor proposed the random key pre-distribution scheme [13], which is based on the idea that each node chooses m keys from a pool of keys of size S. Then two nodes have a link between them if and only if they have a common key. Based on the birthday paradox, a small number of keys chosen by each node are enough to achieve a high probability that two nodes share a common key. Based on Eschenauer and Gligor's work, a number of improvements have been proposed [5, 11, 10, 21]. A common property of these schemes is that each node carries a number of keys (or key materials for some scheme) in their memories. If a node is compromised, all the keys can be revealed to attackers. Since these keys are also used by other nodes, if attackers can compromise a set of nodes, the revealing of the keys stored in their memories can compromise the security of other nodes that are not compromised, which lowers the network resilience.

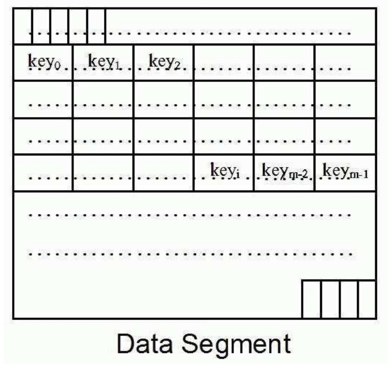


Figure 1: Array of the keys, which is indexing by node ID

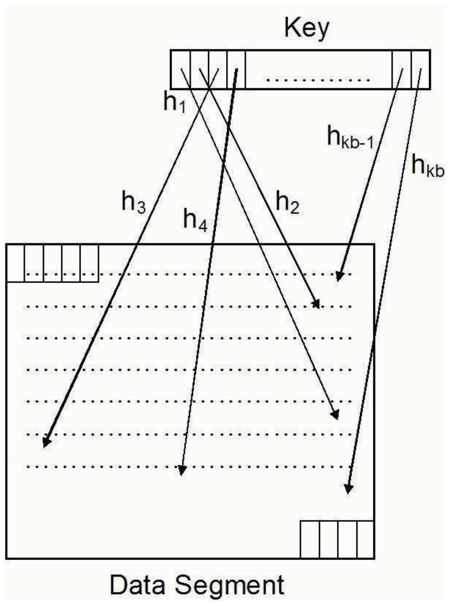


Figure 2: Using hashing function to obfuscate and scramble the keys in data segment

3. DIVERSIFY DATA AND CODE

Secure communication among sensor nodes are based on encrypted communication channels using a collection of keys. For example, in order to establish a secure connection between two nodes in random pairwise keys scheme, we should have a designated key shared between these nodes. All keys are stored in the memory of sensor nodes. Our goal is to protect these keys using obfuscation. The basic computing power of a Mote is provided by an Atmel ATmega 128 processor with 4KB dynamic data RAM, 128KB program ROM. We will use data segment here to refer to the memory dedicated for those keys, which are stored in the 128KB program ROM.

3.1 Data Disguising

Without loss of generality, we assume that keys are stored in an array with indices from 0 to $m-1$ (see Figure 1). Since this data structure is not a secret, once a node is compromised, attackers can easily identify the keys stored on this compromised node. To make attackers' task difficult, we can hide the key storage, making it more difficult for attackers to find the keys. However, hiding things in a program is quite difficult; sooner or later, attackers can find the storage. Since all the nodes save their keys in the same places, once knowing where keys are stored, finding the keys of the next compromised nodes will be a trivial task. Therefore, our goal is two-fold: first, we will make identifying and revealing the keys of the first compromised node a challenge task; second, we will make the data structures for different nodes different, so compromising a subsequent new node is as hard as compromising the first one.

To make identifying the keys of a compromised node a challenging task, we use hash functions to scramble the keys in data segment. Note that this is just the traditional hash functions used in the hash table data structure, rather than the cryptographic one-way hash functions. We use a different hash function for each bit of a key. Assume that the length of each key is kb, we define kb different hash functions, h_1, ..., h_{kb}, where h_j is used for the j-th bit of a key. We apply these hash functions on the index of each key, and the results are the indices of the hash table, where the actual bits of the key are stored. For example, if the index of a key is i, we apply a hash function h_1 on i; we use the first bit of $Keys[h_1(i)]$ to store the first bit of the key i. Similarly, the j-bit of $Keys[h_j(i)]$ stores the j-th bit of the key i. Figure 2 illustrates the scrambling scheme. Given such a scrambled data structure, each key is thus derived using the following equation:

$$K_i = \underset{j=1..kb}{||} \quad bit(Keys[h_j(i)], j)$$

where

$i = 0..m-1$	is Node ID.
m	is the number of keys.
kb	represent size of the key.
$Keys$	array data structure to store the keys.
h_j	hashing function for the j-th bit.
$bit(x, y)$	function that return value of y-th bit in x.
$\|\|$	is the concatenation operator.

We define the following hashing function:

$$h_j(i) = (a_j\, i + b_j)\ mod\ m, \tag{1}$$

where a_j, b_j, and m are hash parameters, and a_j and b_j are different for different hash functions.

There are two issues that we need to consider when selecting hash functions. First, each hash function has to be collision free; otherwise, we might have a undesirable situation where bits of two different keys are stored in the same location. Namely, we need to guarantee that $h_i(k) \neq h_i(k')$ if $k \neq k'$. However, collision free is not necessary for different hash functions; namely it is not a problem if $h_i(k_1)$ and $h_j(k_2)$ collides and have the same value w. Although these two hash values point to the same entry $Keys[w]$ of the hash table, the first one only chooses the i-th bit of $Keys[w]$ for key k_1 and the second one chooses the j-th bit of $Keys[w]$ for key k_2; as long as $i \neq j$, they choose two different bits (see Figure 3).

