The Monitoring Core: A Framework for Sensor Security Application Development

Marco Valero*†, Selcuk Uluagac†, Venkatachalam S.†, Ramalingam K. C.†, and Raheem Beyah†
*Department of Computer Science
Georgia State University
Atlanta, Georgia 30303,USA
mvalero@cs.gsu.edu
†GT CAP Group, The School of ECE
Georgia Institute of Technology
Atlanta, GA 30332, USA
{venkat.subbu, ramalingam.chandrasekar}@gatech.edu
{selcuk, rbeyah}@ece.gatech.edu

***Abstract*—Wireless sensor networks (WSNs) are used for the monitoring of physical and environmental phenomena, and applicable in a range of different domains (e.g., health care, military, critical infrastructure). When using WSNs in a variety of real-world applications, security is a vital problem that should be considered by developers. As the development of security applications (SAs) for WSNs require meticulous procedures and operations, the software implementation process can be more challenging than regular applications. Hence, in an effort to facilitate the design, development and implementation of WSN security applications, we introduce the *Monitoring Core (M-Core)*. The M-Core is a modular, lightweight, and extensible software layer that gathers necessary data including the internal and the external status of the sensor (e.g., information about ongoing communications, neighbors, and sensing), and provides relevant information for the development of new SAs. Similar to other software development tools, the M-Core was developed to facilitate the design and development of new WSN SAs on different platforms. Moreover, a new user-friendly domain-specific language, the M-Core Control Language (*MCL*), was developed to further facilitate the use of the M-Core and reduce the developer's coding time. With the MCL, a user can implement new SAs without the overhead of learning the details of the underlying sensor software architecture (e.g., TinyOS). The M-Core has been implemented in TinyOS-2.x and tested on real sensors (Tmote Sky and MicaZ). Using the M-Core architecture, we implemented several SAs to show that the M-Core allows easy and rapid development of security programs efficiently and effectively.**

***Index Terms*—Wireless Sensor Network Applications, Monitoring Core (M-Core), M-Core Control Language (MCL), Sensor Security Application Development**

I. Introduction

Wireless Sensor Networks (WSNs) are no longer a nascent technology and they are actively deployed as a viable technology in many diverse application domains such as health care, military, and environmental. More companies offer sensor-based solutions, and the usage of billions of networked sensors are envisioned to be deployed on land, in sea, air, and space to detect and predict the environmental changes in an effort to build a globally pervasive nervous system [1]. Given their low cost and multiple functionality, they have been predicted to be one of the ten technologies that will change the world in the next 10 years [2]. Moreover, with recent initiatives such as *Cyber-Physical Systems*, *Internet of Things* [3], and *Planetary Skin* [1], sensor-based applications have gained new momentum in the research community and industry.

To enable more secure wireless sensor networks, the research community has identified many unique security threats, and there has been a tremendous effort to build mechanisms to detect and defend against these threats and a myriad of security solutions have been proposed in the literature. The task of developing security solutions for WSNs is challenging because (1) sensors have limited technical capabilities, and (2) software implementations for security require meticulous procedures and extra operations to provide a desired level of security for applications. However, this process can be expedited and simplified with the assistance of development tools. To facilitate the design, development and implementation of WSN security applications (SAs) we introduce the Monitoring Core (M-Core), a modular, lightweight and extensible middleware.

The M-Core provides information in the form of services that can be simultaneously accessed by multiple processes running on the sensor, and can be used to develop a wide range of SAs. The architecture of the M-Core is divided into sub-components, each of which provides some specific services to the SAs. The design of the M-Core facilitates its usage and understanding, and reduces the overhead of learning the details of the underlying sensor software architecture (e.g., TinyOS). The M-Core also reduces the risk of implementation flaws, which is the most common cause of broken security protocols. Attacks on protocols are often successful when a specific flaw in the implementation of a component is exploited to cause unexpected behaviors [4].

Along with the intuitive architecture of the M-Core, we leverage the collected data to provide developers with useful information in a timely manner to facilitate the development of their solutions. Using the M-Core, developers can spend less time gathering and processing data, and more time designing and implementing their actual algorithms.

To further reduce the time in code development, a new user-friendly domain-specific language, the M-Core Control Language (*MCL*), was developed to facilitate the use of the M-Core and its services. Using the M-Core and the MCL, we implemented four different security solutions which are presented in Section V.

The main contribution of this work include, the devel-

opment of a new service-oriented framework that expedites the development of security applications for WSNs, and the introduction of a new domain-specific language to highly simplify the use of the proposed framework.

The rest of the paper is organized as follows. The related work is discussed in Section II. Section III presents the overview of the M-Core framework and services. The MCL is formally introduced in Section IV and a sample usage is also given. The functional evaluation of the M-Core is presented in Section V. The benefits of the M-Core and MCL, based on the amount of work required for developing new ASs, is discussed in Section VI. We conclude the paper and discuss future work in Section VII.

II. Related Work

Application development in wireless sensors is carried out *close to the operating system level* [5]. This forces the application developer to have a detailed understanding of the underlying system's architecture while programming an application. To ease the programmer's task, many middleware approaches have been proposed to provide a programming abstraction for rapid application development.

