SECURE EXECUTION PARTITIONING OF WEAK DEVICES

THESIS

Presented to the Graduate Council of
Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Sobit Bahadur Thapa, B.E.

San Marcos, Texas

May 2013

SECURE EXECUTION PARTITIONING OF WEAK DEVICES

Committee Members Approved:

Qijun Gu, Chair

Hongchi Shi

Mina S. Guirguis

Approved:

J. Michael Willoughby
Dean of the Graduate College

ACKNOWLEDGEMENTS

I would like to thank Dr. Qijun Gu, my thesis and research advisor, for his long supervision and contribution. Without his guidance and thoughtful support, my research work could not have taken place. I owe a huge debt of gratitude to his kindness and patience. I would also like to thank Dr. Hongchi Shi and Dr. Mina S. Guirguis for agreeing to be on my committee and for their insight and help.

Undoubtedly, I would like to thank my parents and my wife whose support allowed me to pursue my graduate degree.

This manuscript was submitted on March 7, 2013.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

SECURE EXECUTION PARTITIONING OF WEAK DEVICES

by

Sobit Bahadur Thapa, B.E.

Texas State University-San Marcos

May 2013

SUPERVISING PROFESSOR: QIJUN GU

The performance of weak embedded devices like cellular phones and sensors in Wireless Sensor Network (WSN) can be improved significantly by partitioning execution between the nodes in the network. Idle nodes can help busy nodes by executing their code which can lead overall system to achieve high performance gain with increased lifetime. At the same time, security policy of the system can be compromised by exposing private data of a particular node to an external node. Also, attackers can obtain execution flow and critical data by listening to the network because these devices communicate all the time to help each other.

This research studies an optimized framework for distributing and completing tasks securely in a network of weak embedded devices. Analysis of security and performance will be performed on this framework to verify its security, effectiveness and efficiency.

CHAPTER 1

INTRODUCTION

With the fast pace of technological advancement, embedded devices are gaining more computing and communication capacities. Several recent works [18, 24, 28] have demonstrated that their rich computing resources can enable them to perform beyond individual devices. Instead of working for only specific applications, a collection of them can provide a new mobile computing environment where computation inside a single device can be offloaded to other devices in a coordinated manner. This new computing approach can save the resource (such as energy) of a busy device and improve the overall system performance by utilizing the computing resources of nearby idle devices. Such an offloaded computing approach is posed to revolutionize the way we utilize mobile devices in providing new services and supporting emerging applications in the near future.

However, this new computing approach is facing several key challenges. One challenge is how to offload computation to remote devices while managing the sequence of execution inside both the originating device and the collaborating devices. Several code offloading approaches have been developed in mobile cloud [8, 9, 10, 11] that partition an application into several computational components and then offload some of the components to a cloud computing service. This kind of offloading approach treats offloading as an application-oriented service between applications and operating systems. The offloading service connects the cloud service and coordinates the execution of the offloaded application components between the originating device and the cloud service.

Although this type of offloading service is feasible to high-end mobile devices (such as smart phones or robots), it may not be the choice for weak embedded devices (such as sensors) that highly desire a compact and optimized code structure in their applications and thus do not want an additional service. Hence, a new computation offloading scheme is needed that can be tightly integrated into the device programs.

Another challenge is on the security. A few security issues were studied regarding access control [16, 25], integrity [5, 17] and key management [23] for computation offloading in mobile devices. Nevertheless, data confidentiality involved in such offloaded computation is not well studied yet. Because the offloaded computation needs data from the originating device, the remote devices could get access to critical data inside the originating device through the offloaded computation. For example, the offloaded computation may need the key from the originating device to access some other devices or verify offloaded data. If attackers compromise a few remote devices, they may obtain the chance to access such critical data by provide computing service from the compromised devices. Offloaded computation may also expose private information of the originating device. For example, the offloaded computation may need the log of the past operations of the originating device. If the log is disclosed to a compromised device, attackers can figure out what the originating device worked on in the past.

Protecting devices from such data confidentiality and privacy issues is very challenging. One reason is that the computation is usually non-deterministic. It is hard to know if a critical data item may be accessed during the offloaded computation. Another reason is that data is transformed in computation. A data item may be used in computation that produces some results. Then, the offloaded computation may access the results which can disclose the original data item. To protect critical data inside weak embedded devices, we need new lightweight secure measures suitable to weak embedded devices.

To address the above challenges, we propose a new secure task distribution scheme for networked weak embedded devices. The scheme consists of a distributed task scheduler and a run-time data access controller. Differing from the existing code offloading service approaches, the new scheduler is an enhancement to a typical OS scheduler and works as an integral part of an embedded OS. It dispatches the offloaded computation as remote tasks to peer devices. It manages the remote tasks with the local tasks in a seamless manner. The run-time data access controller utilizes the results of dynamic analysis on the embedded applications to identify the critical data flows that may lead to the breach of critical data. It then instruments the application code by tagging and tracking the critical data flows. It performs security check at the task distribution point to ensure that the distributed tasks will not carry any data that may disclose the critical data.

This thesis has the following contributions: (1) distributed task scheduler; (2) data flow analysis; (3) minimum tracking set; (4) sensor implementation.

In this thesis, the computation offload is implemented by modifying the scheduling policy of the operating system. The scheduler of TinyOS is modified to enable the task distribution to peer devices at runtime. An efficient and secure dynamic data flow analysis is implemented by using the optimized results of the static analysis which is performed offline. The scheduler assures the prevention of critical data exposure by checking the security tag value calculated by the dynamic data flow analysis at runtime.

The rest of the thesis is organized as:

Chapter 2 discusses the related works with this thesis.

Chapter 3 describes the overall system overview.

Chapter 4 provides the static analysis scheme.

Chapter 5 presents the secure dynamic data flow analysis.

Chapter 6 shows the implementation of the TinyOS scheduler modification, the static analysis scheme and the secure dynamic analysis.

Chapter 7 presents the environment setup for the experiments and the evaluation of the results.

Chapter 8 discusses the conclusion and future works of the thesis.

CHAPTER 2

RELATED WORKS

2.1 Mobile Cloud

In mobile cloud computing, several schemes were developed to offload computing from mobile devices to the cloud. One approach is based on whole image clone [9], where a clone of a mobile phone is created in cloud. Then, the state of the phone and the clone is synchronized periodically or on-demand. When the mobile phone encounters intensive computation, it suspends the local execution but asks the cloud to continue the execution. Once the cloud completes the execution and returns, the phone resumes its local execution. Another approach partitions mobile programs into modules and utilizes a middleware manager between applications and phone's operating systems to coordinate the offloading and the execution of mobile programs [13]. The middleware manager handles the migration of data and code associated with the offloaded modules. Static and dynamic partition algorithms [10, 11, 13] were developed to optimize the performance of offloading. Apparently, these approaches are not suitable to networked embedded devices as they may not have connections to a cloud.

Because the quality and cost of connections to the cloud may not hold in a mobile environment, a few works [12, 14, 24] have looked into the possibility of utilizing peer mobile devices to provide computing service. But, similar to mobile cloud, a few middleware frameworks were proposed to support offloading mobile code to other mobile devices [12, 24]. The difference is that the middleware frameworks coordinate offloading and execution among mobile devices. Another work [14] developed a probabilistic framework where a program can be partitioned into modules that are then mapped for execution on a set of mobile devices to ensure dealing with uncooperative and malicious devices without the need for connections to the cloud. But, this work emphasized on the selection of peer mobile devices, and did not actually study the mechanism of offloading. In summary, none of these works provide a suitable mechanism to handle offloading among embedded devices.

2.2 Security in Code Offloading

Many existing works on code partition and offloading focused on optimizing the overhead of migration and execution with partitioned code modules. Security is only considered in a few works. One of the works studied the confidentiality and integrity of partitioned programs in a web settings [7]. It assigns security labels to web programs according to a predefined security policy. Then, security checks can be performed on the security labels during the execution of the web programs. It ensures that the web programs are partitioned, placed and executed in the right place securely and efficiently. In another work [31], Sedic was proposed to address the privacy issue in cloud computing. It partitions a computing job according to the security levels of the data in use. Then, it decides the computation offloading based on the allowed replication and placement of data in the public cloud. It ensures that the sensitive data will stay in the private cloud. However, their partition and placement of code and data is deterministic once the security policies are decided. Our work studied non-deterministic situations where the task offloading is only determined at run-time based on the trace of data flow.

In [21], a HybrEx model was developed for confidentiality and privacy in cloud computing. It is an architecture that integrates the public cloud and an organization's provide cloud. It utilizes the public cloud only for safe operations on non-sensitive data and computation, and on the other hand, utilizes the private cloud for the organization's sensitive, private data and computation. In mobile cloud computing, a secure elastic framework was proposed [32] to secure code partition and offloading. The framework is designed for securing weblet-based elastic applications and consists of four components: secure installation, module authentication, secure migration, and permission authorization. However, the framework emphasizes on the management, migration, and execution of offloaded modules of an application though user authentication and key management. The main security goal is the integrity of elastic applications when they are executed at different locations. As these works in general focused on a secure architecture or management framework, they do not directly tackle the security issues in offloaded computation.

2.3 Secure Data Flow Analysis

Static analysis is widely used to find the software vulnerabilities [29] and to construct the static models of a program‘s behavior which can be enforced dynamically. For example, [1, 20] used the static models to enforce the control and

[5] used similar model to enforce the data flow integrity. There are many commercial and open source tools [3, 15, 26] available to perform static analysis and build the static model of a program. But these tools are platform specific and their output cannot be optimized easily. Also, the non-deterministic and the dynamic security analysis techniques like the dynamic data flow analysis cannot use them directly. We have developed a static analysis scheme which uses an intermediate representation during compilation of a program by using the GNU Compiler called SSA. It takes SSA as an input and builds the static data flow graph which can be processed, optimized and used by a dynamic analysis directly.

Information flow based dynamic analysis techniques can determine and enforce security policies at runtime by using the result of a static analysis. [5, 6, 22] used the dynamic data flow analysis by tagging and tracking secure information flows and using a program instrumentation technique. These works rely on static analysis of specific compilers [15] and commercial tools [3, 15, 26]. Also, they focused on the security of the general computer systems which is impractical to use in weak mobile devices like sensors due to the overhead. We have used similar technique to secure the privacy of data during code offloading. But, we have optimized the static graphs to avoid unnecessary program instrumentation and reduced the overhead significantly.

CHAPTER 3

SYSTEM OVERVIEW

3.1 System Model and Assumptions

We envision a network of embedded devices (such as a sensor network) where the devices carry the same program for accomplishing some works. However, differing from a typical homogeneous network, the devices may not be able to execute the whole program alone. Instead, they will only be able to execute a part of the program according and then find a way to collaborate. For example, the network is deployed for monitoring fire in forest as well as processing the detected fire information. All devices carry the monitoring program and the processing program. However, when a fire occurs, only the devices around the area on fire will use the monitoring program to collect the fire information, such as temperature and infrared radiation. But, because of busy monitoring, they may not be able to process the data, for example, analyzing the infrared radiation to identify the signature of burning materials and inner temperature. Hence, they will ask other nearby devices, which are not monitoring, to perform the data processing.

Accordingly, we model the device program as a collection of tasks. Each task represents a set of specific operations. All devices carry the same program, but do not have to execute all the tasks. When a device needs to offload the computation of a task, it dispatches the task to another device. But, in contrast to typical

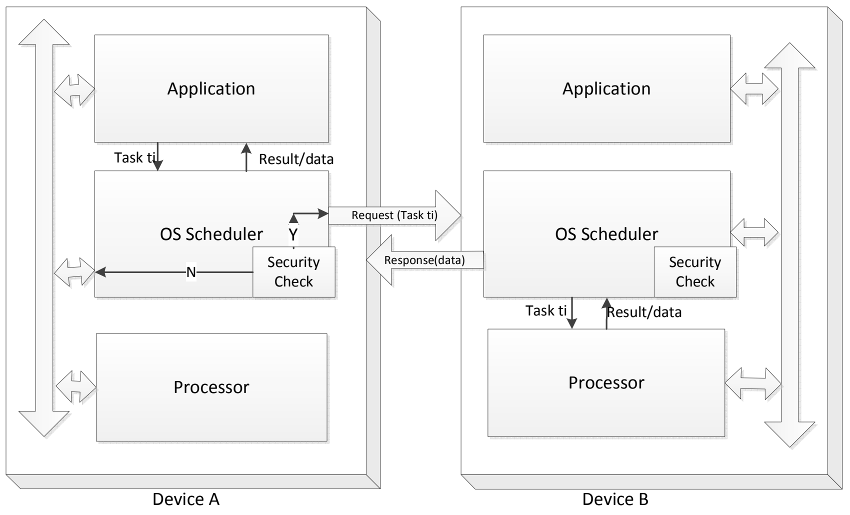

Figure 3.1: Execution flow of the code offloading between two devices

computation offloading in cloud that migrates the code of the computation to remote devices, the task offloading does not migrate the task code to the remote device, because the remote device has the code already. The originating device also dispatches data that is needed by the task. After the task is offloaded, the originating device can move on to the next task. Upon receiving the task and the data, the remote device executes the task and return the result. When the result of the offloaded task returns, the originating device can work on the result if necessary.

3.2 System Workflow

To ensure that the distributed tasks do not disclose sensitive data in the originating devices, the secure task distribution scheme is designed with two phases (illustrated in Figure 3.2) : *offline security analysis* and *run-time secure task distribution*.

