Spring 2016

Improving the security of wireless sensor networks

Mauricio Tellez Nava
James Madison University

Improving The Security Of Wireless Sensor Networks

Mauricio Tellez Nava

A thesis submitted to the Graduate Faculty of

JAMES MADISON UNIVERSITY

In

Partial Fulfillment of the Requirements

for the degree of

Master of Science

Department of Computer Science

May 2016

Dedication

This work is dedicated to my parents Edgar and Miriam. Without your sacrifice of moving away from Bolivia, I would not be standing where I am as a professional. This thesis and all my accomplishments shows that your sacrifice of leaving behind your family, friends and culture was not a waste. I will forever be thankful for you guys.

I love you both.

Mau

Acknowledgments

First, I would like to thank my advisors, Dr. Samy El-Tawab and Dr. Mohammed Heydari, for their support throughout the completion of my thesis. Second, I would like to thank Dr. Florian Buchholz and Dr. Xunhua Wang for serving on my thesis committee. I have been fortunate to have classes with each of them over the last 4 years and it has been a very valuable experience. Third, I would like to thank my family, Edgar, Miriam, Marilia, and Mayra for their support throughout my 6 years at James Madison University.

Finally, I would like to thank my girlfriend, Rachel. Without her love, support, and making sure I took breaks to eat, this year-long journey would not have been possible.

Table of Contents

List of Tables

List of Figures

Abstract

With the rapid technological advancements of sensors, Wireless Sensor Networks (WSNs) have become the main technology for the Internet of Things (IoT). We investigated the security of WSNs in an environmental monitoring system with the goal to improve the overall security. We implemented a Secure Temperature Monitoring System (STMS), which served as our investigational environment. Our results revealed a security flaw found in the bootstrap loader (BSL) password used to protect firmware in the MSP430 MCU. We demonstrated how the BSL password could be brute forced in a matter of days. Furthermore, we illustrate how an attacker can reverse engineer firmware and obtain copies of cryptographic keys. We contributed a solution to improve the BSL password and better protect firmware found in the MSP430 chips. The Secure-BSL software we contributed allows the randomization of the BSL password. Our solution increases the brute force time to decades. The impractical brute force time improves the security of firmware and prevents future reverse engineering tactics. In addition, our Secure-BSL software supports two-factor authentication that allows developers to specify a user-defined passphrase to further protect the MSP430 MCU. Our research serves as proof that any security implemented in a WSN environment is broken if an attacker has access to firmware found in sensor devices.

Chapter 1

Introduction

The Internet of Things (IoT) has become a popular subject in the industry and will soon reach the popularity level of smartphones. With the rapid technological advancements of sensors, Wireless Sensor Networks (WSNs) have become the main technology for IoT [5]. WSNs are composed of a large number of sensors that are physically small, communicate wirelessly among each other, and are deployed without prior knowledge of the network topology [1]. The deployment environments could be over small or large geographical areas in locations that are either public or hostile. Typically, the environments require little human interaction and go unattended for months or even years. The three key characteristics of sensor networks are to continuously monitor surroundings, trigger any alerts based on circumstances occurring, and provision of information on demand [2]. Sensor applications can be broken down into two categories: tracking or monitoring. Tracking examples include the tracking of enemies in a military environment or tracking inventory in a commercial environment [3]. Monitoring examples include monitoring room temperatures in a building or home monitoring used in burglar alarm systems.

The main difference between WSNs and wireless networks is the characteristics of the devices that are part of the network. In wireless networks, most of the connected devices are computers, laptops, tablets, or smart phones. In a WSN, the connected devices are solely sensors that use a machine-to-machine paradigm. Unlike the devices found in a wireless network, sensor devices are very limited in processing power, battery life, communication bandwidth, and memory [4]. In particular, battery energy is the main resource to conserve in a WSN because of the domino effect–it has on other resources. For example, if the battery power decreases then the computational power and communication bandwidth decreases with an overall effect on the sensor. With that being said, current protocols used in wireless networks are meant for devices that support expensive computations and not suitable for sensor devices. WSN research has shown a significant shift to proposing new protocols and algorithms that take into consideration the limited battery life and computation power of sensor devices. A popular research topic has been the security of a WSN environment.

According to Gartner, there will be nearly 26 billion IoT devices by 2020 [5]. As we witness more IoT devices connecting to the Internet, the overall security of systems is more vulnerable. As discussed by OFlaherty, sensor devices will become the favored point of entry for compromising other, bigger targets [5]. Even worst, botnets will find millions of new ”recruits” in the form of zombie appliances or ”sensor devices” [6]. Securing WSNs at an early stage will be a critical role in protecting the future of the Internet. Using todays cryptographic protocols, such as RSA and AES, raises two problems in a WSN environment. The first problem is the fact that these protocols are computationally expensive. The second problem is the fact that these protocols do not protect the nature of sensor devices going unattended for years. A shift in looking at security with different points of view has been seen in WSNs research.

Encryption and authentication are vital components in WSNs. In the past, many researchers have proposed efficient cipher protocols with one goal in mind: to reduce computational power [7][8][9][4][10][11][12][1][13][14][15][16][17][18]. Although Skipjack has been a popular cipher for WSNs, latest research has shifted towards using hardware implementation encryption. The Secure Temperature Monitoring System (STMS) that we have implemented makes use of the AES inline encryption built in to the CC2420 Radio Frequency chips. The STMS is used as our WSN experimental environment to evaluate the overall security of WSNs. In Chapter 3, we discuss the design, the implementation, as well as the test results for our STMS.

Other than efficient ciphers research, there has also been numerous proposals in implementing Public Key Cryptography on WSNs. Using RSA as a key management crypto system for WSNs is possible, however, not practical when it comes to saving battery life on constraint sensor devices. Research has shown a shift towards using ECC as the default key management protocol for implementing PKC in WSNs [19][20][21][22]. Experimental results show that implementing PKC on WSNs is possible and more importantly maintains low computational power requirements. In Chapter 2.3, we highlight survey results of various key management protocols and algorithms proposed for WSNs. Our goal is to provide background information on key management protocols and demonstrate how we have broken all these protocols. In Chapter 5, we demonstrate how we reverse engineered firmware found in one of our STMS nodes and were able to obtain a copy of the cryptographic key used in our secure WSN environment.

Similar to the key management protocols proposed, preventing node capture attacks has been a popular research topic for WSNs. Node capture attacks is a well respected research field within WSN due to the fact that sensor devices are deployed in unattended environments with easy physical access. Unlike computers or servers that are either physically locked or encrypted, sensor devices lack these features. In Chapter 2.4, we highlight various algorithms and schemes proposed to mitigate information leakage from physically capturing nodes. All of the proposed schemes have one thing in common: they all assume that a node has been captured but do not highlight the consequence of capturing a node. The most popular schemes make the assumption that N number of nodes must be compromised before breaking the security of the entire WSN [23][24][25].

Not a lot of research has been done in evaluating node tampering methodologies to extract firmware found in sensor nodes. Most of the research out there has overlooked the consequences of tampering captured nodes. The physical security of sensor nodes is a vital component of the overall security of WSNs. For instance, recent research has shown that the MSP430 MCU family chips are vulnerable to brute force attacks. The bootstrap loader passwords used to protect unwanted access to the micro-controller is weak. The actual password is stored at the same address location as the Interrupt Vector Table (IVT) for the MSP430 chips [26]. Storing the IVT and the password at the same address location suggests that the password is identical to the IVT. Using the IVT as the password makes the physical security of MSP430 chips weak and research has shown that the 256-bit password can be lowered to a 40-bit password [27]. However, brute forcing a 40-bit key space in the best case scenario is impractical because it would take 32 years [28].

In this thesis, we reduced the brute force time from years to a matter of days. Instead of reducing bits of a passwords, we used a different approach of using a password prediction technique. In Chapter 4, we highlight our approach of analyzing password samples and presenting the password patterns found between applications. In Chapter 6, we discuss our contributions of implementing the Secure-BSL software to improve the password used to protect access to the MSP430 through randomization. Our results and evaluations of our improved Secure-BSL password generator are highlighted in Chapter 7. We hope that this thesis can serve as motivation for further research in the physical security of sensors.

Chapter 2

Related Work

In this chapter, we provide related work to support different components of our thesis as well as highlight the overall history of WSNs research. We highlight various sensor applications and provide examples of related temperature monitoring systems. We discuss in detail the two security problems with WSNs that have been popular topics of research. In addition, we present various research in protecting WSNs from node capture attacks. Lastly, we end our chapter with a discussion on the not so popular topic of node tampering attacks.

2.1 Applications

There are numerous WSN applications, but ultimately they all fall into two categories: monitoring and tracking. Monitoring examples include temperature levels, humidity levels, ultra-violet levels, pressure levels, noise levels, etc. Tracking examples include movements of objects, directions of objects, traveling speeds, absence or presence of objects. Furthermore, applications vary depending on the environment where the sensors are deployed. A WSN can be deployed in the following environments: military, environmental, health, public and personal. In a military environment sensors can be an integral component for battlefield surveillance or reconnaissance of opposing forces and terrain [3]. In environmental applications sensors can play a critical role for example in detecting forest fires [29] or monitoring micro climates in crop fields [30]. In a health environment sensors can be used for example to monitor patients' health by implanting a cubic-millimeter computer into a patient's body [31]. In a commercial environment sensors can used for example to detect car thefts [32] or management of inventory tracking [33]. In a personal environment sensors can be integrated on every day appliances [34] and ultimately create a smart home environment [35]. It is up to our imagination on where WSNs can be integrated.

Most of the applications, including the former examples, are deployed in environments that are resource expensive to simulate. Our research concentrates in the physical security of sensor devices and not the actual sensor applications. For example, simulating a Wildfire Monitoring System [36] requires the purchase of expensive equipment that is unnecessary for our research purpose. We instead concentrate in researching sensor applications that are inexpensive to simulate so we don't have to buy expensive hardware. These inexpensive systems have similar security concerns as the more complex systems. Without a loss of generality, we chose a temperature monitoring system application of sensor networks for this research. In fact, we found that temperature monitoring systems using WSN is a popular field of research.

The researchers at Sirindhorn International Institute of Technology (SIIT) implemented a remote temperature monitoring system that collected classroom temperature levels and provided necessary data for the energy consumption management of air conditioning units [38]. The SIIT implementation made use of the Zigbee protocol to implement their network, consisting of a Coordinator, End-Devices and a web server for a GUI. A similar system was implemented at Berkley to monitor room temperatures using the Coordinator and End-Devices approach [39]. The difference between the two is the implementation software used; SIIT used XBee, where as Berkley used Code::Blocks software. Another temperature monitoring system was implemented using MATLAB software, which also made use of the Zigbee protocol [40]. All three systems suggest that building on top of the Zigbee protocol is a popular choice. The Zigbee protocol serves at the network layer and is built on top of the IEEE 802.15.4 protocol. Besides Zigbee, another commonality found between the monitoring systems is the use of unencrypted and unauthenticated communications. Although the Zigbee protocol provides security services, to our knowledge there is no open source implementation of the security features. Furthermore, there is a limited Zigbee open source code available but most Zigbee software is proprietary [41]. Keep in mind that Zigbee is built on top of the IEEE 802.15.4 layer; therefore, if the physical layer is broken then all upper layers, including Zigbee, are no longer secure. We give details about the physical security of motes in Section 2.5. Please note that our temperature monitoring system builds on top of the IEEE 802.15.4 layer.

2.2 Cipher Protocols

One of the main problems with securing WSNs is the limited resources supported by the sensor devices. In particular, resources such as battery power, computational power and memory size are limited. Almost all sensor devices are deployed in open environments and will go unattended for months or even years. Therefore, common encryption systems used by computers or smart phones are not practical for sensor devices. A lot of research has been devoted into proposing cipher protocols that use less computational power, thus increasing the battery lifetime of sensor devices. We highlight popular ciphers that have been proposed for WSNs with the goal to provide information why we chose the in-line AES cipher for our WSN environment.

Previous research has shown a connection between ciphers and the type of micro-controller used. For example, the MD5, SHA-1, IDEA, RC4, and RC5 ciphers were evaluated using 8-bit, 16-bit, and 32-bit micro-controllers. The results showed that 32-bit micro-controllers out performed 8-bit and 16-bit roughly by a factor of two because of the 64-bit block sizes used in these ciphers [4]. On the other hand, 8-bit micro-controllers out perform 32-bit micro-controllers when evaluating the performance of stream ciphers such as LEX [9]. The type of micro-controller and security policies play an important role in selecting the appropriate cipher. Law et al. [1] provides a good verdict of how to choose a suitable cipher and recommends Skipjack as the best cipher choice for WSN. Using a MSP430 emulator, experimental results showed that Skipjack is the most energy efficient compared to RC5, RC6, Rijndael, Twofish, MISTY1, KASUMI, and Camellia [1].

In fact, TinySec uses Skipjack to provide the first fully-implemented link layer security supporting two modes of operation: encryption only (CBC) or encryption and authentication (CBC-MAC) [11]. The research presented by Karlof et al. has been one of the most respected and many researchers have proposed improved versions of TinySec. For example, MiniSec was proposed as an improvement to reduce power consumption of TinySec by using OCB cipher modes to encrypt and authenticate simultaneously [14]. As another example, FlexiSec was proposed to improve the security of TinySec by implementing Bloom-filters to prevent replay attacks [10]. Other protocols, such as the LLSP, have been proposed as a link layer security; however, they're just replicas of TinySec with a few modifications [13]. Besides Skipjack, another cipher family that has been a respected choice for WSNs is the RC4 and RC5 ciphers. The first ever WSN security proposal was the SPINS protocol, which used the RC5 cipher to implement encryption and broadcast authentication [15]. Recent research has shown a shift towards stream ciphers by improving the key generation process of RC4 [12]. All of the proposed ciphers are software implementations that did not take into consideration the efficiency of using hardware implementation.

Although AES is one of the most computational expensive ciphers, it has still been used in at least three research proposals [16] [42] [17]. However, all three proposals made use of the AES software implementation. The micro-controller used in [16] and [42] was the MSP430. The CC2420 RF chips have an inline AES-128 hardware implementation that supports three different modes of operation: encryption only (CTR), authentication only (CBC-MAC) or encryption and authentication (CCM) [43]. Running the hardware implementation version of AES on the CC2420 is significantly more efficient than running the software implementation version of AES on the MSP430. Andersen and Tranberg prove that the CC2420 radio inline CCM security mechanism is 42 times faster and uses

4.5 times less energy than the similar software implementation running on the MSP430 MCU [7]. Furthermore, Buesching and Wolf demonstrated that the throughput of the AES-128 CBC mode implementations to be 1,085kb/s for hardware version and 28kb/s for software version [8]. Tranberg [18] evaluated that the CC2420 security features out perform the TinySec and MiniSec protocols both in security level and efficiency. We make use of the CC2420 inline security feature because of the reasons presented, as well as the fact that we used TelosB motes with our WSN environment.

2.3 Key Management Protocols

In todays Internet, RSA is the most commonly used crypto system to manage the Public Key Infrastructure (PKI). The computational overhead of RSA is significant and unreasonable to implement in WSNs. Lots of research has been done to find algorithms that significantly reduce the resource overhead for key management technologies. We highlight popular key management algorithms that have been proposed and show the major security flaws found on each, when a sensor device has been physically captured.

One of the first key management algorithms was proposed by Jolly et al. Their key management approach pre-deploys a shared symmetric key used for future key exchange and key renewal secure communication [44]. The major security flaw found with the algorithm proposed by Jolly is not knowing when the shared symmetric key has been compromised. An attacker can impersonate the central node by obtaining a copy of the share symmetric key through reverse engineering techniques. Watro el at. proposed TinyPK which makes use of RSA to prevent impersonation attacks [45]. TinyPK is inefficient and does not handle the problem of revoking compromised private keys. A compromised sensor node can be used to maliciously authenticate to the WSN by obtaining a copy of the private keys. The following two examples were proposed back in 2003 when WSNs were first introduced.

Between 2006 and 2015, there was a shift towards using Elliptic Curve Cryptography (ECC) to implement PKI in WSNs. For example, one of the first proposed technologies that made use of ECC was NanoECC. NanoECC was implemented using TinyOS and evaluated on the Mica2 and TelosB sensor nodes. The evaluation results showed that the energy consumption to compute point multiplication in the MSP430-F1611 MCU was 7.95mJ [19]. Although NanoECC is more efficient than TinyPK, the security flaw of not revoking compromised private keys remained in the NanoECC. The researchers at Harvard University also implemented ECC for PKI in WSNs. Their EccM 2.0 PKI software is open source for future research purposes [20]. The EccM 2.0 was tested using Mica2 motes, and compared to NanoEcc , the EccM 2.0 results show higher energy consumptions because bigger key sizes were evaluated. Even if the key size improved security in the EccM 2.0, it is still vulnerable to node capture attacks. The foundation of most key management research has been TinyECC, which also uses ECC for PKI in WSNs. Compared to NanoECC [19] and EccM 2.0 [20], TinyECC's main feature is the flexibility to configure ECC to best fit specific application needs [21]. Similar to the previous two technologies, TinyECC suffers from node capture attacks. Researchers at William and Mary proposed their own ECC implementation which still suffers from the same problem of not securing the private keys [22].

Other proposed algorithms suffer from the same security flaw of not protecting the private keys from being compromised. For example, Nilsoon and Roosta proposed a key management scheme that uses three pre-installed keys: one authentication key, one encryption key and one network key [46]. Du et al. also proposed a scheme that pre-loads a secret key for future key exchange communication [47]. Similarly, Alfandi et al. proposed that each sensor node on a WSN has a certificate signed by a Certificate Authority [48]. Mansour et al. attempted to improve the security of the private keys by using a pre-shared key to renew the private keys [42]. All three proposals suffer from node capture attacks where the private keys can be compromised. Zhang et al. highlighted various pre-deployment schemes, which all suffer from node capture attacks [49]. All of the highlighted technologies have one thing in common: they all do not consider the consequences of capturing a sensor node, reverse engineering data found in memory, and obtaining secret information such as private keys. Without keeping the private keys a secret, all of these proposed solutions are broken.

2.4 Node Capture

Unlike the technologies discussed in Section 2.3, there has been other research proposals that handle cases when a sensor node has been captured. The technologies proposed can be divided into four categories. The first category is key distribution technologies that require large number of nodes to be captured in order to gain any significant results. The second category is key revocation technologies that prevent the compromise of future encrypted information. The third category is intrusion detection systems (IDS) that detect capture nodes in a WSN. The fourth category is intrusion prevention systems (IPS) that prevent the disclosure of sensitive WSN information.

The first category is key distribution technologies that require the capture of large number of nodes to attain any significant gain. Eschenauer and Gligor proposed a pair-wise pre-distribution of keys to allow private sharing of keys between every sensor nodes and avoids large scale WSN compromise [23]. The proposed scheme does not protect the security of small WSNs and only protects distributed sensor networks. Chan et al. proposed an approach to improve the security under small scale attacks at the cost of greater vulnerability to large scale attacks. The approach uses a scheme where cryptographic rekeying increases the resilience of the network against node captures [24]. Xiao et al. key management survey results showed that pairwise key distribution schemes are the best for network resilience by enforcing large fractions of the network to be compromised to attain any significant gain [25]. All of the proposed key distribution algorithms accept a small portion of the WSN to be compromised. If reverse engineering tactics are used to obtain copies of the pair-wised keys then the whole WSN can eventually be compromised.

The second category is key revocation technologies that prevent the compromise of future encrypted information by revoking the keys shared with the compromised sensor node. Wang et al. proposed the KeyRev technology that makes use of a centralized key revocation technique to prevent future information leakage. The functionality of the KeyRev technology makes use of session keys to prevent compromised sensors from obtain new session keys [50]. However, the problem of the KeyRev technology is that it does not know how to detect if a sensor node has been compromised. Chattopadhyay and Turuk proposed a similar approach; however, it is a decentralize approach where each individual node is responsible for revoking keys shared between compromised sensor nodes. The proposed technology made use of a voting scheme where sensor nodes vote on each other whether a sensor is suspected to be a compromised node [51]. Both proposed technologies made the assumptions of having an IDS built in to one of the nodes on the WSN. The key revocation technologies does protect a WSN from node capture attacks; however, making the assumption that a IDS is in place is not reliable.

The third category is intrusion detection systems that detect capture nodes in a WSN. An implemented example of an IDS was presented by Krontiris et al. where an IDS was deployed to detect sinkhole attacks. The rules implemented in their IDS use the overhead route update packet to check for the sender field and produce an alert if the sender ID is not one of the sensor IDs found in the WSN [52]. Although the IDS did not directly detect node capture attacks, it can detect if the capture node has been modified to use a different sensor ID. Another example of an IDS that is more applicable to our scenario of node capture attacks was a proposed scheme by Sun et al. The IDS scheme proposed made use of a pattern classification problem, in which classifiers were designed to classify and observed activities as normal or intrusive [53]. The proposed scheme was not implemented but proposed on paper. Not much research has been published on implementing IDS for WSN because of the characteristics of the technology compared to wireless networks.

The fourth category is intrusion prevention systems that prevent the use of capture nodes to disclose sensitive information about a WSN. Out of all the categories discussed in this section, the IPS category is the most promising in securing WSN from node capture attacks. In particular, Priya and Sathyanarayana proposed a unique method of protecting the security of WSN from node capture attacks by using a threshold secret sharing scheme. The threshold secret sharing scheme divides the master key into several sub keys and as long as the attacker captures fewer than the threshold, he/she cannot reconstruct the master key [54]. The threshold can be application specific and protects the master secret key from being disclosed. Another prevention technology proposed by Soroush et al. is to delete the master key after the key establishment has been accomplished [55]. The proposed scheme makes the assumption that the master key eventually gets deleted from memory in order to make the scheme more secure against node capture attacks [55]. We proved later that this assumption is broken because an attacker can obtain the master key before it is deleted.

2.5 Node Tampering

Node capture attacks are often discussed as a potential vulnerability in WSNs; however, only few papers have been published to prove the feasibility of such attacks [56]. Proposed papers date back to 1997 with the most latest one written in 2008 that highlight node tampering attacks. Skorobogatov describes in detail various node tampering attacks and has classified the techniques into three categories: invasive, semi-invasive, and non-invasive attacks [57]. Invasive attacks require access to the micro-controllers internals and typically make use of expensive equipment used in manufacturing and testing. Semi-Invasive attacks require access to the micro-controllers internals and typically make use of cheaper equipment and less time than invasive attacks. Non-invasive attacks are the easiest that require access to the micro-controllers but does not require additional equipment.

The invasive node capture attacks are more practical in a laboratory environment instead of in-field type of attacks. Anderson and Kuhn discussed an invasive type of attack that used differential fault analysis, where a micro-controller was exposed to low level of ionising radiation to introduce one-bit errors [58]. Other examples from Anderson include the use of chip rewriting attack by specifically targeting gates to overwrite memory locations [58]. A more practical invasive attack was demonstrated by Goodspeed. The invasive attack consisted of implementing the BSLCracker 3.0 hardware that uses a voltage glitching attacks to skip over a ”jz $+0” loop and gain unauthorized entry to the BSL [59]. The attack only applies to two versions of the MSP430: the F1101 and the

F4618 chips.