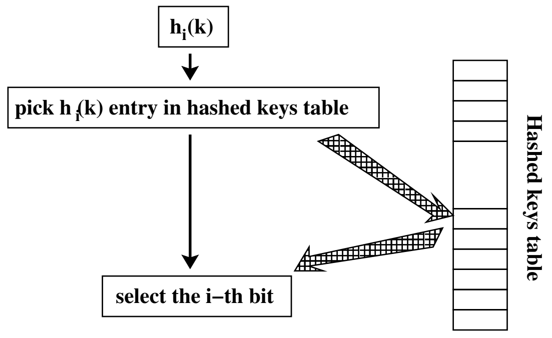


Figure 3: Avoiding collision between hashing functions

Second, hash functions introduce a large amount of unused memory space; we need to keep the unused space to the minimum. To address these two issues, we come up with the following selection criteria:

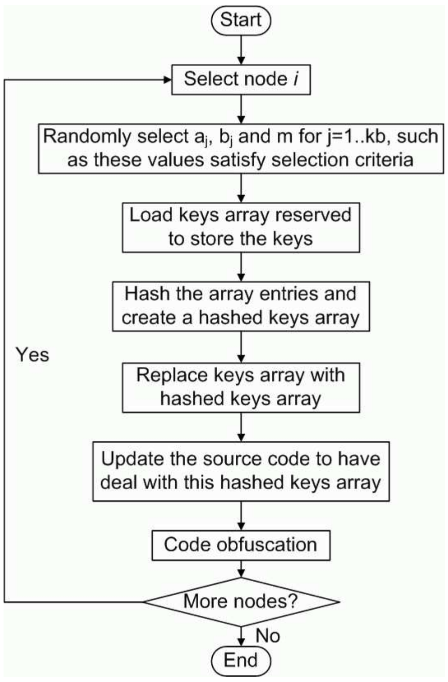
- a_j and b_j are positive integers that affect the number of collisions and space efficiency.
- The larger the value of m is, the lesser the number of collisions, but the more the un-used spaces.
- For faster computation of mod operation, m should be a power of 2; this way, the modulus operation can be implemented using shift, rather than using expensive divisions.
- b_j and m are relatively prime, so that they do not have common factors other than 1.
- Every prime factor of m is also a factor of $a_j - 1$.
- If m is a multiple of 4, $a_j - 1$ should be multiple of 4.

THEOREM 3.1. *Let c_j and d_j be positive integers. When $m = 2^{c_j}$, $a_j = 4d_j + 1$, and b_j is positive odd, h_j is collisions free. Moreover, there is no un-used space.*

PROOF. In the appendix. □

In our scheme, the values of a_j and b_j do not need to be large numbers. This is because each of these value pairs only corresponds to one single bit of a key. Therefore, using brute-force attacks to derive these a_j and b_j values is even harder than using brute-force attacks to directly guess each bit of the key. In other words, if attackers try to derive these hash parameters to get those keys, they have to derive all these hash parameters correctly. Based on this observation, we only use 8 bits for a_j and b_j.

Using different hashing functions for each node will add one layer of difficulty and will achieve our second goal, which is making the process of identifying and revealing the keys for subsequently compromised nodes as hard as that of the first one. That can be done, by randomly selecting values for a_j and b_j that satisfy the constraints above. In order for attackers to identify and reveal the keys for any node, the attackers should find out the values of a_j, b_j and m for $j = 1..kb$. If these values are stored in the data segment too, attackers' tasks become quite easy; they just need to find whether the values are stored. Therefore, it is important to hide these hash parameters.



Start
Select node i
Randomly select a_j, b_j and m for j=1..kb, such as these values satisfy selection criteria
Load keys array reserved to store the keys
Hash the array entries and create a hashed keys array
Replace keys array with hashed keys array
Update the source code to have deal with this hashed keys array
Code obfuscation
More nodes?
Yes
No
End

Figure 4: Keys protection outline

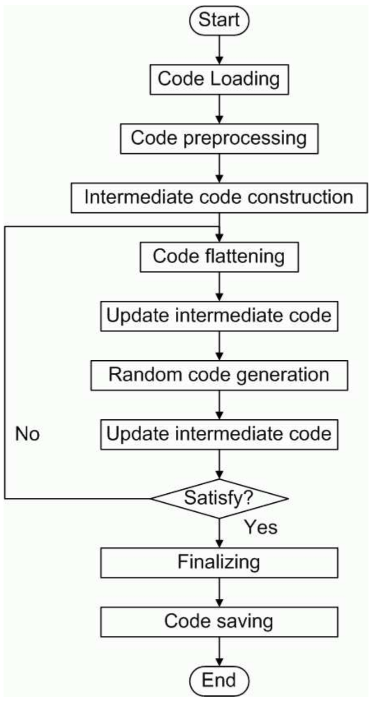


Figure 5: Code obfuscation outline

3.2 Code Obfuscation

We propose to hide a_j,b_j, and m values inside the code segment; then we apply *code obfuscation* to make code analysis a difficult task, so that attackers will find it difficult to extract these values. Our code obfuscator breaks down all important constants in the code and replaces them with pieces of code that construct the values. Moreover, our code obfuscator randomly generates different code for each node; therefore, even if the attackers, after code analysis, have successfully found the locations for these hash parameters in one compromised node, the analysis results will not help them to easily get the locations of the hash parameters in a new node. They have to repeat their code analysis on the new node.

Our code obfuscator consists of several components, including code preprocessing, intermediate code construction, code flattening, random code generation, and finalizing. Each component is responsible for part of the obfuscation, but some of them may run more than once.