Approaches like OASiS [6] and MiSense [7] propose an object-centric, ambient-aware, service-oriented sensornet programming framework to aid the application development in WSNs. In these approaches, a code implementing a basic unit of functionality is wrapped with a defined interface, called service. OASiS [6] provides services which handle dynamic service discovery and configuration, network heterogeneity, constraints management, application and network QoS aware functionalities. Similarly, MiSense [7] aims to provide a content-based publish/subscribe communication model. The service-oriented approach simplifies the work of a developer who can now identify the relevant and useful services from the domain service library and wire them to their application, instead of implementing the service functionality. Although [6] and [7] use a service-oriented programming approach similar to used by M-Core, the developer still has the overhead of learning the details of the framework (e.g., wiring the appropriate service interfaces to their application modules in NesC). Whereas, the M-Core provides a higher-level user-friendly language called the MCL, which simplifies the process of developing and wiring the applications. Also, the M-Core and MCL are specifically designed for security. Thus, the services provided are more security-focused and represent the services used by most security applications in the literature.

Agila is a middleware proposed in [8], which uses a mobile agent-based paradigm for development of WSN applications. Similar to [6] and [7], Agila sits on top of the underlying network infrastructure. However, the middleware proposed in [8] depends on agents, which are in effect, distinct virtual machines with dedicated instruction and data memory, to adapt to the varying application scenario. The work in [8] differs from approaches like [6] and [7], by making use of a higher-level language for simple development of these application-specific agents. However, due to the mobile agent-paradigm, these application-specific mobile agents developed using the high-level language are limited by their size. In contrast to this, the M-Core uses a high-level language, but not a mobile agent-paradigm, and thereby enables rapid WSN application development, and most importantly, it is not limited by the size of the application. Moreover, the M-Core allows the rapid prototyping of security applications, while the other approaches focus on other general purpose applications.

TinyDB [9], takes a data-centric middleware approach, which uses an Acquisitional Query Processing (ACQP) technique to query and gather data from the motes in a WSN. TinyDB considers all motes and sensing physical entities in the WSN as a database system and using the acquisitional query language, a user can send SQL-like queries to collect the sensed data. These queries are processed at the base station, and then dispatched to the sensors in the network. Before the query is dispatched, the optimizer in the base station, with the metadata information about all motes in the network, chooses a query plan with the lowest overall power consumption. TinyDB is a middleware specifically designed for data-centric WSN applications and for data aggregation. It is not for application development, it is a type of macro-programming approach. However, the M-Core, which takes a service-oriented programming approach, has the capability to provide services for the users to develop a data-centric application with emphasis on security. The modular approach for the implementation of the M-Core services, aids the programmers to develop secure data-centric applications, which are scalable and are easily modifiable based on the changing requirements of the applications.

III. The M-Core Architecture

In this section, we discuss the architecture of Monitoring-Core (M-Core) and its interactions with other main components of a sensor. We also present the M-Core services and discuss how they are used to provide services.

A. Design Principles

The M-Core runs on TinyOS [10]. TinyOS is a modular operating system (OS) based on components that are wired together through interfaces to create applications with different functionalities. Following this feature of the OS, we designed the M-Core with a highly modular architecture where every component is independent, and can be easily added and removed without affecting the rest of the system.

The M-Core enables a novel way to provide information for the development of new security applications based on a modular and independent set of services. The M-Core acts as an information manager and dispatcher to facilitate access to the services. The M-Core is divided into sub-components, each of which provides some specific services to the security applications (SAs). Note that any of these sub-components (services) can be easily removed or replaced, and more sub-components can be added to enhance the M-Core's functionality.

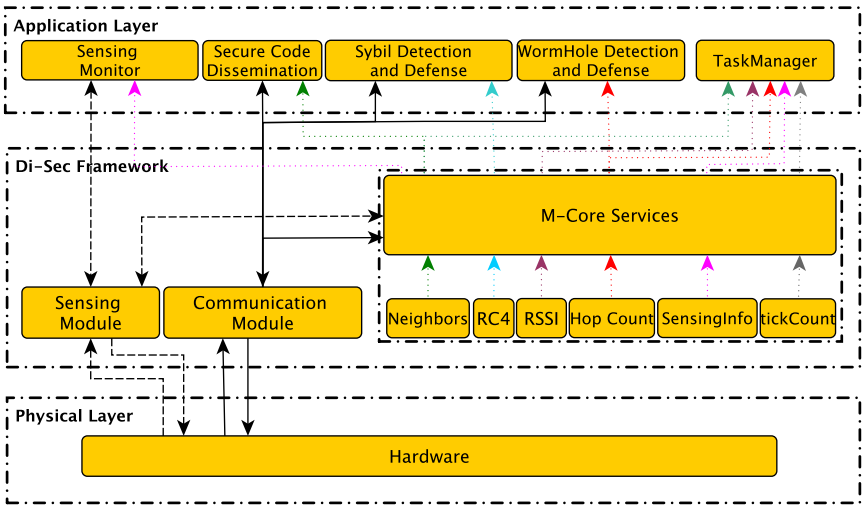

Fig. 1. M-Core data collection and flow.

To create a comprehensive software development architecture, we analyzed the functionality of WSN devices and identified three important functions of sensor devices: sensing physical or environmental conditions, processing collected data, and communicating with other devices. The sensor communication function is an essential component that should be monitored since it is one of the common and easy target for attacks. Therefore, we created our own communication module that controls packets transmitted and received through the radio transceiver and it is the main data source component that feeds the M-Core.