The semi-invasive node capture attacks are more practical for in-field types of attacks. For example, Deng et al. discussed an attack on the MICA motes where a computer, a programming board, and a debugging interface (JTAG) device are all that is need it to physically access the internals of a micro-controller [56]. However, we must keep in mind that the attack only works if the JTAG fuse has not been blown. If the JTAG fuse is blown then access through JTAG is no longer supported and is irreversible [60]. Another example of a semi-invasive attack was demonstrated by Skorobogatov and Anderson by making use of inexpensive laser equipment. They demonstrated that illumination to a target transistor causes it to conduct; thereby inducing a transient fault [61]. Both examples require the use of additional hardware that can be inconvenient when attacks are performed in-field.

The non-invasive node capture attacks are the least expensive and are the ideal choice for in-field types of attacks. A great example of a non-invasive attack was discussed by Becher. The attack exploited the BSL password used to protect MSP430 micro-controllers. Becher demonstrated that the 256-bit BSL password was vulnerable to brute force attacks by reducing the key space to 40 bits [27]. Since the BSL password is stored at the address space as the IVT, the password is identical to the IVT. The 256-bit password was first reduced to 240-bit by considering code addresses to be aligned on a 16-bit word boundary, thus the least significant bit of every interrupt vector is 0. The 240-bit was reduced to 225-bit by considering that the reset interrupt or power-up interrupt was always fixed to the start address of main memory. The 225-bit was reduced to 60-bit by considering that in the worst case scenario the unused interrupts point to the same fixed address. The 60-bit was further reduced to 40-bit by considering that code is placed by the compiler in a contiguous area of memory starting at the lowest flash memory address, thus leaving with a key space of fewer possible addresses. Becher suggested that the brute force time for 40-bit key space would be 128 years. Goodspeed was able to reduce the 128 years to 32 years by changing the baud rate of the MSP430 to the highest supported baud rate of 38400 [28]. Although brute force can be reduced by capturing more nodes, it is not practical to brute force in a matter of years.

This thesis uses a different approach from Becher and Goodspeed. We analyzed the IVT values instead of reducing the number of bits in the key space. We analyzed the passwords to find any patterns that are persistent between sensor applications and use the results to predict future passwords. Our results show that we have lowered the brute force time to a matter of days rather than years.

Chapter 3

Secure Temperature Monitoring System In A WSN Enviroment

In Chapter 2, we study various types of WSN environments and investigated different applications such as monitoring and tracking applications. The number of applications that can be supported by WSNs are numerous; however, most of the systems require an environment that is resource expensive. Simulating a WSN environment can be very expensive [37][30][32][33][36][29], thus the cost outweighs the sole purpose of our research. With that being said, in this chapter we implement the Secure Temperature Monitoring System which we refer to it as STMS. STMS has similar features to the three different temperature monitoring systems implemented by Boosawat et al.[38], Risteska et al. [40] and Mon et al.[39]. However, to our knowledge our STMS is the only temperature monitoring system that encrypts the communication in a WSN. Our STMS is able to securely monitor room temperatures in real time in a university environment such as the Computer Science department at James Madison University. Therefore, STMS is the proper choice to demonstrate the security flaw found in the MSP430 family chips. Our chapter discusses the design of our STMS, implementation, the modes of operation, system set up, system testing, and network monitoring using Wireshark.

3.1 Design

Figure 3.2 shows the WSN topology used with our STMS consisting of a computer used to display temperature levels, a Coordinator used as a base station, a PPPSniffer used to monitor the WSN and three End-Devices used to collect temperature levels. STMS makes use of a star topology where the Coordinator serves as the central node and the three End-Devices serve as the end nodes. The computer serves as an extension of the Coordinator to provide the user with a visual representation of the temperature data collected by each End-Device. Furthermore, the PPPSniffer node is also connected to the computer in order to provide a visual representation of the packets captured by the PPPSniffer.

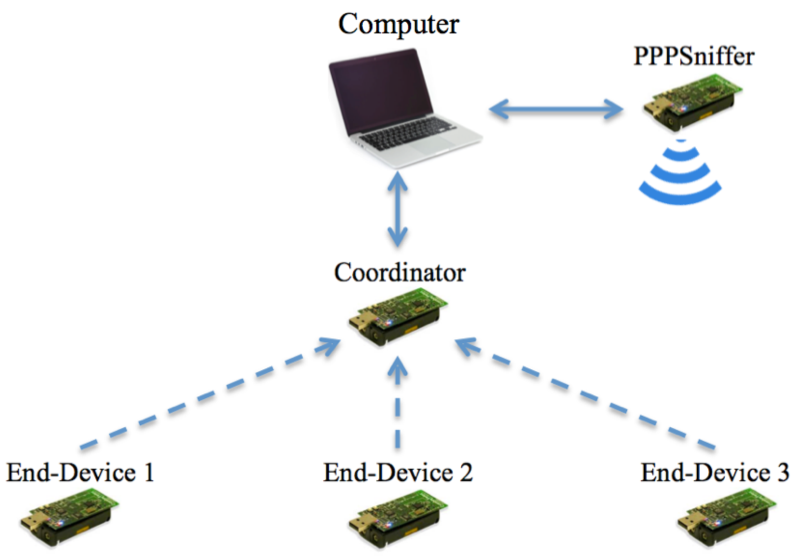


Figure 3.1: The network topology of the STMS. The solid lines represent a physical connection and the dashed lines represent a wireless connection.

- *Computer:*

  The computer has a main role of displaying temperature data in real time. The computer displays temperature data that was sent by the Coordinator through one of the computer's USB serial ports. In addition, the computer also displays capture packets sent by the PPPSniffer through one of the computer's USB serial ports using Wireshark.

- *Coordinator:*

  The Coordinator has three main roles: listening for incoming packets, decrypting packets, and forwarding packets. The Coordinator always listens for any incoming packets sent by the End-Devices. The packets received by the Coordinator are encrypted temperature readings collected by the End-Devices. The Coordinator will decrypt the packets using a network-wide shared key and forward the decrypted data to the laptop. The sole purpose of forwarding the packets to the laptop is to provide the user with a screen to display the temperature readings.

- *End-Device:*

  The three End-Devices roles are: collecting temperature readings, encrypting the temperature data, and sending the temperature data. The three End-Devices are to be placed in three different rooms to periodically collect the current temperature of each room. The End-Device uses a built-in sensor to collect real time temperature readings of each room

[62]. The temperature readings are encrypted by the End-Devices using a network-wide shared key and the encrypted data is sent to the Coordinator.

- *PPPSniffer:*

  The PPPSniffer has two main roles: capturing packets and forwarding the packets. The PPPSniffer is responsible for sniffing all traffic within the STMS. The PPPSniffer uses Point-to-Point Protocol to forward the capture packets to the computer and the computer will display the received packets using Wireshark. The PPPSniffer allows to monitor the network traffic within our STMS and check for secure communication between motes.

3.2 Implementation

The two tasks to set up our WSN environment was to implement our STMS and connect/test the necessary equipment. The hardware subsection provides the specifications of the TelosB motes used within our STMS. The software subsection will provide details of the IDE used and the implementation of our STMS.

3.2.1 TelosB Platform

The three most popular motes for research purposes are: the MICAz motes, IRIS motes, and TelosB motes [2]. The listed three motes are manufactured by MEMSIC, a sensing solutions company [63]. Although all three motes provide similar services, the best choice for our STMS is the TelosB mote. The TelosB comes with an integrated Temperature, Light and Humidity sensor [62], where as the MICAz and IRIS have expansion connecters for Light and Temperature sensors. The TelosB mote did not require us to buy additional sensor hardware since it already comes with the three built in sensors. Additionally, most of the applications in the TinyOS open source project have been tested using TelosB motes and has a great reputation of being popular mote of research [64]. For someone new to the WSN community the use of the TelosB motes eases the learning phase of the thesis project. Therefore, we purchased 5 TelosB motes from MEMSIC at a price of $140 for each mote. Figure 3.2 shows a front and back view of the TelosB mote [65].

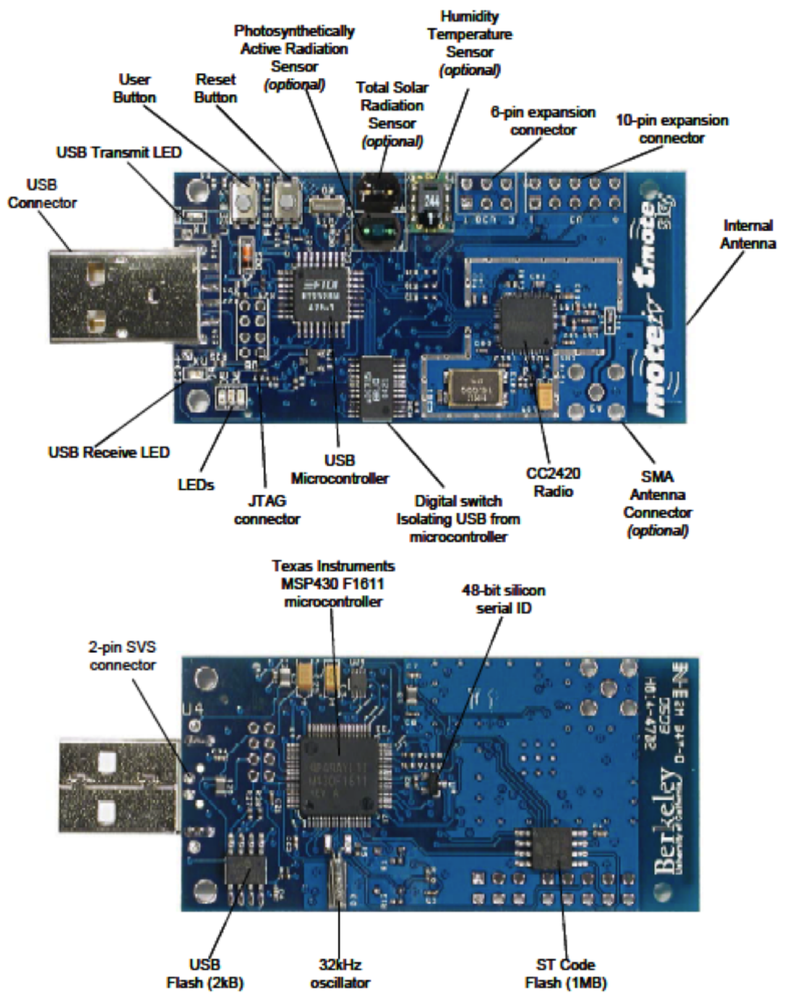

Photosynthetically Active Radiation Sensor (optional)
Humidity Temperature Sensor (optional)
User Button
Reset Button
Total Solar Radiation Sensor (optional)
6-pin expansion connector
10-pin expansion connector
USB Transmit LED
USB Connector
Internal Antenna
USB Receive LED
LEDs
JTAG connector
USB Microcontroller
Digital switch Isolating USB from microcontroller
CC2420 Radio
SMA Antenna Connector (optional)
Texas Instruments MSP430 F1611 microcontroller
48-bit silicon serial ID
2-pin SVS connector
USB Flash (2kB)
32kHz oscillator
ST Code Flash (1MB)

Figure 3.2: The top image shows the front of the TelosB mote. The bottom image shows the back of the TelosB.

MCU Specifications:

The TelosB mote uses the ultra low power Texas Instrument MSP430-F1611 micro-controller. The MSP430-F1611 is a 16-bit RISC processor capable of an extremely low active current consumption that permits the TelosB mote to run for years on a single pair of AA batteries [66]. The MSP430 MCU has an internal digitally controlled oscillator that if the power is at 3.6V may operate at 8MHz. If the TelosB were plugged into a USB port at a power level of 3V then the MCU speed would be roughly 7MHz. If the TelosB mote is powered by a pair of AA batteries then the MCU speeds will vary depending on the power level provided by the batteries. Figure 3.3 shows the supply voltage versus the MCU speed[66]. The voltage versus MCU speeds demonstrates how the battery life affects the overall functionality of the TelosB.

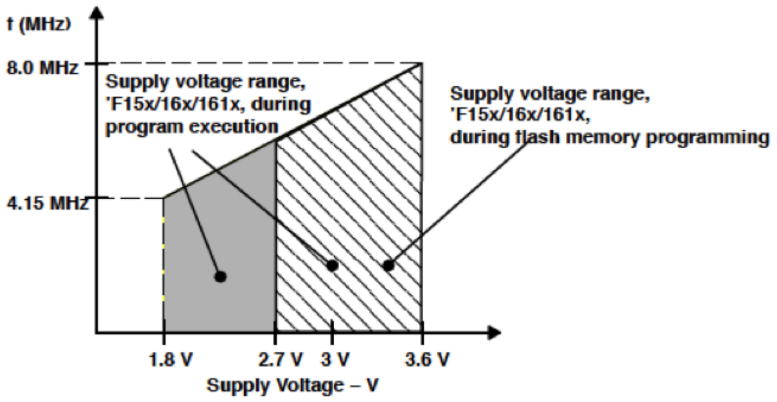


Figure 3.3: Clock frequency vs. supply voltage for MSP430-F15x/F16x/F161x

Memory Specifications:

As shown in Table 3.1, the TelosB mote is equipped with a 10kB of RAM, 48 kB of flash memory, 16kB of EEPROM, and 1024kB of M25P80 External Flash memory [65]. The 10kB of RAM is used when the TelosB mote is powered on to make use of program variables. The 48kB of flash memory is used to store the users program in our case the Coodinator, End-Device or PPPSniffer programs. The 16kB of EEPROM is used for mote hardware configuration purposes like the sensor calibration coefficients. Lastly, the 1024kB for M25P80 External Flash memory is used to store data logging information over time. It is important to note that the security flaw found during our investigations involves the 48kB of flash memory.

Table 3.1: The 4 types of memories found in the TelosB mote.

Memory Type	Size	Purpose
RAM	10kB	Store Variables
Flash Memory	48kB	Store User Code
EEPROM	16kB	Store Mote Configurations
M25P80	1024kB	Store Data Over Time

Radio Specifications:

The TelosB mote is equipped with a Chipcon CC2420 radio chip to support wireless communications [43]. The CC2420 operates on 2.4 GHz to 2.4835 GHz ranges, which means it is in the Industrial, Scientist Medical (ISM) band compliant. The data rate supported by the CC2420 is 250 kbps but data rates may drop depending on the power level. The CC2420 chip is also IEEE 802.15.4 compliant and has a built-in inline AES-128 encryption in which we make use of with our STMS application.

Sensor Specifications:

The TelosB mote has four integrated sensors that do not require additional hardware, unlike the IRIS and MICAz motes [65]. The first two sensors are used to measure light levels, such as measuring photosynthetically active radiation levels (S1087 sensor) and measuring the entire visible spectrum (S1087-01 sensor). The other two sensors are used to measure the humidity levels (SHT11 sensor) and temperature levels (SHT15 sensor) [62]. We only make use of the SHT15 sensor to monitor the temperature levels with our STMS.

The specifications of the TelosB mote suggests that all sensor technologies are resource limited and have low power consumption. Resources such as computation power, memory, and the communication bandwidth are to be kept in mind during a sensor application development process. Unlike computers or smart phones, sensors are very limited in their capabilities and require special protocols that keep low power consumption in mind.

3.2.2 Secure Temperature Monitoring System (STMS)

The Secure Temperature Monitoring System section is broken down into two parts: TinyOS IDE and STMS Implementation. The TinyOS IDE subsection describes the process of setting up an Integrated Development Environment for TinyOS. The STMS Implementation subsection discusses details on how our application works and the roles of each mote in our network.

(A) TinyOS IDE

The first step to set up the WSN environment was to create/install a designated OS that was solely used for research. In our case, we used OS X El Capitan as the host machine and Ubuntu 14.04 TLS as the guest machine using VMWARE Fusion. The Ubuntu is a 64-bit architecture and runs a Linux 3.16.0-57-generic kernel. We choose TinyOS as the WSN OS because of the big community and the popularity usage in sensor research. Other OS for sensors networks include Contiki and LiteOS; however, both lack applications samples for the MSP430 chips[67]. TinyOS is an open source OS designed for low-power wireless devices in which the reader may obtain a free copy from the TinyOS Github Project[68]. TinyOS was written in nesC, which is programming language that incorporates event-driven execution, a flexible concurrency model, and component-oriented application design[69]. Steps to install TinyOS:

1. Unzip the TinyOS project downloaded from Github [68]
2. Open a terminal window and cd into ../tinyos-main/tools
3. Execute the following commands to install the make system for TinyOS
    - $ ./Bootstrap

- $ ./configure
- $ make
- $ sudo make install

4. cd into /etc/apt/sources.list.d and execute the following commands to install TinyOS tools development packages

   - $ sudo echo ”deb http://tinyprod.net/repos/debian squeeze main” >> tinyprod-debian.list
   - $ sudo echo ”deb http://tinyprod.net/repos/debian msp430-46 main” >> tinyprod-debian.list
   - $ sudo apt-get update
   - $ sudo apt-get install msp430-46 nesc tinyos-tools-devel

The next step was to install software that can support an IDE for TinyOS, the software needed are Eclipse and Yeti2. Eclipse can be simply installed by downloading the latest Eclipse SDK version from their website. After installing Eclipse, the next step was to install the Yeti2 Eclipse plugin for TinyOS IDE support. Yeti2 is a nesC editor implemented as a plugin for Eclipse aim to provide developers with the convenience functions expected from a modern development environment [70]. There are other IDE for TinyOS such as NESCDT [71] or TinyDT [72]; however, based on blogs most of the developers recommend using Yeti2 due to its simplicity and user-friendly style.
Steps to install Yeti2:

1. Open Eclipse and go to Help > Install New Software
2. Click on Add to add a new repository
3. Use the following to create a new repository:
   - $ Name: Yeti2
   - $ Location: http://tos-ide.ethz.ch/update/site.xml
4. Make sure that the following boxes are checked:
   - $ Yeti 2 Core
   - $ Yeti2 Optional
   - $ Yeti2 Environments (make sure the Yeti 2 TinyOS 2.x in Cygwin is unchecked)
5. Install Yeti2 and let Eclipse restart

6. Open the TinyOS perspective by going to Window > Open Perspective > Other
7. Choose TinyOS and click OK
8. Set up the plugin environments by going to Window > Preferences
9. Select TinyOS > Environments > TinyOS 2.x unix-environment
10. Make sure TinyOS Root Directory, TinyOS TOS Directory, TinyOS Application Directory, and TinyOS App Makerules reference the TinyOS installation

If all steps were followed correctly the reader should now have a TinyOS IDE for developing WSN applications. In our case, we used our TinyOS IDE to implement STMS.

(B) STMS Implementation

The contribution of this thesis is to investigate the security flaw found in the MSP430-BSL password that is used to protect flash memory firmware. To demonstrate the outcomes of dumping flash memory firmware, we reverse engineered our STMS End-Device application to obtain the cryptographic keys. With that being said, our STMS is built on top of Layer 1 and Layer 2 of the OSI model. Figure 3.4 shows the comparison of the OSI model between wireless networks and wireless sensor networks [73]. The IEEE 802.15.4 standard is similar to the 802.11 but the protocol is built on the idea of low power consumption and limited transport rates of 250Kbps [74]. We make use of the IEEE 802.15.4 protocol at the data link layer to establish an association between the Coordinator and an End-Device. We also make use of the CC2420 inline AES-128 encryption at the physical layer to encrypt the communication between the Coordinator and the End-Devices [43]. If the security is broken at the physical layer by obtaining copies of the cryptographic keys through reverse engineering then all upper layers, like the 6LoWPAN protocol, Zigbee protocol and CoAP protocol will be broken because of the OSI model domino effect. Additional reasons why we build on top of the L1 and L2 layers is to limit the code size added when using the CoAP or 6LoWPAN protocols. TinyOS provides sample applications that make use of the 6LoWPAN or CoAP protocols, but when attempting to build the examples we encounter a ”region rom overflowed by N number of bytes” error. Making use of the upper layers (6LoWPAN, Zigbee or CoAP) exceed the allotted TelosB 48kB rom size when compiling the application.

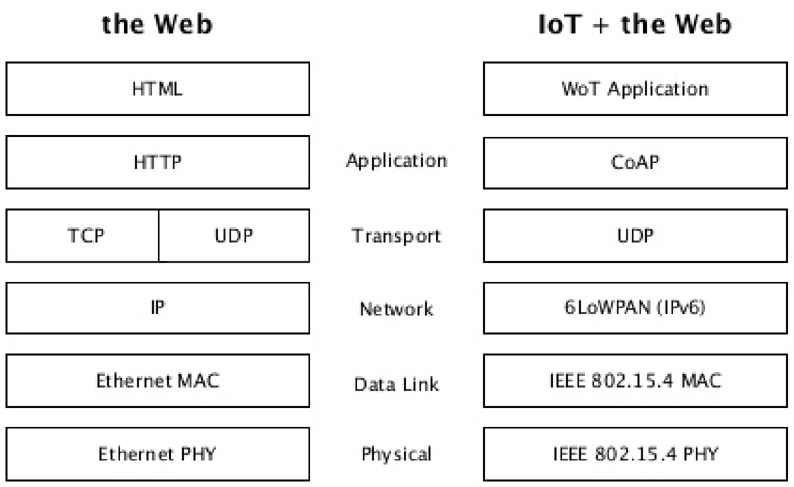


Figure 3.4: A comparison of the OSI model between a wireless network (left) vs. a wireless sensor network (right).

We have implemented two versions of our Secure Temperature Monitoring System: we call the first version IEEE802154 and the second version L1-Secure. The IEEE802154 version is built on top of the IEEE 802.15.4 MAC layer, which allows an End-Device to associate with the Coordinator before sending temperature reading packets. The communication in the IEEE802154 version is unencrypted and the purpose is to demonstrate how motes can join the STMS network. The L1-Secure version is built on top of the Physical layer (L1), which allows an End-Device to encrypt the data before sending temperature reading packets to the Coordinator. The L1-Secure version does not require the End-Device to associate with the Coordinator before sending the encrypted packet. We have implemented two versions (IEEE802154 and L1-Secure) of STMS because at the time of this writing there is no implementation of the secure MAC layer using the IEEE 802.15.4 protocol [74]. We have successfully built a secure WSN solely using the L1 layer, thus avoiding any implementation required to include the IEEE 802.15.4 MAC layer. Since the purpose of this thesis is the security of firmware at the L1 layer and not the upper layers, then building on top of the L1 layer suffices the proof of concept. The STMS code has been made open source for research purposes and can be downloaded from our GitHub project[75].