3.2.1 *Code preprocessing*

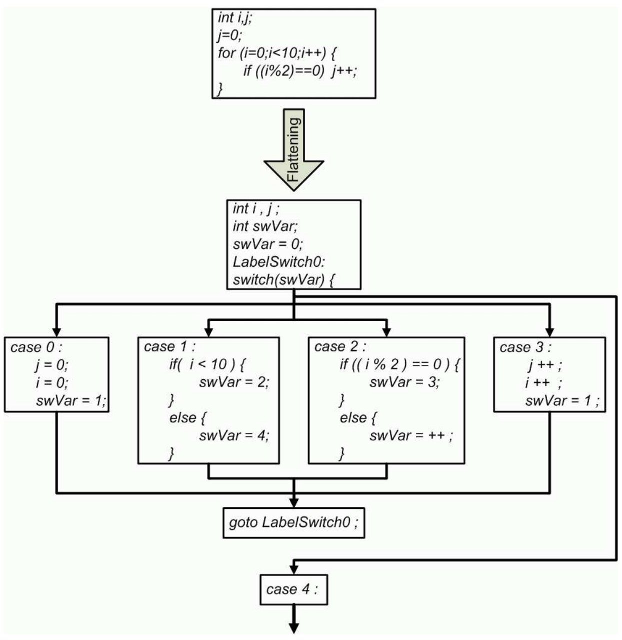
Code preprocessing is responsible for preparing the code for the other stages of obfuscation. It removes comments, standardizes loop and control instructions, splits variable declarations, and formats instructions. The main goal of this component is to make the job of the other stages easier. Dividing the obfuscator into separate stages simplifies the design, implementation, and testing.

3.2.2 *Intermediate code construction*

We need to transform a program to another semantically-equivalent program; our transformation is performed at the source code level. However, directly transforming the original source code written in NesC or C languages is a difficult task. We take a widely-adopted approach: first, we convert the source code of the original program into an intermediate code representation; and then we construct the target program from the intermediate code, which is usually much easier to deal with and can be processed faster [31, 33].

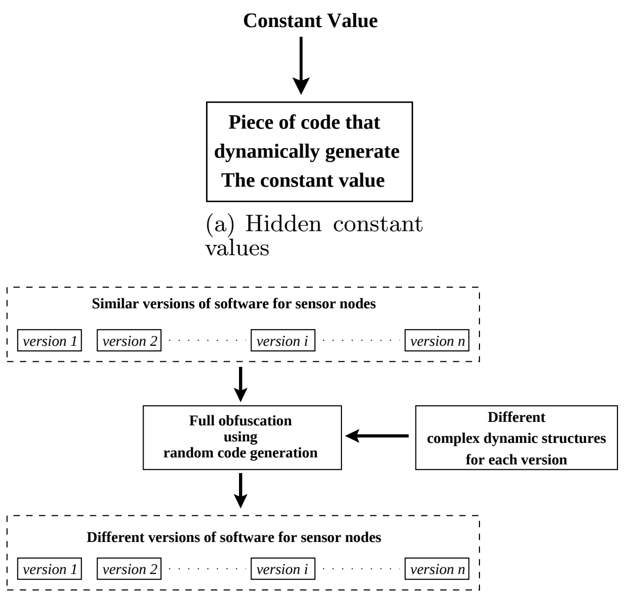
Converting source code into an intermediate code and working on this intermediate code has many advantages: intermediate code is simpler than the original code; it is more flexible to read and process intermediate code; the structures of intermediate code are clearer; tools are available for standard intermediate code. Moreover, using intermediate code simplifies the design, implementation, and maintenance of code obfuscator.

In the obfuscation related research, SUIF-2 [1] is a widely-adopted tool for creating intermediate representation. SUIF-2 is a compiler infrastructure project, which is co-funded by DARPA and NSF. It is a new version of the SUIF compiler system, a free infrastructure designed to support collaborative research in optimizing and parallelizing compilers. However, Since SUIF-2 was not developed for NesC language (the language used by TinyOS), a number of constraints and structures are not supported by SUIF-2. NesC has several characteristics that differ from other languages [15], such as the separation of construction and composition, the specification of component behavior in terms of set of interfaces, and the bidirectional interfaces. Moreover, in nesC, components are statically linked to each other via their interfaces, and nesC is designed under the expectation that code will be generated by whole-program compilers. Furthermore, nesC has it own statements that do not exist in the closest language C, such as connecting interfaces instructions, and events; it has a restricted variables declaration order where all variables should declared before any other statement. A few statements syntax also differs between nesC and C languages. All these differences require


```
int i,j;
j=0;
for (i=0;i<10;i++) {
      if ((i%2)==0)  j++;
}
```

```
int i , j ;
int swVar;
swVar = 0;
LabelSwitch0:
switch(swVar) {

case 0 :
    j = 0;
    i = 0;
    swVar = 1;

case 1 :
    if(  i < 10 ) {
          swVar = 2;
    }
    else {
          swVar = 4;
    }

case 2 :
    if (( i % 2 ) == 0 ) {
          swVar = 3;
    }
    else {
          swVar = ++ ;
    }

case 3 :
    j ++ ;
    i ++  ;
    swVar = 1 ;

goto LabelSwitch0 ;

case 4 :
```

Figure 6: Example of code flattening


Constant Value
Piece of code that dynamically generate The constant value
Similar versions of software for sensor nodes
version 1
version 2
version i
version n
Full obfuscation using random code generation
Different complex dynamic structures for each version
Different versions of software for sensor nodes
version 1
version 2
version i
version n

(a) Hidden constant values

(b) generate different versions

Figure 7: Objectives of random code generation

us to write a new front tool for SUIF to compile and read nesC code. Therefore, we decided to develop our own tool instead of using SUIF.

We use the Extensible Markup Language (XML) as our intermediate representation because of a number of reasons: (1) XML documents are easily readable and self-describing: it is easy for a person to read and understand a well-design xml document. (2) XML is interoperable: XML is open for everyone. (3) XML documents can be hierarchical: it is easy for XML to represent any complex hierarchical structure such as parsing tree. (4) We do not have to write the parser: there are many xml parsers available either as stand-alone programs or as object-based components, such as the XML parser in Microsoft .Net. We use the XML parser provided in C#.