In the offline security analysis phase, the tasks of device programs are analyzed to identify security-related information and then are instrumented for run-time protection. After device programs are developed, a security analysis tool analyzes

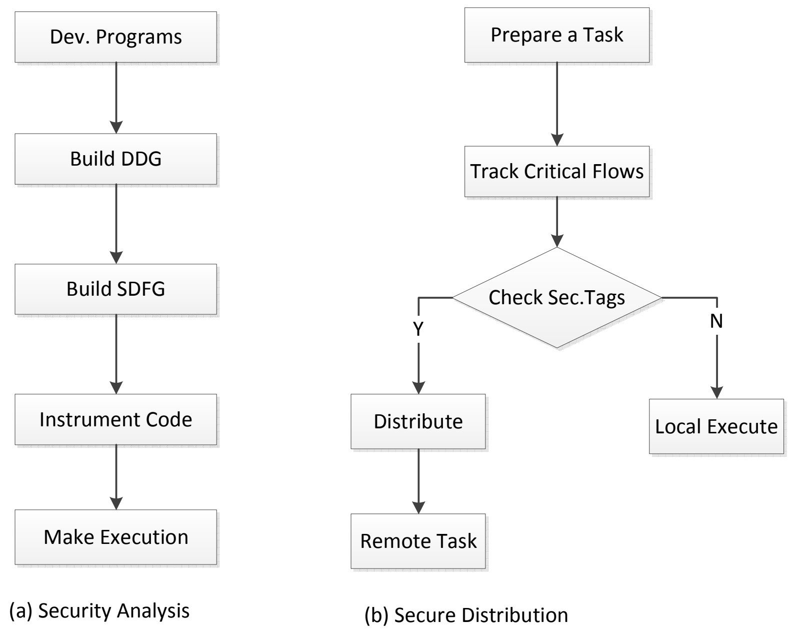

Figure 3.2: Execution Framework

the data flow of distributed tasks. It produces a data dependency graph (DDG) to find possible computations that use the security-critical data and then produces a secure data flow graph (SDFG) to find potential breach if tasks are distributed. As data dependency is not deterministic at run time, the tool instruments the device programs with a minimum amount of code that embeds security tags to track the usage of the security-critical data. The resulting device program is then rebuilt to make the final executable.

In the run-time secure task distribution phase, the security tags are tracked with the execution of preparing a task until when the task needs to be dispatched. Then, the security tags are checked against security policy. If the security check passes, the task and its associated data are dispatched to a remote device. Otherwise, the task is executed locally.

CHAPTER 4

STATIC ANALYSIS

We have developed a scheme to analyze the static behavior of a program. Our scheme takes one of the intermediate representations of the program during a compilation process called Static Single Assignment (SSA) form as an input. The input is preprocessed and transformed into a graph data structure to make it suitable for data flow analysis. The set of Data Flow Graph (DFG) is computed using a static data flow analysis algorithm which goes through each line of the program and constructs a DFG. Finally, it computes the subset of the DFG which has a path to the distribution points of the program.

4.1 Static Single Assignment Form (SSA)

The SSA [27] form is an established intermediate representation of a program during compilation. It is based on the theory that a variable is defined only once in a single program. If there are multiple assignments to a single variable, the SSA construction phase generates a new variable by renaming that variable. The SSA is widely used in modern compiler designs [2] [19] for optimization and transformation. We have used it to perform an information flow based security analysis of a program.

```
        x = 5; //Definition
        x = 10; //Redefinition
```

Figure 4.1: Example of definition and redefinition of a variable

```
        ...........
        if (x > 0)
                z = m;
        else
                z = n;
        ...........
```

Figure 4.2: Control flow merge

4.1.1 SSA terminologies and examples

1. Definition and Redefinition of a variable

   A variable is said to be defined when it is first initialized in a program and redefined when it is assigned again with a different value.

   In Figure 4.1, the variable ‘x’ is defined in one place and redefined again in another place. In the SSA form of the program, these are completely different variables and named differently as ‘X1’ and ‘X2’.

2. Control flow merge and PHI function

   Figure 4.1 is an example of a program section covering the control flow merge. The variable ‘z’ can be either ‘m’ or ‘n’ according to the value of ‘x’ at the runtime. So, there will be a flow merge at point ‘z’ from ‘m’ and ‘n’ depending upon the condition variable ‘x’. The SSA transformation introduces a special function called PHI function to handle this situation as shown in Figure 4.3.

Figure 4.4 is an example of a code segment in a non-SSA form on the left hand side and in the SSA form on the right hand side.

```
        if (X1 > 0)
        Z1 = m;
        else
        Z2 = n;

        Z = PHI(Z1,Z2);
```

Figure 4.3: PHI Node insertion

```
non-SSA form

x = i;
if (x > 0)
        y = m;
else
        y = n;
```

```
SSA form

X = I;
IF (X > 0)
        Y1 = M;
ELSE
        Y2 = N;

# Y3 = PHI {Y1,Y2}
```

Figure 4.4: A simple code section in non-SSA and SSA form

4.2 Static Data Flow Analysis

A program represented in the SSA form is a very efficient and accurate means of static code analysis. Data flow information in SSA form is explicit because each variable is assigned exactly once. Reassignment of a variable in the original program corresponds to a new variable in the SSA form. So, if we travel through the definition and usage of a variable from the start to the end of the control flow, we come up with a data flow representation which is a DFG.

4.2.1 Preprocessing SSA

To build a data flow graph for a complete program, we need to first preprocess the SSA form of the individual procedures of the program as illustrated in the Algorithm 1. During preprocessing, functions which include task distributions are extracted, processed and copied into a new file. The new file is used for the analysis as it includes sections of the program involved in the task distribution. Also, the SSA is changed into an inter-procedural representation by making the function calls inline. The dependency between the arguments and the parameters of a function call is maintained by inserting a code for assigning parameters to the arguments. Also, the link between the result variable and the return variable is introduced through an assignment. The data flow analysis using the preprocessed SSA is inter-procedural.

Algorithm 1 An algorithm to preprocess the SSA form of a program

```
Require: SSA return Preprocessed SSA $PSSA$
                                         ▷ Search and save functions with distribution
 1: for Function Fi with a distribution function do
 2:     Save $Fi$ to $FunctionWithDistList$
 3: end for
                   ▷ Preprocess and save all functions with task distribution in new file $PSSA$
 4: for $FunctionFi \in FunctionWithDistList$ do
 5:     Search function $Fi$ in SSA
 6:     Insert tag $< START\_Ti >$ in $PSSA$
 7:     seek first line of the function definition
 8:     repeat
 9:         line $Li$ from the SSA
10:         if $Li$ is simple statement then
11:             copy $Li$ to $PSSA$
12:         else if $Li$ is function call then
13:             search function definition
14:             seek to first line of function definition
15:             Insert link from function arguments to parameters
16:             read line $Li$ from the SSA
17:             if $Li$ is simple statement then
18:                 copy $Li$ to $PSSA$
19:             else if $Li$ is return statement then
20:                 Insert link from return to result           ▷ Nested function call
21:             else if $Li$ is function call then
22:                 goto 13
23:             end if
24:         end if
25:     until $Li$ is a distribution point
26:     Insert tag $< END\_Ti >$ in $PSSA$
27: end for
```

Figure 4.5 is an example which demonstrates how we preprocess and achieve

```
A1 = 1;
B1 = 3
//call with arguments and return
C1 = foo1(A1,B1);
//call that changes global variable
foo2();
C2 = C1+global_var;
//call with no effect
foo3();
distribute (C2);

Function definitions:
//function with parameter + return
foo1 (P1,P2){
Z1 = P1+P2;
return Z1;
}

//function with global variable
foo2()
{
        I1= 1;
        J 1= 2;
        K1 = I1*J1
        GLOBAL_VAR1=K1;
}
//independent call
foo3(){
P1 = 10;
P2 = 20;
P3 = P2     P1;
}
```

```
A1 = 1;
B1 = 3
//inline foo1()
//C1 = foo1(A1,B1);
//make argument link
P1 = A1;
P2 = B1;
Z1 = P1+P2;
//make return link
C1 = Z1;

//inline foo2()
I1= 1;
J 1= 2;
K1 = I1*J1
GLOBAL_VAR1=K1;

C2 = C1+ GLOBAL_VAR1;
//inline foo3
P1 = 10;
P2 = 20;
P3 = P2     P1;

distribute (C2);
```

Figure 4.5: An example of SSA preprocessing

inter-procedural representation of the SSA form of a program. It has three functions to show the different cases which are made inline during preprocessing. The first function ‘foo1’ has both arguments and return values. It is recorded by assigning the return variable to the result variable ‘C1’ and local variables ‘A1’ and ‘B1’ to ‘P1’ and ‘P2’ respectively, which are the parameters of the function. In the function ‘foo2’, we have shown an example of indirect data reference via the global variable. The function does not return anything but it updates the global variable. The function call ‘foo3’ has nothing to do with the caller. It neither brings a data out from the caller nor returns it back. So, that is a separate data flow graph and has no effect in our final data flow analysis by making it inline.

4.2.2 Data Flow Graph

Definition 4.2.1. *A Data Flow Graph (DFG) is a directed graph representing the data dependency between a number of operations and functions of a program. The complete set of DFG of a program provides all possible information flows in a system. The nodes of the DFG are the variables involved in statements and the edges are their dependencies.*

A DFG is constructed by representing the statements of a program in a graphical form and connecting them together. Figure 2 is an algorithm to construct a set of DFG from a SSA which reads each statement as an input and generates the corresponding DFG. The resulting set represents the information flows of the whole system from which all possible data dependencies can be derived.

Algorithm 2 Algorithm to get Data Flow Graph (DFG) from SSA

```
Require: Pre-processed SSA form of a program return Set of DFG
1: for statement s do
2:     Generate NodeList
3:     for Vertex V1 in NodeList do
4:         find the destination node V2 in NodeList
5:         insert edge E between V1 and V2
6:     end for
7: end for
```

The conventions used in our static analysis for the DFG construction are listed below:

1. Nodes

    (a) Variables

    (b) Constants

   (c) Operators

   The variables, constants and operators are represented as the nodes of a DFG. For example, for the program statement ‘Z = X + Y’, the nodes of the graph are ‘X’, ‘+’, ‘Y’ and ‘Z’. The operator nodes exclude the assignment operator since it is represented by an edge.

2. Edges

   The edges represent assignments, operations and conditions of the statements in a program. They connect different type of nodes and have the following possibilities of connections:

   (a) Variable to Variable (Eg: x = a, edge from a to x)

   (b) Constant to Variable (Eg: x = 5, edge from 5 to x)

   (c) Variable to Operator (Eg: z = x + y, edge from y to +, x to +)

   (d) Operator to Variable (Eg: edge from + to z)

We have defined two special types of nodes and edges for a DFG which are listed below:

1. Merge Edge

   A Merge Edge is defined as an edge in a data flow graph which represents a simple operation from the source nodes to the destination node.

   Example: ‘Z = X + Y;’

In this statement, ‘X’ and ‘Y’ go through an addition operator to merge into the target node Z. In the corresponding DFG, ‘X’ and ‘Y’ point to ‘+’ operator and ‘+’ finally points to ‘Z’ via a Merge Edge. An example of a Merge Edge is shown graphically in Figure 4.9

2. Selection Edge and Diamond Node

   A Selection Edge is a special type of edge which connects nodes of conditional statements and loops. We use a special type of node, which has diamond shape called Diamond Node, for these statements.

   Example: ‘I = (J <99)? X: Y;’

   Here, the conditional operator is represented by a Diamond Node. The condition variables ‘J’ and ‘99’ are incoming nodes to the Diamond Node and the candidate nodes ‘X’ and ‘Y’ are the nodes connected to it via the Selection Edges. After performing selection operation, the Diamond Node points to the result node ‘I’. An example of a Diamond Node with the Selection Edges is shown graphically in Figure 4.11.

4.2.3 Methods of Constructing Data Flow Graph

The methods of constructing a DFG from the statements of a program are listed below:

1. Simple Assignment

   The DFG for a simple assignment presented in Figure 4.6 is shown in Figure 4.7.

```
a = b;
NodeList
a,b
Edges
b.next = a
```

Figure 4.6: Data structure of the DFG for a simple assignment

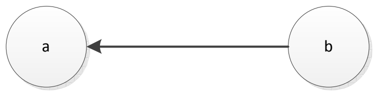

Figure 4.7: DFG for a simple assignment

2. Simple Operation

```
x = y + z;
NodeList
x, y, +, z
Edges
z.next = +
y.next = +
+.next = x
```

Figure 4.8: Data structure of the DFG for a simple operation

The DFG for a simple operation presented in Figure 4.8 is shown in Figure 4.9.

3. Decision Structure

The DFG for a decision operation presented in Figure 4.10 is shown in Figure 4.11.

4. Loop Structure

The corresponding data flow graph for the Figure 4.12 is shown in Figure 4.13. It shows that the loop variables preserve the dependency with the variables inside the loop.