- *Coodinator:*

  As previously mentioned, the Coordinator is the central node of our STMS. Since we have two versions of the Coordinator the operations slightly vary between the IEEE802154 version and the L1-Secure version. Both versions support the operations of listening for incoming packets and forwarding the received data. The difference is that the IEEE802154 version supports association responses and the L1-secure version supports decryption. To support association responses the IEEE802154 version uses the MAC sub-Layer Management Entity (MLME) interface provided by TinyOS. The MLME interface provides access to the management services provided by the MAC sub-layer. In particular, the following MLME components are used: MLME_RESET, MLME_SET, MLME_START, MLME_ASSOCIATE, and MLME_COMM_STATUS. The MLME_RESET allows the Coordinator to request a MLME reset and initialize the MAC to be able to use other MAC primitives. The MLM_SET is used to set the values on the Personal Area Network Information Base (PIB), which is a database that stores information to manage the MAC layer. MLME_START is used to begin acting as Coordinator of a PAN and will always be called after the MLME_SET. The MLME_ASSOCIATE is used to notify the Coordinator that an End-Device has requested to associate with the PIB. The MLME_COMM_STATUS is used to indicate communication status between the Coordinator and the End-Devices. As far as listening for incoming packets, the IEEE802154 version uses the MCPS_DATA interface where as the L1-Secure version uses the Receive interface. The decryption operation of the L1-Secure version makes use of the CC2420Keys interface to set or "send" the key from the MSP430 flash memory to the CC2420 chip. When a packet is received the payload is decrypted by the CC24020 chip using a network wide shared key. The actual data content received by the Coordinator is a 14-bit digital readout value that is converted to Fahrenheit using the follow formula in (3.1)[62]. The final operation requires both versions to forward the temperature reading to the Ubuntu VM in order to print the Node ID and Fahrenheit value on the screen. The C printf library is used to display the temperature readings on the screen using the Serial port terminal tool available on the Ubuntu Software Center.

$$-39.3 + (0.018 \times data) \tag{3.1}$$

- *End-Device:*

  The End-Devices are the end-nodes of our STMS; they are responsible for collecting temperature readings of various locations. Similar to the Coordinator, we have two versions of the End-Device with slight variation between the IEEE802154 version and the L1-Secure version. Both versions support the operations of collecting temperature readings and sending temperature packets to the Coordinator. The difference is that the IEEE802154 version supports association requests, where as the L1-secure version supports encryption. To support association requests the IEEE802154 version uses the MAC sub-Layer Management

Entity (MLME) interface provided by TinyOS. In particular, the following MLME components are used: MLME_RESET, MLME_SET, MLME_GET, and MLME_ASSOCIATE. The MLME_RESET, MLME_SET, and MLME_ASSOCIATE have the same roles as discussed in the Coordinator section. The new interface used with the End-Device is the MLME_GET, which is used to get specific information about the PIB. Both versions require collecting temperature readings using the built-in temperature sensor SHT15. The SensirionSht11C interface is used by both versions to read temperature levels in real time using the SHT15 sensor. The IEEE802154 reads temperatures every 5 seconds, where as the L1-Secure reads temperatures every 3 seconds. The temperature reading is a 14-bit digital output value that is sent to the Coordinator along with the Node ID. However, before sending the data the L1-Secure version encrypts the payload of the packet by using the CC2420Keys interface and CC2420SecurityMode interface. The CC2420Keys interface is used to set or "send" the network wide shared key from the MSP430 chip to the CC2420 chip. The CC2420SecurityMode interface is used to set AES-128-CTR security mode in order to encrypt the payload portion of the packets. The final operation requires both versions to send the temperature packets to the Coordinator. The IEEE802154 version uses the MCPS_DATA interface to send the temperature packets to the Coordinator after an association has been established. The L1-Secure version uses the AMSend interface to send the encrypted temperature packets to the Coordinator with no pre-association establishments.

- *PPPSniffer:*

  The PPPSniffer node has one main role of capturing packets being sent from the End-Devices to the Coordinator in our STMS. TinyOS provides the PPPSniffer application that can be linked to Wireshark. The application is found under the /tinyos-main/apps/PPPSniffer directory of the TinyOS Github Project [68]. The application receives and snoops packets on the IEEE 802.15.4 channel and forwards the packets to the Ubuntu VM using the Point-to-Point protocol. The Ubuntu VM then uses Wireshark to allow us to view details about the packets being sent on our STMS network or any WSN.

3.3 System Testing

The following section will provide the steps to set up STMS along with the PPPSniffer to display the capture packets using Wireshark. We assume the TinyOS IDE (refer to Section 3.2.2) has been setup and the STMS code has been downloaded from our GitHub project[75]. To avoid any path compilation errors, it is important that the downloaded STMS code is stored under the /tiny-main/apps directory. The first step of setting up the WSN environment is to plugin the TelosB motes to USB ports found on a computer. Figure 3.5 shows a USB Hub we setup to allow us to plugin all our TelosB motes to the computer that is running the Ubuntu VM. Furthermore, using a terminal window on the Ubuntu VM we ran the motelist command to list all the current motes recognized by the Ubuntu VM. Figure 3.6 displays the output of the motelist commands which matches the labels and TelosB motes seen in Figure 3.5. We are now ready to flash the PPPSniffer, IEEE802154, and L1-Secure applications to the plugged in TelosB motes.

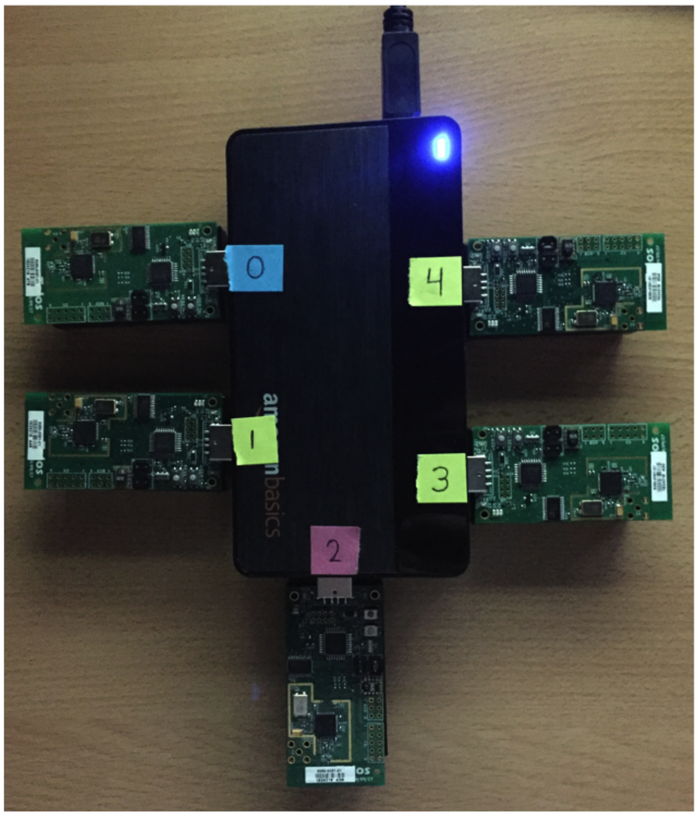
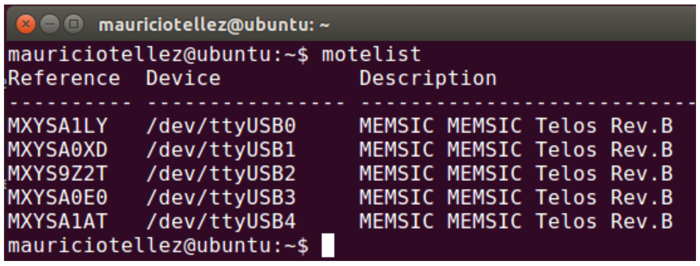


Figure 3.5: A picture taken of our USB hub and TelosB motes and the corresponding port number



```
mauriciotellez@ubuntu:~$ motelist
Reference  Device           Description
---------- ---------------- ---------------------------
MXYSA1LY   /dev/ttyUSB0     MEMSIC MEMSIC Telos Rev.B
MXYSA0XD   /dev/ttyUSB1     MEMSIC MEMSIC Telos Rev.B
MXYS9Z2T   /dev/ttyUSB2     MEMSIC MEMSIC Telos Rev.B
MXYSA0E0   /dev/ttyUSB3     MEMSIC MEMSIC Telos Rev.B
MXYSA1AT   /dev/ttyUSB4     MEMSIC MEMSIC Telos Rev.B
mauriciotellez@ubuntu:~$
```

Figure 3.6: The output of the mostelist command

- **PPPSniffer Setup**

  The PPPSniffer requires the Wireshark tool, which can installed by either obtaining a copy from the Ubuntu Software Manager or using the apt-get command. The two procedures to setup the PPPSniffer will be the same for both STMS versions. The setup consists of first flashing the PPPSniffer application to the mote that is plugged in to the /dev/ttyUSB2 port. The second step is to establish an internet link between the /dev/ttyUSB2 port and the ppp interface using the pppd command.

  Steps to setup PPPSniffer:

  1. Open a terminal window
  2. $cd /opt/tinyos-main/apps/PPPSniffer
  3. $make telosb cc2420x_32khz install,1 bsl,/dev/ttyUSB2
  4. $sudo pppd debug passive noauth nodetach 115200 /dev/ttyUSB2 nocrtscts nocdtrcts lcp-echo-interval 0 noccp noip ipv6 ::23,::24
  5. The last line of the previous command output should be "Script /etc/ppp/ipv6-up finished (pid 14415), status = 0x0"
  6. The terminal window should be left open

- **STMS Setup**

  The setup procedures between the two versions of IEEE802154 and L1-Secure are the same except they differ in flashing the appropriate application. However, the same procedures should be followed to flash the Coordinator and the three End-Devices on either versions. Setting up the Coordinator requires additional steps compared to the PPPSniffer setup steps. The first step is to flash the Coordinator application to the mote that is plugged in to the /dev/ttyUSB0 port. The second step is to establish an internet link between the /dev/ttyUSB0 port and the ppp interface using the pppd command. The third step is add the ppp1 interfaces to the list network interfaces by using the ifconfig command. The last step is to open up Wireshark and start capturing packets on the ppp0 interface.

  Steps to setup Coordinator:

  1. Open a terminal window
  2. $cd /opt/tinyos-main/apps/STMS/IEEE802154/Coordinator/src (for IEEE802154 version) or

     $cd /opt/tinyos-main/apps/STMS/L1-Secure/Coordinator/src (for L1-Secure version)

3. $make telosb install.1 bsl,/dev/ttyUSB0
4. $sudo pppd debug passive noauth nodetach 115200 /dev/ttyUSB0 nocrtscts nocdtrcts lcp-echo-interval 0 noccp noip ipv6 ::25,::26
5. The output of the previous command should be interval messages of
   ”sent [LCP ConfReq id=0x1 <asyncmap 0x0> <magic 0xc9580e> <pcomp> <accomp>]”
6. The terminal window should be left open
7. Open a new terminal window
8. $sudo ifconfig ppp1 add fec0::101/64
9. $sudo wireshark
10. Wireshark will start and proceed to capture packets on the ppp0 interface.

The steps to setup the End-Devices is straight forward because it only requires flashing the End-Device application to the three End-Device motes.
Steps to setup End-Devices:

1. Open a terminal window
2. $cd /opt/tinyos-main/apps/STMS/IEEE802154/End-Device/src (for IEEE802154 version) or
   $cd /opt/tinyos-main/apps/STMS/L1-Secure/End-Device/src (for L1-Secure version)
3. $make telosb install.4 bsl,/dev/ttyUSB1 (for sensor in Room 4)
4. $make telosb install.6 bsl,/dev/ttyUSB3 (for sensor in Room 6)
5. $make telosb install.8 bsl,/dev/ttyUSB4 (for sensor in Room 8)

3.3.1 IEEE802154 Capture Packets & Analysis

The IEEE802154 version of the STMS makes use of the LED lights for demonstration purposes and checking if the application is working as expected. As shown in Figure 3.7 there are various lights turned on for each TelosB. The TelosB mote in port USB0 is the **Coordinator** which blinks the green LED light for every successful End-Device association and blinks the blue light for every packet received. The TelosB motes in ports USB1, USB3 and USB4 are the the **End-Devices**, which turn on the green LED light when successfully establishing a association with the Coordinator and blink the blue LED light for every packet sent. The TelosB mote in port USB2 is the **PPPSniffer** which blinks the green LED light for every packet capture and the red LED light for every capture packet sent to the computer.

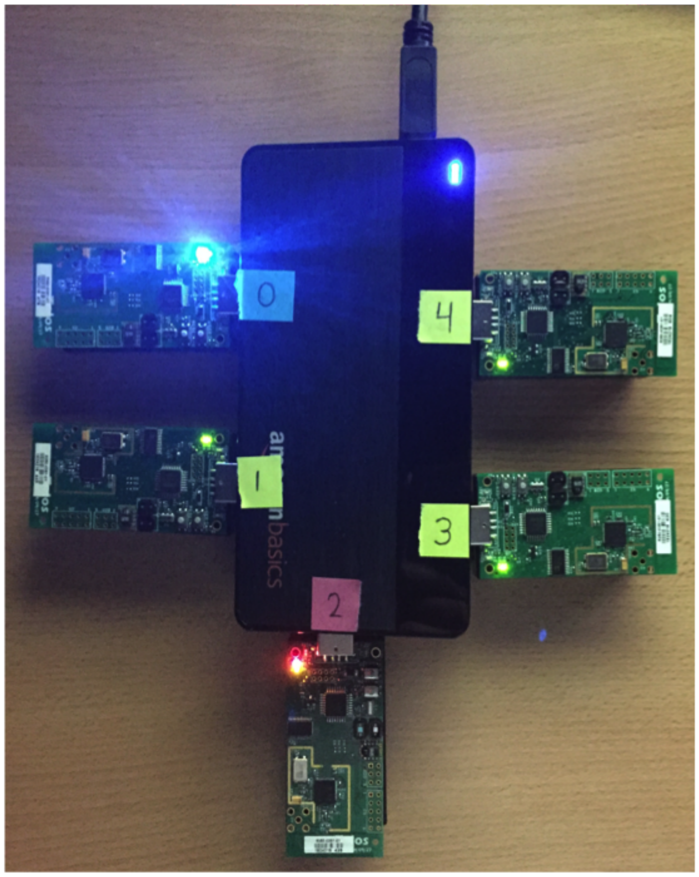
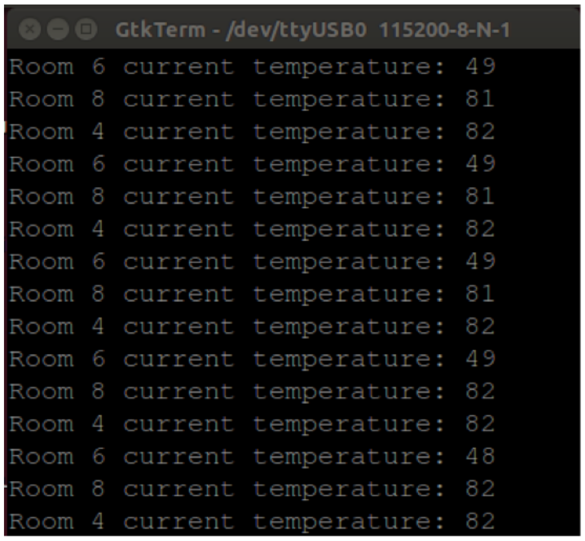


Figure 3.7: A picture taken of our USB hub and TelosB motes with the IEEE802154 version and PPPSniffer running

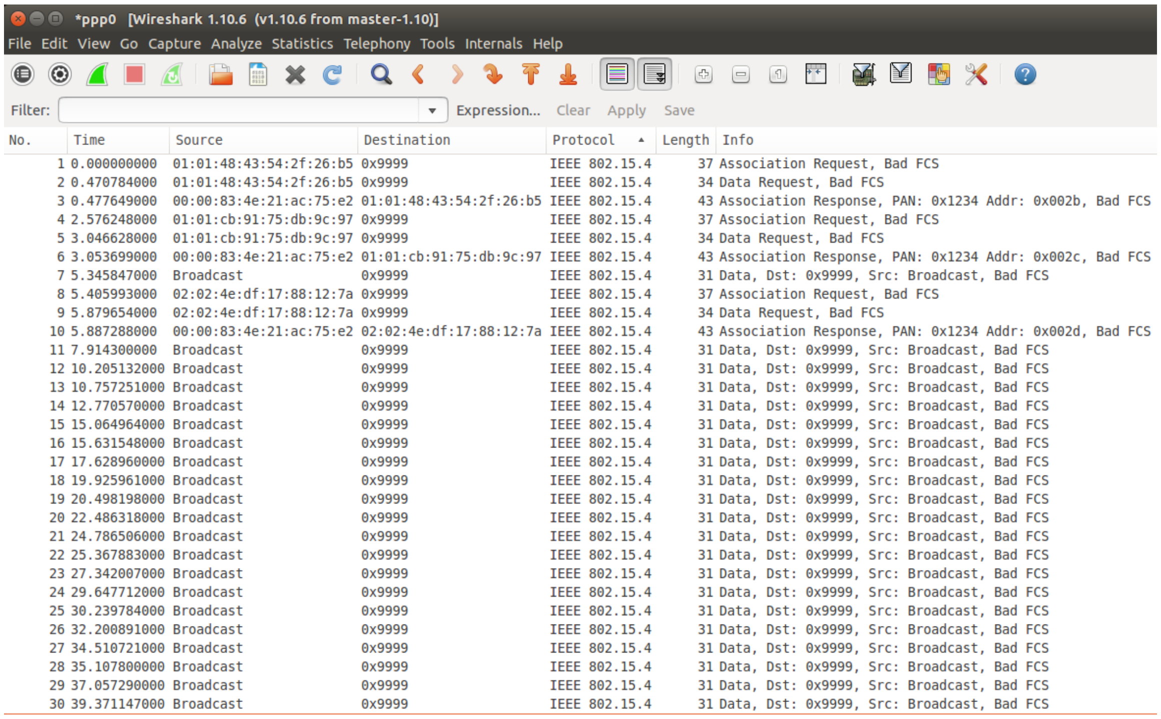
To display the actual temperature readings collected from the End-Device we use the Serial Port Terminal (SPT), which can be installed through the Ubuntu Software Center. Once opened the SPT requires setting the proper configurations by clicking on "Configurations > Port" then setting the Port field to /dev/ttyUSB0 and Baud Rate field to 115200. Figure 3.8 shows the results of displaying the data received by the Coordinator which is plugged in to port USB0. As we can see from the results, the sensors located in Room 4 and Room 8 were reading temperature levels of 81^oF or 82^oF. The two sensors were located in the same graduate lab room in the Computer Science Department at James Madison University. The environment for sensor 6 was changed to demonstrate that sensors can be powered by batteries and may collect different temperature levels. We used two double AA batteries to power sensor 6 and placed the sensor in our graduate lab refrigerator. As seen in the results, the temperature readings for Room 6 were 48^oF or 49^oF, which was the same temperature set in the refrigerator. We must point out that the temperature readings may be affected by 1^oF or 2^oF due to heat generated by the MSP430 when constantly being used.



```
Room 6 current temperature: 49
Room 8 current temperature: 81
Room 4 current temperature: 82
Room 6 current temperature: 49
Room 8 current temperature: 81
Room 4 current temperature: 82
Room 6 current temperature: 49
Room 8 current temperature: 81
Room 4 current temperature: 82
Room 6 current temperature: 49
Room 8 current temperature: 82
Room 4 current temperature: 82
Room 6 current temperature: 48
Room 8 current temperature: 82
Room 4 current temperature: 82
```

Figure 3.8: The output generated by the serial port terminal, which displays temperature readings received by the Coordinator

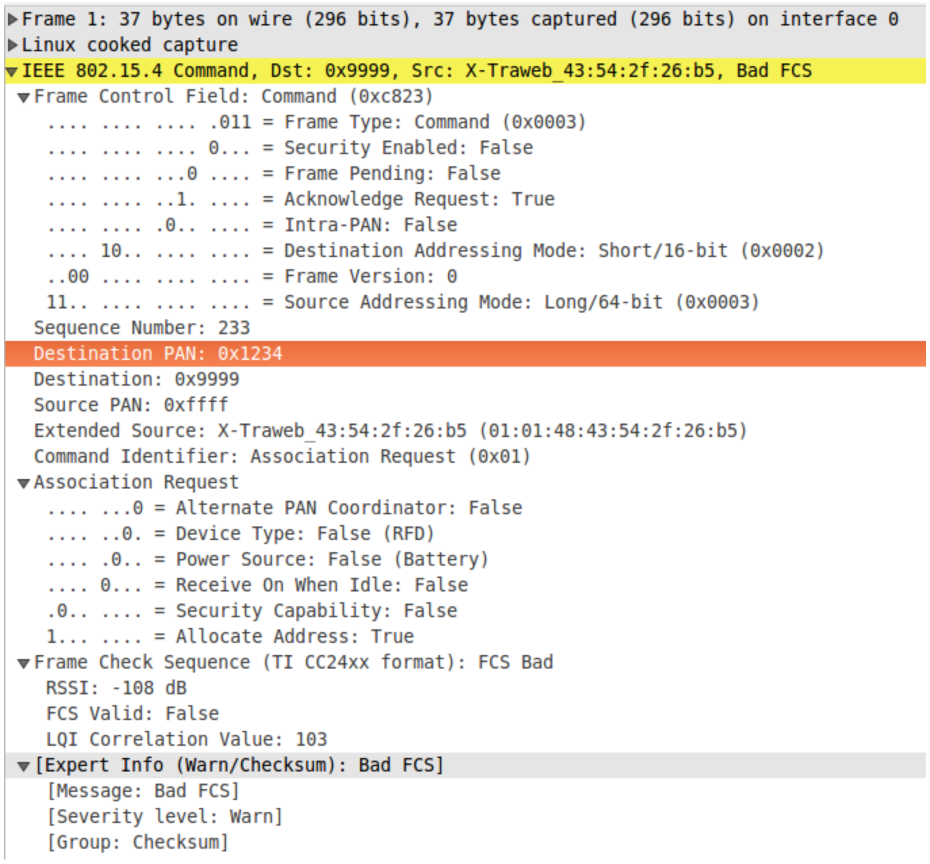
The information collected by Wireshark shows detailed traffic going across our STMS IEEE802154 network and the results have been saved in the IEEE802154.pcapng file, which can be accessed through our Github project [75]. Before analyzing the information, Wireshark must be configured properly to interpret IEEE 802.15.4 packets. First click on "Edit > Preferences...", then on the Wireshark Preferences - Profile window select "Protocols > IEEE 802.15.4" and make sure that the "TI CC24xx FCS format" box is checked. Apply the new settings and, once the Wireshark Preferences - Profile window closes, right click on any of the packets and click on "Decode as...". On the Wireshark Decode As window select the link tab and make sure the value for the Ethertype 0x86dd value "IEEE 802.15.4" is selected. The capture packets between the Coordinator and the End-Devices will be decoded as IEEE 802.15.4 format and will make the interpretation process easier. Figure 3.9 shows the results of capturing packets going across the STMS IEEE802154 network. As we can see from the results, packets 1, 2 and 3 belong to the association between the Coordinator and the first End-Device sensor. Packets 4, 5, and 6 belong to the association between the Coordinator and the second End-Device sensor. Packets 8, 9, and 10 belong to the association between the Coordinator and the third End-Device sensor. The rest of the packets are the temperature readings collected by the End-Devices and sent to the Coordinator.