The M-Core collects its data from three components: the communication module, the sensing module, and the OS. All the data and packets going through the communication (COMM) module are also copied to the M-Core for collection and analysis. For each outgoing packet, the COMM module notifies the M-Core whether the transmission was successful or not. For all incoming packets, the COMM module passes a copy to the M-Core even though the packet is not addressed to that specific sensor node. The M-Core can also be connected to the sensor suites using a sensing module (SENSE) to monitor external phenomena. Using the SENSE module, the M-Core can intercept, monitor, and record all external sensing measurement values and requests. The operating system (i.e., TinyOS) also provides important data that can be leveraged by the M-Core to generate suitable information for security software development.

The benefits of using the M-Core for developing new WSNs software solutions are the following: (1) It has a built-in modular and flexible software architecture that provides an easy means to add, remove, and replace sub-components. (2) It is a lightweight monitoring and control layer invisible to upper layers. (3) It is easy to activate and use. (4) The provided services can be easily expanded. (5) The M-Core reduces the design and development time of software solutions.

The M-Core architecture is shown in Figure 1, where we also illustrate five SAs and six M-Core sub-components, each of which provides a service. Each SA includes the implementation of the necessary behavior utilizing the M-Core services. For instance, to implement a Sybil attack detection program, a developer might use the *rssivalue* interface (service) provided by the *RSSI* sub-component of the M-Core. The list of implemented M-Core sub-components and services are summarized in Table I and will be further described in the next sub-section, and the details of our implemented SAs will be discussed in Section V.

B. M-Core Services

In TinyOS jargon, *interfaces* are used to interconnect *components*. Each interface define *commands* and *events* that can be used by developers of new SAs to interact with existing solutions. Our implemented M-Core sub-components and services are presented in Table I and their descriptions are as follows:

1) ***commScan****:* The *commScan* sub-component can be used to detect the status of the communication channel based on two parameters: the number of consecutive successfully transmitted packets, and the average number of received packets per second. Both parameters can be obtained by the commands *getConsecutiveSuccess()* and *getpps()*, respectively. To collect information about the successfully transmitted packets, we configured the COMM module to notify the M-Core whether a packet is successfully sent or not by signaling two different events, *packetsuccess()* and *packetfail()*, respectively. For the received packets per second calculation, we count all the packets received by the node and average them every second. The COMM module passes a copy of all messages to the M-Core even when the packets are not addressed to the sensor.

This sub-component also provides a command *setThreshold()* that allows users to define a threshold for a minimum

TABLE I
M-CORE SUB-COMPONENTS

sub-component	Interface	Commands/Events	Action
commScan	channelinfo	getConsecutiveSuccess	Returns the number of consecutive sent packets
		getpps	Returns the number of received packets per second
		setThreshold	Sets the threshould for acceptable consecutive sent packets
packetCount	packetcount	getPacketCount	Returns the total received packets
		lostPacket	Returns the number of lost packets by node
packetSize	packetsize	getPacketSize	Calculates and returns the size of a packets
		getReceivedPacketSize	Returns the average size of all the received packets
RSSI	rssivalue	getRssiTable	Returns neighbors RSSI table
		initRssiTable	Initializes the neighbors RSSI table
LQI	lqivalue	getLqiTable	Returns neighbors LQI table
		initLqiTable	Initializes the neighbors LQI table
hopCount	hopcount	hopcount	Initiates the hop count process
		hopcountDone	Notifies the nodes when the hopcount value is ready
neighbors	neighbors	request	Triggers a neighbor discovery message
	neighborsinfo	getNeighbors	Returns the number of current neighbors
		initNeighbors	Initializes current neighbors table
commAttributes	commAttributes	setCommChannel	Changes the communication channel
		setTransPower	Adjusts the transmission power a the specified value
tickCount	tickcount	getTicks	Returns the number of CPU ticks to transmit a packet
sensingInfo	sensingstat	getAvgSenseValue	Returns the average sensed value aggregated at the mcore
RC4	encryptI	encrypt	Encrypts/Decrypts a message based on a secret key
remoteExecution	remoteexec	execute	Executes command provided by another M-Core component

acceptable successfully transmitted packet rate. If this threshold is not reached, the *commScan* sub-component will notify the upper layer by signaling an event. Some other possible uses of this sub-component include the detection of correct (or incorrect) functioning of the radio transceiver, the amount of traffic in the channel, and congestion in the network.

2) ***packetCount****:* The *packetCount* sub-component provides the interface *packetcount* with one command (*getPacketCount()*) to return the total number of packets received by the sensor, and one event *lostPacket()*, which is signaled every time a packet is lost by one of the node's neighbors. For the lostPacket event the sensor keeps track of all its neighbor's transmissions and maintains a four-tuple table to simplify the detection of a neighbor losing or dropping packets. The four-tuple contains the ID of the node who created the packet ($NodeS$), the ID of the node forwarding a packet ($NodeF$), the sequence number for the combination ($NodeS, NodeF$), and the total lost packets for the same combination. Hence, one can detect if any of the forwarding nodes F is dropping packets generated at node S. This sub-component can be used to detect unreliable communication links as well as malicious activities in the network.

3) ***packetSize****:* The *packetSize* sub-component is used to get information about the size of the packets. The *packetsize()* interface provides two commands: *getPacketSize()* which was designed for the users to directly ask the M-Core for payload size of a specific packet, and *getReceivedPacketSize()* which returns the average payload size of all received packets. Average payload size information can be used in conjunction with the received packets per second information (provided by the *getpps()* command) to calculate network throughput.