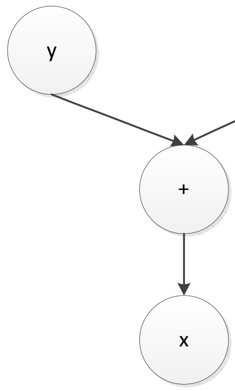

Figure 4.9: DFG for a simple operation

```
If (X1 == 1)
        Y1 = 10;
Else
        Y2 = 20;
Y3 = PHI (Y1 | Y2)

NodeList
X1, Y1, Y2, Y3, PHI, 1, 10, 20

Edges
X1.next = PHI_Y
1.next = PHI_Y
10.next = Y1
20.next = Y2
Y1.next = PHI_Y
Y2.next = PHI_Y
PHI_Y.next = Y3
```

Figure 4.10: Data structure for the DFG of a decision structure

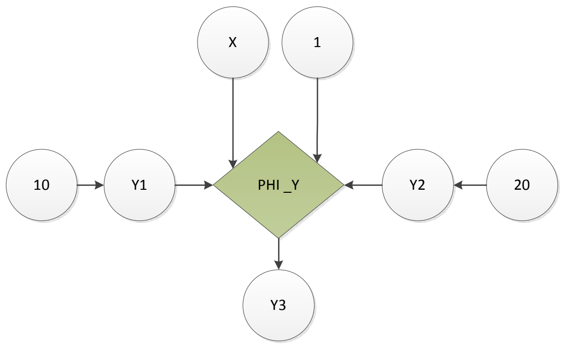

Figure 4.11: DFG for a simple decision operation

```
Code:
for (i = j; i < k; i++){
        l = m-n;
        x = 2;}

NodeList
i, j, k, l, m, n, +, -, 1, 2, x
Edges
j.next = i
k.next = i
1.next = +
i.next = +
+.next = i
m.next = -
n.next = -
-.next = l
i.next = l
2.next = x
i.next = x
```

Figure 4.12: Data structure of the DFG for a simple loop

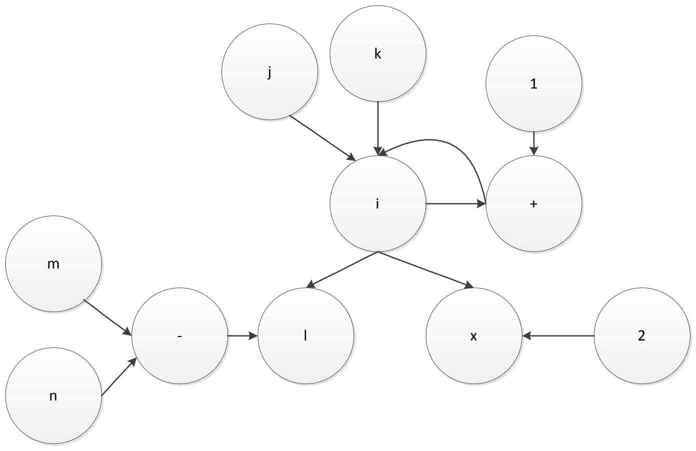

Figure 4.13: DFG for a simple loop

4.2.4 Data Dependency Graph

Definition 4.2.2. *A Data Dependency Graph (DDG) to a particular point of a program is a subset of the DFG which has a path to that point.*

After constructing the set of DFG from the pre-processed SSA form of a program, finding out the data flow to a particular data point is next step in static analysis. We have to travel backward from that point to all possible data flow paths in the set of DFG. The DDG construction process follows a graph traversal algorithm which starts from a distribution point and travels back to the ending nodes in a set of DFG. The intermediate nodes and edges between the starting and the ending nodes propagate dependency from input variables and constants to the distribution point of the program.

Figure 3 is an algorithm to construct a DDG from the set of DFG of a program. It starts from the distribution point and travels back recursively to find the nodes and edges connected to it.

Algorithm 3 Algorithm to get DDG from the set of DFG

```
Require: StartNode, Set of DFG return DDG
                                                  ▷ Push StartNode S to STACK
 1: Push S to the STACK
 2: Save node S to DDG
 3: repeat                                                    ▷ Get first node
 4:     v = POP STACK
 5:     if v is unmarked then
 6:         mark v
 7:     end if                                                   ▷ Keep node v
 8:     Save node v to DDG
 9:     for edge(w,v) connected to v do
10:         Push w to the STACK
11:     end for
12: until STACK is empty
```

Proposition 4.2.1. *A DDG constructed from a set of DFG by the Algorithm 3 preserves all secure information.*

Proof by contradiction. There are two types of DFG in a complete set of DFG of a program.

1. Set of DFG which has path to the distribution points

2. Set of DFG which has no path to the distribution points

Algorithm 3 filters out the DFG set 1 which has a path to the distribution point of the program to construct a DDG. Therefore, every node of a DDG has a path to the distribution point and can be disclosed by the Task Distributor.

Let us assume that there is a secure node ‘S’ in a DFG which is not included in a DDG but can be disclosed by the Task Distributor. This means that ‘S’ has a path to the distribution point of the program. But, it contradicts with the Algorithm 3 since it includes every node that has path to the distribution point and ‘S’ should have been included in a DDG. Hence, A DDG constructed from a set of DFG by the Algorithm 3 preserves all secure information.

$\square$

CHAPTER 5

SECURE DYNAMIC ANALYSIS

5.1 Overview of Dynamic Analysis

The static analysis takes the SSA representation of a program as an input, computes a set of DFG and gives a set of DDG as the final output. The DDG set contains information about the dependency between the variables of the program. A particular dependency chain in a DDG is called a Data Flow Path, which represents a path that a variable can take during the program execution.

For the security of a system, a task that tries to send out secure data should not be executed. In other words, the Data Flow Path ending at a distribution point should not contain any secure variables. For the sequential structures of a program, the static data flow information is deterministic and it can determine the exact Data Flow Path of the variables. In contrast, the Data Flow Path of complex structures like decisions and branches is non-deterministic because it can change according to the runtime situation. So, the static analysis alone cannot determine the selection of the Data Flow Path after a decision or branching point. In order to find it accurately, a dynamic data flow analysis is required.

Let us consider a simple example as shown in Figure 5.1. Here, variables ‘m’ and ‘n’ follow a simple sequential structure where the data flow can be easily traced by the static analysis. But, the variable ‘y’ is conditional upon variable ‘x’ and it might

```
a = 5; b = 10;
m = a+b;
n = 5;
.....
if (x > 0)
        y = m;
else
        y = n;
```

Figure 5.1: A simple example to illustrate need of dynamic analysis

take either ‘m’ or ‘n’ during runtime. Since the static analysis cannot answer on which variable the result ‘y’ will be dependent during execution, dynamic data flow analysis is needed.

Our static analysis is based on the SSA form of the program which is generated during the optimization pass of a program compilation. The compiler introduces a PHI node as shown in Figure 4.4 to handle a control flow merge. The DFG of the example in Figure 5.1 follows a similar structure with multiple data flows merging into the PHI node. The result of the PHI operation is one of the flows that the system selects during execution. Since the static analysis cannot determine the result of the operation, we need dynamic data flow analysis.

The dynamic data flow analysis handles the complex situations like decisions and branches in a program by using program instrumentation technique, which keeps record of the data flows during runtime. The tags associated with variables are initialized and updated according to their usage and propagated with the Data Flow Path. This technique can determine exactly which Data Flow Path the variable has followed during execution.

In this work, we are interested to find out whether the current execution flow of the program passed through the data flow path involving secure data or not, rather than tracking particular data flow path. In other words, we are concerned with the data dependency of a particular point of the program with secure variables. The Secure Dynamic Analysis (SDA) tracks the data dependency with secure data nodes by doing a program instrumentation. It takes the DDG and the SSA form of a program as input. and trims down the DDG by removing non secure flows. Then, it finds out tag initialization and update points in the SSA form of the program. Finally, it initializes, propagates and updates tag values to the data flow paths involving secure variables. The tag value can be checked at a particular point to determine the data dependency on secure variables. The program can be compiled after changing it back to the normal form from the SSA form.

5.2 Secure Data Flow Graph

Each data flow path of a DDG ending at a node represents the possible data flow of the system to that node. For the large programs, the length of the data flow paths could be very long and they might cross with each other as well. This makes security analysis harder and inefficient. We can reduce the complexity of the graph by removing the data flow paths which do not contribute to the security information.

Definition 5.2.1. *A Safe Data Flow (SaDF) path is defined as an incoming data flow path to a particular node in a DDG which does not contain any secure nodes.*

Definition 5.2.2. *A Secure Data Flow (SeDF) path is defined as a data flow path that includes secure nodes on it and ends at the distribution point of the program.*

Definition 5.2.3. *A Secure Data Flow Graph(SDFG) is a subgraph of DDG on which any SaDF is a sub path of SeDF.*

Definition 5.2.4. *A Complete Secure Data Flow Graph is a maximum SDFG that includes all secured nodes of a DDG. We use SDFG notation to refer a Complete SDFG throughout our work.*

A SDFG can be computed from a DDG by keeping SeDF paths and trimming out SaDF paths which are not the sub paths of a SeDF. The Algorithm 4 presents a method to find out SeDF and SaDF in a DDG. It prepares a list of secure nodes ‘SecureNodeList’ and sets a flag ‘HasSecureChild’ on the nodes having direct and indirect dependency with secure variables. It takes the distribution point of the DDG as the ‘StartNode’ and travels back until it visits complete nodes.

The output of the Algorithm 4 is used as an input by the Algorithm 5 to construct the SDFG. The algorithm trims out the set of SaDF from a DDG and constructs the SDFG. It constructs it by copying the set of SeDF to the final graph but excluding the set of SaDF, which are not part of the SeDF. The set of edges that does not contain a secure node or a node with ‘HasSecureChild’ flag set as true, is a SaDF which is not a part of SeDF.

Proposition 5.2.1. *A SDFG preserves all security information present in a DDG.*

Proof. The set of edges ‘SE’, which contain secure nodes and marked nodes having a downstream link to the distribution point, is a SeDF. Similarly, the set of edges, ‘NE’, which do not have secure nodes and marked nodes is an independent SaDF which is not a part of the SeDF. The set ‘NE’ neither contains secure nodes nor the

Algorithm 4 Find SeDF and secure nodes in a DDG

```
Require: DDG, StartNode return SecureNodeList,DDG with HasSecureChild flags set
    on Nodes
                              ▷ Get all connected nodes to StartNode and push in a STACK
 1: for all connected nodes v to StartNode do
 2:     Push v to the STACK
 3: end for
                                                                 ▷ Recursively find SeDF
 4: repeat                                                               ▷ Get first node
 5:     v = POP STACK
 6:     if v is unmarked then
 7:         mark v
 8:     end if
 9:     if v is secure node then
10:         mark and save v to SecureNodeList      ▷ Mark all parents of v has secure
    child
11:         repeat
12:             get parent p of v
13:             if p has secure child flag is false then
14:                 mark p has secure child true
15:             end if
16:             make p as node v
17:         until p has secure child flag true OR p is StartNode
18:     end if
19:     for all connected nodes v do
20:         Push v to the STACK
21:     end for
22: until STACK is empty
```

Algorithm 5 Get SDFG from a DDG by trimming out all independent SaDF

```
Require: DDG, StartNode, SecureNodeList return SDFG
                          ▷ copy DDG to SDFG using SecureNodeList and parent list
 1: for all connected nodes v to StartNode do
 2:     Push v to the STACK
 3: end for
 4: repeat                                                      ▷ Get first node
 5:     v = POP STACK
 6:     if v is unmarked then
 7:         mark v
 8:     end if
 9:     if v in SecureNodeList OR v has secure child flag is true then   ▷ Keep node v
10:         save node v
11:         for all connected nodes p do
12:             Push p to the STACK
13:         end for
14:     else                         ▷ This is a node of SaDF that can be trimmed
15:         break
16:     end if
17: until STACK is empty
```

nodes that are dependent upon secure nodes. From the definition of the security, 'NE' does not need to be secured. Hence, the construction of the SDFG by removing 'NE' preserves security information.

□

5.3 Code Instrumentation

Code instrumentation is a standard technique used in dynamic data flow analysis which instruments a program to track the dynamic behavior during execution. It records the definition and usage of a variable in the data flow path by assigning tag values. The final tag value calculated can provide the information of the data nodes which have been used by the current execution flow in order to reach a particular point in the program.

In general tag based dynamic information flow tracking systems[22], all variables have their own associated tags. A tag can be a metadata that annotates a variable or a pointer to another data structure which annotates the variable. Whenever there is an operation, the tag associated with resulting variable is updated according to the tag associated with the operand variables. The numeric constants do not have tags. At any particular execution point, the tag associated with a variable can be checked to find the data dependency.

We have used similar techniques to implement the dynamic data flow analysis for the security analysis of the Task Distributor. In contrast with general techniques, our implementation utilizes the SDFG to avoid unnecessary codes for tag update and propagation. The SDFG covers all of the SeDF paths and helps to skip the SaDF paths, which are not required for security analysis. It reduces the analysis cost and achieves better performance without compromising the security.

5.3.1 Tag Assignment and Propagation

Definition 5.3.1. *A Tag Score is defined as a numeric score associated with a node of a SDFG.*

Definition 5.3.2. *A Tag Value is defined as the total tag score of the nodes in a data flow path of a SDFG.*

The SDA assigns a numeric Tag Score to the nodes of a SDFG. It gives a constant positive numeric tag score to a secure variable and a tag score of zero to a normal variable. It also gives initial Tag Value of zero to all edges. A variable which has a positive tag score signifies that it is a secure variable or a global variable derived from a secure variable.

The General Information Flow Tracking framework [22] and similar traditional tag based analysis techniques update and propagate the tag values using the tags of operands to the result of an operation. This is equivalent to the nodes and the edges of a DFG. If an edge visits a secure variable node, it gets the tag value according to the tag score associated with that variable. Whenever an edge branches into multiple edges or number of edges merge into a single edge, the new edge gets the tag value from the input edges. At the distribution point of the program, the tag value of the incoming edge can be checked to decide whether current execution flow has a data dependency on secure variables or not. If not, it should have a tag value of zero.

5.3.2 Tag Propagation without Any Reduction

The initial implementation of our tag initialization and propagation technique is performed by assigning tag scores to all variables in a SDFG and propagating tag values via the edges. Figure 5.2 is a simple example of the tag assignment and the propagation. It associates and initializes the tag scores to each of the variables in the sample code. These tag scores are assigned, updated and propagated via the edges of the graph to the end of the program as shown.