*ppp0 [Wireshark 1.10.6 (v1.10.6 from master-1.10)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression...
Clear
Apply
Save

```
No.  Time         Source                  Destination             Protocol      Length Info
   1 0.000000000  01:01:48:43:54:2f:26:b5 0x9999                  IEEE 802.15.4     37 Association Request, Bad FCS
   2 0.470784000  01:01:48:43:54:2f:26:b5 0x9999                  IEEE 802.15.4     34 Data Request, Bad FCS
   3 0.477649000  00:00:83:4e:21:ac:75:e2 01:01:48:43:54:2f:26:b5 IEEE 802.15.4     43 Association Response, PAN: 0x1234 Addr: 0x002b, Bad FCS
   4 2.576248000  01:01:cb:91:75:db:9c:97 0x9999                  IEEE 802.15.4     37 Association Request, Bad FCS
   5 3.046628000  01:01:cb:91:75:db:9c:97 0x9999                  IEEE 802.15.4     34 Data Request, Bad FCS
   6 3.053699000  00:00:83:4e:21:ac:75:e2 01:01:cb:91:75:db:9c:97 IEEE 802.15.4     43 Association Response, PAN: 0x1234 Addr: 0x002c, Bad FCS
   7 5.345847000  Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
   8 5.405993000  02:02:4e:df:17:88:12:7a 0x9999                  IEEE 802.15.4     37 Association Request, Bad FCS
   9 5.879654000  02:02:4e:df:17:88:12:7a 0x9999                  IEEE 802.15.4     34 Data Request, Bad FCS
  10 5.887288000  00:00:83:4e:21:ac:75:e2 02:02:4e:df:17:88:12:7a IEEE 802.15.4     43 Association Response, PAN: 0x1234 Addr: 0x002d, Bad FCS
  11 7.914300000  Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  12 10.205132000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  13 10.757251000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  14 12.770570000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  15 15.064964000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  16 15.631548000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  17 17.628960000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  18 19.925961000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  19 20.498198000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  20 22.486318000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  21 24.786506000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  22 25.367883000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  23 27.342007000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  24 29.647712000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  25 30.239784000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  26 32.200891000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  27 34.510721000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  28 35.107800000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  29 37.057290000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
  30 39.371147000 Broadcast               0x9999                  IEEE 802.15.4     31 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
```

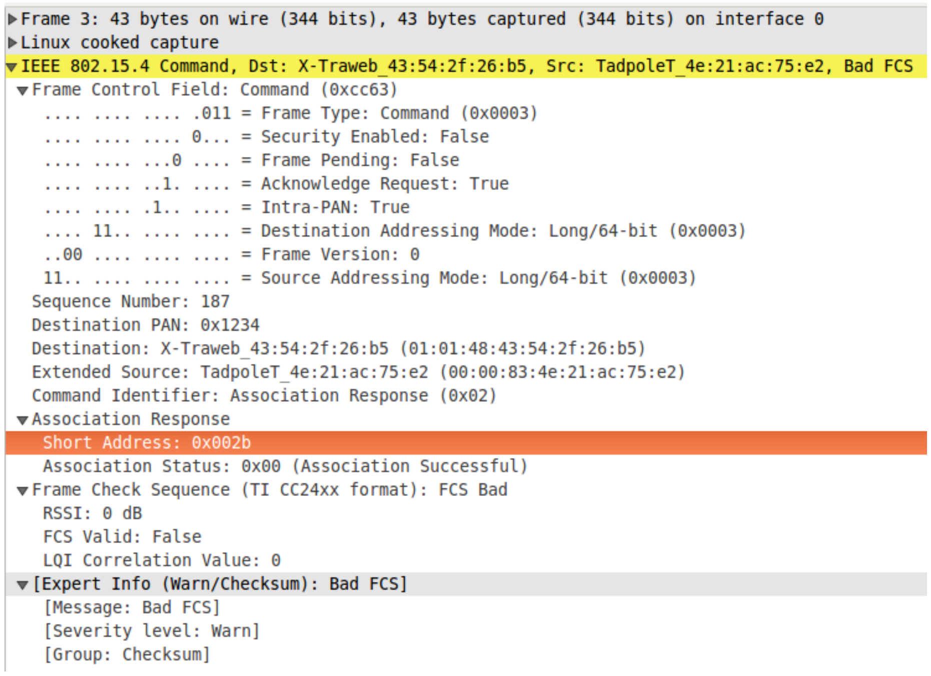
Figure 3.9: The packets going across the STMS IEEE802154 network.

Figure 3.10 shows details about packet 1, which is the Association Request packet sent by the End-Device to the Coordinator. Some of the interesting fields found are the Destination Personal Area Network (PAN), Destination, Source PAN, and Extended Source. A sensor device will either have an extended address (also known as "unique address") or a short address that was allocated by the PAN Coordinator when the End-Device associated[74]. The destination PAN identifier address 0x1234 allows communication between PANs and serves transmissions between End-Devices across independent networks. The destination address 0x9999 is the short address of the Coordinator and was predefined along with the destination PAN during the development of the STMS IEEE802154. The source PAN address 0xffff is the short address of the End-Device and by default is set to 0xffff if the End-Device has not yet been associated with a PAN. The extended source address 01:01:48:43:54:2f:26:b5 is the unique address of the End-Device requesting to join the PAN. As shown in Figure 3.11, the response sent by the Coordinator to the End-Device is an Association Response packet. In this case, the extended source address 00:00:83:4e:21:ac:75:e2 is the unique address of the Coordinator. The short address assigned to the End-Device by the Coordinator is 0x002b, which is unique with the 0x1234 PAN. The Coordinator assigns short addresses to new End-Devices that join the network by incrementing from the previously assigned address. In this case the next End-Device that joins the network will receive the value of 0x002c. The short address value will be reset to 0x0000 when the Coordinator has rebooted by pressing the reset button on the motes. Once the association has been established the communication continues by sending temperature readings from the End-Device to the Coordinator. Figure 3.12 shows the hex values found on the payload section of the packet. The payload consists of 8 bytes, where the first four bytes 0x0004, or 4, is the Node ID and the last 4 bytes 0x1b13, or 6913, is the temperature data. If we convert the 6913 to Fahrenheit using the Sensirion formula we get $-39.3 + (0.018 \times 6913)$ or 85^oF. The temperature value has increased because the motes were running for a long period of time; therefore, sending temperature packets every 3 seconds affected the temperature sensor by 3^oF.



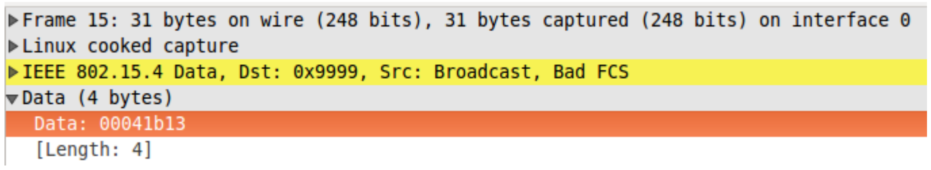
```
▶Frame 1: 37 bytes on wire (296 bits), 37 bytes captured (296 bits) on interface 0
▶Linux cooked capture
▼IEEE 802.15.4 Command, Dst: 0x9999, Src: X-Traweb_43:54:2f:26:b5, Bad FCS
 ▼Frame Control Field: Command (0xc823)
    .... .... .... .011 = Frame Type: Command (0x0003)
    .... .... .... 0... = Security Enabled: False
    .... .... ...0 .... = Frame Pending: False
    .... .... ..1. .... = Acknowledge Request: True
    .... .... .0.. .... = Intra-PAN: False
    .... 10.. .... .... = Destination Addressing Mode: Short/16-bit (0x0002)
    ..00 .... .... .... = Frame Version: 0
    11.. .... .... .... = Source Addressing Mode: Long/64-bit (0x0003)
  Sequence Number: 233
  Destination PAN: 0x1234
  Destination: 0x9999
  Source PAN: 0xffff
  Extended Source: X-Traweb_43:54:2f:26:b5 (01:01:48:43:54:2f:26:b5)
  Command Identifier: Association Request (0x01)
 ▼Association Request
    .... ...0 = Alternate PAN Coordinator: False
    .... ..0. = Device Type: False (RFD)
    .... .0.. = Power Source: False (Battery)
    .... 0... = Receive On When Idle: False
    .0.. .... = Security Capability: False
    1... .... = Allocate Address: True
 ▼Frame Check Sequence (TI CC24xx format): FCS Bad
    RSSI: -108 dB
    FCS Valid: False
    LQI Correlation Value: 103
 ▼[Expert Info (Warn/Checksum): Bad FCS]
    [Message: Bad FCS]
    [Severity level: Warn]
    [Group: Checksum]
```

Figure 3.10: An example of the association request packet sent from the End-Device to the Coordinator.



```
▶Frame 3: 43 bytes on wire (344 bits), 43 bytes captured (344 bits) on interface 0
▶Linux cooked capture
▼IEEE 802.15.4 Command, Dst: X-Traweb_43:54:2f:26:b5, Src: TadpoleT_4e:21:ac:75:e2, Bad FCS
 ▼Frame Control Field: Command (0xcc63)
    .... .... .... .011 = Frame Type: Command (0x0003)
    .... .... .... 0... = Security Enabled: False
    .... .... ...0 .... = Frame Pending: False
    .... .... ..1. .... = Acknowledge Request: True
    .... .... .1.. .... = Intra-PAN: True
    .... 11.. .... .... = Destination Addressing Mode: Long/64-bit (0x0003)
    ..00 .... .... .... = Frame Version: 0
    11.. .... .... .... = Source Addressing Mode: Long/64-bit (0x0003)
  Sequence Number: 187
  Destination PAN: 0x1234
  Destination: X-Traweb_43:54:2f:26:b5 (01:01:48:43:54:2f:26:b5)
  Extended Source: TadpoleT_4e:21:ac:75:e2 (00:00:83:4e:21:ac:75:e2)
  Command Identifier: Association Response (0x02)
 ▼Association Response
   Short Address: 0x002b
   Association Status: 0x00 (Association Successful)
 ▼Frame Check Sequence (TI CC24xx format): FCS Bad
   RSSI: 0 dB
   FCS Valid: False
   LQI Correlation Value: 0
 ▼[Expert Info (Warn/Checksum): Bad FCS]
   [Message: Bad FCS]
   [Severity level: Warn]
   [Group: Checksum]
```

Figure 3.11: An example of the association response packet sent from the Coordinator to the End-Device.



```
▶Frame 15: 31 bytes on wire (248 bits), 31 bytes captured (248 bits) on interface 0
▶Linux cooked capture
▶IEEE 802.15.4 Data, Dst: 0x9999, Src: Broadcast, Bad FCS
▼Data (4 bytes)
  Data: 00041b13
  [Length: 4]
```

Figure 3.12: An example of the payload portion found in the packet sent from the End-Device to the Coordinator.

3.3.2 L1-Secure Capture Packets & Analysis

The L1-Secure version of STMS also makes use of the LED lights for demonstration purposes and checking if the application is working as expected. As shown in Figure 3.13, there are fewer lights turned on compared to the IEEE802154 version. The TelosB mote in port USB0 is the Coordinator, which blinks the blue LED light for every packet received. The TelosB motes in ports USB1, USB3 and USB4 are the the End-Devices, which blink the blue LED light for every packet sent. The TelosB mote in port USB2 is the PPPSniffer, which blinks the green LED light for every packet captured and the red LED light for every capture packet sent to the computer.

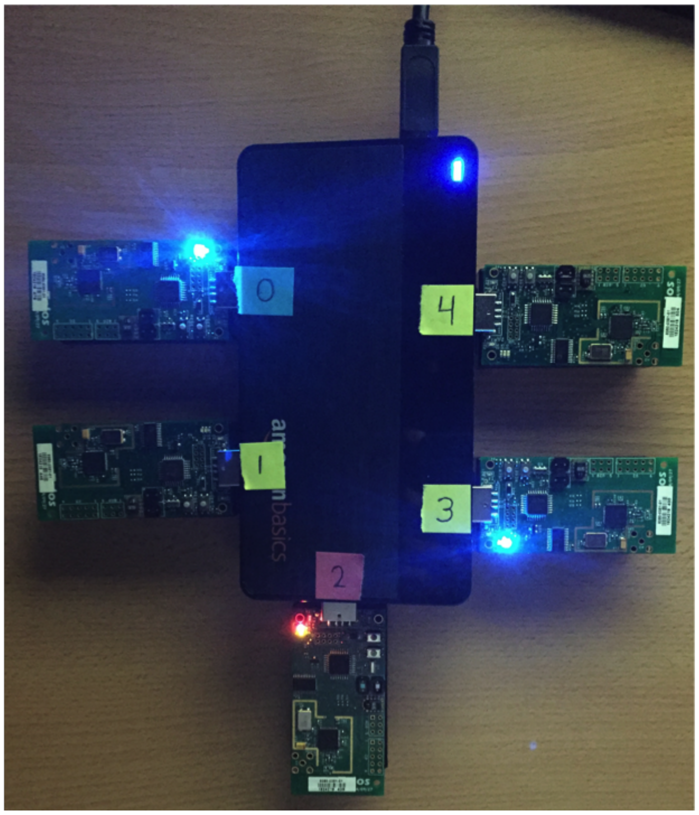
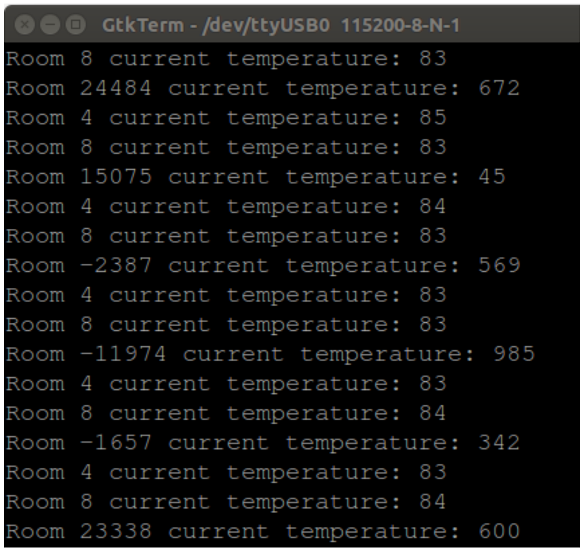


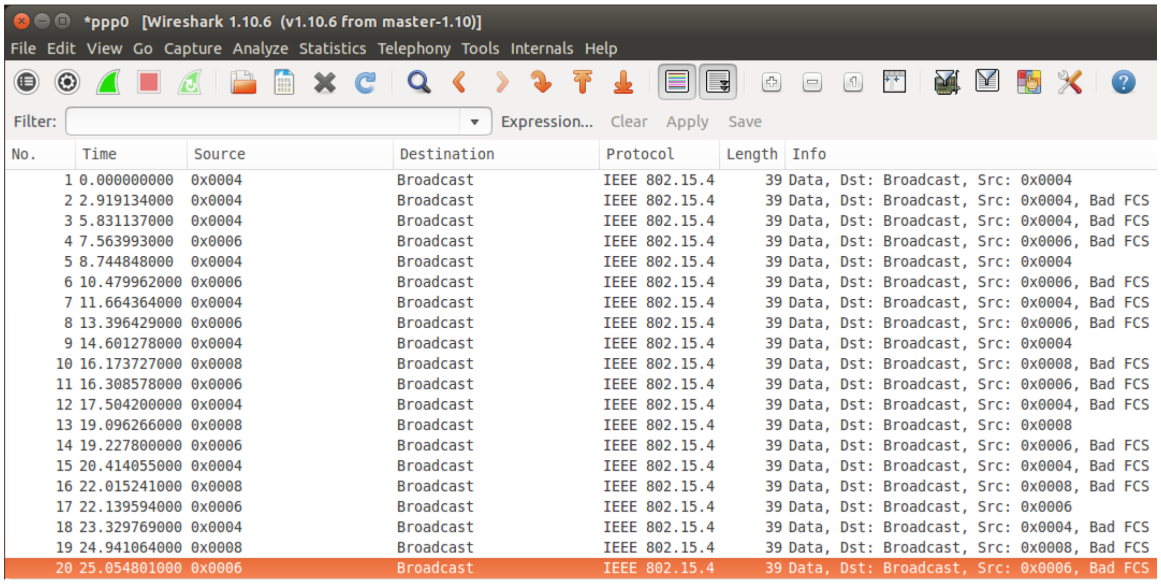
Figure 3.13: A picture taken of our USB hub and TelosB motes with the IEEE802154 and PPPSniffer running

Similar to the IEEE802154 version, we display the data received by the Coordinator using the Serial Port Terminal (SPT) tool in our L1-Secure version. Figure 3.14 shows the results of displaying the data received by the Coordinator plugged in to port USB0. The results show the sensors located in Room 4 and Room 8 were reading temperature levels of 83^oF to 85^oF. Again, the sensors were reading higher temperatures because the motes had been running for an extended period of time, thus making the MSP430 chips produce heat. If the sensors were programmed to the collected temperature measurements every 3 hours, instead of 3 seconds, then the temperature measurements would be more accurate. We can also observe from the results that the sensor in Room 6 was no where to be found and, instead, random data was displayed. We modified the code for the sensor in Room 6 to use a different key than the network wide shared key, thus the Coordinator was not able to decrypt the message sent by the sensor in Room 6. The purpose was to demonstrate that if the correct key is not used then the Coordinator can be programmed to ignore the "garbage," or random data.



```
Room 8 current temperature: 83
Room 24484 current temperature: 672
Room 4 current temperature: 85
Room 8 current temperature: 83
Room 15075 current temperature: 45
Room 4 current temperature: 84
Room 8 current temperature: 83
Room -2387 current temperature: 569
Room 4 current temperature: 83
Room 8 current temperature: 83
Room -11974 current temperature: 985
Room 4 current temperature: 83
Room 8 current temperature: 84
Room -1657 current temperature: 342
Room 4 current temperature: 83
Room 8 current temperature: 84
Room 23338 current temperature: 600
```

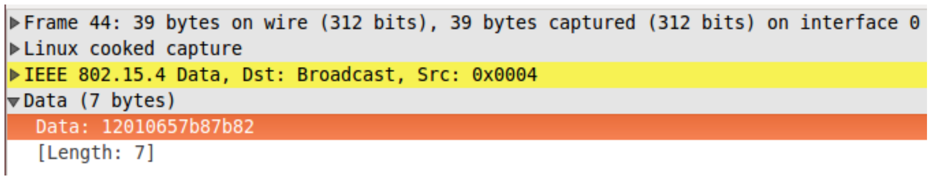
Figure 3.14: The output generated by the serial port terminal, which displays temperature readings received by the Coordinator


*ppp0 [Wireshark 1.10.6 (v1.10.6 from master-1.10)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression...
Clear
Apply
Save

```
No.    Time         Source     Destination   Protocol      Length Info
     1 0.000000000  0x0004     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0004
     2 2.919134000  0x0004     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0004, Bad FCS
     3 5.831137000  0x0004     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0004, Bad FCS
     4 7.563993000  0x0006     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0006, Bad FCS
     5 8.744848000  0x0004     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0004
     6 10.479962000 0x0006     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0006, Bad FCS
     7 11.664364000 0x0004     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0004, Bad FCS
     8 13.396429000 0x0006     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0006, Bad FCS
     9 14.601278000 0x0004     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0004
    10 16.173727000 0x0008     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0008, Bad FCS
    11 16.308578000 0x0006     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0006, Bad FCS
    12 17.504200000 0x0004     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0004, Bad FCS
    13 19.096266000 0x0008     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0008
    14 19.227800000 0x0006     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0006, Bad FCS
    15 20.414055000 0x0004     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0004, Bad FCS
    16 22.015241000 0x0008     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0008, Bad FCS
    17 22.139594000 0x0006     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0006
    18 23.329769000 0x0004     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0004, Bad FCS
    19 24.941064000 0x0008     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0008, Bad FCS
    20 25.054801000 0x0006     Broadcast     IEEE 802.15.4     39 Data, Dst: Broadcast, Src: 0x0006, Bad FCS
```

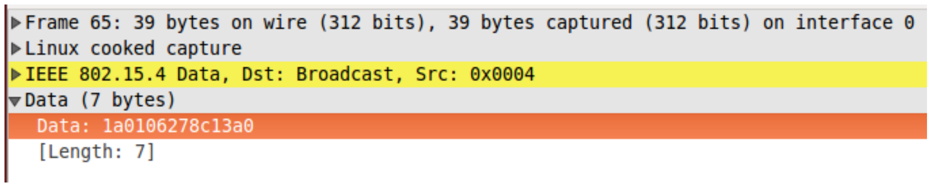
Figure 3.15: The packets going across the STMS L1-Secure network.

The detailed traffic collected by Wireshark for the STMS L1-Secure network is found in the L1-Secure.pcapng file, which can be accessed through our Github project[75]. The same procedures discussed in the previous sample were used to Decode the packets to IEEE 802.15.4 format. Figure 3.9 shows the results of capturing packets going across the STMS L1-Secure network. All of the packets are data packets sent by the End-Devices to the Coordinator. We do not see any Association packets because the L1-Secure version does not require the End-Devices to associate with the Coordinator. The End-Devices will collect temperature levels, encrypt the payload of a packet and "broadcast" the temperature packets on the network. Therefore, any motes within the range of communication will be able to capture the broadcast packets but will not be able to interpret the data without the encryption key. Figure 3.16 shows an example of a mote ”PPPSniffer” that is within the range collecting the packets going across the L1-Secure network. The samples show the hex values found on the payload section of three packets sent by the sensor in Room 4 to the Coordinator. The data is supposed to be the Node ID along with the temperature data; however, the data appears to be random. The payload portion of the packets are being encrypted by the network wide shared key used by the Coordinator and End-Devices. If the PPPSniffer knew the network wide shared key then the eavesdropper would be able to decrypt the traffic in the L1-Secure network.



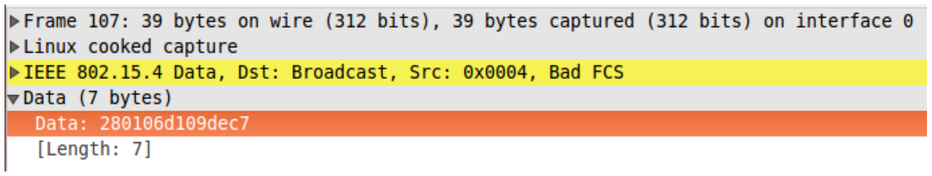
```
▶Frame 44: 39 bytes on wire (312 bits), 39 bytes captured (312 bits) on interface 0
▶Linux cooked capture
▶IEEE 802.15.4 Data, Dst: Broadcast, Src: 0x0004
▼Data (7 bytes)
   Data: 12010657b87b82
   [Length: 7]
```

(a) Sensor 4 Sample 1



```
▶Frame 65: 39 bytes on wire (312 bits), 39 bytes captured (312 bits) on interface 0
▶Linux cooked capture
▶IEEE 802.15.4 Data, Dst: Broadcast, Src: 0x0004
▼Data (7 bytes)
   Data: 1a0106278c13a0
   [Length: 7]
```

(b) Sensor 4 Sample 2



```
▶Frame 107: 39 bytes on wire (312 bits), 39 bytes captured (312 bits) on interface 0
▶Linux cooked capture
▶IEEE 802.15.4 Data, Dst: Broadcast, Src: 0x0004, Bad FCS
▼Data (7 bytes)
   Data: 280106d109dec7
   [Length: 7]
```

(c) Sensor 4 Sample 3

Figure 3.16: An example of the encrypted payload portion found in the packet sent from the End-Device in Room 4 to the Coordinator.