4) ***RSSI and LQI****:* The receiver signal strength indication (*RSSI*) and link quality index (*LQI*) are independent sub-components, but the way they collect data and operate is very similar. A copy of every packet received by the COMM module is passed to each of these sub-components where they parse the message to extract the RSSI and LQI values respectively. The sub-components maintain a neighbors' table and all the extracted values are averaged with the corresponding values from the tables. Our current implementation supports $CC2420$, and $RF230$ radios, which are used by many of the sensors currently available in the market. However, support for other radios can be easily added. The information provided by these two sub-components can be used for defining reliable links in routing protocols, and also for device fingerprinting for authentication.

5) ***neighbors****:* The *neighbors* sub-component provides two interfaces, *neighbors* and *neighborsinfo*, to gather neighbor information. Using the *request()* command from the *neighbors* interface, the sensor initializes a neighbor discovery process where it broadcasts a discovery message that is acknowledged by all the neighbors within its transmission range. Once the acknowledgement is received, the sensor refreshes its neighbor table with updated information. Accordingly, the *getNeighbors()* command provided by the *neighborsinfo* interface returns the neighbors' table, which can be initialized or reset with the *initNeighbors()* command. The information provided by this sub-component can be used to dynamically create communication routes for mobile WSN deployments.

6) ***commAttributes****:* The *commAttributes* sub-component is used to adjust communication attributes such as the communication channel and the transmit power using the *setCommChannel()* and *setTransPower()* commands respectively. These attributes can also be modified at run time, which is convenient for users that need to tune their application while running.

7) ***tickCount****:* The *tickCount* can be used to verify the CPU load. The command *getTicks()* provided by the *tickcount* interface initiates the transmission of an arbitrary packet to

calculate the number of CPU ticks elapsed from the message creation until the confirmation of the transmission. This value varies according to the CPU load, and can be used to establish a threshold for minimum or maximum CPU utilization. This sub-component can also be used for security to observe abnormal activities onboard.

8) ***sensingInfo****:* The *sensingInfo* sub-component collects information about the sensor readings and keeps an average of the measured sensing values. Sensor information is important to monitor since it can be used to verify the correct functioning of the sensor itself. For instance, an overloaded sensor might report a higher temperature readings due to overheating.

9) ***RC4****:* An instance of the *RC4* [11] encryption algorithm is also provided as a service in the M-Core. The *encrypt* command provided by the *encryptI* interface is used to *encrypt* and *decrypt* a message that is passed as a parameter along with the encryption key of a certain desired size.

10) ***hopCount****:* The *hopCount* is used to enable all the nodes to determine their hop-count values with respect to the cluster head. It provides the interface *HopCount* with a command *hopcount()* to initiate the hop-count service and an event *hopcountDone()* which is signaled once the nodes have determined their hop-count values. This service can be used periodically or on-demand.

11) ***remoteExecution****:* The *remoteExecution* is a sub-component used to remotely call commands from other M-Core sub-components to initialize variables as well as configure sensors attributes. This component can be used to initialize neighbors, RSSI, and LQI tables, to remotely change the channel and transmit power, and to adjust any parameter available in the M-Core.

To further facilitate the use of the M-Core and its services, we developed the M-Core Control Language which is explained in the following section.

IV. The M-Core Control Language (MCL)

To easily use the services provided by the M-Core, we have created a new domain specific language: the M-Core Control Language (MCL). In this section, we introduce the MCL, present the formal grammar of the language, and provide an example that shows it can be used to activate, deactivate or create new programs.

A. Rationale for the MCL & Formal Definition

The M-Core was designed to provide a full and extensible set of services for the development of new SAs. However, a programmer needs to learn the underlying M-Core architecture (e.g., modules, interfaces, events, and commands) and do additional implementation (e.g., wiring in TinyOS) to integrate existing software solutions with the M-Core and also for creating new ones. Moreover, this situation may be exacerbated given the sophistication needed to implement programs on sensors for a novice programmer (this is evaluated in Section VI). Therefore, the MCL has been designed to address these issues. It utilizes the sub-components defined in the M-Core and simplifies the programmer's work to easily activate, deactivate or create their SAs by automatically generating important programming components needed for the underlying M-Core architecture (e.g., configuration files, module files and wiring). The MCL is a language consisting of a small set of keywords. It was developed with Python. The formal definition of the grammar of the MCL using the Extended Backus-Naur Form (EBNF) is given in Listing 1.

Listing 1. Formal definition of MCL with EBNF.

```
MCL ::= 'START', SPACES,
        { KEYWORDS, '(', EXPRESSIONS, ')', SPACES },
        'END' ;

KEYWORDS ::= 'ACTIVATE' | 'STOP' | 'SET' |
             'ASSOCIATE' | 'DISSOCIATE' | 'NEW' ;

EXPRESSIONS ::= PARAMETERS, { [ ',', SPACES,
                PARAMETERS ] }, [ ',', SPACES, VALUE ] ;

PARAMETERS ::= [a-zA-Z]\w* ;

VALUE ::= \d* ;

SPACES ::= ' '* ;
```

A program written with MCL starts and ends with the keywords, *START* and *END*. Between these, one can use the other keywords *ACTIVATE*, *STOP*, or *NEW* to activate, deactivate or create the modules, respectively. A programmer can even define its own variables using the *SET* keyword.