3.2.3 Code Flattening

An effective way to make code analysis difficult is to make the control flow of a program difficult to understand. Control flow graph (CFG) is graph representation of all paths that might be traversed through a program during its execution. Each node in a control flow graph represents a basic block with no jump statement or jump target (label), while each edge represents a jump statement from the end of a source block to the beginning of a destination block. Control flow graph is essential not only to static analysis tools but also to many compiler optimizations.

Code flattening is a method for scrambling the control flow of a program by code transformation. It transforms the code into another code, in which all jump statements depend on dynamic values and may jump to any basic block. In other words, control flow graph will be transformed into a partial complete graph that is useless for static analysis. We need to use code flattening to eliminate and destroy the control flow graph (skeleton) of program code so that static analysis can not be used efficiently anymore.

Code flattening, first proposed by Wang [31], is performed in two steps. In the first step, all high-level control instructions are transformed into their equivalent if-then-goto instructions. In the second step, goto instructions are modified such that their target addresses are determined dynamically.

One way to do code flattening is to transform the control flow graph of each function into one big switch statement. In which, every case statement represents an edge in the flow graph before the next control statement (see Figure 6). In this manner, each function consists of one big switch statement, which hides the control flow skeleton of that function, while the branches are replaced with data-dependent instructions. However, the process of flattening can be reversed easily, because the data causing jumps between cases are constants. In our obfuscator, we replace each constant with a piece of code that deals with pointers. Injecting these pieces of code into the original code make static analysis of the code difficult, because the shape of the control flow graph is now dependent on the data that cannot be found out using static analysis. More details about injecting these codes are given in the next subsection (random code generation).

Code flattening can be easily achieved if the source code has already been transformed to an intermediate code, which represents the syntactic structure tree of the source code. Given the control flow that can be extracted easily from the tree, each edge in the graph can be transformed as one case in the flattening switch statement. The complexity of this process is as hard as processing and querying intermediate code, so that the more flexible and simpler intermediate code is, the easier the flattening process is.

3.2.4 Random code generation

To diversify sensor node, we use random code generation to create different version of the same code for each node, as well as to hide constant data. In order to make the output code hard to analyze, random code generation uses a randomly generated dynamic data structure. The inserted

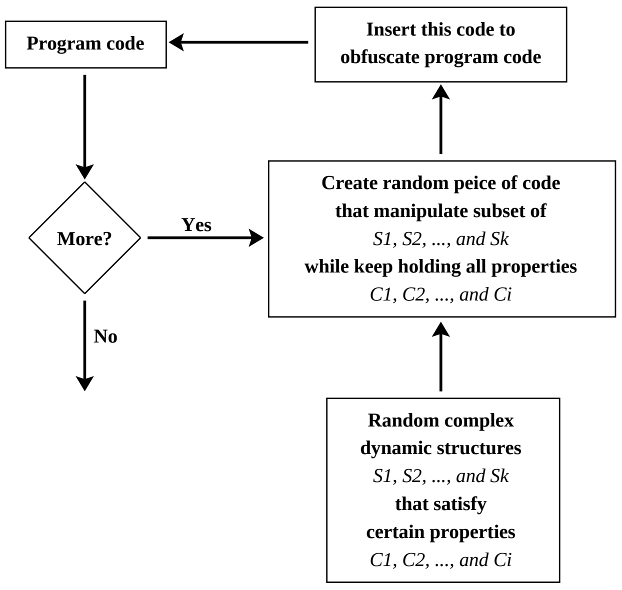

Program code
Insert this code to obfuscate program code
More?
Yes
No
Create random peice of code that manipulate subset of
S1, S2, ..., and Sk
while keep holding all properties
C1, C2, ..., and Ci
Random complex dynamic structures
S1, S2, ..., and Sk
that satisfy certain properties
C1, C2, ..., and Ci

Figure 8: Random code generation cycle

code handles and processes this structure in such a way that certain properties about the structure are preserved, where these properties are randomly created. The structure involves a lot of pointers and dynamic data structures, because it is well-known that analyzing code that deals with pointers and dynamic data structures is a difficult task and cannot be done easily using static analysis. These pieces of code that we generate can be used to construct constant data, so we do not need to store these constant data in plaintext in the code. Moreover, they can also be used to insert control flow instruction that does not affect the output of the obfuscated code. As a result of that, attacker should identify and understand the inserted code in order to analyze the obfuscated code. Since the dynamic data structure and its reserved properties are randomly generated, attacker should start from zero every time he/she analyzes the code of a newly captured sensor node; the correlation of code among nodes is very low.

The objectives of random code generation can be summarized as the following (see Figure 7):

1. Generating a different version of code for each sensor node. Without random code generation, the codes running on each sensor node are not diversified.
2. Hiding constant values in the code, by replacing them with dynamically generated values based on a sequence of expressions.

The process of random code generation is described in the following:

- Each version is based on different and randomly generated complex dynamic structure S_1, ..., S_k, such that no two versions share the same dynamic structures (see Figure 8). S_i could be any dynamic data structure that is complex to analyze or understand, such as dynamic graph data structures.
- Each piece of code inserted to the program code is a randomly generated code that manipulate S_1, ..., S_k while preserving the properties C_1, ..., C_i about the structures, such that no two pieces of code are similar. C_i represents a certain property about S_1, ..., S_k. Values storing in certain locations have certain property (e.g. fix value, even, odd, or follow a certain relation), or some pointers are always point to different isolated graphs.
- C_1, ..., C_i are different from one version to another.
- Using code flattening to disguise the control flow and to make the task of finding similarity among versions difficult. In this case, static analysis will not be much useful.

In order to make it more difficult for attackers, code flattening and random code generation can be applied more than once until we get a best result within the resource constraints. Since sensor nodes have limited resources, it may limit any obfuscator from producing a un-breakable code. However, as we have stated before, the objective of our proposed scheme is to make the cost of breaking a non-trivial number of sensor nodes prohibitively high.