3.4 Chapter Summary

In this chapter, we contributed a working implementation of our Secure Temperature Monitoring System. We discussed our approach of using a star topology, where the central node was the Coordinator and the end nodes were the End-Devices. The STMS design has two versions: one version supporting associations between sensors and the second version supporting encrypted communications. We used TinyOS to implement our STMS as well as the TelosB motes to test our STMS. The association between the Coordinator and End-Device was first established before sending the temperature reading to the Coordinator. The temperature readings are encrypted by the End-Devices before sending them to the Coordinator. We deployed a TelosB mote as a packet sniffer to check the functionality of the STMS. Our results show that our STMS was successfully able to associate sensors that join the network, as well as encrypt the communication within our STMS network.

Chapter 4

Breaking The MSP430-BSL Password

With the Secure Temperature Monitoring System in place, this chapter discusses how the security of a WSM environment can be broken. As mentioned in Chapter 2 there has been numerous proposals presented to improve the security of WSNs; however, most of them are concentrated in the Application, Network, and Link layers [4][9][1][11][14][13][15][12][16][42][17][8][18]. Most of the security research papers make node capture attack assumptions with no details on the outcome of having direct physical access to the sensor node [23][24][25][51][50][52][53][54][55]. Therefore, the purpose of Chapter 4 is to evaluate the time needed to break into a TelosB mote and dump firmware. First, a brief discussion of the two methods used to program MSP430 chips will be reviewed. Next, the password used to protect access to the MSP430 firmware will be analyzed. Lastly, the process of brute forcing the password to get access to MSP430 and dump the firmware will be investigated.

4.1 Programming the MSP430

The MSP430 can be programmed in two different ways. The first way is through the use of JTAG connector, which requires additional hardware. The second and more convenient way is through the use of the USB serial bootstrap loader (BSL).

1. **JTAG**

   The TelosB motes support MSP430 programing through the use of the JTAG 8-pin 2mm connector. In addition to programming, the JTAG connector allows on-chip debugging, single stepping through code, reading from memory, and writing to memory. Texas Instrument provides the FET debugger, which is a USB interface that connects to the JTAG interface to allow programing and debugging MSP430 code [60]. For the most part the JTAG connector is mainly used by developers to test the functionality of their equipment. With that being said, access to the JTAG is unprotected, allowing anyone with a JTAG adapter full access to code store in flash memory or RAM. In our case, the JTAG interface allowed us to access the STMS application and locate the encryption keys used for our secure WSN environment. Furthermore, proprietary firmware can be easily downloaded using the JTAG interface and dumped code can be replicated for personal benefit [57]. To prevent unwanted access to the JTAG interface, the MSP430 provides a feature of fuse blowing. The JTAG port is protected by a fuse; blowing the fuse completely disables the entire JTAG test circuitry in the micro-controller and is irreversible [60]. Once the fuze is blown the only way to program the MSP430 is through the USB bootstrap loader.

2. **BSL**

   Unlike the JTAG interface, access to the USB bootstrap loader (BSL) is password protected, which makes it ideal for after production. Furthermore, the MSP430-BSL interface provides access to flash memory as well as RAM, making the programing task easier by not requiring additional hardware. Even after the JTAG fuse has been blown, the BSL interface continues to function as expected making it optimal for in the field mote maintenance. The MSP430-BSL is a unique code located in a factory-masked boot ROM, which does not allow write or erase access to prevent the BSL code from being altered. The BSL code consists of commands that use the UART protocol with RS232 interfacing to allow communication between a computer and the MSP430 [26]. The key feature of the MSP430-BSL is the restricted commands allowed if the user does not have the password. Table 4.1 shows the commands allowed with and without the password.

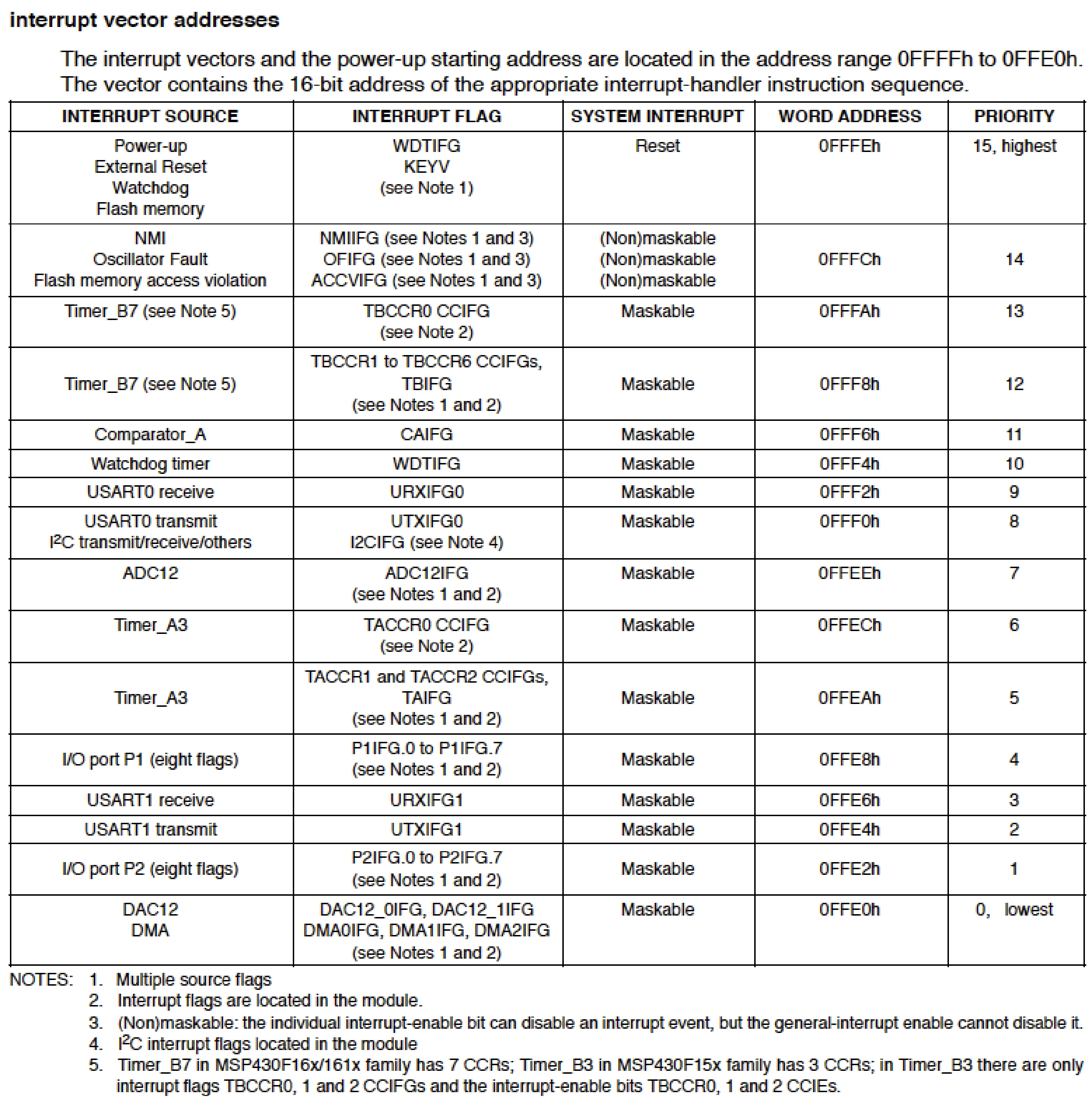
Table 4.1: MSP430-BSL Commands Allowed Without vs. With The Password

Command	Without Password	With Password
Mass Erase	Yes	Yes
Transmit BSL Version	Yes	Yes
Change Baud Rate	Yes	Yes
Receive Password	Yes	Yes
Receive Data Block	No	Yes
Transmit Data Block	No	Yes
Erase Segment	No	Yes
Erase Check	No	Yes
Set Memory Offset	No	Yes
Load Program Counter	No	Yes
Start User Program	No	Yes

In Table 4.1 full access to the MSP430 is allowed only if the user provides the correct password. If no password is provided then the only useful commands allowed to the user is to send the password to the MSP430 or mass erase all of the content in the MSP430 flash memory. The mass erase will allow full access to the mote since the password will be reset to default (details will be provided in Section 4.2). Mass erasing the flash memory defeats the purpose of getting access to the firmware stored in the MSP430 mote. Therefore, providing the correct password is ideal to obtaining meaningful information about the applications stored in sensor nodes. We are interested in the Transmit Data Block command, which is only allowed if the correct password has be entered. The Transmit Data Block command allowed us to dump the firmware found in flash memory, which later can be reverse engineered to get encryption keys used in a secure WSN environment. Furthermore, proprietary firmware can also be obtained if the correct password is entered. We have left details about the password for a dedicated section since the main security flaw of MSP430 is found with the password used for the BSL.

4.2 Analyzing The BSL Password

The MSP430-BSL used a 32-bytes password to protect firmware found in flash memory. As discussed in the previous section, the memory address used for the MSP430-BSL password was the same as the Interrupt Vector Table (IVT); therefore, the BSL password was identical to the IVT. Since the password was the IVT, the BSL password resides at the top of the memory at address 0xFFFE to the memory address 0xFFE0 (refer to Figure 5.1). Furthermore, the 32-bytes password was broken down into sixteen values, each 2-bytes (16-bit addressing) in size. Each of the 2-bytes values corresponds to the address of an interrupt handler. With that being said, Figure 4.1 shows the Interrupt Vector Table for the MSP430F15x, MSP430F16x, and MSP430F161x chips [66]. In order to understand how the MSP430-BSL password is generated we had to understand how the IVT was created. First of all, the address that was generated for each interrupt vector handler was dependent on the application used and the compiler version. For instance, if the application being compiled used various event handlers then the values in the IVT would change accordingly. Furthermore, different compiler versions would generate different address values for each interrupt vector handler. Second of all, we must discuss in detail how the address for each interrupt handler was determined by examining the 16 interrupt handlers supported by the MSP430 chips. Before discussing each interrupt handler, we must highlight some of the data found in Figure 4.1. As we can see from priority column, the highest priority level of 15 was located at the top of the memory address (0xFFFE). The priority level decreased as the memory address decreased all the way up to the lowest priority level of 0, located at the bottom of the memory address (0xFFE0). As we discuss more about each interrupt handler, we will reference the values located in each column of Figure 4.1 [66].



interrupt vector addresses

The interrupt vectors and the power-up starting address are located in the address range 0FFFFh to 0FFE0h. The vector contains the 16-bit address of the appropriate interrupt-handler instruction sequence.

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
Power-up External Reset Watchdog Flash memory	WDTIFG KEYV (see Note 1)	Reset	0FFFEh	15, highest
NMI Oscillator Fault Flash memory access violation	NMIIFG (see Notes 1 and 3) OFIFG (see Notes 1 and 3) ACCVIFG (see Notes 1 and 3)	(Non)maskable (Non)maskable (Non)maskable	0FFFCh	14
Timer_B7 (see Note 5)	TBCCR0 CCIFG (see Note 2)	Maskable	0FFFAh	13
Timer_B7 (see Note 5)	TBCCR1 to TBCCR6 CCIFGs, TBIFG (see Notes 1 and 2)	Maskable	0FFF8h	12
Comparator_A	CAIFG	Maskable	0FFF6h	11
Watchdog timer	WDTIFG	Maskable	0FFF4h	10
USART0 receive	URXIFG0	Maskable	0FFF2h	9
USART0 transmit I²C transmit/receive/others	UTXIFG0 I2CIFG (see Note 4)	Maskable	0FFF0h	8
ADC12	ADC12IFG (see Notes 1 and 2)	Maskable	0FFEEh	7
Timer_A3	TACCR0 CCIFG (see Note 2)	Maskable	0FFECh	6
Timer_A3	TACCR1 and TACCR2 CCIFGs, TAIFG (see Notes 1 and 2)	Maskable	0FFEAh	5
I/O port P1 (eight flags)	P1IFG.0 to P1IFG.7 (see Notes 1 and 2)	Maskable	0FFE8h	4
USART1 receive	URXIFG1	Maskable	0FFE6h	3
USART1 transmit	UTXIFG1	Maskable	0FFE4h	2
I/O port P2 (eight flags)	P2IFG.0 to P2IFG.7 (see Notes 1 and 2)	Maskable	0FFE2h	1
DAC12 DMA	DAC12_0IFG, DAC12_1IFG DMA0IFG, DMA1IFG, DMA2IFG (see Notes 1 and 2)	Maskable	0FFE0h	0, lowest

NOTES: 1. Multiple source flags
2. Interrupt flags are located in the module.
3. (Non)maskable: the individual interrupt-enable bit can disable an interrupt event, but the general-interrupt enable cannot disable it.
4. I²C interrupt flags located in the module
5. Timer_B7 in MSP430F16x/161x family has 7 CCRs; Timer_B3 in MSP430F15x family has 3 CCRs; in Timer_B3 there are only interrupt flags TBCCR0, 1 and 2 CCIFGs and the interrupt-enable bits TBCCR0, 1 and 2 CCIEs.

Figure 4.1: The interrupt vector table for the MSP430F15x, MSP430F16x, and MSP430F161x chips.

- *0xFFFE: WDTIFG, KEYV*

  The watchdog timer interrupt flag (WDTIFG) is responsible for handling any system resets due to any security key violations (KEYV), reset on VCC power-on, or a reset on application malfunction [60].

- *0xFFFC: NMIIFG, OFIFG, ACCVIFG*

  The non-maskable interrupt flag (NMIIFG) is responsible for handling non-recoverable hardware errors. The oscillator fault interrupt flag (OFIFG) is responsible for handling any oscillator rate errors. The flash memory access violation interrupt flag (ACCVIFG) is responsible for handling any erase or programming requests sent while the flash memory is busy [76].

- *0xFFFA: TBCCR0 CCIFG*

  The Timer_B7 is one of the two general purpose 16-bit counters that counts up until the value in the TBCCR0 register is reached. The capture/compare interrupt flag (CCIFG) is set once the counter has reached the value store in TBCCR0 register. The CCIFG will be set until the interrupt request has been handled and will be reset to 0 after completion[77].

- *0xFFF8: TBCCR1 to TBCCR6 CCIFGs, TBIFG*

  The Timer_B7 is one of the two general purpose 16-bit counters that counts up until the value in either TBCCR1 through TBCCR6 registers is reached. The capture/compare interrupt flag (CCIFG) is set once the counter has reached the value store in either TBCCR1 through TBCCR6 registers. The CCIFG will be set until the interrupt request has been handled and will be reset to 0 after completion. The timer rollover interrupt flag (TBIFG) will be set when the timer counts down from the value in TACCR0 register to 0 [77].

- *0xFFF6: CAIFG*

  The comparator_a interrupt flag (CAIFG) is responsible for handing any analog-to-digital conversions, battery-voltage supervision, and monitoring of external analog signals [60].

- *0xFFF4: WDTIFG*

  The watchdog timer interrupt flag (WDT) is responsible for handling any system resets software problem that occurs. The watchdog timer can also be configured to serve as an interval timer and will generate interrupts at selected time intervals[66].

- *0xFFF2: URXIFG0*

  The UART and SPI receive interrupt flags (URXIFG0) are responsible for handling serial data communication through the universal synchronous/asynchronous receiver peripheral. The USART0 supports synchronous SPI, asynchronous UART and I2C communication protocols using the receive buffer channel[66].

- *0xFFF0: UTXIFG0 I2CIFG*

  The UART and SPI transmit interrupt flags (URXIFG0) are responsible for handling serial data communication through the universal synchronous/asynchronous transmit peripheral.

The USART0 supports synchronous SPI, asynchronous UART and I2C communication protocols using the transmit buffer channel [66]. The I2C interrupt flag is set when I2C is being used for the serial data communication.

- *0xFFEE: ADC12IFG*

  The ADC12 interrupt vector flag (ADC12IFG) is responsible for handling efficient analog-to-digital conversions through the use of a 16 work conversion-and-control buffer[66]

- *0xFFEC: TACCR0 CCIFG*

  The Timer_A3 is one of the two general purpose 16-bit counters that counts up until the value in the TACCR0 register is reached. The capture/compare interrupt flag (CCIFG) is set once the counter has reached the value store in TACCR0 register. The CCIFG will be set until the interrupt request has been handled and will be reset to 0 after completion[77].

- *0xFFEA: TACCR1 and TACCR2 CCIFGs, TAIFG*

  The Timer_A3 is one of the two general purpose 16-bit counters that counts up until the value in either TACCR1 or TACCR2 registers is reached. The capture/compare interrupt flag (CCIFG) is set once the counter has reached the value store in any of the TACCR1 or TACCR2 registers. The CCIFG will be set until the interrupt request has been handled and will be reset to 0 after completion. The timer rollover interrupt flag (TAIFG) will be set when the timer counts down from the value in TACCR0 register to 0 [77].

- *0xFFE8: P1IFG.0 to P1IFG.7*

  The port 1 interrupt flag (P1IFG.#) is responsible for handling general Input and Output interrupts. Each interrupt flag has the I/O following description [66]:

  - IFG.0 - clock signal TACLK input
  - IFG.1 - capture: CCI0A input and compare: Out0 output/BSL transmit
  - IFG.2 - capture: CCI1A input, compare: Out1 output,
  - IFG.3 - capture: CCI2A input, compare: Out2 output
  - IFG.4 - signal output
  - IFG.5 - compare: Out0 output
  - IFG.6 - compare: Out1 output
  - IFG.7 - compare: Out2 output

- *0xFFE6: URXIFG1*

  The UART and SPI receive interrupt flags (URXIFG1) are responsible for handling serial data communication through the universal synchronous/asynchronous receiver peripheral. The USART1 supports synchronous SPI and asynchronous UART communication protocols using the receive buffer channel[66].

- *0xFFE4: UTXIFG1*

  The UART and SPI transmit interrupt flags (URXIFG1) are responsible for handling serial data communication through the universal synchronous/asynchronous transmit peripheral. The USART1 supports synchronous SPI and asynchronous UART communication protocols using the transmit buffer channel[66].

- *0xFFE2: P2IFG.0 to P2IFG.7*

  The port 2 interrupt flag (P2IFG.#) is responsible for handling general Input and Output interrupts. Each interrupt flag has the I/O following description [66]:

  – IFG.0 - output

  – IFG.1 - clock signal at INCLK

  – IFG.2 - capture: CCI0B input/Comparator_A output/BSL receive

  – IFG.3 - compare: Out1 output/Comparator_A input

  – IFG.4 - compare: Out2 output/Comparator_A input

  – IFG.5 - external resistor defining the DCO nominal frequency

  – IFG.6 - 12-bit ADC/DMA channel 0 external trigger

  – IFG.7 - compare: Out0 output

- *0xFFE0: DAC12_0IFG, DAC12_1IFG, DMA0IFG, DMA1IFG, DMA2IFG*

  The DAC12 interrupt flags (DAC12_0IFG and DAC12_1IFG) are responsible for handling any digital-to-analog conversion and are used in conjunction with the DMA controller. The DMA interrupt flags (DMA0IFG, DMA1IFG and DMA2IFG) are responsible for handling any movement of data from one memory address to another without CPU intervention[66].

4.3 Breaking The BSL Password

The following section will be broken down into four subsections in order to help the reader understand the process of breaking the MSP430-BSL password. The first section will discuss the time it takes to brute force the 32-bytes MSP430-BSL password. The second section will discuss how a sample of passwords were collected using various sensor applications. The third section will discuss the technique used in this thesis to predict the MSP430-BSL password. Lastly, the fourth section will discuss the time it takes to break the MSP430-BSL password using our password prediction technique.

4.3.1 Brute Forcing The MSP430-BSL 32-bytes Password

To brute force the 32-bytes (256-bit) MSP430-BSL password required a significant amount of time. A key that is 256-bits in length has a key space of 2^{256} or 1.1579×10^{77} possible keys. Brute forcing a 256-bit key space is unreasonable but for the sake of argument we evaluated how long it takes to brute force the MSP430-BSL 32-bytes password using our TelosB motes. To demonstrate how long it takes to check a password we used our IEEE802154 Coordinator application as an example. The correct password for our tests is the IVT table generated by the compiler when building out IEEE802154 Coordinator application. We implemented a python script that calculated the time it took to check the correct password vs the incorrect password with our TelosB mote. The pwd-time-check.py script can be accessed through our Github project [75]. Figure 4.2 shows the time it took to check 1, 200, 400, 600, and 1000 correct passwords and incorrect passwords. The data presented in the graph is an average time of 10 trials performed to check the correct password (Table A.1) and 10 trials to check the incorrect password (Table A.2). The results show the time to guess a correct password vs a incorrect password is equivalent. Our results are roughly the same as the ones presented by Becher et al. [27].

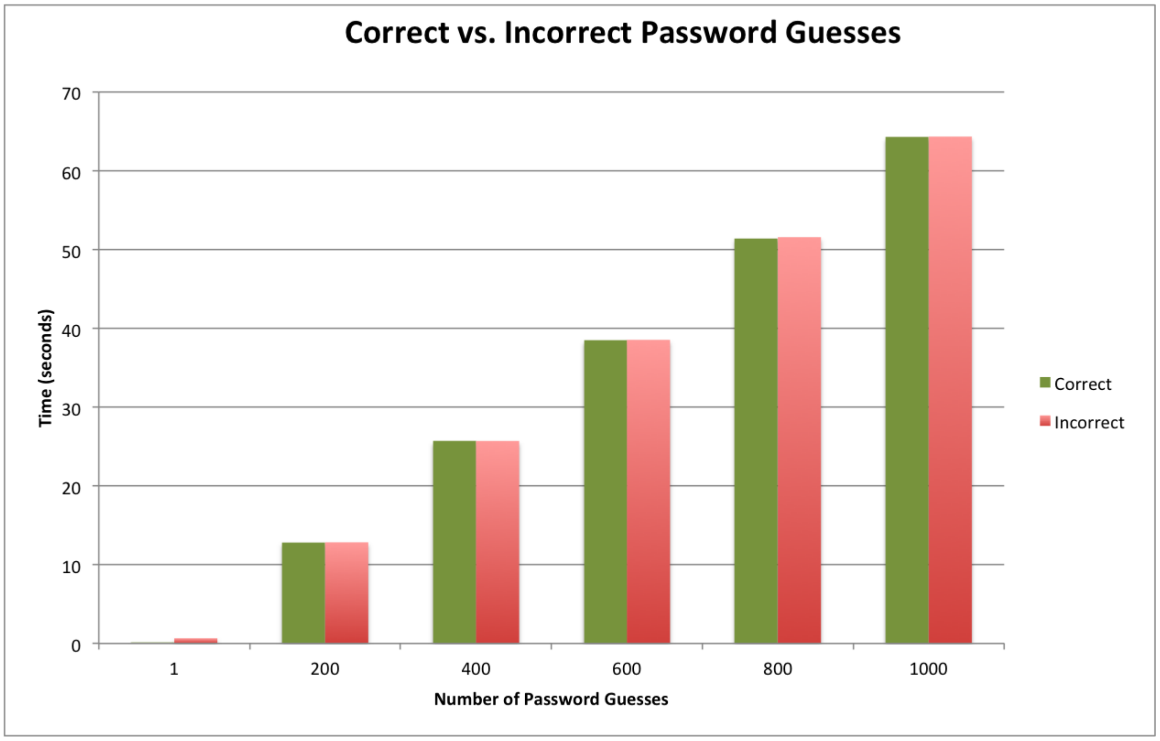


Figure 4.2: The bar graph above displays the time it takes in seconds to guess N number of correct passwords and incorrect passwords.