B. Sample Usage

In this sub-section, we show a sample usage of MCL. In our realistic scenario, the user implements four WSN programs that can be interchangeably activated, deactivated, or even run in parallel using the MCL. The MCL script written by the user is given in Figure 2 (actual code snippet in the middle). Specifically, the user instructs the M-Core to activate existing program D1 and deactivate D2 and D3. The user also creates a new program D4 into the M-Core, and sets the specific activation time and specifies that it use the *tickCount* sub-component of the M-Core. In the example, *ACTIVATE* enables the existing D1 and specifies the D1 starting time. *ASSOCIATE* is used to connect the D1 to the sub-components of the M-Core. *STOP* simply disables the existing D2 and D3 that might not be used at run time and disconnects them from the sub-components. Moreover, *NEW* adds the new D4 program configurations into the M-Core and generates a new template file for the D4 module implementation. With this one keyword (*NEW*), the users can start writing their own detailed program behavior in the newly created file without worrying about the underlying details of the M-Core and TinyOS. As seen in the figure, a user would be able to handle existing programs and create new ones with simple keywords. Most importantly, the conversion from MCL to the necessary underlying components (i.e., side files in Figure 2) of the M-Core and all the integration process are automatically handled by the MCL.

V. M-Core Functional Evaluation

To evaluate the functionality of the M-Core and the MCL we implemented four security applications:

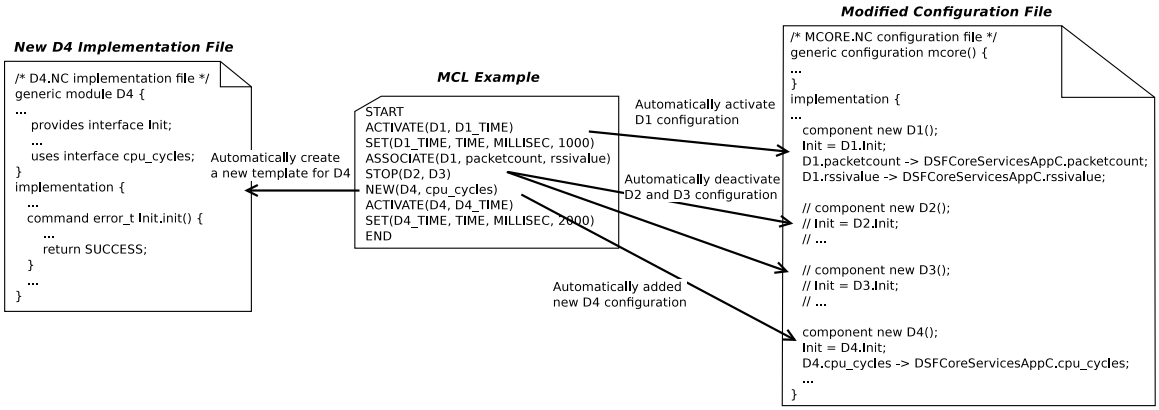

Fig. 2. A realistic example usage of MCL.

- The detection and defense against *Wormhole* attacks.
- The detection and defense against *Sybil* attacks.
- A *secure code dissemination* using Deluge [12].
- A task manager application for sensors that we call *TinyOSTaskManager*.

A. *Wormhole Detection and Defense Application*

The Wormhole attack [13], is one of the potential threats targeting ad hoc and sensor networks. To launch this kind of attack, an adversary connects two distant points in the network using a *tunnel* with low latency, to deceive distant sensors and make them believe they are neighbors. Once the routes are established and the network traffic starts using the wormhole link, the attacker can retrieve sensitive information or disrupt the network communication.

For this experiment, the complete wormhole detection and defense mechanism was implemented using the M-Core architecture. As shown in Figure 3, the scenario consisted of four legitimate nodes and an attacker node placed near node 4. Using the attacker node, the wormhole link (i.e., faster link) is established between node 4 and the cluster head (node 1).

For the wormhole detection mechanism, we use the *hop-count service* provided by the M-Core. During the initialization of the topology, the *hop-count* service is called by the cluster head and all the nodes estimate their corresponding hop-counts from the cluster head. The hop-count service can also be called periodically or whenever there is a change in the topology (e.g., new node addition). At the end of each hop-count service, all nodes update their corresponding *node id* of their preceding node towards the cluster head and cache it.

In the normal communication procedure, each node which initiates a transmission towards the cluster head sets its *current hop* field to its estimated *hop-count* and sends the packet. Each subsequent node decrements this *current hop* value and compares with its own estimated *hop-count*. It forwards the packet only if both the values match. If not, it immediately broadcasts an alert message. On reception of the alert message, the nodes revert back to their cached *preceding node id* for sending packets towards the cluster head.

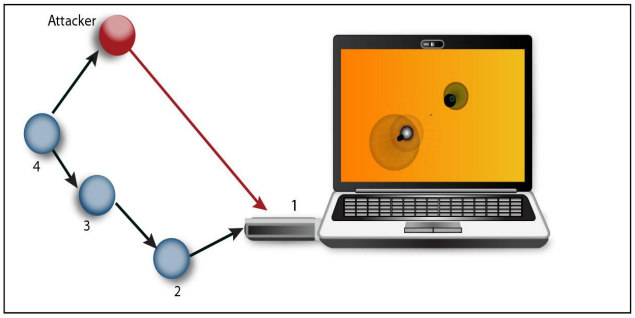

Fig. 3. Wormhole scenario.