In this example, secure variables are represented with a name starting with ‘SEC’. Initially, the tag score of ‘1’ is given to all secure variables and ‘0’ is given to all non secure variables. The tag values of the edges are updated according to the variables they have used as shown in Figure 5.2. Let us say that we are interested to see whether the variable 'FINAL' has data dependency on a secure variable in the program or not. When the instrumented program is executed, the tag values of the data flow paths are updated according to the runtime condition which propagates the dependency information to the end point. We can check the tag value associated with the edge 'TAG_EDGEFINAL' associated with variable 'FINAL'. If it is greater than zero, we can conclude that it has data dependency on secure variables in the program.

5.4 Minimum Set of Tags

Assigning Tag Scores to all nodes of a SDFG and propagating the data flow information via the edges is inefficient. We introduce the concept of Trails as defined in the Definition 5.4.4 to optimize the program instrumentation by avoiding unnecessary initialization and updates. The SDFG can be changed into a Trail Graph as defined in the Definition 5.4.5 which reduces nodes of a SDFG into the Trail Nodes and the edges to the Trails. The Trail Graph optimizes and preserves the complete information of a SDFG required for dynamic security analysis.

Definition 5.4.1. *A node in a SDFG that branches out to multiple edges is defined as a Branch Node.*

Definition 5.4.2. *A node in a SDFG that has multiple incoming edges is defined as a Merge Node.*

```
int SEC_J,SEC_A,SEC_B;

int I, K, X, Z, N, N, N1, N2;

//Assignment
I =  SEC_J;
K = I;

//Branch
X = K;
Z = K;

//Condition (Merge)
IF (M > 0)
        N1 = SEC_A + SEC_B;
ELSE
        N2 = Y;

N = PHI (N1,N2)

//OPERATION
L = X + N;

FINAL = L;
```

```
int SEC_J,SEC_A,SEC_B;
//Assign constat score5
int TAG_SEC_J, TAG_SEC_A, TAG_SEC_B =  5;

int I, K, X, Z, N, N, N1, N2;
int TAG_I, TAG_K, TAG_X,TAG_Y, TAG_Z = 0;
int TAG_N TAG_N1, TAG_N2 = 0;

//Assignment
I =  SEC_J;
TAG_EDGEI = TAG_SEC_J;
K = I;
TAG_EDGEK = TAG_EDGEI + TAG_I;

//Branch
X = K;
TAG_EDGEX = TAG_EDGEI + TAG_K;
Z = K;
TAG_EDGEZ = TAG_EDGEK + TAG_K;

//Condition
IF (M > 0)
        //New EDGE
        N1 = SEC_A + SEC_B;
        TAG_EDGEN1 = TAG_SEC_A + TAG_SEC_B
        + TAG_EDGEM;
ELSE
        //New EDGE
        N2 = Y;
        TAG_EDGEN2 = TAG_Y + TAG_EDGEM;

N = PHI (N1,N2);
TAG_EDGEN = TAG_EDGEN1 biwiseOR TAG_EDGEN2;

//OPERATION
L = X + N;
TAG_EDGEL = TAG_EDGEX + TAG_EDGEN;

FINAL = L;
TAG_EDGEFINAL = TAG_EDGEL + TAG_L;
```

Figure 5.2: An example of tag assignment and propagation

Definition 5.4.3. *The Trail Nodes is defined as the set of Secure Nodes, Merge Nodes and Branch Nodes in a SDFG.*

Definition 5.4.4. *A Trail is defined as a flow between a pair of Trail Nodes which does not contain other Trail Nodes in between.*

Definition 5.4.5. *A Trail Graph is defined as a graph in which each edge is a Trail.*

The minimum set of tags are listed in List 5.4.

1. Minimum Set of Tag Score

   Secure Nodes and Global Variables contain secure information which should be tagged.

2. Minimum Set of Tag Value

   Trails propagate Tag Scores and they have Tag Values associated with them.

The same example in Figure 5.2 can be changed to the example in Figure 5.3 after removing unnecessary tag updates from the code.

Proposition 5.4.1. *The program instrumentation with minimum tag set achieves the same security as the tag propagation without any reduction.*

Proof. Let us consider the set of secure nodes 'S' and the set of non-secure nodes 'N' in a Trail. Tag propagation without any reduction assigns tag scores to both 'N' and 'S'. The total tag value of the Trail is the total tag score of 'N' and 'S' in the Trail. But, the concept of minimum tag sets excludes the set 'N'. In both cases, the total tag value of the Trail remains same since 'N' does not include any security

```
int SEC_J,SEC_A,SEC_B;

int I, K, X, Z, N, N, N1, N2;

//Trail
I =  SEC_J;
K = I;

//Banch Point from K
X = K;
Z = K;

//CONDITION
IF (M > 0)
        N1 = SEC_A + SEC_B;
ELSE
        N2 = Y;

N = PHI (N1,N2)

//OPERATION
L = X + N;

FINAL = L;
```

```
int SEC_J,SEC_A,SEC_B;
//Assign constat score5
int TAG_SEC_J, TAG_SEC_A, TAG_SEC_B =  1;

int I, K, X, Z, N, N, N1, N2;

//Trail T1
I =  SEC_J;
TAG_T1 = TAG_SEC_J;
// No tag udpate
K = I;

//Branch Point
X = K;
TAG_TRAIL2 = TAG_TRAIL1;
Z = K;
TAG_TRAIL3 = TAG_TRAIL1;

//CONDITION
IF (M > 0)
        //New TRAIL
        N1 = SEC_A + SEC_B;
        TAG_TRAIL4 =
        bitwiseOR(TAG_SEC_A, TAG_SEC_B)
ELSE
        //New TRAIL
        N2 = Y;
        TAG_TRAIL5 = TAG_Y;

N = PHI (N1,N2);
TAG_TRAIL6 =
getSelectedTrail (TAG_TRAIL4,TAG_TRAIL5);

//OPERATION
L = X + N;
TAG_TRAIL7 =
bitwiseOR(TAG_TRAIL2,TAG_TRAIL6);

//L is in TRAIL7
//No update
FINAL = L;
```

Figure 5.3: An example of tag assignment and propagation with minimum tag sets

information. Hence, tags with minimum tag set achieves the same security level as that without any reduction. □

5.4.1 Finding Tag Update Points

Before inserting the instrumentation code into the actual source code for the dynamic analysis, we need to find out the locations where tag assignment and updates are needed. In our dynamic analysis, this step can be performed by using the Trail Graph. We have to travel throughout the SDFG to find the Trail Graph. The Trail Nodes are the points which require tag update. The locations in the source code corresponding to the Trail Nodes are the points where instrumentation is required.

The Algorithm 6 locates the tag update points in a program. It takes the SDFG and the stack of all start nodes of it as an input and returns the number of lists containing information about the Trail Graph and Trail Nodes where the tag update is needed. The output includes the following list:

1. Trail Definitions.
2. Name and association of Trail Nodes in a Trail.
3. Trails merging through the Merge Nodes
4. Trails coming out of the Merge Nodes.
5. Trails branching out from the Branch Nodes.

Algorithm 6 Find tag update points and Trails in SDFG

```
Require: $SDFG$,$StartNodeStack$
    return $SecureTrailList$,$SecureNodeToTrailList$,$TrailMergeToPHINodeList$,$PhiNodeToTrail$,
    $TrailMergeToOperationList$,$OperatorToTrail$,$TrailToBranchNode$,$BranchNodeToTrailList$
 1: POP first node $Si$ from $StartNodeStack$
 2: get all children of $Si$ as $ChildList$
 3: for $Nodei \in ChildList$ do                               ▷ Define a new Trail for each branch
 4:     define $Trail_i$
 5:     save $Trail_i$ to $SecureTrailList$
 6:     if $Si$ is secure node then
 7:         associate secure node $Si$ to the Trail $Trail_i$ and save to $SecureNodeToTrailList$
 8:     else if $Si$ is Operator node then
 9:         associate Operator Node $Si$ to the Trail $Trail_i$ and save to $OperatorToTrail$
10:     else if $Si$ is PHI node then
11:         associate PHI Node $Si$ to the Trail $Trail_i$ and save to $PhiNodeToTrail$
12:     else if $Si$ is present in $TrailToBranchNode$ then
13:         associate branching Node $Si$ to the Trail $Trail_i$ and save to
    $BranchNodeToTrailList$
14:     end if                                            ▷ Conditions for the new Trail generation
15:     repeat
16:         if $Nodei$ is PHI node then
17:             associate Trail $Trail_i$ to the PHI node $Si$ and save to
    $TrailMergeToPHINodeList$
18:             PUSH $Nodei$ to $StartNodeStack$
19:         else if thenNodei is an Operator node
20:             associate Trail $Trail_i$ to the Operator node $Si$ and save to
    $TrailMergeToOpNodeList$
21:             PUSH $Nodei$ to $StartNodeStack$
22:         else if thenNodei is branching node
23:             associate branching node $Nodei$ to Trail $Trail_i$ and save $TrailToBranchNode$
24:             PUSH $Nodei$ to $StartNodeStack$
                                                          ▷ Where no Trail generation is required
25:         else
26:             assign $Nodei$ to $Si$
27:             if $Si$ is secure node then
28:                 associate Trail $Trail_i$ to the secure node $Nodei$ and save to
    $SecureNodeToTrailList$
29:                 get child of $Si$ and assign to $Nodei$ to continue on loop
30:             end if
31:         end if
32:     until $Nodei$ is a PHI node or an Operator Node or a node with multiple child
33:     POP first node $Si$ from $StartNodeStack$
34:     get all children of $Si$ as $ChildList$
35: end for
```

5.4.2 Inserting Tag Initialization and Propagation Code

The output list of the Algorithm 6 is used to instrument the program. The code for Tag Score initialization for secure variables and Tag Value initialization for secure Trails is inserted first. A constant numeric value is given as the Tag Score to all secure variables and Tag Values of all Trails is initialized to zero. The Trails update their Tag Values when they meet the Trail Nodes. The tag reset code for each task is inserted in the Task Scheduler after the completion of each task.

The Algorithm 7 presents a process of inserting the tag update and propagation code in the SSA form of the program. It takes the SSA form and the output of the Algorithm 6 as the input. It goes through all update points and inserts the tag update and propagation code.

Proposition 5.4.2. *Attacker cannot manipulate the tag value easily.*

Proof. The tag propagation is a bitwise OR between the tag values of the Trails. To get a zero result, the attacker should be able to change the tag values to zero for all Trails. So, it is not easy to manipulate the tag value of the Trails to break the security. □

Proposition 5.4.3. *Tag instrumentation does not introduce additional vulnerability.*

Proof. The DFG of the original program and the instrumented program are separate and there is no link between them because the instrumentation creates new variables for the tag assignment and propagation. Also, the control flow of the instrumentation is in sequence with the control flow of the original program without

Algorithm 7 Insert tag assignment and propagation code

```
Require: $SSA$,$SecureTrailList$,$SecureNodeToTrailList$,$TrailMergeToPHINodeList$,$PhiNodeToTrail$,
    $TrailMergeToOperationList$,$OperatorToTrail$,$TrailToBranchNode$,$BranchNodeToTrailList$
        return $SecureSSA$
                                    ▷ Initialize numeric tag score to all Secure Nodes
 1: for $SecureNode_i \in SecureNodeToTrailList$ do
 2:     Insert Code $TagSecureNode_i = SomeNumericTagScore;$
 3: end for
                                    ▷ Initialize Trails to all tasks of the program
 4: for $Task_i \in ProgramTaskList$ do
 5:     Insert Code $Task_iTrailList = findAndSaveTaskTrails$
 6: end for
                                    ▷ Initialize 0 Tag Value to all Trails
 7: for $Trail_i \in SecureTrailList$ do
 8:     Insert Code $TagTrail_i = 0$;
 9: end for
                                    ▷ Insert tag update code for the Secure nodes
10: for $SecureNode_i, Trail_i \in SecureNodeTrailList$ do
11:     search $SecureNode_i$ in $SSA$
12:     Insert Code $TagTrail_i = bitwiseOR(TagTrail_i, TagSecureNode_i)$;
13: end for
                                    ▷ Insert tag update code for the Operator nodes
14: for $Operator, Trail_i \in OperatorToTrail$ do
15:     find $Operator$ in SSA
16:     insert Code $TagTrail_i = bitwiseOR$ ($getAllTrails$ associated with the Operator in
    $TrailMergeToOperatorList$)
17: end for
                                    ▷ Insert tag update code for the PHI nodes
18: for $PHINode, Trail_i \in PhiNodeToTrail$ do
19:     find $PHINode$ in $SSA$
20:     insert Code $TagTrail_i = getTagvalueOfTheTrail$(select a Trail $Trail_i$ from
    $TrailMergeToPhiNodeList$ where $Trail_i$ is in $CurrentTrailList$)
21: end for
                                    ▷ Insert tag update code for the Branching nodes
22: for $BranchingNode, TrailToBi \in TrailToBranchNode$ do
23:     for $BranchingNode$, $Trail_i$ in $BranchNodeToTrailList$ do
24:         find $BranchingNode$ in SSA
25:         insert Code $TagTrail_i = TrailToBi$
26:     end for
27: end for
                                    ▷ Insert reset code in Task Scheduler after the completion each task
28: for $Trail_i \in Task_iTrailList$ do
29:     Insert Code $TagTrail_i = 0$;
30: end for
```

introducing additional branches. Hence, the instrumentation does not introduce any additional vulnerability to the original program. □

5.5 Discussion

5.5.1 Handling Functions

A function call in a program takes out the data dependency from the caller and carries it in on the function return. Since we have made functions inline during the preprocessing step of the SSA form before building data flow graphs, we have to pay special attention during program instrumentation phase to handle them accurately.

The preprocessing step of the static analysis section keeps a record of function call points, return points and the variables used for arguments which is useful during dynamic analysis. A function which has a record corresponding to a node in the SDFG should be instrumented like the example presented in the Figure 5.4. It uses an array to pass the tag values from the caller and a variable to return the tag value from the function to the resulting variable. The input tag values are calculated according to the Trail Tag Values in which the arguments are held. The return tag value is equal to the tag value of the Trail in which the return variable is held. Everything else inside a function follows the same process mentioned in previous sections.