3.2.5 *Finalizing*

This is the final stage, in which we conduct some further obfuscation, such as removing comments, changing the formatting, changing identifiers (e.g. method and variable names) into meaningless names, and removing debug information. We also pre-process the intermediate code to be transformed back to nesC code.

3.3 Optimization

Sensor nodes (e.g. Mica2) have limited memory, limited CPU power, and limited batteries. Our scheme output should run within these constraints and limitations, while achieving our diversity goals. When the program source code is really large, resources slack will be small which will have direct effects on the diversity quality and performance. There are a number of methods to optimize the performance of our scheme within a short resource slack. All these methods are based on the facts that not all codes have the same importance from the security prospective. In this subsection, we consider several optimization techniques.

The first technique reduces the number of constant values that need to hidden using temporary variables. For example, if we have two variables that need to be set to the same value, we can set one of the variables to that value and get the value of the other variable from the first one. In this manner, the number of constant values appearing in the source code is reduced.

The second technique leaves some functions un-obfuscated. These functions can be selected before obfuscation starts by using special tags as indicator. Functions that are not considered for obfuscation are less important, from the security prospective, than the other functions. Namely, being able to understand the source code for these functions does not reveal much information about the keys.

The third technique randomly ignores some constant value from our random code generation. In the same manner as we have in the second technique, values that need not to be considered for obfuscation should be selected before obfuscation starts by using special tag. Values are not equally important. For example, values related to hashing functions

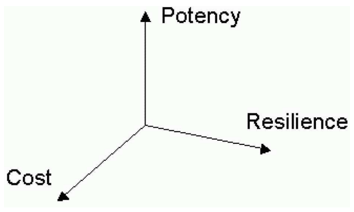


Figure 9: Obfuscation transformation measures, according to [33]

such as a_j, b_j, and m are extremely important than values related to other different tasks. Instead of using tags to mark values that need to be obfuscated, we can prioritize these values so that the obfuscator can compare the importance of each value with other. In this way, obfuscator can start with the highest priority values first; if resource permits, the obfuscator can continue obfuscating the values with lower priority, and so on until we reach the desirable results.

4. PERFORMANCE EVALUATION

We have implemented out scheme on Mica2 sensor nodes, we present the performance results in this section. We divide this section into 3 parts. First, we explain the evaluation metrics; second, we show the configuration of our experiments; finally, we present our results and explain their meaning.

4.1 Performance Metrics

To evaluate the effectiveness and efficiency of our obfuscation scheme, a number of things need to be measured. Collberg et al. classify the criteria for evaluating a code obfuscation scheme into three categories [7] (also see figure 12):

- Potency: measure of code complexity for human to understand.
- Resilience: measure how hard and protected the code is against automatic deobfuscator.
- Cost: measure how much resource is needed by the obfuscated code.

Based on the above classification, and considered the special properties of sensor nodes, we have developed the following metrics, and will use them to evaluate our scheme.

1. Running Time: The running time is a very important factor and constraint for wireless sensor network, because of its limited power and the fact that data might be time sensitive. Obfuscated code always need more time than the original code; however this increase should be reasonable and should not exceed the resource capabilities and application constraints. Any obfuscated code that takes much longer time to run than its original code will be inapplicable.

2. Memory: sensor nodes have limited amount memory; for example Mica2 motes have 128K bytes of program flash memory (ROM) and 4k bytes of physical RAM. Our scheme should be able to work within these limits, and should not generate programs that exceed the memory constraints.

3. Lines of code: the oldest and most commonly used measure of source code program length is the number of lines of code (LOC). It was a naive way to measure the complexity of a program. Two reasons why LOC is popular are that it is easy to calculate, and most developers have an intuition as to what a line of code is. LOC metric has never been formalized and defined clearly by any standards organization. There are two major types of LOC measures: physical LOC and logical LOC. Physical LOC is mainly the number of physical lines in the text of the program's source code while logical LOC measures the number of statements which is specifically defined by their corresponding programming languages. In our paper, we use the most commonly used physical LOC–the count of non-blank and non-comment lines in the text of a program's source code.

4. Cyclomatic complexity: Cyclomatic complexity, developed by Thomas McCabe, is one of the most widely and effective software metrics [22, 23]. It is used to measure the complexity of a program by measuring the number of linearly-independent paths in the program. The cyclomatic complexity of a software module is calculated from a control flow graph of the module using the following formula:

$$CC = E - N + P$$

where CC is cyclomatic complexity, E is the number of edges of the graph, N is the number of nodes of the graph, and P is the number of connected components.

4.2 Experiments Configuration

In this subsection, we present our experiment configuration, which includes obfuscator input, the obfuscator itself, and output.

Input: The RC5 application program was our input for obfuscator. It is a program that uses the RC5 encryption/decryption algorithm to encrypt/decrypt a sequence of data block. RC5 is a fast block cipher designed by Rivest in 1994.

RC5 has many features that make it a desirable choice for sensor networks, such as:

1. It is a parameterized algorithm with a variable key size, variable number of rounds, and a variable block size. The key can range from 0 bits to 2040 bits in size, while the number of rounds can range from 0 to 255. The block size can be chosen as 32 bits for experiments and evaluation, 64 bits for DES replacement, or 128 bits for high security application. This provides a great flexibility to RC5 in term of performance, speed, and security.

2. The encryption routine consists of three primitive operations; integer addition, bitwise XOR, and variable rotation. This simplicity of RC5 makes it easy to implement and analyze.

3. Heavy use of data-dependent rotations and the mixture of different operations provide the security of RC5.

4. RC5 is a fast block cipher, which make it a good choice for wireless sensor networks.

There are different encryption/decryption algorithms that can be used instead. However we have selected RC5 for the features it provides beside that it has been implemented in TinySec, which is commonly used in the TinyOS community. We have selected the following values as the configuration of our experiments.

- Number of rounds is 12.
- Block size is 8 bytes (64 bits).
- Key size is 8 bytes (64 bits).
- TinySec implementation.