Whenever a node receives an advertisement packet for a shorter (or faster) route towards the cluster head, it starts sending packets to the cluster head through this new route. However, it still holds the previous *preceding node id* in its cache. Now, when the attacker node placed near node 4 advertises a shorter route to the cluster head, node 4 sends packets to the cluster head through the attacker. When node 1 receives these packets, it identifies a mismatch between its own *hop-count* and the *current hop* associated with each of these packets. It broadcasts an alert message once the threshold (number of packets arriving with a *hop-count* mismatch) is reached, which directs node 4 to revert back to its cached *preceding node id* and resume transmission. Now, the route through the attacker is avoided and also both ends of the wormhole link are identified using the *hop-count* and *current hop* parameters of the node which initiated the alert message.

Figure 4 shows the time taken for each packet transmitted by node 4 to reach the cluster head. It takes around 14.019 milliseconds in the normal scenario (without attacker). After 8 packets have been transmitted successfully, the attacker is introduced. As shown there is a significant decrease in the time taken (around 6.746 milliseconds) for packets 9-13 to reach the cluster head. This is due the fact that these packets (9-13) now take the faster route through the attacker. The cluster head detects the wormhole attack once the threshold is reached (five

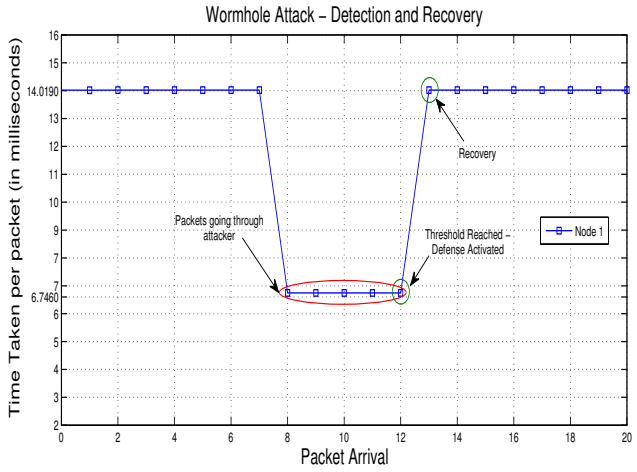

Fig. 4. Wormhole results.

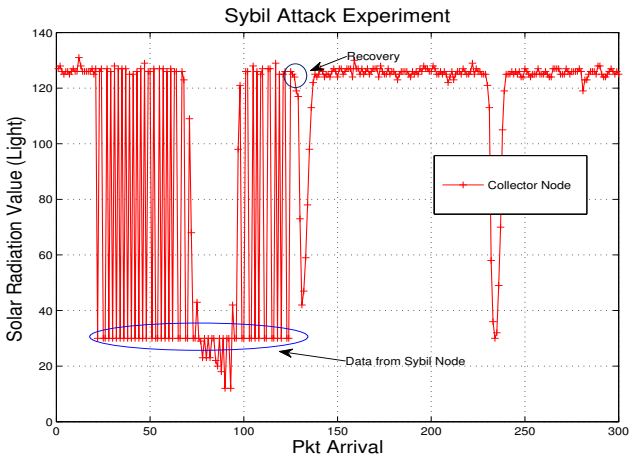

Fig. 5. Sybil Results.

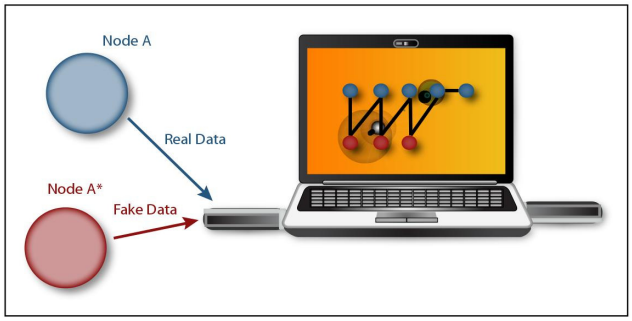

Fig. 6. Sybil Scenario.

packets arriving with hop-count mismatch) and broadcasts the alert message. After recovery, the subsequent packets (14^{th} packet onwards) take the previously cached route which takes around 14.091 milliseconds to reach the cluster head. Note that the contribution of this work is not the technique used to defend against Wormhole attacks, rather the use of the M-Core and the MCL to facilitate the development of an AM.

B. *Sybil Detection and Defense Application*

The Sybil attack consists of one or more malicious nodes that use legitimate nodes identifiers (IDs) to inject fake data into the network. Our attack scenario consists of a malicious node (Sybil node) impersonating other legitimate nodes by broadcasting messages with one or multiple node IDs [14]. In our scenario, we use a RSSI-based approach to detect the Sybil attack. For the Sybil detection the M-Core provides a RSSI table containing average RSSI values for each neighbor. This table is updated every time a packet arrives to the sensor since the packet is passed to the M-Core and the RSSI value is extracted and averaged. We collect at least 10 sample packets from each neighbor to calculate the RSSI average and define an upper and lower threshold for the RSSI. Note that all values and thresholds in our framework are configurable.

The setup of the experiment consists of 2 legitimate sensors: one sampler and one collector. The sampler gets light intensity measurements and transmits the values to the collector. The collector receives and displays the data. We also have 1 Sybil sensor that impersonates the sampler and injects false data into the network which is received by the collector node (Figure 6). When our SA detects a Sybil message, it discard the packet.