The tag values associated with these variables are reset after the return statement to revert them to their initial state. By doing so, there will be no side effect introduced when the function is called from the program sections where the

```
Function Call:

X = foo (Y,Z);
.................

Function Definition:

foo (I,J){
............
............
return K;
}
```

```
After Instrumentation:

//Initialize
define TAG_ARRAY[n];
RETURN_TAG = 0;

Function Call:
//Pass tag values
TAG_ARRAY[0] = Trail_Y.tag_value;
TAG_ARRAY[1] = Trail_Z.tag_value;

X = foo (Y,Z);

// Read return tag
X.tag_value = RETURN_TAG;

//Reset
reset TAG_ARRAY[n];
RETURN_TAG = 0;

Function Definition:
foo (I,J){

// Get caller's tag values
I.tag_score = TAG_ARRAY[0];
J.tag_score = TAG_ARRAY[1];
.............
.............
//Tag Out
RETURN_TAG = TRAIL_K.tag_score;
.............
}
```

Figure 5.4: An example of handling functions during code instrumentation

DFG does not have pass to the distribution point. In that situation, variables that pass tag values are in their initial state.

5.5.2 Compiling Secure Code

The SSA form of a program cannot be compiled directly. It is used frequently on top of other intermediate representations having direct correspondence to it. Although, there are certain techniques available to interpret the SSA form [30], these techniques are not perfect and have complexity to implement. Our SDA produces a secure code in the SSA form, which is hard to compile and execute.

We have transformed the SSA form back to the source code to make it directly compilable. Due to the lack of a perfect tool, which transforms the SSA form back into its original program code, we have done the transformation manually. Our conversion process takes the original source code, the SSA form of the program and the secure SSA form of the program as the input. It finds the difference between the instrumented and the original SSA form, converts it into the equivalent source code representation using standard conversion techniques[4] and instruments the source code accordingly. The output is the secure code which can be compiled, executed and tested. Algorithm 8 describes the general steps used to get the secure source code from the secure SSA form.

Algorithm 8 Converting from secure SSA form to source code

```
Require: $SSA$,$SecureSSA$,$SourceCode$ return $SecureSourceCode$
1: Find $differencesections$ between $SSA$ and $SecureSSA$
2: for $difference \in differencesections$ do
3:     Trace locations of the $difference$ in $SSA$ and $SecureSSA$
4:     Find corresponding locations in $SourceCode$
5:     Compare $SourceCode$, $SSA$ and $SecureSSA$
6:     Convert $difference$ into equivalent source code $differenceSourceCode$
7:     Get $SecureSourceCode$ by inserting $differenceSourceCode$ into $SourceCode$)
8: end for
```

CHAPTER 6

IMPLEMENTATION

In this section, the implementation of a task distributor as well as security analysis is presented. We have implemented our work in TinyOS operating system using MICAz motes and MIB520 programming boards. We have modified the TinyOS scheduler for task distribution to neighboring devices and ensured the security of the system by using static and dynamic data flow analysis techniques.

6.1 Task Distributor

6.1.1 Overview of TinyOS

TinyOS[] is a free and open source operating system for embedded devices such as wireless sensors. It is an event-driven operating system which allows a device to sleep and keeps it in low power state most of the time. Whenever there is an event, the device wakes up to perform the task corresponding to that event. Each event posts a task to the operating system. Tiny-OS has a task scheduler to handle these tasks which is a general FIFO queue. A task is not executed immediately after it is posted but will be eventually executed in the order it was posted. Tasks cannot preempt each other. Only interrupts can preempt tasks and interrupts. After completing all tasks, the device goes back to the sleep state which is the low power state of the system.

TinyOS programs are written in a C dialect, called nesC, and translated into C when compiled. A typical nesC application is built out of components which are assembled together to form a whole program. Components can interact with each other via interfaces and each component either provides or uses interface. They provide interface for their users and use interfaces provided by other components to perform their operations. An interface specifies a set of functions to be implemented by its providers and a set of functions to be implemented by its users. This allows a single interface to represent complex interactions between the components of an application. A component can call other components by using commands, which respond back to the caller by signaling events after the completion of operations.

6.1.2 Tasks and Scheduler in TinyOS

In TinyOS, a task specification is actually a function, which has the ‘task’ keyword but doesnt have arguments and return value. The main difference between functions and tasks is that tasks cannot be called directly from the code. They have to be posted to the scheduler using the ‘post’ keyword and executed later following the task execution policy. When a task is defined and posted, it is not executed immediately. Rather, the task is added to the task queue and the scheduler executes it sometime later.

Figure 6.1 shows the procedure of defining and calling tasks in TinyOS. The nesC compiler automatically understands these keywords and transforms them to the appropriate code in compile time.

Tasks are posted via the TaskBasic interface provided by the scheduler of

```
//Define Task
task Foo()
{
//Some Logic
}
................
//Post Task
post Foo();
```

Figure 6.1: An example of task definition and post in TinyOS

TinyOS whenever the ‘postTask’ command of TaskBasic interface is used. Once this method is called, it adds the task to the task queue. The scheduler checks the task queue during scheduling and signals the ‘runTask’ event of the TaskBasic interface which executes the code of the task.

Figure 6.2 presents the signature of the TaskBasic interface.

```
interface TaskBasic<precision_tag>{

//Post tasks
  async command error_t postTask();

 //Run tasks
  void  event runTask();

}
```

Figure 6.2: The TaskBasic Interface

Figure 6.3 is the code transformed automatically by the nesC compiler for the task keyword and the post task method used in TinyOS applications.

```
module a {
  ...
  uses interface TaskBasic as foo;
}
implementation {
//Define Task
  event void foo.runTask() {
    ...}
//Post Task
    call foo.postTask();
  }
```

Figure 6.3: Equivalent task code transformed by nesC compiler

The scheduler is the main component responsible for handling tasks and events in TinyOS. The default task execution policy is FIFO: tasks are executed in the order they are posted. TinyOS goes in low power mode when there is no job to handle. Whenever there is an event such as an I/O interrupts, it posts a task to the task queue. The scheduler runs in an infinite loop to check the task queue. If there is a task in the queue, the scheduler signals the runTask event to execute the task. Once all tasks are executed, the scheduler goes back to the low power sleep mode.

The default scheduler is the TinySchedulerC component in TinyOS and its default implementation is a module named as SchedulerBasicP. The default scheduler component is a configuration which wires the SchedulerBasicP module with other components. Figure 6.4 is the code snippet of the TinySchedulerC component which provides Scheduler and TaskBasic interfaces. Figure 6.5 is the code snippet of the SchedulerBasicP module.

```
configuration TinySchedulerC {
        provides interface Scheduler;
        provides interface TaskBasic[uint8_t id];
        }
        implementation {
                components SchedulerBasicP as Sched;
                components McuSleepC as Sleep;
                Scheduler = Sched;
                TaskBasic = Sched;
                Sched.McuSleep -> Sleep;
        }
```

Figure 6.4: The TinySchedulerC Component

```
module SchedulerBasicP {
        provides interface Scheduler;
        provides interface TaskBasic[uint8_t taskID];
        uses interface McuSleep;
}
```

Figure 6.5: The SchedulerBasicP Component

The parameterized interface TaskBasic provided by the scheduler uses the unique() function to obtain unique task identifiers which is used to post tasks and later used to dispatch and run tasks. The McuSleep interface is used by the scheduler to turn the device into the low power sleep mode when the task queue is empty.

The Scheduler interface provided by TinyOS has commands for initialization and running tasks, and is used by TinyOS to execute tasks as illustrated in Figure 6.6.

```
interface Scheduler {
        command void init();
        command bool runNextTask(bool sleep);
        command void taskLoop();
}
```

Figure 6.6: Scheduler Interface

The ‘init’ command is used to initialize the task queue. The taskLoop() is an infinite loop which checks task identifiers in the task queue. If it finds the queue empty, it continues low power sleep mode. Otherwise, it signals the TaskBasic.runTask event to execute the first task in the queue. If there are more tasks to execute, it executes the runNextTask command.

```
command void Scheduler.taskLoop(){
        for (;;)
        {
         uint8_t nextTask;

         atomic
         {
                while ((nextTask = popTask()) == NO_TASK)
                {
                        call McuSleep.sleep();
                }
         }
         signal TaskBasic.runTask[nextTask]();
        }
}
```

Figure 6.7: The Scheduler Task Loop

Figure 6.8 shows the postTask command in the TaskBasic interface that provides task posting feature to TinyOS applications.

```
async command error_t TaskBasic.postTask[uint8_t id]()
  {
    atomic return pushTask(id) ? SUCCESS : EBUSY;
  }
```

Figure 6.8: TaskBasic Post Task

Figure 6.9 shows the pushTask function which pushes unique task identifiers to the task queue. It is implemented using an array data structure.

6.1.3 Architecture of Task Distributor

We have implemented a modified version of tinyOS scheduler which has complete functionality of remote task distribution. We have modified the tinyOS scheduler to enable tasks posting to neighboring nodes and to get results back from them using radio components. It does security analysis on the data that is going to be distributed with a task and stops distribution if it violates security policy. It posts tasks for local execution if remote node is not discovered or does not reply to the sender.

Figure 3.1 represents the modified architecture of TinyOS scheduler. An application posts tasks to the scheduler for execution. There are two types of tasks: basic and distributable. Basic tasks are posted via the basic task interface as described in Section 6.3 where as distributable tasks are posted via the new task interface provided by the task distributer. The distributable tasks can be posted to idle neighbors by the scheduler for efficiency of computation.

```
bool pushTask( uint8_t id )
{
        if( !isWaiting(id) ){
                  if( m_head == NO_TASK )
                  {
                        m_head = id;
                        m_tail = id;
                  }
                  else
                  {
                        m_next[m_tail] = id;
                        m_tail = id;
                  }
                  return TRUE;
        }
        else
        {
                return FALSE;
        }
        }
```

Figure 6.9: The Push Task Method

The scheduler also contains a security analysis component to make distribution decision. It does data flow analysis on data carried by a particular task to check if it depends upon secure variables of a program. If it finds data dependency on secure variables in the program, it stops the task distribution and posts it locally. Also, in case of communication failure, error message or no reply from the remote node, it posts the task back for local execution.

6.1.4 Changing TinyOS Scheduler

The default scheduler of a typical application in TinyOS can be replaced by a customized scheduler. The application developer should make a component named TinySchedulerC in the application directory which requests the system to replace the default configuration at compilation time. This component provides a wiring of the desired scheduler implementation. Figure 6.10 is an example for the Blink application that uses TinySchedulerC explicitly.

```
configuration BlinkAppC {
}
implementation {
        components BlinkC, TinySchedulerC;
...................
}
```

Figure 6.10: The Blink application trying to override the default scheduler

If we want to have a scheduler that provides the task distribution functionality which is capable of posting tasks to local node or remote node at run time, we should modify the default scheduler. We have provided this feature through an additional distribution interface similar to the TaskBasic interface.

The TinySchedulerC component provides an additional interface called TaskDist for the task distribution to which is similar to the TaskBasic interface provided by TinyOS for basic tasks. Figure 6.11 is the new TaskSchedulerC component modified from the original version presented in Figure 6.4. It has wiring with the modified scheduler SchedulerDistP.

The modified scheduler has two additional task queues called Distribution Task Queue and Received Task Queue in addition to the Basic Task Queue. Each queue has been implemented using an array similar to the basic tasks in TinyOS. Figure 6.5 presents the modified task loop for the task distribution. Figure 6.7 shows the task loop in default TinyOS.

The default implementation of the scheduler presented in Figure 6.5 is modified to the SchedulerDistP as shown in Figure 6.13. The new implementation provides an additional interface called the TaskDist interface together with the TaskBasic interface. The TasDist interface posts tasks to a remote node instead of a local node. Upon failures or security policy violations, it can post tasks in the local queue

```
#include "hardware.h"
#include "Timer.h"

configuration TinySchedulerC {
  provides interface Scheduler;
  provides interface TaskBasic[uint8_t id];
  provides interface TaskDist[uint8_t id];
}
implementation {
  components SchedulerDistP as Sched;
  components McuSleepC as Sleep;

 //LocalTime Interface to handle failures
 components new CounterToLocalTimeC (T32khz) as counter;

//Radio component used for task distribution
  components RadioProviderP as RPP;

//Wire-through of TinySchedulerC and SchedulerDistP
  Scheduler = Sched;

//Basic tasks posted via TaskBasic
  TaskBasic = Sched.TaskBasic;

//Dist tasks posted via TaskDist
  TaskDist = Sched.TaskDist;

//SchedulerDistP uses RadioControl to distribute tasks
  Sched.RadioControl -> RPP.RadioControl;

//SchedulerDistP resets when MCU goes to sleep
  Sched.LocalTime -> counter;
  Sched.McuSleep -> Sleep;


}
```

Figure 6.11: The modified TinySchedulerC component

for local execution. The TaskDist interface is shown in Figure 6.14.

When a task is executed from the received task queue, the ‘runTask’ event associated with the task identifier is signaled from the scheduler with the corresponding code. This is similar to the basic task execution technique where corresponding code inside a function with the ‘task’ keyword is executed by the scheduler.