Although we believe that obfuscating the entire whole TinyOS code will give better results, we have obfuscated only the application code rather than the entire TinyOS code. We have done that because when the TinyOS executable file – which is loaded to sensor node – is generated, it mainly consists of two separate parts: one for the application and the other for the TinyOS libraries. Mixing them together is difficult, but will be the pursue of our future work. Furthermore, the parts of code that we do not obfuscate include the library codes and machine-dependent code that are not much important from the security prospective. They are not dealing with hashing keys, parameters, or even the main logic of our code, so that revealing these parts and being able to read them and understand them does not help attackers much to achieve their goals.

It is really important to notice that, we do not just obfuscate the RC5 algorithm; we apply our scheme to the entire RC5 application code and the libraries it uses. However, we do not obfuscate the TinyOS libraries.

Obfuscator: As we mentioned above, we have different phases; hashing, flattening, and constant hiding and code generation. We have done the following experiments based on the configuration of these phases:

- Original program: in which we simply run the original un-obfuscated code.
- Data obfuscation: in which we apply the data obfuscation only; the goal is to scramble the keys using hashing functions.
- Data and one pass code obfuscation (Flattening only): beside our data obfuscation, we apply one pass of code flattening only.
- Full one pass obfuscation: in which a full one pass data obfuscation and code obfuscation is applied. Since full obfuscation will increase time and memory usage, we have also implemented four different optimization (they are described in the descending order in terms of their obfuscation strength): (1) full one pass obfuscation 1, where we do not have any optimization; (2) full one pass obfuscation 2, where the number of constant values that need to hidden is reduced using temporary variables; (3) full one pass obfuscation 3, where the functions that do not need to be obfuscated are identified using special tags; (4) full one pass obfuscation 4, where some constant values are ignored in our random code generation.

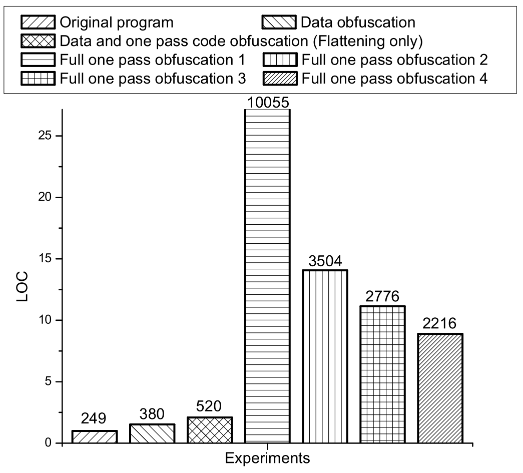


Figure 10: Lines of code (LOC)

Output: We have compiled and built the output codes to run under Mica2 platform. We used the default compilation and building flags and options that are defined in the Makerules file. The optimization flag is defined as -Os which is the default value too. We used the same rules for all experiments.

4.3 Experiments Results

We have plotted the results as bar charts. In each figure, we set the performance of the original code as the basic unit 1, so the y-axe shows the relative performance data of the obfuscated code to the original code. The absolute performance values are written on the top of each bar.

4.3.1 Code Complexity

To demonstrate how hard it is for attackers to analyze our obfuscated code, we use two well-known metrics: one is to use the number of lines, and the other is to use the cyclomatic complexity.

Our scheme and obfuscator are expected to increase the number of logical instruction and lines of source code. Increasing lines of codes (LOC) represents the increase in the complexity of the code, because new lines are used to add complex structure and code to our original input code. Figure 10 shows the increase in LOC. We have found that applying data obfuscation and code flattening only does not increase LOC much. This is because data obfuscation does not add much code, while code flattening focuses on hiding the skeleton of control flow rather than increasing lines of code. However, the full code obfuscation increases the lines of code very significantly. The increase of LOC will be reduced when optimization is applied.

Cyclomatic complexity is one of the most widely-used and effective software metrics. Cyclomatic complexity metric is collected by a static analyzer from source code. We plot our results in Figure 11. The results show that our code obfuscation increases the cyclomatic complexity quite drastically. For example, the full obfuscation (without optimization) can increase the cyclomatic complexity by 95 times; even with optimization (e.g. level 3), the cyclomatic complexity can still be increased by 27 times. It is believed that the cyclo-

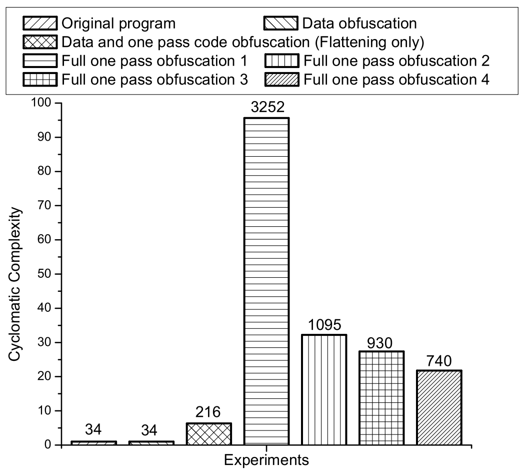


Figure 11: Cyclomatic complexity

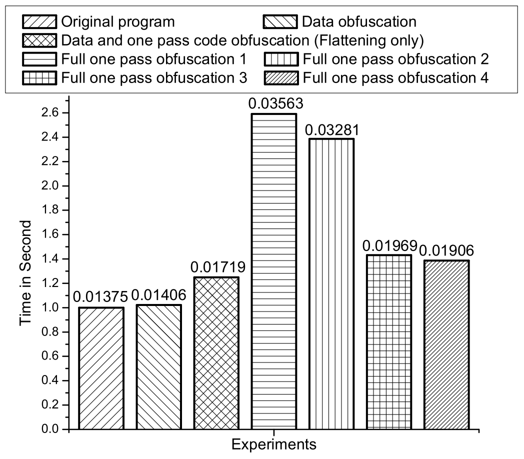


Figure 12: Running time in seconds

matic complexity gives a good indicator on how difficult a code can be understood by human.