Figure 5 shows the results of our experiment including the data fluctuation caused by the injections and the detection and recovery point. As seen in the figure, the M-Core is able to support Sybil scenario as well. While implementing this RSSI approach we notice that there are some false positives due to the unstable nature of the communication channels and RSSI. This is not the optimal defense against Sybil attacks, but we are showing that the M-Core provides useful services for security.

C. *Secure Code Dissemination Application*

Our secure code dissemination application is a secure version of Deluge. Deluge is the de-facto TinyOS network reprogramming tool. Since it does not provide any communication security, we used the RC4 M-Core service to create a secure version of Deluge. The M-Core provides the RC4 stream cipher [11] encryption and decryption algorithm as a service by providing the *encryptI* interface. Since the RC4 ciphers are generated using a symmetric operation, the SA can use the same *encrypt* function call of the *encryptI* interface to encrypt and decrypt the data.

To perform this encryption or decryption operation, our secure deluge passes the secret key, size of secret key, plaintext and size of plaintext as input to the *encrypt* function call. After encrypting the plaintext, the RC4 service will overwrite the input plaintext with the resultant ciphertext data. Hence applications have to make only one function call before sending and after receiving the data to encrypt and decrypt the data stream, respectively. This RC4 encryption is a simple algorithm which is platform independent and can work with all versions of TinyOS. The implementation of the secure code dissemination using the M-Core is presented in Figure 7.

The memory sizes of the secure code dissemination application with and without M-Core are: (1) M-Core: 2743

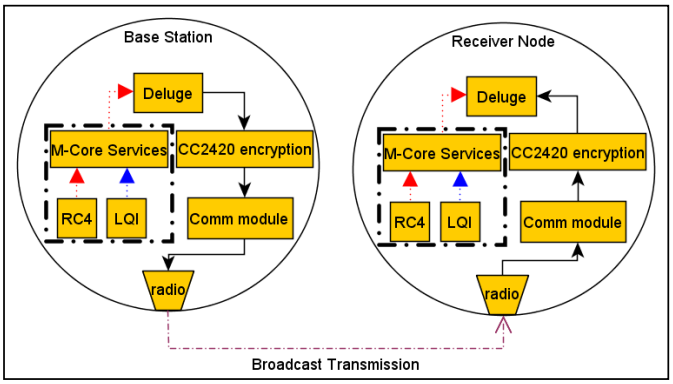

Fig. 7. Secure Code Dissemination Scenario.

bytes (RAM) and 43094 bytes (ROM). (2) Plain: 2743 bytes (RAM) and 42916 bytes (ROM). And the transmission times in milliseconds are: (1) M-Core: 2432 and (2) Plain: 2236. This demonstrates that the M-Core only adds a small overhead to the secure Deluge AM.

D. *TinyOSTaskManager Application*

Using the M-Core services, we also developed a simple task manager-like program that allows one to monitor different activities on the sensor (*TinyOSTaskManager*). The monitor application collects information about the CPU ticks, number of neighbors, number of active M-Core services, packets in/sec, packets out/sec, and total packets received. The status of the different parameters are displayed in a java interface. Figure 8 shows a screen capture of the *TinyOSTaskManager*.

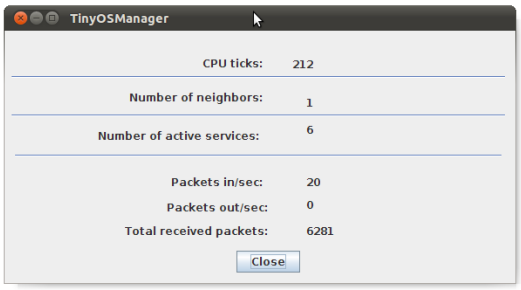

Fig. 8. TinyOSTaskManager.

Besides the security applications presented in this paper, the M-Core architecture and services can be leveraged to develop a complete security system. For instance, in our previous work [15], we used the M-Core to develop *Di-Sec*, a comprehensive security framework for sensor networks that can defend against all kinds of attacks.

Note that the security applications illustrated above are based on existing solutions in the literature and were presented as examples. The novelty (and contribution of this work) lies in the fact that they were quickly developed using the M-Core and MCL.

VI. M-Core and MCL Development Costs

To evaluate the benefits of the M-Core and the MCL, we describe the amount of work (i.e., costs) required to develop new software solutions with and without our framework. We divided the development costs in two categories: *learning* costs and *implementation* costs.

A. *Learning Costs*

For the development of new WSN software programs in TinyOS, one needs to understand the concepts of *modules, configurations, interfaces*, and *wiring*. *Modules* (or *components*) are the basic building blocks of a TinyOS program since they implement the program's executable logic and include some specific behaviors. For one module to be able to call and use the functions provided by another module, we need *configuration* files to map the set of provided functions in one component to a set of required functions in another component (*interfaces*). In TinyOS, connecting two components through an interface is called *wiring*. Whenever a developer wants to create a new program, he/she must define the program requirements and then identify the components that provide the required functionality. Once the components are identified, the developer needs to find the interfaces to communicate with those components and implement their events and learn how to use their commands. Moreover, when some functionalities are not implemented in the required components, the developer has to implement it himself.

Given that the M-Core provides all the required information (services) for the development of new security applications within a single component, a developer only needs to add an M-Core component in their programs and call any service directly through the M-Core. These services provided by the M-Core take care of parsing all the messages and sensing values and abstracts most of the system level configurations from the application developers. Therefore, the amount of work required to create new application is much smaller when using the M-Core and it aids the security application developers to write efficient defense modules for various security attacks and in providing security fixes for the deployed WSN applications, without worrying about the system configuration.