The TaskDist interface has an additional ‘replyBack’ command and an ‘updateLocalData’ event. When a device completes a task received from a remote device, it sends the result back using the ‘replyBack’ command. After getting

```
//Low power scheduler loop; if no tasks, sleep; else run task
  command void Scheduler.taskLoop()
  {
    for (;;)
    {
//Distribution task identifier
      uint8_t nextDTask = NO_TASK;
//Received task identifier
      uint8_t nextRTask = NO_TASK;
//Basic task identifier
      uint8_t nextMTask = NO_TASK;

//Infinite Task Loop
      atomic
      {
        while (
//Check tasks in distribution queue
               (nextDTask = popDTask()) == NO_TASK &&
//Check tasks in basic task queue
               (nextMTask = popMTask()) == NO_TASK &&
//Check tasks in received task queue
               (nextRTask = popRTask()) == NO_TASK)
        {
//No tasks, sleep
          call McuSleep.sleep();
        }
      }
//Distribute tasks
      if (nextDTask != NO_TASK) {
        atomic distributeTask(nextDTask);
      }
//Run basic tasks
      else if (nextMTask != NO_TASK) {
        signal TaskBasic.runTask[nextMTask]();
      }
//Run task from received task queue
      else if (nextRTask != NO_TASK) {
        signal TaskDist.runTask[nextRTask](r_data[nextRTask]);
      }
    }
  }
```

Figure 6.12: The modified scheduler Task Loop for the task distribution

```
module SchedulerDistP {
  provides interface Scheduler;
  //Default Task Handler
  provides interface TaskBasic[uint8_t id];
  //Distribution Task Handler
  provides interface TaskDist[uint8_t id];
}
```

Figure 6.13: The SchedulerDistP Comoponent

execution result from the remote device, the sender of the task signals the ‘updateLocalData’ event to reflect the completion of task. The scheduler signals this event only after a result message is received from the remote node.

```
interface TaskDist{

//Post tasks to Remote Node
  async command error_t postTask(uint32_t data);

//Run tasks request received from remote node
  void  event runTask(Data rdata);

//Reply back to sender with result if needed
  command bool replyBack(uint32_t reply);

//Update local data once reply received from remote node
  void event updateLocalData(uint32_t rdata);

}
```

Figure 6.14: The TaskDist Interface

The tasks for distribution are given the highest priority. The local tasks are given a lower priority. The tasks received from remote devices are given the lowest priority. Whenever the scheduler finds tasks in the distribution queue, it distributes corresponding task to remote nodes. If it finds tasks in the received task queue, it executes corresponding tasks and sends the results to the task originators. The basic task execution approach is unchanged.

The 'distributeTask' method checks the security tags of the associated data with tasks and decides whether it can be exposed to the remote nodes or not. If there is no security risk, it sends the task execution requests to remote nodes, which have the same task code. The receiver node posts the received tasks to the received task queue for the execution request and sends result back to sender after completion of it. If risks are present, it simply dispatches the execution signal locally, which is equivalent to posting tasks to the local task queue. Figure 6.15 presents the method responsible for posting a task execution request to a remote node.

The scheduler uses a component called RadioControl which contains radio interfaces used for the communication. It is used to send a task request to a remote

```
//Distribute Task to Remote Node
void distributeTask(uint8_t remoteTask){
        boolean success;
//Backup task_id and data for handling failures
        s_tasks[remoteTask] = remoteTask;
        s_data[remoteTask] = d_data[remoteTask];

//Check security tag
//(SEC_THRESOLD is minimum secure tag value)
if (tag_value[remoteTask] > SEC_THRESOLD){
        //There is security risk, execute to local node
        signal RadioControl.executeToLocalNode(remoteTask);}
        //No Security risk
         else{
                  busy = TRUE;
                  success = call RadioControl.sendTask(remoteTask,d_data[remoteTask]);
                  //If failure, execute to local node
                  if (!success){
                        signal RadioControl.executeToLocalNode(remoteTask);
                  }
                 }
        busy = FALSE;
        return;
}
```

Figure 6.15: The Task Distribution Method

device and receive the computed result back. Also, the receiver uses it to receive a task request and send the computed result back. For the reliability of communication, the RadioControl component uses acknowledgements and timers. If a receiver does not acknowledge the sender, it reports an error to the scheduler. Also, if it successfully sends a task to a remote node, it starts a timer for the corresponding task. If the receiver does not reply back with the result within some fixed time interval or it replies back with an error, it signals a failure to the scheduler. Then the scheduler executes the task locally.

6.1.5 Using Task Distributor

To use the Task Distributor, an application developer should write a program by following the template provided in Figure 6.16. Also, the developer needs to include few additional nesC files listed in 6.1.5 in the application directory. These files are used by the Task Distributor to override the default scheduler of TinyOS.

1. TinySchedulerC.nc

2. SchedulerDistP.nc

3. TaskDist.nc

4. RadioControl.nc

5. RadioProviderApp.nc

6. RadioProviderP.nc

7. sender.h

```
module a {
  ...
  uses interface TaskDist as x;
}
implementation {
//Basic task
 task foo(){
 }
//Posting Basic task
 post foo();
//Post distribution task from application
  call x.postTask();
//Signalled by scheduler to run dist task
  event void x.runTask(Data rdata) {
.....task logic.........

//If it signaled from received queue, reply back sender
//equals is string comparison method
  if equals(rdata->type,'rQueue'){
    //Initialize data and send Back
          x.replyBack(Data);
        }
  }
//Signalled by scheduler to update result from remote node
   event void x.updateLocalData() {
        ...
   }
}
```

Figure 6.16: Template of Task Distribution Framework

```
//Configuration of application (FooC.nc)
//Used to create unique task identifier for each instance
#define UQ_TASK_DIST "TinySchedulerC.TaskDist"
configuration FooC {
  ...
}
implementation {
  components FooP, TinySchedulerC;
  FooP.BlinkTask -> TinySchedulerC.TaskDist[unique(UQ_TASK_DIST)];
  FooP.SenseTask -> TinySchedulerC.TaskDist[unique(UQ_TASK_DIST)];
}
//Implementation of application (FooP.nc)
module FooP {
  uses interface TaskDist as BlinkTask
  uses interface TaskDist as SenseTask
}
implementation {
//Result of sensing
   uint8_t sense_data;
// The TaskBasic example, written with keywords
  task void localTask() { ... some logic ... }
// Complete dist task example with sense use case
    event void SenseTask.runTask(Data rdata) {
// Record sense data function logic
        sense_data = SenseTask();
         //If it signalled from received queue, reply back sender
      if equals(rdata->type,'rQueue'){
        //Initialize data and send back to sender
          Data->data = sense_data;
          x.replyBack(Data);
        }
      }
// The Dist example, written with taskname.runTask method
  event void BlinkTask.runTask(Data rdata) {
  ... some logic ... }
void internal_function() {
  ..........some logic.....
//Posting distribution tasks example
            call SenseTask.postTask(20);
            call BlinkTask.postTask(100);
//Posting basic task example
        post localTask();
  }
//Signalled by scheduler to update result from remote node
     event void x.updateLocalData(Data rdata) {
        //local sense data value is updated to reflect the sense execution
        sense_data = rdata->data;
     }
}
```

Figure 6.17: Complete example of using task distributor

Figure 6.17 is a TinyOS example application which uses the Task Distributor. The 'BlinkTask' and the 'SenseTask' are distributable tasks and the 'localTask' is a basic task. The scheduler can send a request to a remote device to execute a distributable task if the request does not carry sensitive information of the application. The basic tasks are always executed locally.

6.2 Security Assurance

6.2.1 Overview

We have developed a static analysis tool which does data flow analysis of a program and finds out the risky task distributions involving secure information. But the static analysis cannot determine the exact behavior of the program at runtime. We have implemented a dynamic data flow analysis technique to trace the data flow during execution. Finally, we have implemented a security enforcement mechanism, which uses the result of dynamic analysis and blocks the task distribution, which tries to send out the secure data out of the system.

6.2.2 Static Data Flow Analysis

The Gnu C compiler produces Static Single Assignment (SSA) representation during compilation of a program. The compiler does data flow analysis on program code using the SSA form for optimization. We have used this representation to construct the Data Flow Diagrams for security analysis.

6.2.2.1 SSA Representation

The static analysis tool takes SSA form of the program as an input. The SSA form is an intermediate representation of a program during the compilation process. We can get the SSA form of any program, which can be compiled by the Gnu C (gcc) compiler by passing '-fdump-tree-ssa' option in compilation command. Figure 6.18

is the command used to compile nesC program and produce the program in SSA form during the compilation process.

```
FLAG=-Os -Wall -fdollars-in-identifiers
--param max-inline-insns-single=100000
CC=gcc -b avr -V 4.1.2 -mmcu=atmega128 -fdump-tree-ssa
all: app.c
        $(CC) $(FLAG) -o app.exe app.c
```

Figure 6.18: gcc command to get SSA form of program

We have written a Perl script to preprocess the SSA form and make it suitable for the data flow analysis. It follows the algorithm presented in Figure 1 which takes the SSA form as an input and gives the processed SSA as an output. It extracts code sections of a program involved in a task distribution and makes the function calls inside an extracted section inline. The preprocessed SSA is used by the data flow analysis algorithm to build the set of data flow graph. Figure 6.19 presents a snippet of a C program on the left hand side and the preprocessed SSA on the right hand side.

6.2.2.2 Building Data Flow Graph

The static analysis tool reads each line of a preprocessed SSA and represents it in a graphical structure called a DFG. The Perl script generates the nodes as defined in Definition 1 of a statement using regular expressions and keeps their dependencies, which are edges as defined in Definition 2 of the graph using a multidimensional associative array (hash) data structure. The associative array uses the nodes and line number of the statement as keys and the corresponding dependent nodes as values. These key-value pairs represent adjacent nodes of a DFG with an edge from

```
//Variable Definition
uint8_t x, y, z,i ,
j, sec_data1,m,n;
//sec_data1: Secure
sec_data1 = 5;
x = 2;
n = 3;
y = x+1;
z = y+5;
z = z+1;
//Condition
if  (z%2 == 0)
    i = sec_data1;
else{
    m = n+7;
    i = m+2;
        }
j = i;
call remoteTask1.postTask(j);
```

```
<<START>>: data_35
start1_24 = 0;
sec_data1_25 = 5;
x_26 = 2;
n_27 = 3;
y_28 = x_26 + 1;
z_29 = y_28 + 5;
z_30 = z_29 + 1;
D.19312_31 = (int) z_30;
D.19313_32 = D.19312_31 & 1;
if (D.19313_32 == 0)
goto <L14>;
else goto <L15>;
<L14>:;
i_42 = sec_data1_25;
goto <bb 10> (<L16>);
<L15>:;
m_40 = n_27 + 7;
i_41 = m_40 + 2;
#i_2 = PHI <i_42(8),i_41(9)>;
<L16>:;
j_33 = i_2;
D.19317_34 = (uint32_t) j_33;
data_35 = D.19317_34;
<<END>>: data_35
```

Figure 6.19: C code with corresponding SSA representation

the key to the value. We have used GraphViz tool in Perl to represent the associative array in a graphical image.

Figure 6.21 is an example of a Perl program for processing simple statements of a program. We have complex scripts to handle complex operations like decisions and loops. Figure 6.20 is an example of hash representation of program statements. Here, variable ‘x’ has two outgoing edges in the graph with destination nodes ‘OP1’ and ‘z’ and there is an edge between node ‘y’ to ‘j’ and node ‘z’ to ‘i’. It is recorded by using an associative array ‘dfg_array’ as shown.

Figure 6.19 is an example of representing a statement into a DFG using an associative array in Perl.

```
        //Statements
        1: y = x+1;
        2: z = x;
        ......
        5: i = z;
        .......
        10: j = y;

        //Hash Representation of statements to construct Graph
        dfg_aray{x}{1} = 'OP1';
        dfg_aray{1}{1} = 'OP1';
        dfg_aray{OP1}{1} = 'y';
        dfg_array{x}{2} = z;
        dfg_array{z}{5} = i;
        dfg_array{y}{10} = j;
```

Figure 6.20: Manipulation of Perl Hash to DFG

```
@nodes = split(/=/,$line);
$counter = 0;
foreach ($line){
$counter++;
@node_array = split ($operatorlist_regex,$nodes[1]);
$arraySize = scalar (@node_array);
#If line is complex operation
#Eg: x= y+z-2;
if ($arraySize > 1)
        {
        #Give all operators in a line to same name
        $operator = "OP".$counter;
        # For all operands point to operator
        # Operator finally points to result (LHS)
        foreach $data (@node_array)
        {$dfg_array{"$data"}{$counter} = $operator;}
         #Node 0 is LHS node (result)
         $dfg_array{$operator}{$counter} = "$nodes[0]";
        }
#Else, simple assignment (eg: x = y;)
else
{
$dfg_array{"$nodes[1]"}{$counter} = "$nodes[0]";
}
}
```

Figure 6.21: Perl associative array representation of a SSA for DFG

6.2.2.3 Building Data Dependency Graph

The output of the previous section is a set of DFG of a program represented in the SSA form. Since we do not need all of these graphs for security analysis, we need to filter out subset of them. The subset is the set of DDG which includes all variables and their dependencies reaching to distribution points of the program. We have

```
#Start from distribution point
$value = $distribution_point;
sub dfa
{
        #Get connected nodes to start point
        @key_array = get_hash_keys_by_value($value);

        if ( is_empty($key_array) && is_empty($stack) ){
                return;
        }
        else {
        foreach ($key_array){
                if (!marked($key)
                #Push into stack
                push(@node_stack , $key);
        }
        #Pop key
        $key = shift(@node_stack);

        #Mark key
        $mark($key);

        #Save dependency information
        save_path($key,$value);

        #Recursive call (value = key)
        dfa ($key)
        }
}
```

Figure 6.22: Perl script to get a DDG for a distribution point

written a recursive Perl program which finds out all of the task distribution point in a program, takes each distribution points as the start point and extracts the set of DFG ending to that point. The result is the set of DDG of those sections of the program which can distribute tasks at runtime.

Figure 6.22 is a Perl function which starts from a distribution point of a program, searches dependent variables backward in a set of DFG and results a DDG as final output. It follows the Algorithm 3 as defined in Definition 4.2.4.