4.3.2 Running Time

For real-time applications, time is an important factor. In this experiment, we measure the running time of the program under mica2 platform. We compare the running time of the obfuscated code with that of the original code. Figure 12 depicts the results. From the figure, we can see that the increase caused by data obfuscation is trivial (only 2%). When we apply the data obfuscation and code flattening together, the running time is increased by 25%. When we apply full obfuscations, the running time is increased quite substantially. Our results have also shown that when optimization level is increased, the running time can drop. Of course, as we have shown before, aggressive optimization can negate some of the obfuscations and can thus make code analysis easier for attackers. However, It is worth to note that when we use the "full one pass obfuscation 3", the running overhead is reduced to just 43%. As we can see from Figure 10 and Figure 11, at this level of optimization, the complexity of the obfuscated code is still reasonably high. Therefore, this level of optimization seems a good tradeoff between code performance and complexity.

It should also be noted that the running-time overhead is just for encryption/decryption, which only accounts for a small portion of the running time in most applications. Therefore, the impact of the code obfuscation on applications is quite small.

4.3.3 Memory

Memory is also a limited resource in sensor nodes, especially the RAM, because Mica2 nodes have 4kbytes of RAM and 128 Kbytes of ROM. We have measured the memory usage for both original RC5 code and the obfuscated RC5 code. Our results are depicted in Figure 13 for RAM and in Figure 14 for ROM.

Our results have shown that code obfuscation does not have any impact on the RAM usage. However, it does have a substantial impact on ROM usage. From the results, we can see that without any optimization, the full code obfuscation

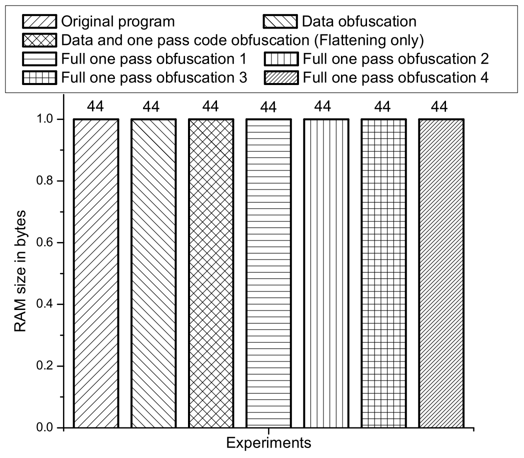


Figure 13: RAM in bytes

can make the code about 12 times as large. With certain degree of optimization, the size can be reduced to 6 times as large for "full one pass obfuscation 3". These increases are justifiable because more code should be inserted to achieve and scrambling and data obfuscation. Although the increase of ROM usage is substantial, we believe that since RC5 is the only program that we need to obfuscate, the impact on the overall ROM usage (including applications) will still be reasonable. Moreover, since the unused ROM will be mostly wasted once a sensor is deployed, it is desirable to use those unused ROM space for code obfuscation purpose.

5. CONCLUSION

Our paper shows that diversity approaches can be applied to sensor nodes and TinyOS. We have implemented a scheme to diversify sensor nodes to improve resilience against capture. Our scheme is a diversified key protection scheme for sensor network, which diversifies data and code segments by creating different and obfuscated data and code segment for

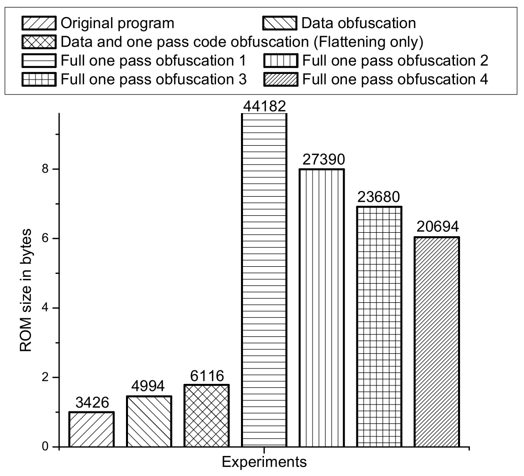


Figure 14: ROM in bytes

each node in the network. We have implemented our scheme for Mica2 motes.

Although sensor nodes have limited resources, the way nodes are deployed does not require a perfect obfuscation scheme. Our scheme achieves the following goal: attackers' effort on compromising one node cannot save their efforts on compromising another node. Therefore, achieving non-repeatability is a unique objective for code obfuscation in sensor networks because of the large number of sensor nodes in typical sensor-network applications.

There are many open research areas for applying diversity to enhance security in wireless sensor networks, such as applying diversity for key predistribution scheme, localization in wireless sensor networks, energy management, and network management. Our results have shown that using diversity to enhance security in sensor networks is a viable solution.

6. REFERENCES

[1] The suif 2 complier system, 2005.

[2] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114, August 2002.

[3] E. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, and D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *In Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 281– 289, October 2003.

[4] C. Blundo, A. De Santis, A. Herzberg, S. Kutten, U. Vaccaro, and M. Yung. Perfectly-secure key distribution for dynamic conferences. *Lecture Notes in Computer Science*, 740:471–486, 1993.

[5] H. Chan, A. Perrig, and D. Song. Random key predistribution schemes for sensor networks. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 197, Washington, DC, USA, 2003. IEEE Computer Society.

[6] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, dec 2002.

[7] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, July 1997.

[8] D. Dasgupta. Immunity-based intrusion detection system: A general framework. In *In Proceedings of 22nd National Information Systems Security Conference (NISSC)*, pages 147–160, 1999.

[9] Y. Deswarte, K. Kanoun, and J-C. Laprie. Diversity against accidental and deliberate faults. In *CSDA '98: Proceedings of the Conference on Computer Security, Dependability, and Assurance*, page 171, Washington, DC, USA, 1998. IEEE Computer Society.