The results show us that on average the TelosB mote is able to check 15 passwords per second regardless if the password is correct or incorrect. However, we were able to increase the passwords check significantly by changing the baud rates. The results in Figure 4.2 are an average of checking the passwords using the default BSL baud rate of 9600. Our TelosB mote supported baud rates up to 38400; therefore, significantly increasing the number of passwords checked per second. Using the same pwd-time-check.py script we added a code to change the default baud rate of 9600 to the maximum supported TelosB baud rate of 38400. Figure 4.3 shows the comparison of using the default baud rate of 9600 to a faster TelosB baud rate of 38400. Similar procedures were followed by checking 1, 200, 400, 600, and 1000 passwords with 10 trials (Table A.3). The results show us that on average the TelosB mote using a baud rate 38400 was able to check 38 passwords per second regardless if the password was correct or incorrect. That is 23 passwords more per second than using the default baud rate of 9600. Our results are roughly the same as the ones presented by Goodspeed [28].

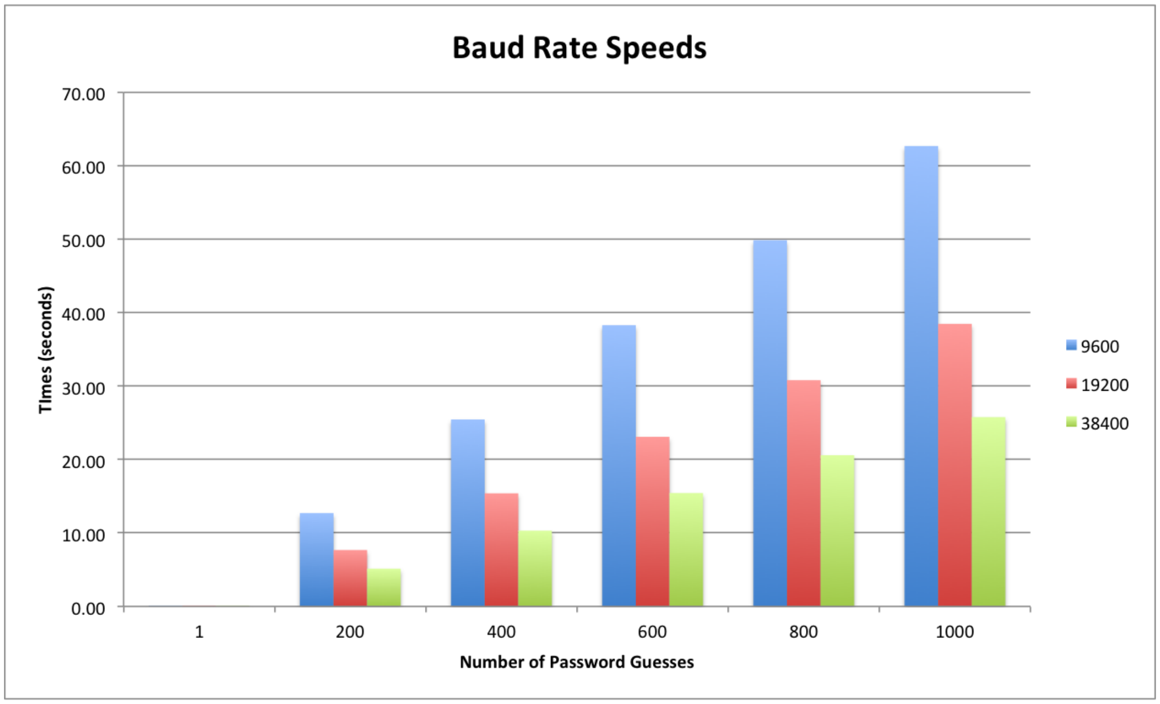


Figure 4.3: The bar graph above display the time it takes in seconds to guess N number of correct passwords using different baud rates (9600 vs 19200 vs 38400).

With that being said, using a baud rate of 38400 allowed us to guess 38 passwords per second or 2280 passwords per minute. To brute force a key space of 2^{256} would require 9.66×10^{67} (4.1) years! Becher et al. [27] was actually able to reduce the 9.66×10^{67} years to 128 years by evaluating MSP430-BSL password bits. The 256-bit password was reduced to a 40-bit password by considering that MSP430 instructions must be even-aligned (240-bit), the reset vector always points to the beginning of main memory (225-bit), not all interrupts are used (60-bits), and assuming that the program uses 2kb of flash memory (40-bits)[27]. Furthermore, Goodspeed[28] was able to reduce the 128 years to 32 years by making use of the 38400 baud rate. However, 32 years is still not practical and further key space reduction is necessary to make the breaking of the MSP430-BSL password plausible.

$$((((2^{256} \div 2280) \div 60) \div 24) \div 365) \qquad or \qquad 9.66 \times 10^{67} \tag{4.1}$$

4.3.2 Generating Password Samples

The best way to reduce the brute force times for the MSP430-BSL password is to use a pattern/prediction technique instead of reducing the bits. In order to predict a password we first had to collect password samples to analyze and evaluate. Since the MSP430-BSL password is identical to IVT, then we needed various MSP430 sensor applications to generate password samples. Luckily for us, TinyOS provides numerous sensor applications under their /tinyos-main/apps directory[68]. A bash script was implemented to iterate through all the applications under the /apps directory to build each application. Building the application generates an Intel HEX (main.ihex) file, which was the binary representation of the application in ASCII text format. The main.ihex file contains the IVT; therefore, the bash script uses a python script to dump the IVT from the main.ihex file. Both the bash script generate-pwds.sh and the pythons script dump-pwd.py can be accessed through our Github project [75].

Unfortunately, out of the 200 TinyOS application samples, only 93 were able to be successfully compiled and build. There are three main reasons why the other 107 applications did not compile. First of all, not all applications were properly ported from TinyOS 1.x to TinyOS 2.x; therefore, some applications have compile-time errors [78]. Second of all, the TelosB motes only have 48kb of flash memory; therefore, if the application was bigger than 48kb then an error occurred. Third, not all applications were built for the TelosB mote; therefore, a few of them were solely meant for MICA motes. With that being said, the password sample size of 93 applications showed promising results for future password predictions.

4.3.3 Password Pattern Algorithm

The following section proposes the algorithm used to predict future passwords and reduce brute force times for the MSP430-BSL password. The 93 passwords obtained from the generate-pwds.sh script were used to create the password pattern algorithm. Before proposing our algorithm it is important to first discuss how the 32-bytes password is identical to the IVT. The dump-pwd.py script outputs a copy of the data stored at address 0xFFE0 to 0xFFFF. The 32 bytes copied from the the address range is the MSP430-BSL password or IVT. The following 32 bytes dumped correspond to the TinyOS Blink application.

```
0xe2 0x43 0xe2 0x43 0xe2 0x43 0xe2 0x43 0xe2m 0x43 0x7c 0x49 0x5e 0x49 0xe2 0x43
0xe2 0x43 0xe2 0x43 0xe2 0x43 0xe2 0x43 0xc0 0x49 0xa2 0x49 0xe2 0x43 0x00 0x40
```

The 32-bytes map to the 16 interrupts found in the IVT of the MSP430 chips. With that being said, the 32-bytes can be broken down into sixteen 2-bytes blocks. The values in each block corresponds to the address of the Interrupt Service Routine (ISR). The values must be translate to little endian to obtain the corresponding memory address. Table 4.2 shows the conversion of the 32-bytes to the corresponding IVT entry for the Blink TinyOS application. As we can see from Table 4.2, 11 out of 16 IVT entries are duplicates. These entries are duplicates because they point to the same address in memory. The IVT entries point to the same address in memory because those interrupts are unused.

Applications that have unused interrupts have lower brute force times because the password size is 12-bytes (6 IVT entries, 2-bytes each) or 96-bits. The password size of 96-bits is still not practical and not all applications have 11 unused interrupts.

Interrupt Flag	**Address**	**Priority**	**Value**
WDTIFG, KEYV	0xFFFE	15	0x4000
NMIIFG, OFIFG, ACCVIFG	0xFFFC	14	0x43e2
TBCCR0 CCIFG	0xFFFA	13	0x49a2
TBCCR1 to TBCCR6 CCIFGs, TBIFG	0xFFF8	12	0x49c0
CAIFG	0xFFF6	11	0x43e2
WDTIFG	0xFFF4	10	0x43e2
URXIFG0	0xFFF2	9	0x43e2
UTXIFG0 I2CIFG	0xFFF0	8	0x43e2
ADC12IFG	0xFFEE	7	0x43e2
TACCR0 CCIFG	0xFFEC	6	0x495e
TACCR1 and TACCR2 CCIFGs, TAIFG	0xFFEA	5	0x497c
P1IFG.0 to P1IFG.7	0xFFE8	4	0x43e2
URXIFG1	0xFFE6	3	0x43e2
UTXIFG1	0xFFE4	2	0x43e2
P2IFG.0 to P2IFG.7	0xFFE2	1	0x43e2
DAC12 0IFG, DMA2IFG	0xFFE0	0	0x43e2

Table 4.2: A sample of the IVT for the Blink application

For example, the PPPSniffer application provided by TinyOS had less unused interrupts compared to the Blink application. Table 4.3 shows the IVT values for the PPPSniffer TinyOS application. As we can see from Table 4.3, 7 out of 16 IVT entries are unused , which shortens the password brute force size to 20-bytes (10 IVT entries, 2-bytes each), or 160 bits.

Interrupt Flag	**Address**	**Priority**	**Value**
WDTIFG, KEYV	0xFFFE	15	0x4000
NMIIFG, OFIFG, ACCVIFG	0xFFFC	14	0x4680
TBCCR0 CCIFG	0xFFFA	13	0x71d0
TBCCR1 to TBCCR6 CCIFGs, TBIFG	0xFFF8	12	0x71ee
CAIFG	0xFFF6	11	0x4680
WDTIFG	0xFFF4	10	0x4680
URXIFG0	0xFFF2	9	0x73c8
UTXIFG0 I2CIFG	0xFFF0	8	0x743a
ADC12IFG	0xFFEE	7	0x4680
TACCR0 CCIFG	0xFFEC	6	0x714e
TACCR1 and TACCR2 CCIFGs, TAIFG	0xFFEA	5	0x7168
P1IFG.0 to P1IFG.7	0xFFE8	4	0x7214
URXIFG1	0xFFE6	3	0x4680
UTXIFG1	0xFFE4	2	0x4680
P2IFG.0 to P2IFG.7	0xFFE2	1	0x7316
DAC12 0IFG, DMA2IFG	0xFFE0	0	0x4680

Table 4.3: A sample of the IVT for the PPPSniffer application

With that being said, it was important to test whether there was a correlation between the number of unused interrupts and the code size of the application. One would assume the bigger the application the more likely interrupts would be used. We wrote a script to convert the 32-bytes passwords to their corresponding IVT entries (little endian) for each application. Due to the size of the password sample, the table with all the IVT entries was not added to the Appendix A; however, the results along with the convert-endianess.py script can be access through our Github project [75]. Using the results from the convert-endianess.py script, we wrote a python script that iterated through all the applications and determined the number of unused interrupts for each application. Furthermore, an additional bash script (get-apps-size.sh) was implemented to iterate through all the TinyOS applications and obtain the code size of each application. Table A.4 shows the results of the scripts in descending order (from largest to smallest) for code size. The results show us the assumption made was incorrect and there was no correlation between code size and the number of unused interrupts. For instance, /apps/TCPEcho application was 35-kB in size and had 7 unused interrupts, where as the /apps/tests/storage/block application was 9-kB is size and had the same number of unused interrupts. The relation between the code size of an application to the number of unused interrupts was not coherent and suggests that interrupts are only used if the application specifically makes use of the interrupt. Applications that have unused interrupts point to an address in memory with the following instruction:

```
address: br #0x4321
...
0x4321: reti
```

The designated memory address simply executes the RETI instruction to end an interrupt service routine; hence, the reason why we see IVT entry duplicates regardless of the code size. The next step was to determine exactly how many applications have N number of unused interrupts. Table 4.4 shows the number of applications that have either eleven, ten, nine, seven, six, five or four unused interrupts. We will reference Table 4.4 later on in this section to help explain the password pattern algorithm used to reduce brute force times for the MSP430-BSL password.

Table 4.4: The number of unused interrupts with the number of applications

Unused Interrupts	Applications
Eleven	13
Ten	3
Nine	9
Seven	47
Six	2
Five	12
Four	7
Total Applications	93

Even though the unused interrupts shortens the key space it is still not practical to brute force without determining patterns found in the IVT entries. To obtain more intel about the IVT entries, we implemented our password pattern algorithm using python. Our password pattern algorithm consists of first sorting the addresses found in each IVT entry and second taking the differences between the sorted addresses. Since the values stored in each IVT entry point to an ISR address then the output of the sort-order.py script is the ISR addresses in ascending order. The python script also provided the corresponding IVT entry position to assist in maintaining the order of the IVT entries. Keeping track of the IVT entries order is very important since the order of the interrupts matters when reconstructing the password. The rest of this section will discuss how the sorted IVT entry values were used to our advantage to reduce the brute force time of the MSP430-BSL password.

First, sorting the addresses found in the IVT entries showed us which interrupts were most commonly unused in our applications. Figure 4.4 shows the interrupts that were unused in our applications sample. The IVT entry 15 always points to the start of the program (0x4000); thus, it will always be the same for all applications. The IVT entries 0, 10, 11 and 14 were never used by the applications we analyzed (refer to Figure 4.5). The IVT entries 5, 6, 12, and 13 were always used by applications we analyzed and corresponded to the Timer_B and Timer_A interrupt handlers. The results make sense since TinyOS is an event-driven operating system that relies heavily on triggers such as timers[79].

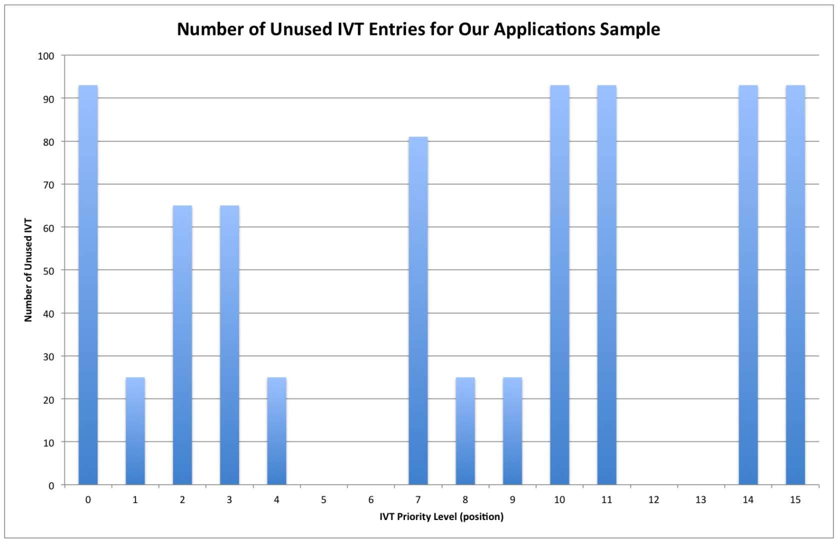


Figure 4.4: The number of unused IVT entries for our application sample size.

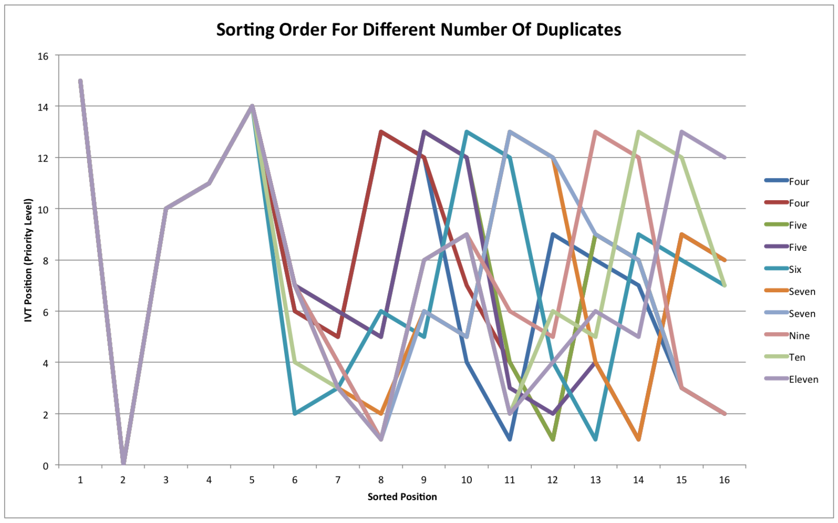


Figure 4.5: Patterns found when sorting the IVT entries.

Second, using the sorted addresses of the IVT entries we analyzed the differences between two consecutive addresses. We implemented a python script (IVT-Entry-Diff.py) that iterated through all the sorted addresses in the IVT for each application and took differences between the addresses. The results of the script showed significant consistency between applications when evaluating the differences between IVT entry values. The consistency between addresses found in the IVT entries allowed use to significantly decrease the brute force time of the MSP430-BSL password.

4.3.4 Brute Forcing Using Password Prediction

Before evaluating the consistent differences between addresses found in the IVT entries we must point out that the order of the IVT entires matter. Since the positions of the IVT entries changed after sorting, the order of the new sorted IVT entry positions are important when reconstructing the IVT or password. To demonstrate our theory of reducing the brute force time for the MSP430-BSL password we provide experiment results of 3 different categories of unused interrupts. The categories are 7 unused interrupts (Category 1), 11 unused interrupts (Category 2) and 5 unused interrupts (Category 3).

Experiment 1

Experiment 1 consisted of applications that had 7 unused interrupts and the applications sample size is 47 applications. Most of the applications of our sample fell within Category 1; therefore, having a more promising brute force success. The results of the IVT-Entry-Diff.py script showed us a consistency difference between addresses found in the IVT entries of applications that have 7 unused interrupts . Figure 4.6 shows an example of the differences between the sorted addresses found in the IVT entries of 10 applications. For example, for App42 (/apps/tests/TestAM), the value stored in IVT entry 15 (p15) is 0x4000 and the value stored in IVT entry 0 (p0) is 0x450E then the difference between p0 and p15 is 0x50E (1294). We also see that the IVT entries p0, p10, p11, p14, p7, p3, and p2 are the IVT entry duplicates (unused interrupts) because the difference between these values will always be 0.

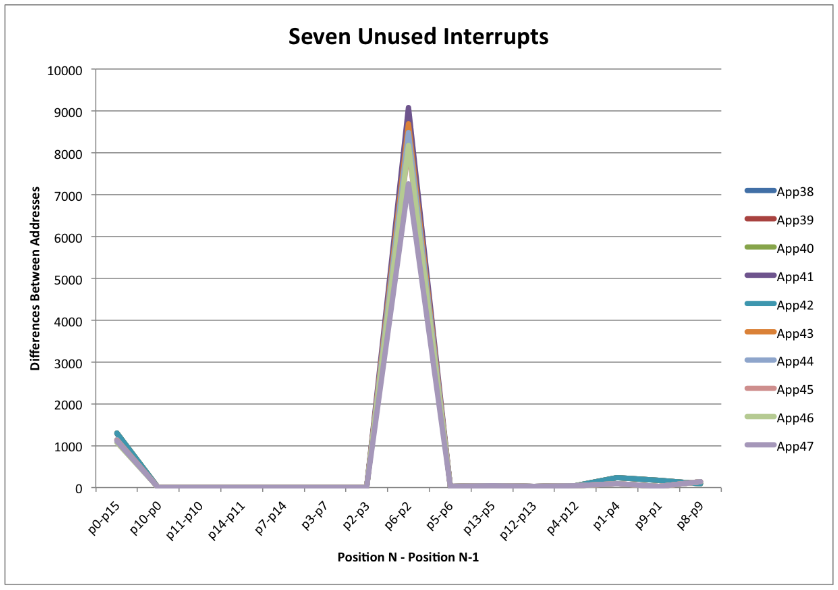


Figure 4.6: The patterns found when taking the difference between two consecutive sorted address found in the IVT entries

To predict the value stored in IVT entry 0 as well as all the other unused interrupts, we evaluated the differences found between entry 0 and entry 15 with all the 47 applications. Figure 4.7 shows results of taking the difference between entry 0 and entry 15. As we can see from the results, the difference between the address stored in entry 0 and the address stored in entry 15 (0x4000) will always fall within the range of 1,000 (0x3E8) to 2,000 (0x7D0). Therefore, to predict the address in IVT entry 0 we must brute force all possible even numbers between 0x43E8 and 0x47D0. Since the address had to be an even number we had 500 possible combinations to try when brute forcing values for IVT entries p0, p10, p11, p14, p7, p3, and p2.

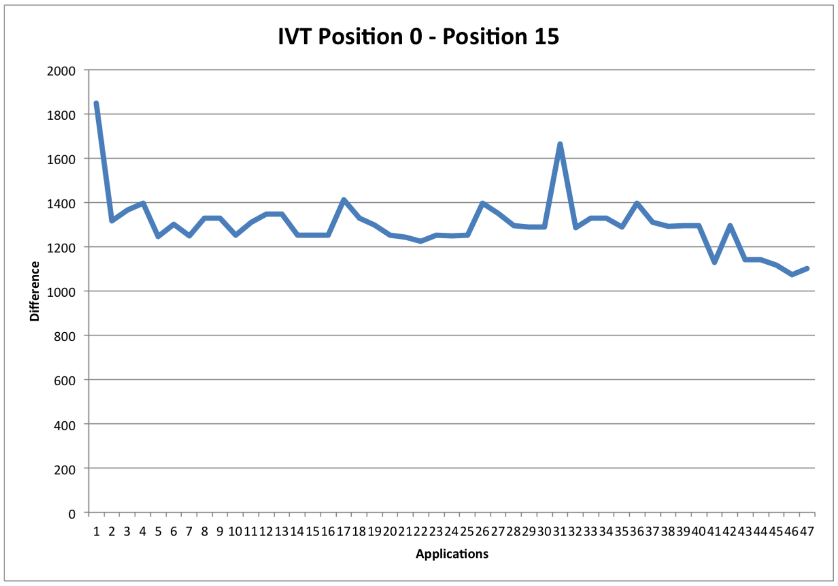


Figure 4.7: The patterns found when taking the difference between IVT entry 0 and entry 15

The next value that is not a duplicate in Figure 4.6 is the difference between IVT entry 6 and IVT entry 2. Figure 4.7 shows the results of taking the difference between entry 6 and entry 2. As we can see from the results the difference between the address stored in entry 6 and the address stored in entry 2 will always fall within the range of 5,000 (0x1388) to 30,000 (0x7530). Furthermore, we can see the difference between the p6 to p2 decrease from application 1 to 47 that is because the applications were sorted from largest to smallest in code size. Therefore, since all but one application that have 7 unused interrupts are roughly 10kB in size (reference Table A.4), then it is safe to say the smallest application with 7 unused interrupts would be 9kB in size. We can further define the range to brute force the value in IVT entry 6 from 7,000 (0x1B58) to 29,000 (0x7148). Since the address had to be an even number we had 11,000 possible combinations to try when brute forcing values for IVT entry 6.