B. *Implementation Costs*

The M-Core provides a simple architecture that eases the design of new software solutions. As shown in Figure 1, the architecture of the M-Core allows the developer to create multiple upper layer programs running individually or in parallel, and it is also possible to have a stack of layers using the M-Core services. To compare the implementation costs, we discuss the amount of work to create a simple SA that maintains a table with the node neighbors' RSSI information. The evaluation is based on the number of lines of code to write and the number of files to modify for a basic code setup of a new program. This is taken as an evaluation metric because security application developers often need a rapid application development platform to aid them in providing security fixes for an on going security attack in WSN. Also,

TABLE II
IMPLEMENTATION COMPARISON FROM THE DEVELOPER'S PERSPECTIVE

	No M-Core	**M-Core only**	**M-Core & MCL**
Lines of code	20 for new component	20 for new component	5 for new program
	8 for new configuration	8 for new configuration	
	4 for component wiring	4 for M-Core wiring	
	8 for every additional event per interface	8 for every additional event per interface	
	55 for RSSI extraction and tables	RSSI extraction and tables are provided as M-Core services	RSSI extraction and tables are provided as M-Core services
Total	**91**	**36**	**5**
Files to modify	2 modules	1 New module	1 MCL program
	1 configuration	1 configuration	1 module
	1 interface	1 M-Core	
	1 headers		
Total	**5**	**3**	**2**

most of the existing middleware approaches, do not emphasis security application development and this metric evaluates the use of M-Core as a rapid application development platform for a security application developer.

As seen in Table II, using the M-Core and MCL only takes 5 lines of code to develop our simple program compared to 91 lines of code without the M-Core. For our evaluation, we used a simple scenario, but savings are amplified when developing more complex SAs.

VII. CONCLUSION AND FUTURE WORK

In this work we introduced the Monitoring Core (M-Core). The M-Core is a lightweight and extensible software layer that allows for the monitoring of both internal and external activities simultaneously to provide information in the form of services. The service-oriented architecture of the M-Core facilitates and expedites the development of security applications (SAs) for WSNs. Moreover, a new domain specific language, the M-Core Control Language (MCL) has been introduced to easily use M-Core. Both the M-Core and the MCL along with five example SAs have been implemented and tested on real sensors. As an overall contribution, this work realized an architecture that can be leveraged by developers to expedite the development of security application for sensor networks. In our future work, we will increase the number of M-Core services and provide a quantitative analysis and better evaluation of the MCL.

ACKNOWLEDGMENT

This work was partly supported by NSF Grant No. CNS-1052769

REFERENCES

[1] "Planetary Skin," http://www.planetaryskin.org/home.
[2] J. Bort, "10 technologies that will change the world in the next 10 years," http://www.networkworld.com/news/2011/071511-cisco-futurist.html, 2011.
[3] "The Internet of Things," http://www.theinternetofthings.eu/.
[4] G. Ormazabal, S. Nagpal, E. Yardeni, and H. Schulzrinne, "Principles, systems and applications of IP telecommunications. services and security for next generation networks," H. Schulzrinne, R. State, and S. Niccolini, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Secure SIP: A Scalable Prevention Mechanism for DoS Attacks on SIP Based VoIP Systems, pp. 107–132.
[5] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Comput. Surv.*, vol. 43, no. 3, pp. 19:1–19:51, Apr. 2011.
[6] M. Kushwaha, I. Amundson, X. Koutsoukos, S. Neema, and J. Sztipanovits, "Oasis: A programming framework for service-oriented sensor networks," in *the 2nd International Conference on Communication Systems Software and Middleware (COMSWARE)*, jan. 2007, pp. 1 – 8.
[7] J. M. Prinsloo, C. L. Schulz, D. G. Kourie, W. H. M. Theunissen, T. Strauss, R. Van Den Heever, and S. Grobbelaar, "A service oriented architecture for wireless sensor and actor network applications," in *the Proceedings of the annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries (SAICSIT)*, 2006, pp. 145–154.
[8] C.-L. Fok, R. Gruia-Catalin, and C. Lu, "Agilla: A mobile agent middleware for self-adaptive wireless sensor networks," *in the ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 3, pp. 16:1–16:26, Jul. 2009.
[9] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *in the ACM Transactions on Database Systems (TODS)*, vol. 30, no. 1, pp. 122–173, Mar. 2005.
[10] TinyOS homepage, http://www.tinyos.net/.
[11] B. A. Forouzan, *Cryptography & Network Security (1st edition)*. McGraw-Hill, 2007.
[12] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," *in the Proceedings of the 2nd International conference on Embedded Networked Sensor Systems*, pp. 81–84, 2004.
[13] Y.-C. Hu, A. Perrig, and D. Johnson, "Wormhole attacks in wireless networks," *in the IEEE Journal on Selected Areas in Communications*, vol. 24, no. 2, pp. 370 – 380, feb. 2006.
[14] M. Demirbas and Y. Song, "An RSSI-based scheme for sybil attack detection in wireless sensor networks," in *the Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks (WOWMOM)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 564–570.
[15] M. Valero, S. S. Jung, A. S. Uluagac, Y. Li, and R. Beyah, "Di-Sec: A Distributed Security Framework for Heterogeneous Wireless Sensor Networks." in *In the Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, Orlando, FL, USA, Mar. 2012.