[10] W. Du, J. Deng, Y. S. Han, S. Chen, and P. Varshney. A key management scheme for wireless sensor networks using deployment knowledge. In *Proceedings of the IEEE INFOCOM'04*, pages 586–597, Hongkong, China, March 7-11 2004.

[11] W. Du, J. Deng, Y. S. Han, and P. Varshney. A pairwise key pre-distribution scheme for wireless sensor networks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 42–51, Washington DC, USA, October 27–31 2003.

[12] R. J. Ellison, D. A. Fisher, R. C. Linger, H. F. Lipson, T. A. Longstaff, and N. R. Mead. Survivability: Protecting your critical systems, 1999.

[13] L. Eschenauer and V. D. Gligor. A key management scheme for distributed sensor networks. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 41–47, New York, NY, USA, 2002. ACM Press.

[14] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[15] D. Gay, P. Levis, D. Culler, and E. Brewer. nesc 1.1 language reference manual, May 2003.

[16] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. P. Pfleeger, J. S. Quarterman, and B. Schneier. Cyberinsecurity: The cost of monopoly. Technical report, 2003.

[17] M. A. Hiltunen, R. D. Schlichting, C. A. Ugarte, and G. T. Wong. Survivability through customization and adaptability: The cactus approach. In *In Proceedings of DARPA Information Survivability Conference and Exposition*, pages 294–307, 2000.

[18] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280, New York, NY, USA, 2003. ACM Press.

[19] R. C. Linger. Systematic generation of stochastic diversity as an intrusion barrier in survivable systems software. In *HICSS '99: Proceedings of the Thirty-Second Annual Hawaii International Conference on System Sciences-Volume 3*, page 3062, Washington, DC, USA, 1999. IEEE Computer Society.

[20] C. Linn and S. Debray. Obfuscation of executable

code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, New York, NY, USA, 2003. ACM Press.

[21] Donggang Liu and Peng Ning. Establishing pairwise keys in distributed sensor networks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 52–61, New York, NY, USA, 2003. ACM Press.

[22] Thomas J. McCabe. A complexity measure. *IEEE Transactions of Software Engineering*, SE-2(4):308–320, December 1976.

[23] Thomas J. McCabe and Charles W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, 1989.

[24] 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, Oct. 2000.

[25] R. Nowak and U. Mitra. Boundary estimation in sensor networks: Theory and methods. 2nd International Workshop on Information Processing in Sensor Networks, April 2003.

[26] R. D. Nowak. Distributed em algorithms for density estimation and clustering in sensor networks. In *IEEE Transactions on Signal Processing, Special Issue on Signal Processing in Networking.* IEEE Press, 2003.

[27] A. J. O'Donnell and H. Sethu. On achieving software diversity for improved network security using distributed coloring algorithms. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 121–131, New York, NY, USA, 2004. ACM Press.

[28] A. Perrig, R. Szewczyk, V. Wen, D. E. Culler, and J. D. Tygar. SPINS: security protocols for sensor netowrks. In *Mobile Computing and Networking*, pages 189–199, 2001.

[29] N. Roux, J-S. Pegon, and M. Subbarao. Cost adaptive mechanism to provide network diversity for manet reactive routing protocols. In *In proceedings of MILCOM 2000*, pages 287–291, Oct. 2000.

[30] D. Wagner. Resilient aggregation in sensor networks. In *SASN '04: Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*, pages 78–87, New York, NY, USA, 2004. ACM Press.

[31] C. Wang. *A Security Architecture for Survivability Mechanisms.* PhD thesis, Department of Computer Science, University of Virginia, October 2000.

[32] F. Wang and R. Uppalli. Sitar: A scalable intrusion-tolerant architecture for distributed services, 2003.

[33] G. Wroblewski. *General Method of Program Code Obfuscation.* PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.

[34] Y. Zhang, H. Vin, L. Alvisi, W. Lee, and S. K. Dao. Heterogeneous networking: a new survivability paradigm. In *NSPW '01: Proceedings of the 2001 workshop on New security paradigms*, pages 33–39, New York, NY, USA, 2001. ACM Press.

7. APPENDIX

Proof of Theorem 3.1. In order to prove that h_j is collision free and there is no un-used space, we have to prove that the following conditions are satisfied

- a_j and b_j are positive integers.
- b_j and m are relatively prime, so that they do not have common factors other than 1.
- Every prime factor of m is also a factor of $a_j - 1$.
- If m is a multiple of 4, $a_j - 1$ should be multiple of 4.

These conditions as we explained in our paper, represents the require selection criteria in order the hashing function to be collision free and does not have un-used space. These rules are based on linear-congruential generators and theory of congruences.

Since

$$\begin{aligned} d_j \text{ is a positive integer} \quad &\Rightarrow 4d_j + 1 \text{ is positive integer} \\ &\Rightarrow a_j \text{ is positive integer} \end{aligned} \tag{2}$$

$$b_j \text{ is positive odd} \quad \Rightarrow b_j \text{ is positive integer} \tag{3}$$

From 2 and 3, the first condition is satisfied.

Since b_j is odd, then 2 is not a prime factor for b_j.
However, $m = 2^{c_j}$ which means that 2 is the only prime factor for m.
Therefore m and b_j do not share any prime factor.
Therefore m and b_j are relatively prime and the second condition is satisfied.

Since

$$4d_j|2 \quad \Rightarrow a_j - 1|2 \tag{4}$$

$$m = 2^{c_j} \quad \Rightarrow \text{ 2 is the only prime factor of m} \tag{5}$$

From 4 and 5, every prime factor of m is also a prime factor of $a_j - 1$, and condition 3 is satisfied.

Since

$$4d_j|4 \quad \Rightarrow a_j - 1|4 \quad a_j - 1 \text{ is multiple of } 4 \tag{6}$$

From 6, condition is satisfied.

Therefore all conditions satisfied and the theorem is proven.