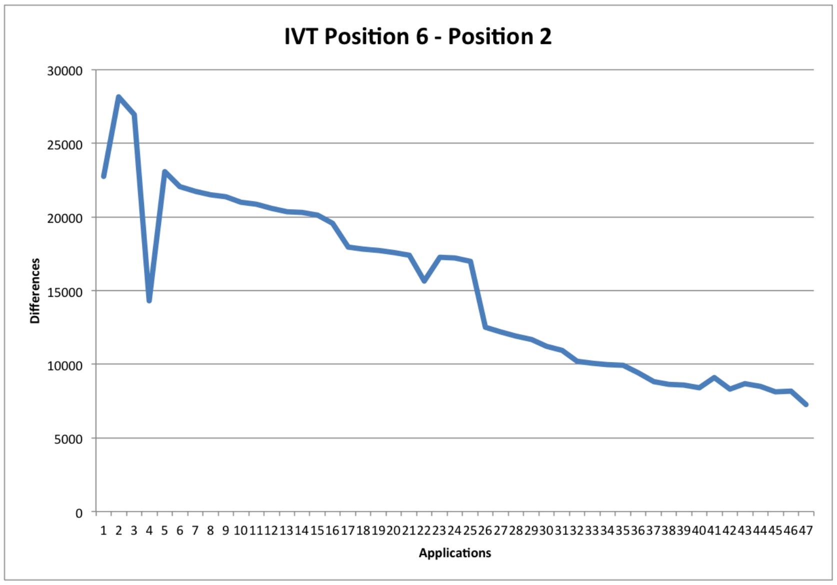


Figure 4.8: The patterns found when taking the difference between IVT entry 6 and entry 2

The next value in Figure 4.6 is the difference between IVT entry 5 and IVT entry 6. In Figure 4.9 the point blue shows results of taking the difference between entry 5 and entry 6. As we can see from the results, the difference between the address stored in entry 5 and the address stored in entry 6 will always be 30 (0x1E). Therefore, we would only have to add 30 (0x1E) to the address stored in the IVT entry 6 in order to obtain the address in IVT entry 5. We had 1 possible combination to try when brute forcing values for IVT entry 5. The next value in Figure 4.6 is the difference between IVT entry 13 and IVT entry 5. In Figure 4.9 point red shows results of taking the difference between entry 13 and entry 5. As we can see from the results, the difference between the address stored in entry 13 and the address stored in entry 5 will always be 38 (0x26). Therefore, we would only have to add 38 (0x26) to the address stored in the IVT entry 5 in order to obtain the address in IVT entry 13. We had 1 possible combination to try when brute forcing values for IVT entry position 13. The next value in Figure 4.6 is the difference between IVT entry 12 and IVT entry 13. In Figure 4.9 point green shows results of taking the difference between entry 12 and entry 13. As we can see from the results, the difference between the address stored in entry 12 and the address stored in entry 13 will always be 30 (0x1E). Therefore, we would only have to add 30 (0x1E) to the address stored in the IVT entry 13 in order to obtain the address in IVT entry 12. We had 1 possible combination to try when brute forcing values for IVT entry position 12. The next value in Figure 4.6 is the difference between IVT entry 4 and IVT entry 12. In Figure 4.9 point purple shows results of taking the difference between entry 4 and entry 12. As we can see from the results, the difference between the address stored in entry 4 and the address stored in entry 12 will always be 38 (0x26). Therefore, we would only have to add 38 (0x26) to the address stored in the IVT entry 12 in order to obtain the address in IVT entry 12. We had 1 possible combination to try when brute forcing values for IVT entry 4.

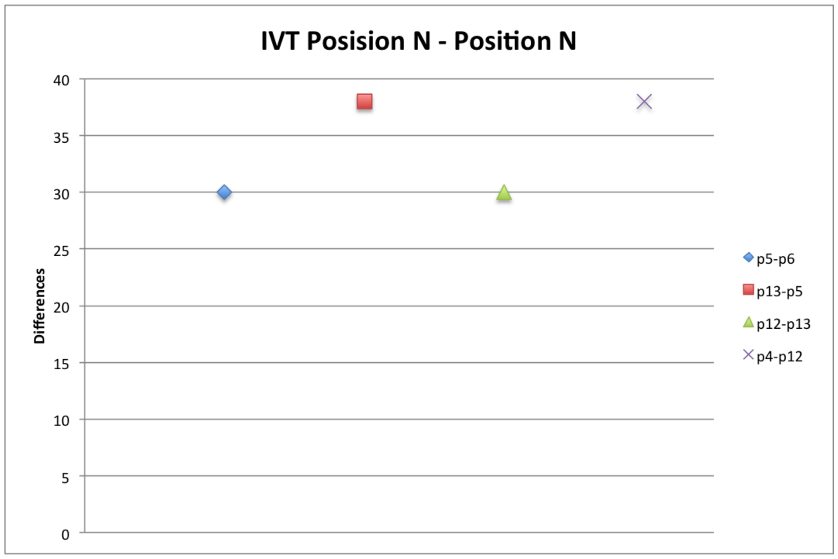


Figure 4.9: The patterns found when taking the difference between the next four values of IVTs

The next value in Figure 4.6 is the difference between IVT entry 1 and IVT entry 4. Figure 4.10 shows results of taking the difference between entry 1 and entry 4. As we can see from the results, the difference between the address stored in entry 1 and the address stored in entry 4 will always fall within the range of 240 (0xF0) to 265 (0x109). Therefore, we would only have to add between 240 (0xF0) and 265 (0x109) to the address stored in the IVT entry 4 in order to obtain the address in IVT entry 1. Since the address had to be a even number we had 12 possible combinations to try when brute forcing values for IVT entry 4.

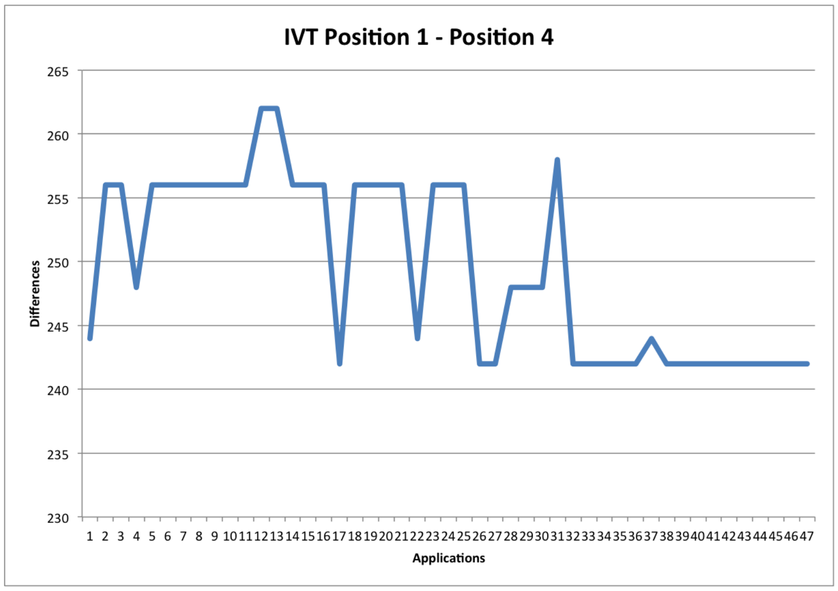


Figure 4.10: The patterns found when taking the difference between IVT entry 1 and entry 4

The next value in Figure 4.6 is the difference between IVT entry 9 and IVT entry 1. Figure 4.11 shows results of taking the difference between entry 9 and entry 1. As we can see from the results, the difference between the address stored in entry 9 and the address stored in entry 1 will always be either 174 (0xAE) or 178 (0xB2). Therefore, we would only have to add either 174 (0xAE) or 178 (0xB2) to the address stored in the IVT entry 1 in order to obtain the address in IVT entry 9. We had 2 possible combinations to try when brute forcing values for IVT entry 9.

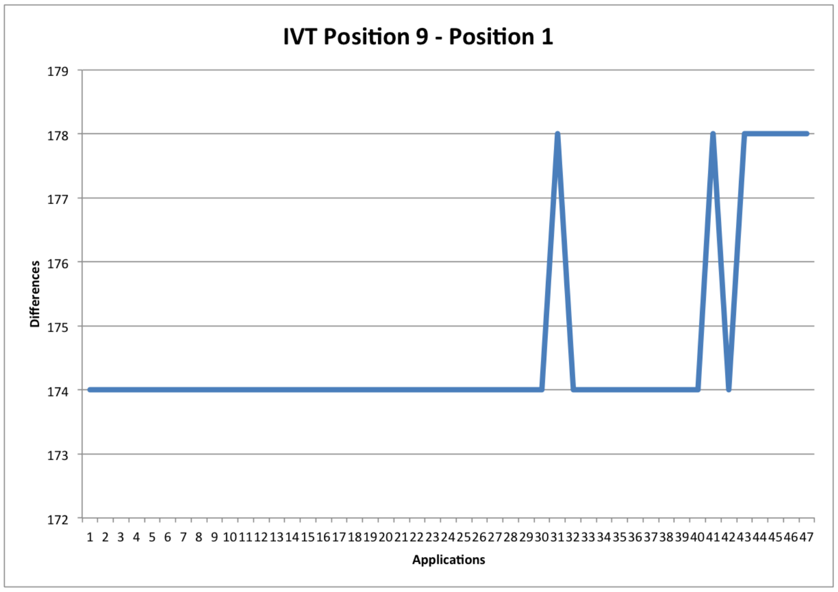


Figure 4.11: The patterns found when taking the difference between IVT entry 9 and entry 1

The next value in Figure 4.6 is the difference between IVT entry 8 and IVT entry 9. Figure 4.12 shows results of taking the difference between entry 8 and entry 9. As we can see from the results, the difference between the address stored in entry 8 and the address stored in entry 9 will always be either 62 (0x3E) or 92 (0x5C). Therefore, we would only have to add either 62 (0x3E) or 92 (0x5C) to the address stored in the IVT entry 9 in order to obtain the address in IVT entry 8. We had 2 possible combinations to try when brute forcing values for IVT entry 8.

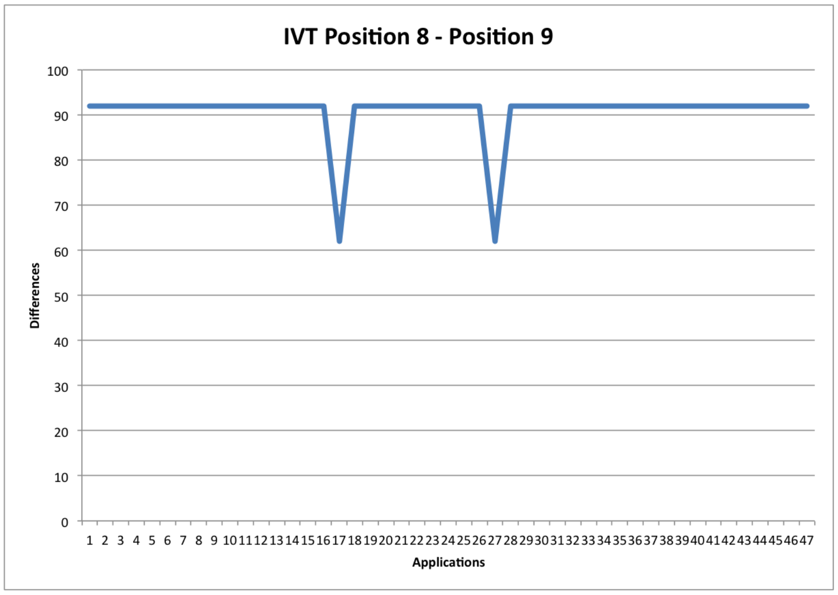


Figure 4.12: The patterns found when taking the difference between IVT entry 8 and entry 9

The analysis results show us that the key space for applications that had 7 unused interrupts would roughly be 264,000,000 (4.2) possibilities. Using the baud rate of 38400, a total of 2,280 passwords can be brute forced in 1 minute (refer to Figure 4.3). With that being said, to brute force the MSP430-BSL password with applications that had 7 unused interrupts it would take 80 (4.3) days.

$$500 \times 11000 \times 1 \times 1 \times 1 \times 1 \times 12 \times 2 \times 2 \qquad or \qquad 264,000,000 \tag{4.2}$$

$$(((264000000 \div 2280) \div 60) \div 24) \qquad or \qquad 80 \tag{4.3}$$

Experiment 2

Experiment 2 consisted of applications that had 11 unused interrupts and the application sample size used was 13 applications. The results of the IVT-Entry-Diff.py script showed us a consistent difference between addresses found in the IVT entries of applications that had 11 hlunused interrupts. Figure 4.13 shows the results of differences found between the sorted addresses of 13 applications.

We can see see that the IVT entries p0, p10, p11, p14, p7, p3, p2, p1, p8, p9 and p4 are the unused interrupts because the difference between these values are always 0. Using the same approach used in Experiment 1, we evaluated the differences between addresses found in the IVT entry where there were spikes because of a significant difference. The differences found between the address stored in position 0 and the address stored in position 15 (0x4000) will fall within the range of 100 (0x64) to 1,500 (0x5DC). Since the address had to be an even number we had 750 possible combinations to try when brute forcing values for IVT entries p0, p10, p11, p14, p7, p3, p2, p1, p8, p9 and p4. Next, the differences found between the address stored in IVT Entry position 6 and the address stored in IVT Entry position 4 will fall within the range of 200 (0xC8) to 9,000 (0x2328). Since the address had to be an even number then we have 4400 possible combinations to try when brute forcing values for IVT entry position 6. Next the differences found between the address stored in IVT entry 5 and the address stored in IVT entry 6 will either be 158 (0x9E) or 30 (0x1E). We had 2 possible combinations to try when brute forcing values for IVT entry 5. Next, the differences found between the address stored in IVT entry 13 and the address stored in IVT entry 5 will always be 38 (0x26). We had 1 possible combination to try when brute forcing values for IVT entry 13. Next, the differences found between the address stored in IVT entry 12 and the address stored in IVT entry 13 will always be 30 (0x1E). We had 1 possible combination to try when brute forcing values for IVT entry 12. The analysis results show us that the key space for applications that had 11 unused interrupts would roughly be 6,600,000 (4.4) possibilities. With that being said, to brute force the MSP430-BSL password with applications that have 11 unused interrupts it would take 2 (4.5) days.

$$750 \times 4400 \times 2 \times 1 \times 1 \qquad or \qquad 6,600,000 \tag{4.4}$$

$$(((6600000 \div 2280) \div 60) \div 24) \qquad or \qquad 2 \tag{4.5}$$

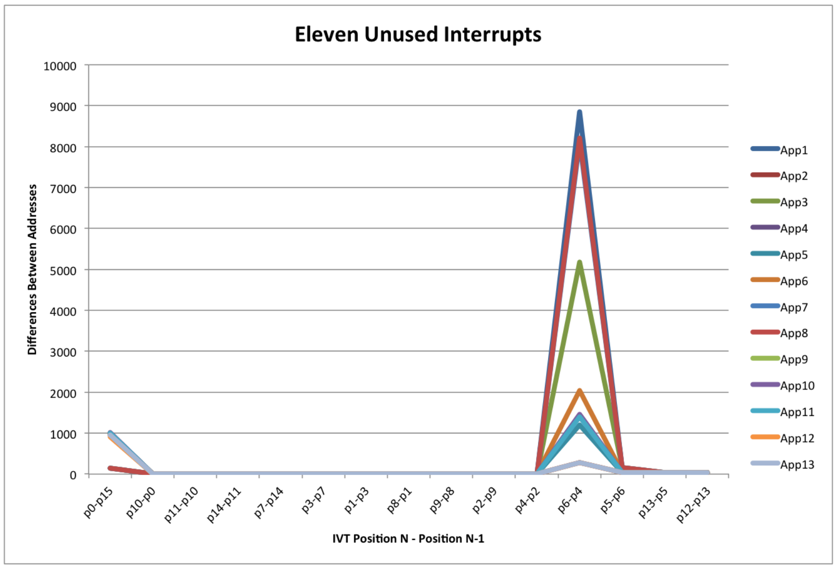


Figure 4.13: The patterns found when taking the difference between two consecutive sorted address found in the IVT entries

Experiment 3

Experiment 3 consisted of applications that had 5 unused interrupts and the application sample size used was 12 applications. The results of the IVT-Entry-Diff.py script showed us a consistent difference between addresses found in the IVT entries of applications that had 5 unused interrupts . Figure 4.14 shows the results of differences between the sorted addresses found in the IVT entries of 12 applications. We can see see that the IVT entries p0, p10, p11, p14, and p7 are the unused interrupts because the difference between these values are always 0. Using the same approach used in Experiment 1 and 2, we evaluated the differences between addresses found in the IVT entries where there were spikes due to a significant difference. The differences found between the address stored in entry 0 and the address stored in entry 15 (0x4000) will fall within the range of 1,000 (0x3E8) to 2,000 (0x7D0). Since the address had to be an even number we had 500 possible combinations to try when brute forcing values for IVT entries p0, p10, p11, p14, and p7. Next, the differences found between the address stored in IVT entry 6 and the address stored in IVT entry 7 will fall within the range of 10,000 (0x2710) to 30,000 (0x7530). Since the address had to be an even number we had 10,000 possible combinations to try when brute forcing values for IVT entry 6. Next, the address found in IVT entry 5 will always be 30 (0x1E), the IVT entry 13 will always be 38 (0x26), and the IVT entry 12 will always be 30 (0x1E0). Therefore, all three IVT entries had 1 possible combination to try when brute forcing for IVT entries 5, 13, and 12. Next, the differences found between the address stored in IVT entry 4 and the address stored in IVT entry 12 will either be 42

(0x2A) or 38 (0x26). We had 2 possible combinations to try when brute forcing values for IVT entry 4. Next, the differences found between the address stored in IVT entry 1 and the address stored in IVT entry 4 will either be 242 (0xF2), 244 (0xF4), 248 (0xF8) or 256 (0x104). We had 4 possible combinations to try when brute forcing values for IVT entry 1. Next, the differences found between the address stored in IVT entry 9 and the address stored in IVT entry 1 will either be 174 (0xAE) or 178 (0xB2). We had 2 possible combinations to try when brute forcing values for IVT entry 9. Next, the differences found between the address stored in IVT entry 8 and the address stored in IVT entry 9 will always be 92 (0x5C) or 256 (0x100). We had 2 possible combinations to try when brute forcing values for IVT entry 8. Next, the differences found between the address stored in IVT entry 3 and the address stored in IVT entry 8 will always be 174 (0xAE) or 34 (0x22). We had 2 possible combinations to try when brute forcing values for IVT entry 3. Next, the differences found between the address stored in IVT entry 2 and the address stored in IVT entry 3 will always be 134 (0x86) or 92 (0x5C). We had 2 possible combinations to try when brute forcing values for IVT entry 2. The analysis results show us that the key space for applications that had 5 unused interrupts would roughly be 640,000,000 (4.6) possibilities. With that being said, to brute force the MSP430-BSL password with applications that have 5 unused interrupts it would take 194 (4.7) days.

$$500 \times 10000 \times 2 \times 4 \times 2 \times 2 \times 2 \times 2 \qquad or \qquad 640,000,000 \tag{4.6}$$

$$(((640000000 \div 2280) \div 60) \div 24) \qquad or \qquad 194 \tag{4.7}$$

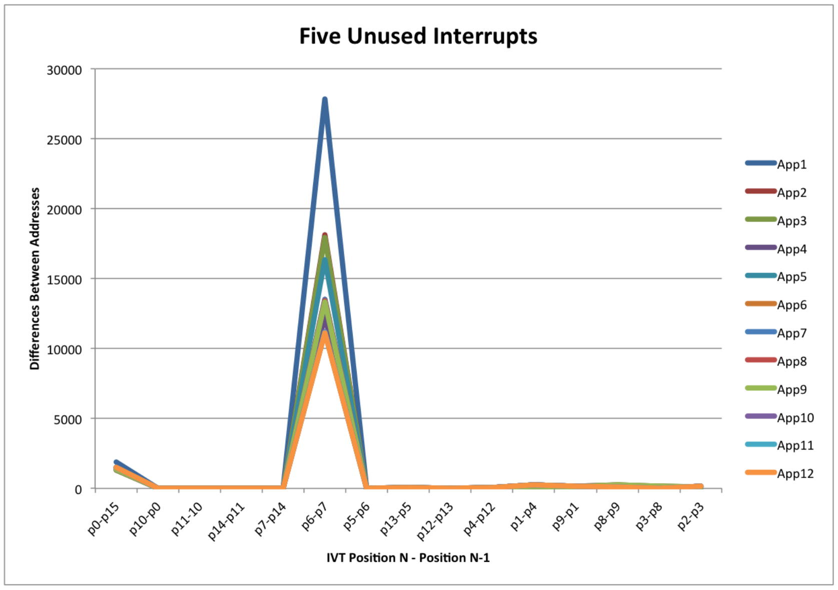


Figure 4.14: The patterns found when taking the difference between two consecutive sorted address found in the IVT entries

Our samples prove that the time estimates made by Becher et al. [27] of 128 years and Goodspeed[59] of 32 years to brute force the MSP430-BSL password can be decreased to a matter of days. Keep in mind that if our application's sample sizes were in the thousands instead of 93 applications then we would be able to further decrease the brute force times to a matter of hours. Furthermore, a bigger sample size can be useful in determining patterns in applications that make use of various interrupts. Although our application's sample was small, there was a variety of interrupts used as well as a variety of unused interrupts. With our sample size we were able to find significant patterns between various applications, which allowed us to to reduce the brute for times. Even if the brute force times takes couple days, brute forcing using our approach is still practical by simultaneously brute forcing multiple capture motes. Making use of more motes is plausible because End Devices in a WSN will most likely have the same application stored in flash memory. The only difference found would be the sensor ID or any cryptographic keys used with the sensor node. Since the MSP430-BSL passwords are only affected by using specific interrupts then identical passwords will be found in similar applications. Therefore, since most of the applications in our sample fall in the category of 7 unused interrupts then capturing 10 motes would decrease the brute force time to roughly 8 (4.8) days. Brute forcing in 8 days is more practical than brute forcing in 128 years or even 32 years.

$$80days \div 10motes \qquad or \qquad 8 \tag{4.8}$$

4.4 Chapter Summary

In this chapter, we contributed investigation results of analyzing the BSL passwords used to protect access to the MSP430 MCU. We demonstrated that the password used to restrict BSL access is identical to the IVT values. We highlighted that the address range used to store the password is the same as the address range used to store the IVT values. Using TinyOS sample applications, we generated a password sample size of 93 to evaluate any patterns found between applications. We found significant patterns between passwords by sorting the addresses found in the IVT and taking the differences between addresses. Using the results from the applications we analyzed we lowered the brute force time from years to only days. We proof the brute force time for applications with 11 unused interrupts to be 2 days. To brute force the applications that have 7 unused interrupts it would take 80 days and for applications that have 5 unused interrupts it would take 194 days.