Figure 6.23 presents the DDG of the SSA presented in the Figure 6.19 using the Perl program which is demonstrated in Figure 6.22.

To prevent a system from exposing the security critical data out of it, we have to block the task distribution, which can send data having dependency on secure

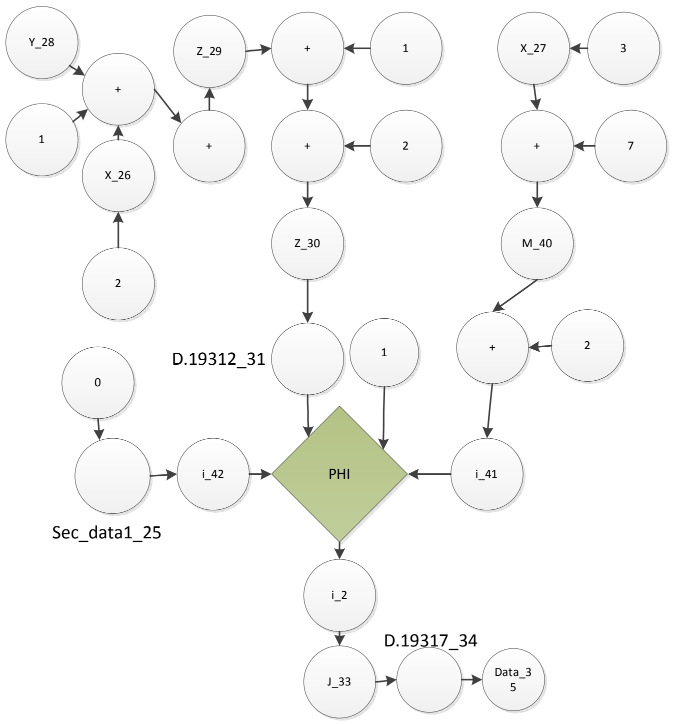

Figure 6.23: The DDG of the Example 6.19

variables. The static data flow graph can show potential dependency between variables but it cannot tell exact dependency at runtime. The presence of complex structures like branches and conditions make data dependency complex and dynamic. To find out which data flow path current execution followed needs runtime analysis called Dynamic Data Flow Analysis. We have implemented a dynamic data flow analysis by using the program instrumentation technique. It helps the Task Scheduler check the dependency of outgoing data on secure variables and block the corresponding task distribution.

6.2.3 Dynamic Data Flow Analysis

The dynamic analysis constructs a set of SDFG from the set of DDG which is the output of static analysis. The SDFG is further optimized into a graph structure called Trail Graph which reduces the nodes of a SDFG into the Trail Nodes and the edges to the Trails. The Trail Graph optimizes and preserves complete information of a SDFG required for dynamic security analysis. The analysis assigns Tag Scores to secure variables and propagates them by using the Trails of the Trail Graph. The Trails propagate the dependency information of secure variables to the distribution point using the Tag Values associated with them. At the distribution point, the Tag Value can be checked to determine whether current execution has data dependency with secure variables or not.

6.2.3.1 Constructing SDFG

We have written a Perl script, which computes a SDFG from a DDG by keeping the SeDF paths and trimming out the SaDF paths, which are not the sub paths of a SeDF. It follows the algorithms described in the Section 5.2. The Perl script presented in Figure 6.24 finds out the set of SeDF and SaDF in the DDG. It prepares a list of secure nodes ‘SecureNodeList’ and sets a flag ‘HasSecureChild’ to the nodes having direct and indirect dependency with secure variables. The‘StartNode’ input is the distribution point from where it travels back until it visits complete nodes of the DDG. Then, it trims out the set of SaDF from the DDG and results a SDFG as the final output by following the Algorithm 5. Finally, the script copies the set of SeDF to the final graph but excludes the set of SaDF which are not part of the SeDF. The set of edges that does not contain a secure node or a node with ‘HasSecureChild’ flag set as true is a SaDF which is not a part of SeDF.

The SDFG of the Example 6.19 is shown in Figure 6.25. It illustrates that the

```
sub sdfg
{
        #Get nodes of edges connected to the startnode
        @key_array = get_hash_keys_by_value($value);

        if ( is_empty($key_array) && is_empty($stack) ){
                return;}
        else{
                foreach ($key_array){
                        if (!marked($key)
                        #Push into stack
                        push_stack ($key);
                }

        #Pop key (Get first connected vertes)
        $key = pop_stack();

        #Mark key (visited)
        $mark($key);

        # Is key is secure variable, save to list
        if (is_secure($key)){
                # save to intermediate secure graph array
                addnode($key,$securenodelist);

                # mark all parents of current node has securechild
                mark_parent_hassecchild($key);
        }
        #Recursive call (value = key)
        sdfg ($key)

        # Function to save final result graph
        # Uses securenode list  and has_secchild flag to copy
        copy_final_result();

        }
}
```

Figure 6.24: Perl script to get Secure Data Flow Graph

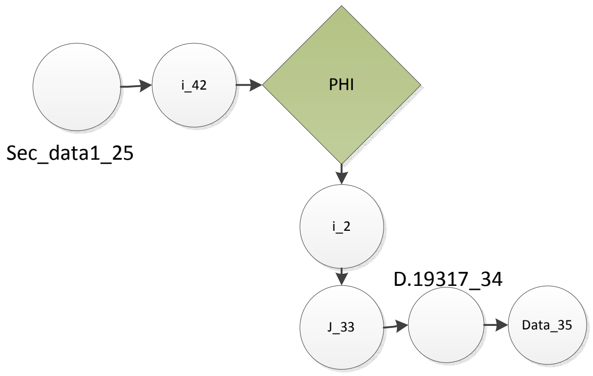

Figure 6.25: The SDFG of the Example 6.19

DDG presented in Figure 6.23 is optimized significantly by cutting off the nodes that do not have dependency on secure variables.

6.2.3.2 Finding Tag Update Points

Neither all nodes of a SDFG require Tag Scores nor do all edges require Tag Values to propagate secure data dependency. As mentioned in the dynamic analysis section 5.4, a SDFG can be optimized into a Trail Graph by using the concept of minimum tag sets. The nodes of the graph are called Trail Nodes which are connected to each other via the Trails. The Trail Nodes include Secure Nodes, Merge Nodes and Branch Nodes of a SDFG and these are the only points where the tag update is needed. Figure 6.26 presents a Perl script which finds the tag update points in a SDFG by using the Algorithm 6. It takes start nodes of a SDFG as an input and gives a list of Trails and Trail Nodes as an output. The output is used later for the implementation of the program instrumentation.

Figure 6.27 presents the Trail Graph of the SDFG presented in Figure 6.25. It illustrates that 7 nodes of the SDFG are reduced to 3 nodes of the Trail Graph. These are the points in the program section presented in Figure 6.19 where the instrumentation is needed.

6.2.3.3 Program Instrumentation

After finding the tag update points, the final step of dynamic analysis is the program instrumentation. Figure 6.28 presents the general steps for inserting an extra code in a program. It follows steps presented in Algorithm 7 by taking the output of the Algorithm 6 implemented in Figure 6.26 as an input. First, it inserts Tag Score and Tag Value initialization code for secure nodes and Trails. Then, it

```
sub FindTagUpdatePoints(){
$StartNodeStack = $_[0];
$path_hash = $_[1];
my @SecureTrailList;
$s1 = shift(@StartNodeStack);
@children = get_value_by_key($s1,$path_hash);

#Iterate through each connected nodes
foreach my $n1 (@children){
        #Define a new trail for each branch
        my $Trail = "Trail".$i;
        push(@SecureTrailList, "$Trail");
#Process start node
         if (isSecureNode($n1)){ #s1 is secure?
                $SecureNodeToTrailList{"$si"}{$i} = $Trail;
         }
         if (isOperator($s1)){ #s1 is an operator (Merge Node)?
                $OperatorToTrailList{"$si"}{$i} = $Trail;
         }
         if (isPHI($s1)){ #s1 is a PHI (Merge Node)
                $PhiToTrailList{"$si"}{$i} = $Trail;
         }
         if (hasNchild($s1)){ #s1 has n children (Branch Node)
                $BranchNodeToTrailList{"$si"}{$i} = $Trail;
         }
#Process connected node
        do {
         print "Processing␣child";
         if (isOperator($n1)){
                $TrailMergeToOperatorList{"$Trail"}{$i} = $n1;
                push(@StartNodeStack,$n1);
         }
         if (isPHI($n1)){
                $TrailMergeToPhiList{"$Trail"}{$i} = $n1;
                push(@StartNodeStack,$n1);
         }
        if (hasNchild($n1)){
                $TrailToBranchNode{"$Trail"}{$i} = $n1;
                push(@StartNodeStack,$n1);
         }
        if (isSecureNode($n1)){
                $SecureNodetoTrailList{"$n1"}{$i} = $Trail;
                #Do not add to stack, but go ahead
                $n1 = getchildren ($n1,$path_hash);      }
        } while (isOperator($n1) || isPHI($n1) || hasNchild($n1) );
        # Reinitialize
        $s1 = shift(@StartNodeStack)
        @children = get_value_by_key($s1,$path_hash)
} #End for loop
} #End of the function
```

Figure 6.26: Perl script to find tag update points

finds the corresponding location of Trail Nodes in the program and inserts the tag update and propagation code by using the input list. Also, it inserts the tag propagation code for the functions by following the method presented in the Section 5.5.1. An example of program instrumentation with minimum tag sets is presented

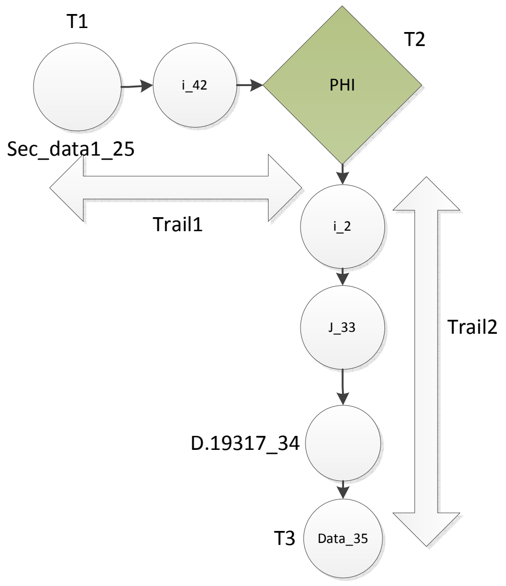

Figure 6.27: The Trail Graph of the SDFG 6.25

in Figure 5.3.

6.2.3.4 Securing Distribution

At the distribution point, the Task Scheduler checks the tag value of the outgoing Trail Node. If it finds it greater than zero, it blocks the distribution and posts it for local execution as shown in Figure 6.29 to protect the privacy.

6.2.3.5 Compiling Secure Code

The SSA form of the program is not directly compilable. We have to change it back to original form of the program to compile, execute and test. Due to lack of a perfect tool and method, we have performed this step manually. We followed the steps presented in the Algorithm 8 to change a secure SSA back into the original program form. First, we compare the SSA and the secure SSA forms of a program to find the difference. Then, we find and insert the difference into the corresponding

```
sub insert_tag_update_code(){
$SecureNodeToTrailList = $_[0];
$SecureTrailList = $_[1];
$OperatorToTrailList = $_[2];
$PhiToTrailList = $_[3];
$BranchNodeToTrailList = $_[4];
$file = $_[5];
#Initialize numeric tag score to all Secure Nodes
foreach my $sec_node (@SecureNodeToTrailList){
        write_tagscore_definition( $file, $sec_node);
}
#Initialize Tag Value of zero to all Trails
foreach my $trail (@SecureTrailList){
        write_tagvalue_initialization( $file, $trail);
}
#Find and insert tag update code for secure nodes
foreach my $sec_node,$trail (@SecureNodeToTrailList){
        write_secure_tagvalue_update( $file, $sec_node,$trail);
}
#Find and insert tag update code for operator nodes
foreach my $operator_node,$trail (@OperatorToTrailList){
        write_merge_tagvalue_update( $file, $operator_node,$trail);
}
#Find and insert tag update code for PHI nodes
foreach my $phi_node,$trail (@PhiToTrailList){
        write_merge_tagvalue_update( $file, $phi_node,$trail);
}
#Find and insert tag update code for branching nodes
foreach my $branch_node,@trail (@BranchNodeToTrailList){
        write_branch_tagvalue_update( $file, $branch_node,@trail);
}
# Find tasks and insert TrailList
        find_insert_tasktrail_code($file);
# Find and insert reset code for tasks
        find_insert_reset_tasktrail_code($file);
# Find and insert tag propagation code for functions
        find_insert_function_tag_code($file);
} #end of the function
```

Figure 6.28: General Perl script for program instrumentation

```
void distributeTask(uint8_t remoteTask){
   //Backup task and data associated dat
   s_tasks[remoteTask] = remoteTask;
   s_data[remoteTask] = d_data[remoteTask];
   //Check tag value and make decision
   if ( getTagValue(remoteTask,d_data[remoteTask]) > 0){
                signal TaskDist.runTask[remoteTask](d_data[remoteTask]);
        }
   else{
        call RadioControl.sendTask(remoteTask,d_data[remoteTask]);
        busy = TRUE;
        }
}
```

Figure 6.29: Checking security tag value at the distribution point

location in the original program by changing the difference in original program form by following standard conversion steps. Finally, the code is compiled and tested.

CHAPTER 7

EXPERIMENTS AND EVALUATION

We conducted experiments on TinyOS applications to evaluate the security and and overhead of the secure task distribution schemes.

7.1 Experimental Settings

We modified four TinyOS applications, which are included as example programs in TinyOS, to use the secure task distribution scheme. They are Blink (A1), Sense (A2), RCToLeds (A3), and RSToLeds (A4). Because these applications themselves have only local tasks, they are modified to have both local and remote tasks. New variables and functions are added to these applications for the experiments. The added variables include secure and non secure variables which allow us to perform security analysis with different type of nodes. The added functions are used to evaluate the correctness of inter-procedural security analysis. These functions are of a variety of types, including the functions that take or do not take arguments and that return or do not return values. Furthermore, we developed one testing application (A5) in which we can create arbitrary variables, functions and tasks to add complexity into applications for evaluation. The summary of these experimental applications is shown in Table 7.1.

To evaluate the secure task distribution scheme, we measured two categories of

Table 7.1: Experimental Applications

Apps	Vars	Sec-Vars	Funcs	Loc-Tasks	Rem-Tasks
A1	4	1	2	1	1
A2	4	1	1	1	1
A3	8	2	2	1	1
A4	3	1	1	2	1
A5	25	5	6	3	3

metrics. One category is of *security metrics*, including the metrics reflecting the characteristics of DDG, SDFG and Trail Graph (TG) discussed in Section 4 and the correctness of security check discussed in Section 5. The second category is of *overhead metrics.* As security code is added into original programs, it incurs overhead to affect performance of the instrumented programs. In particular, we examined the overhead incurred to code and data respectively.

7.2 Security Analysis and Evaluation

7.2.1 Graph Properties

The security analysis transforms a device program to the DDG and then the SDFG to find the data variables and the executions that may disclose the originator's data. Then, the TG is created to find and insert the tag update code to trace the secure data at run time. Hence, the achieved security is determined by the properties of the graphs.

Table 7.2 presents the properties of DDG, SDFG and TG of the experimental programs. The graphs are produced by the security analysis tool using Algorithms 3, 5 and 6. The properties include the number of nodes and the number of edges of the graphs. The properties show that even though many nodes (data variables) are

Table 7.2: Graph Properties

Apps	DDG		SDFG		TG	
	#Node	#Link	#Node	#Link	#Node	#Trail
A1	14	14	7	6	3	2
A2	20	22	8	7	3	2
A3	15	14	12	11	8	6
A4	20	22	7	6	7	5
A5	69	89	37	36	20	13

used in device programs, only a small portion (less than 30%) of the nodes, i.e. the trail nodes, are indeed needed for tag update and security check. The minimum tag set discussed in Section 5.4 is an effective technique to identify the nodes that are related to security.

7.2.2 Correctness of Security Check

We also checked the correctness of security check with the instrumented security code to ensure that originator data security is enforced. The correctness is measured by two metrics. One is true positive (TP) rate that is the percentage of secure nodes that are denied to be sent to neighbors. The other is true negative (TN) rate that is defined as the percentage of non-secure nodes that are allowed to be sent to neighbors.

For comparison, we inspected three different security enforcement. One is no security. The second is our scheme that only secures applications with minimum tag set. The third is a stricter security enforcement that a remote task is denied as long as a part of a SeDF is executed. Table 7.3 shows the results. Our scheme using minimum tag set achieves the desired originator data security. When no security is enforce, all secure nodes are dispatched with remote tasks to neighbors. When

Table 7.3: Correctness of Security Check

Apps	No Security		Minimum Tag Set		Stricter Security	
	TP	TN	TP	TN	TP	TN
A1	0	1	1	1	1	0.64
A2	0	1	1	1	1	0.71
A3	0	1	1	1	1	0.43
A4	0	1	1	1	1	1.00
A5	0	1	1	1	1	0.65

security enforcement is stricter, a portion of non-secure nodes are denied and the applications are over-protected.

7.3 Overhead Analysis and Evaluation

Because new code is inserted to the program to assign tag scores and tag values and track and update tags, it incurs overhead in both code and data. For comparison, we inspected four variations of the experimental programs. The first (V1) is the base line that uses the original applications. The second (V2) are the applications with additional secure data and new functions. The third (V3) are the second variation of applications with the distributed task scheduler. The last (V4) are the third variation of applications with the inserted security code.

7.3.1 Code Overhead

Table 7.4 shows the code overhead of the four variations of experimental applications. For all applications, their V2's code sizes are a little larger than their V1's code sizes because of the added functions.

The difference between V2 and V3 is not uniform. For A1 and A2, their V3's code sizes are significantly larger than their V2's. It is because V3 adds the

Table 7.4: Code Overhead

App	V1 (KB)	V2 (KB)	V3 (KB)	V4 (KB)
A1	84.9	89.4	434.4	464.8
A2	142.1	148.8	493.8	524.2
A3	416.7	422.9	435.0	483.4
A4	475.6	481.8	491.6	532.0
A5	403.1	430.7	441.7	554.6

distributed task scheduler that contains an additional radio component. However, for the other three applications, their V3's code sizes are just a little larger than V2's, because the applications have the radio component in themselves already.

The security code contributes to the difference between V3 and V4. Because of using minimum tag set, the overall code overhead is reduced significantly. It is mainly determined by the complexity of TG. As discussed in Section 5, the security code include the tag update code, the tag initialization code, and the security check code. The tag update code is inserted at each trail node and is an additive function over a set of tag values assigned to incoming trails. Hence, the total code overhead can be estimated as $c_t(x+y)+c_i+c_c$, where x is the number of trail nodes, y is the number of trails, c_t is the unit size of each tag update block, c_i is the code size of tag initialization, and c_c is the code size of security check. The difference between V3 and V4 well reflects the estimation based on the properties of TG in Table 7.2.

7.3.2 Data Overhead

New variables are introduced during instrumentation of an application to tag and track the critical data flows. The data memory overhead is proportional to the Trail Graph properties since a variable is introduced for each Trails and Trail Nodes. The

Table 7.5: Data Overhead

App	V1 (B)	V2 (B)	V3 (B)	V4 (B)
A1	51	52	405	410
A2	47	51	388	393
A3	300	304	392	406
A4	312	316	388	400
A5	253	253	396	429

size of each of these variables is one byte. So, if there are ‘X’ Trails and ‘Y’ Trail Nodes, the data memory overhead is ‘X+Y’ bytes.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

In this thesis, a secure task distribution scheme for a network of weak embedded devices is proposed. The scheduler of TinyOS is modified to dispatch the offloaded computation as remote tasks to peer devices. To ensure that the distributed tasks do not disclose sensitive data in the originating devices, a static security analysis as well as a run-time security protection mechanism is proposed and implemented.

The static analysis uses the SSA form of a program to construct a set of DFG. The subset of the DFG ending to the distribution point is called a DDG which can show potential risky task distributions via the data flow paths. The dynamic analysis instruments the application code by tagging and tracking the critical data flows computed by the static analysis. It performs a security check at the task distribution point to ensure that the distributed tasks will not carry any data that may disclose secure information.

The experiment results show that the total overhead of the Task Distributor is minimal and it is highly acceptable for the weak mobile devices like sensors. The static analysis builds the complete set of DFG of a program accurately and represents all possible data flows. The dynamic analysis confirms the secure task distribution during runtime.

8.2 Future Work

Use of global variables can introduce indirect data dependency in a program. For example, the last use of a global variable on different execution flow may leave it dependent with a secure variable but the dynamic data flow analysis may not know the history and treat it as a normal data. To handle this situation, we can instrument the program code to calculate the last tag score associated with the global variables by computing data dependency graphs on them. This can be done by repeating the same algorithms proposed in the static and the dynamic analysis sections of this thesis. A global variable can be treated as the end point of a dependency graph similar to the distribution point of a program. The final tag value associated with the global variable can be used by the dynamic analysis as the security tag score.

In this thesis, the dynamic analysis instruments a program in the SSA form, which cannot be compiled directly. Due to the lack of a perfect tool to convert it back to the original form, it is done manually. It follows the standard SSA conversion theories, which can be automated.

BIBLIOGRAPHY

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. (2009). Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4:1–4:40.

[2] Appel, A. W. (1999). *Modern Compiler Implementation in ML*. Cambridge University Press.

[3] Balakrishnan, G., Gruian, R., Reps, T., and Teitelbaum, T. (2005). Codesurfer/x86 - a platform for analyzing x86 executables. In *Proceedings of the 14th international conference on Compiler Construction*, CC'05, pages 250–254, Berlin, Heidelberg. Springer-Verlag.

[4] Briggs, P., Cooper, K. D., Harvey, T. J., and Simpson, L. T. (1998). Practical improvements to the construction and destruction of static single assignment form. *Software Practice & Experience*, 28(8):859–881.

[5] Castro, M., Costa, M., and Harris, T. (2006). Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 147–160, Berkeley, CA, USA. USENIX Association.

[6] Chang, W., Streiff, B., and Lin, C. (2008). Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 39–50, New York, NY, USA. ACM.

[7] Chong, S., Liu, J., Myers, A. C., Qi, X., Vikram, K., Zheng, L., and Zheng, X. (2007). Secure web applications via automatic partitioning. In *Proc. of ACM SOSP*, pages 31–44.

[8] Christensen, J. H. (2009). Using RESTful web-services and cloud computing to create next generation mobile applications. In *Proc. of ACM SIGPLAN*, pages 627–634.

[9] Chun, B.-G. and Maniatis, P. (2009). Augmented smartphone applications through clone cloud execution. In *Proc. of Usenix HotOS*.

[10] Chun, B.-G. and Maniatis, P. (2010). Dynamically partitioning applications between weak devices and clouds. In *Proc. of ACM Workshop on Mobile Cloud Computing Services: Social Networks and Beyond*.

[11] Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. (2010). MAUI: making smartphones last longer with code offload. In *Proc. of ACM MobiSys*, pages 49–62.

[12] Eom, H., St Juste, P., Figueiredo, R., Tickoo, O., Illikkal, R., and Iyer, R. (2012). SNARF: a social networking-inspired accelerator remoting framework. In *Proc. of the Workshop on Mobile Cloud Computing*, pages 29–34.

[13] Giurgiu, I., Riva, O., Juric, D., Krivulev, I., and Alonso, G. (2009). Calling the cloud: enabling mobile phones as interfaces to cloud applications. In *Proc. of ACM/IFIP/USENIX International Conference on Middleware*, pages 83–102.

[14] Guirguis, M., Ogden, R., Song, Z., Thapa, S., and Gu, Q. (2011). Can You Help Me Run These Code Segments on Your Mobile Device? In *Proc. of IEEE Globecom*.

[15] Guyer, S. and Lin, C. (Feb.). Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2):342–357.

[16] Huan, L., Hang, L., and Xia, Y. (Dec.). Access control technology research in embedded operating system. In *Embedded Software and Systems, 2005. Second International Conference on*, pages 7–14.

[17] Huang, H., Hu, C., and He, J. (July). To verify embedded system software integrity with tcm and fpga. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 1, pages 65–70.

[18] Huerta-Canepa, G. and Lee, D. (2010). A virtual cloud computing provider for mobile devices. In *Proc. of ACM Workshop on Mobile Cloud Computing Services*, pages 1–6.

[19] Keith Cooper, L. T. (2003). *Engineering a Compiler*. Morgan Kaufmann Pub.

[20] Kiriansky, V., Bruening, D., and Amarasinghe, S. P. (2002). Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA. USENIX Association.

[21] Ko, S. Y., Jeon, K., and Morales, R. (2011). The HybrEx model for confidentiality and privacy in cloud computing. In *Proc. of USENIX conference on Hot topics in cloud computing*.

[22] Lam, L. C. and cker Chiueh, T. (2006). A general dynamic information flow tracking framework for security applications. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pages 463 –472.

[23] Lu, Y.-F., Kuo, C.-F., and Pang, A.-C. (2012). A novel key management scheme for wireless embedded systems. *ACM SIGAPP Applied Computing Review*, 12(1):50–59.

[24] Miluzzo, E., Cáceres, R., and Chen, Y.-F. (2012). Vision: mClouds - computing on clouds of mobile devices. In *Proc. of the ACM workshop on Mobile cloud computing and services*, pages 9–14.

[25] Naedele, M. (Aug.). An access control protocol for embedded devices. In *Industrial Informatics, 2006 IEEE International Conference on*, pages 565–569.

[26] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. (2002). Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK. Springer-Verlag.

[27] Ron Cytron, Eanne Ferrantej, B. K. R. M. N. W. F. K. Z. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13:451–490.

[28] Satyanarayanan, M. (2010). Mobile computing: the next decade. In *Proc. of ACM Workshop on Mobile Cloud Computing Services.*

[29] Shankar, U., Talwar, K., Foster, J. S., and Wagner, D. (2001). Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 16–16, Berkeley, CA, USA. USENIX Association.

[30] von Ronne, J., Wang, N., and Franz, M. (2004). Interpreting programs in static single assignment form. In *Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators*, IVME '04, pages 23–30, New York, NY, USA. ACM.

[31] Zhang, K., Zhou, X., Chen, Y., Wang, X., and Ruan, Y. (2011). Sedic: privacy-aware data intensive computing on hybrid clouds. In *Proc. of ACM CCS*, pages 515–526.

[32] Zhang, X., Schiffman, J., Gibbs, S., Kunjithapatham, A., and Jeong, S. (2009). Securing elastic applications on mobile devices for cloud computing. In *Proc. of ACM workshop on Cloud computing security*, pages 127–134.

VITA

Sobit Bahadur Thapa was born in Kaski, Nepal on April 14, 1983, the son of Shree Kaji Thapa and Durga Thapa. In 2006, he received the Bachelor's Degree in Computer Engineering from the Thribhuvan University, Nepal. In 2011, he entered the Graduate College of Texas State University-San Marcos. Together with Dr. Qijun Gu, he published "Can you help me to run codes on your mobile device?" in IEEE Globicomm 2011.

Permanent Address: Sarangkot-7, Kaski
Nepal

This thesis was typed by Sobit Bahadur Thapa.