Chapter 5

Reverse Engineering MSP430 Applications

In this chapter we demonstrate the process of reverse engineering MSP430 applications found in flash memory. As shown in Table 4.1, having the password gives anyone access to all the commands that can be used with the MSP430. An attacker who knows the password will most likely be interested in the commands that allow the exchange of information between a computer and the MSP430. In particular, the attacker will be interested in the "Transmit Data Block" command to allow a computer to extract binary data found in MSP430 flash memory [26]. Dumping MSP430 applications found in flash memory will allow an attacker to either make copies of the binary or reverse engineer the binary. For example, an attacker may be interested in making copies of proprietary applications found in flash memory for redistribution purposes. More importantly, an attacker is going to be interested in finding important information about a WSN application by reverse engineering the sensor applications. For example, an attacker would be interested in reverse engineering our STMS application to obtain a copy of the cryptographic keys. We prove that the asymmetric cryptography proposals [44][45][19][20][46][42][48][49] are broken through reverse engineering. For all these proposals the private keys are pre-deployed on the sensor device; therefore obtaining a copy of the keys allows attackers to authenticate themselves to the WSN. Reverse engineering MSP430 applications will also allow the attacker to understand the functionality of the system and ultimately use the captured node as an entry point to the bigger targets [5]. In most cases the central unit will be the base station of a WSN that is most likely plugged in to a computer or server. With that being said, the purpose of Chapter 5 is to provide an example of how an attacker can reverse engineer MSP430 applications and obtain critical information about a WSN. First, we will discuss the characteristics of the MSP430 flash memory and provide instructions on how to dump MSP430 applications found in flash memory. We captured one of our End-Devices from the L1-Secure STMS to demonstrate the process of reverse engineering to obtain a copy of the network wide shared key.

5.1 Flash Memory Dump

Understanding the memory layout of the MSP430 flash memory is an important step before dumping MSP430 applications and reverse engineering. Figure 5.1 shows the flash memory for the MSP430. For our purposes we will concentrate on the MSP430-F161x 48kB since that is the MSP430 version used with our TelosB mote. However, other MSP430 versions are very similar with the only difference being the starting address of memory regions. The flash memory is partitioned into 512-byte segments and it is the smallest size of flash memory that can be erased [76]. The flash memory is broken down into 3 regions: information memory, RAM and main memory. There is no difference in the operations of main memory and information memory besides segment size and physical address[60]. The information memory region consists of two segments: A and B, both 128-bytes in size. The main memory region consist of 95 segments starting at Segment 0 and going up to Segment 94; each segment is 512-bytes in size. The RAM region starts at address 0x1100 and it is 10,239 bytes in size. Flash memory grows downward; therefore, Segment 0 starts at address 0xFFFF and ends at address 0xFE00. Segment 1 starts at address 0xFDFF and ends at address 0xFC00 and so on until Segment 94, which starts at address 0x41FF and ends at address 0x4000. The main memory starts at address 0x4000 and ends at address 0xFFFF. Since user code (the application) is stored in main memory then dumping data found within the main memory region is of our interest.

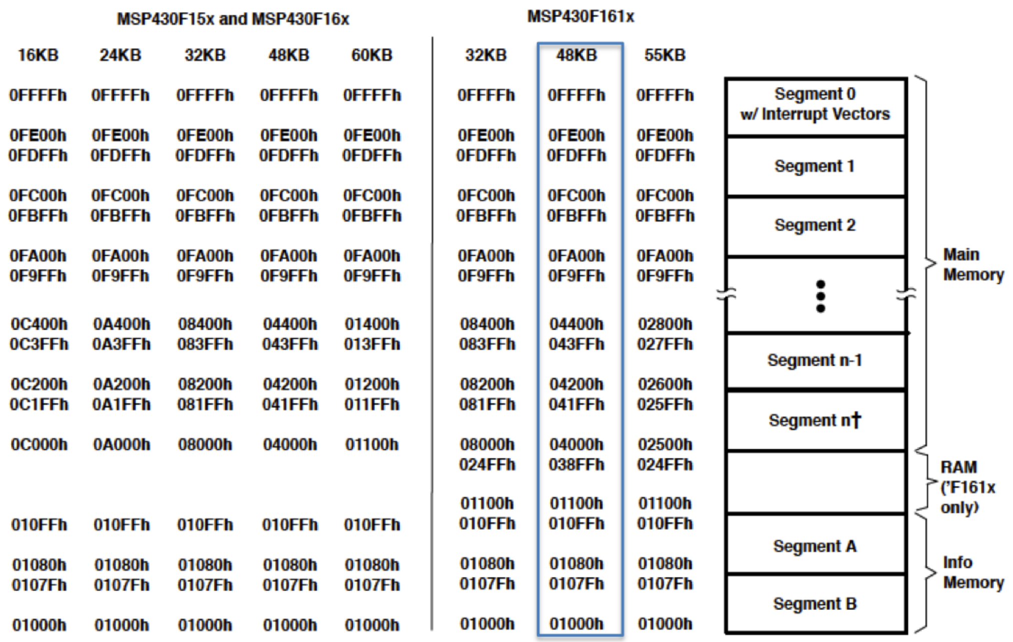

MSP430F15x and MSP430F16x
MSP430F161x
16KB 24KB 32KB 48KB 60KB
32KB 48KB 55KB
0FFFFh 0FFFFh 0FFFFh 0FFFFh 0FFFFh 0FFFFh 0FFFFh 0FFFFh
0FE00h 0FE00h 0FE00h 0FE00h 0FE00h 0FE00h 0FE00h 0FE00h
0FDFFh 0FDFFh 0FDFFh 0FDFFh 0FDFFh 0FDFFh 0FDFFh 0FDFFh
0FC00h 0FC00h 0FC00h 0FC00h 0FC00h 0FC00h 0FC00h 0FC00h
0FBFFh 0FBFFh 0FBFFh 0FBFFh 0FBFFh 0FBFFh 0FBFFh 0FBFFh
0FA00h 0FA00h 0FA00h 0FA00h 0FA00h 0FA00h 0FA00h 0FA00h
0F9FFh 0F9FFh 0F9FFh 0F9FFh 0F9FFh 0F9FFh 0F9FFh 0F9FFh
0C400h 0A400h 08400h 04400h 01400h 08400h 04400h 02800h
0C3FFh 0A3FFh 083FFh 043FFh 013FFh 083FFh 043FFh 027FFh
0C200h 0A200h 08200h 04200h 01200h 08200h 04200h 02600h
0C1FFh 0A1FFh 081FFh 041FFh 011FFh 081FFh 041FFh 025FFh
0C000h 0A000h 08000h 04000h 01100h 08000h 04000h 02500h
024FFh 038FFh 024FFh
01100h 01100h 01100h
010FFh 010FFh 010FFh 010FFh 010FFh 010FFh 010FFh 010FFh
01080h 01080h 01080h 01080h 01080h 01080h 01080h 01080h
0107Fh 0107Fh 0107Fh 0107Fh 0107Fh 0107Fh 0107Fh 0107Fh
01000h 01000h 01000h 01000h 01000h 01000h 01000h 01000h
Segment 0 w/ Interrupt Vectors
Segment 1
Segment 2
Segment n-1
Segment n†
Segment A
Segment B
Main Memory
RAM ('F161x only)
Info Memory

Figure 5.1: The layout of the MSP430-F161x flash memory

Before discussing details on dumping MSP430 applications found in flash memory, we need to discuss exactly how a TinyOS application is flashed into main memory. Figure 5.2 shows the four-phase process of flashing a TinyOS application into MSP430 flash memory. The first phase compiles the source code (TinyOS application) using the nescc compiler and outputs a native C version of the application. The second phase compiles the native C application using the gcc compiler and outputs an ELF object file. The third phase converts the ELF object file to an IHEX object file using the msp430-objcopy command. The fourth, and final, phase flashes the binary to flash memory by reading the ASCII Text representation of the binary using the tos-bsl bootstrap loader software.

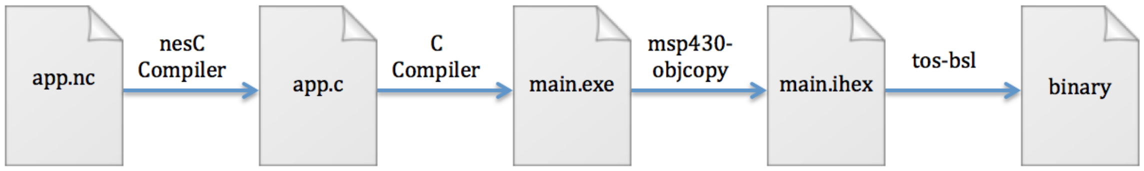


Figure 5.2: The process of flashing a TinyOS application to the MSP430 flash memory

We created a modified version of the tos-bsl to fit our specific memory dump needs. Our modified version allows a user to pass as a parameter the file that contains the password obtained during the brute force process. The original tos-bsl requires the main.ihex file to be passed as the password file because the main.ihex file contains a copy of the password or the IVT. The IVT values are the last 32 bytes of the main.ihex file which are sent to the MSP430 to allow full access to the chip. Our dump-binary.py script took four parameters: the USB port, the password file, the start address, and the size in bytes to dump from flash memory. For example, since we used a TelosB mote equipped with a MSP430-F1611, the start address would be 0x4000 and the dump size would be $0xFFFF - 0x4000$ or 0xBFFF. The output of the script was stored in a file called app-bin, which contained the binary that was found in flash memory of the capture mote. We used our dump-binary.py script to dump the binary found in our ”capture mote”, which has a copy of the L1-Secure End-Device application. The app-bin file contained the MSP430 applications that was found in the L1-Secure End-Device mote and will be used to demonstrate the reverse engineering process.

5.2 Disassemble and Code Analysis Process

The two tools that are the most useful for reverse engineering MSP430 applications are msp430-objcopy and msp430-objdump. The msp430 tools was installed when the MSP430 Tool Chain was installed. One additional tool that could be useful is the msp430static which can be downloaded from the msp430static source forge website[80]. The subsection below will demonstrate the process of converting the dumped MSP430 application , disassembling the MSP430 application , and analyzing the assembly instructions. Regardless of the application found in a mote, the disassembly process will apply to all MSP430 applications and the only difference between the readers dumped MSP430 application and our dumped MSP430 application will be the code analysis of the application itself.

5.2.1 msp430-objcopy

Assuming the reader has used our script to dump the MSP430 application , the following section will describe the conversion process of the dumped MSP430 application. The app-bin file generated by the dump-binary.py script contains binary data that is used by the MSP430 to execute the sensor application. Before disassembling the binary we converted the binary in app-bin file to an ELF format using the msp430-objcopy command. The msp430-objcopy allows to copy and translate content from a source object file to a destination object file in a format different than (or the same as) the source object file. The supported targets for the msp430-objcopy are: elf32-msp430, elf32-little, elf32-big, srec, symbolsrec, verilog, tekhex, binary, and ihex. We were interested in converting from a binary object file to an elf32-msp430 object file. The desired conversation was performed by executing the following command:

$msp430-objcopy -I binary -O elf32-msp430 –set-start 0x4000 app-bin app-elf-msp430

The –set-start flag allowed us to specify the start address of where the converted program began. Setting the start address to 0x4000 eased the reverse engineering process because it maintained the default start address of main memory. Note that not all dumped MSP430 binaries support setting the start address and unfortunately for us that was the case. However, we wrote a python script that converts the addresses to the corresponding MSP430 main memory address, which will be discussed in the following section.

5.2.2 msp430-objdump & msp430static

The following section will describe the process to disassemble the dumped MSP430 application found in a mote. The msp430-objdump displayed information about MSP430 object files, such as the disassembled contents of all sections in a object file. We used the following command to disassemble the app-elf-msp430 object file created during the conversation process.

$msp430-obdump -D app-elf-msp430 > app-disassemble.txt

After executing the msp430-objdump command, the app-disassemble.txt contained the disassembled version of the MSP430 application that was found in flash memory. In our case the app-disassemble.txt file contained assembly instructions of the L1-Secure End-Device application. At this point, one can start analyzing the assembly instructions to ultimately collect important characteristics of the application. However, before diving into the code analysis phase we must point out two characteristics we found that hold true for all MSP430 applications. The first characteristic found was an address conversion problem when converting the dump binary to an ELF format using msp430-objcopy. The MSP430 applications do not support the –set-start flag option to set the start address to 0x4000 instead of a dumped start address of 0x0. Since opcodes are dumped as is, the disassembly instructions will reference an address as it is stored in flash memory. For example, the code found at address 0x4020 in flash memory is not the same as the code found at address 0x4020 in the disassemble dumped version. The equivalent code will instead be stored at address 0x20 in the disassemble dumped version. We were able to fix this problem by implementing a python script (convert-addr.py) that converted the address of each instruction in the app-disassemble.txt file to the corresponding main memory address by adding 0x4000. The second characteristic found was the use of stripping to optimize binary code size. Reducing code size by stripping makes the reverse engineering process more difficult because useful information, such as symbols, no longer existed. We must also point out that stripping the symbols in a binary also prevented us from using the msp430static tool, which could have significantly sped up the reverse engineering process. In case the reader does encounter a situation where the binary is not stripped then it is important to highlight how the msp430static can be useful. The msp430static is a disassembly analysis tool for working with MSP430 application images and can be downloaded from the msp430static source forge website[80]. The msp430static uses Perl to load the disassembled version of a binary into a SQLite3 database, and through macros or ”non-parameterized queries” one can quickly filter certain characteristics of the disassembled binary. The msp430static relies heavily on symbols to be able to organize the database for the disassembled version of the binary[81]. The disassembly analysis can be sped up by using macros to quickly display all the TinyOS functions that have been used with an application or even display blocks of instructions that are of interest. Lastly, we must also point out that the main.exe file generated during the compile process was not stripped (refer to Figure 5.2); therefore, this file can be used with the msp430static for practice purposes.

5.2.3 Code Analysis

Up to this point, the reverse engineering process of dumping MSP430 applications and disassembling dumped MSP430 applications has been the same regardless of the application. However, the code analysis process will vary depending on the functionalities of the application. Since the foundation of our L1-Secure STMS implementation solely used the physical layer, then any future application making use of the CC2420 inline encryption will most likely have a similar code analysis process. Furthermore, research suggests WSNs are moving towards encryption using hardware implementation due to being 42 times faster and using 4.5 times less energy than similar software implementation running on the MSP430 MCU[7]. With that being said, we provide the reader with an overview of the MSP430 assembly language and contribute an example on how we were able to efficiently find the network wide shared key of our L1-Secure STMS application.

The MSP430 interconnects a 16-bit RISC Microprocessor (MPU), peripherals, and memory through the use of a memory address bus (MAB) and memory data bus (MDB). Figure 5.3 shows the functional block diagram of the MSP430-F161x MCU. The von-Neumann architecture has one address space shared with flash memory, RAM, peripherals and special function registers[60]. Words are used to address instructions and, as mentioned in Chapter 4, words are only located at even addresses. As a Reduce Instruction Set Computer (RISC), the MSP430 only has 15 registers, 7 addressing modes and a total of 27 instructions. Table 5.1 shows all 15 registers, where registers 4 through 15 are general-purpose registers[60]. The PC/R0 program counter register is used to point to the next instruction be executed. The SP/R1 stack pointer register is used by the MCU to store the return address of calls or interrupts. The SR/CG1/R2 status register is used as a source and destination register or as a constant generator register. The CG2/R3 is a constant generator register that generates six commonly used constants without requiring an additional 16-bit word of program code.

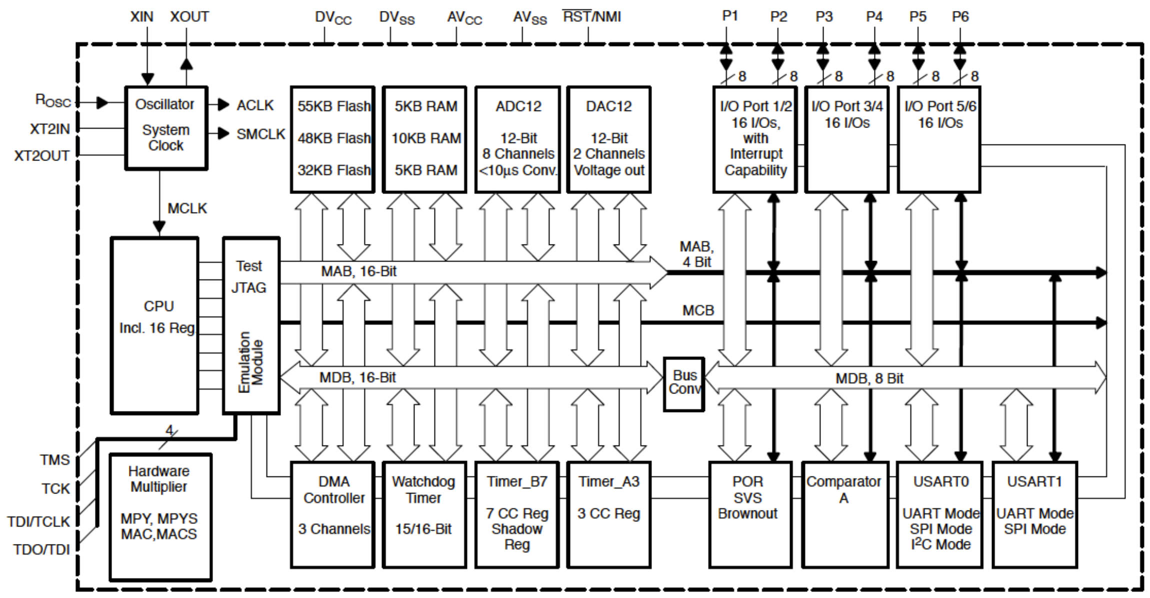


Figure 5.3: The functional block diagram for the MSP430-F161x MCU

Table 5.1: The 15 registers supported by the MSP430.

Purpose	Register
Program Counter	PC/R0
Stack Pointer	SP/R1
Status Register	SR/CG1/R2
Constant Generator	CG2/R3
General-Purpose Registers	R4-R15

Table 5.2 shows the 7 addressing modes supported by the MSP430 MCU (examples can be found in the MSP430x1xx Family User's Guide manual[60]). Essentially, the addressing modes dictate the 7 different ways that the operands of an instruction should be identified. The most common addressing modes that are encountered by MSP430 assembly instructions are the register mode, absolute mode, and the immediate mode.

Table 5.2: The 7 addressing modes supported by the MSP430

Addressing Mode	Syntax
Register Mode	Rn
Indexed Mode	X(Rn)
Symbolic Mode	ADDR
Absolute Mode	&ADDR
Indirect Register Mode	@Rn
Indirect Auto-Increment	@Rn+
Immediate Mode	#N

The MPS430 instruction set is composed of 27 core instructions and 24 emulated instructions. Table 5.3 shows all 51 instructions with the emulated instructions marked with the † symbol[60]. The difference between the two types of instructions is that the core instructions have unique op-codes, where as emulated instructions do not have unique opcodes. The emulated instructions are used for making assembly code more readable/writable and are replaced automatically by the assembler with an equivalent core instruction. All instructions in Table 5.3 that have one or more operands are word instructions used to access word data or word peripherals. Only the instructions marked with (.B) support byte instructions used to access byte data or byte peripherals. The columns labeled with V, N, Z, and C represent the status bit columns. The overflow bit (V) is set when the results of an arithmetic instructions overflows the signed-variable range. The negative bit (N) is set when the result of a byte or word instruction is negative. The zero bit (Z) is set when the result of a byte of work instruction is zero. The carry bit (C) is set when the result of a byte or work instructions produces a carry. The instructions will either affect the status bit (*), not affect the status bit (-), clear the status bit (0) or set the status bit (1).

Table 5.3: The instruction set of the MSP430.

Instruction	Operation	V	N	Z	C
ADC(.B)† dst	$dst + C \rightarrow dst$	*	*	*	*
ADD(.B) src,dst	$src + dst \rightarrow dst$	*	*	*	*
ADDC(.B) src,dst	$src + dst + C \rightarrow dst$	*	*	*	*
AND(.B) src,dst	$src \wedge dst \rightarrow dst$	0	*	*	*
BIC(.B) src,dst	$\neg src \wedge dst \rightarrow dst$	-	-	-	-
BIS(.B) src,dst	$src \vee dst \rightarrow dst$	-	-	-	-
BIT(.B) src,dst	$src \wedge dst$	0	*	*	*
BR† dst	$dst \rightarrow PC$	-	-	-	-
CALL dst	$PC + 2 \rightarrow stack, dst \rightarrow PC$	-	-	-	-
CLR(.B)† dst	$0 \rightarrow dst$	-	-	-	-
CLRC†	$0 \rightarrow C$	-	-	-	0
CLRN†	$0 \rightarrow N$	-	0	-	-
CLRZ†	$0 \rightarrow Z$	-	-	0	-
CMP(.B) src,dst	$dst - src$	*	*	*	*
DADC(.B)† dst	$dst + C \rightarrow dst(decimally)$	*	*	*	*
DADD(.B) src,dst	$src + dst + C \rightarrow dst(decimally)$	*	*	*	*
DEC(.B)† dst	$dst - 1 \rightarrow dst$	*	*	*	*
DECD(.B)† dst	$dst - 2 \rightarrow dst$	*	*	*	*
DINT†	$0 \rightarrow GIE$	-	-	-	-
EINT†	$1 \rightarrow GIE$	-	-	-	-
INC(.B)† dst	$dst + 1 \rightarrow dst$	*	*	*	*
INCD(.B)† dst	$dst + 2 \rightarrow dst$	*	*	*	*
INV(.B)† dst	$\neg dst \rightarrow dst$	*	*	*	*
JC/JHS label	Jump if C set / Jump if higher or same	-	-	-	-
JEQ/JZ label	Jump if equal / Jump if Z set	-	-	-	-
JGE label	Jump if greater or equal	-	-	-	-
JL label	Jump if less	-	-	-	-
JMP label	Jump unconditionally	-	-	-	-
JN label	Jump if N set	-	-	-	-
JNC/JLO label	Jump if C not set / Jump if lower	-	-	-	-
JNE/JNZ label	Jump if not equal / Jump if Z not set	-	-	-	-
MOV(.B) src,dst	$src \rightarrow dst$	-	-	-	-
NOP†	No operation	-	-	-	-
POP(.B)† dst	$@SP \rightarrow dst, SP + 2 \rightarrow SP$	-	-	-	-
PUSH(.B) src	$SP - 2 \rightarrow SP, src \rightarrow @SP$	-	-	-	-
RET†	$@SP \rightarrow PC, SP + 2 \rightarrow SP$	-	-	-	-
RETI	Return from interrupt	*	*	*	*
RLA(.B)† dst	Rotate left arithmetically	*	*	*	*
RLC(.B)† dst	Rotate left through C	*	*	*	*
RRA(.B) dst	Rotate right arithmetically	0	*	*	*
RRC(.B) dst	Rotate right through C	*	*	*	*
SBC(.B)† dst	$dst + 0xFFFF + C \rightarrow dst$	*	*	*	*
SETC†	$1 \rightarrow C$	-	-	-	1
SETN†	$1 \rightarrow N$	-	1	-	-
SETZ†	$1 \rightarrow C$	-	-	1	-
SUB(.B) src,dst	$dst + \neg src + 1 \rightarrow dst$	*	*	*	*
SUBC(.B) src,dst	$dst + \neg src + C \rightarrow dst$	*	*	*	*
SWPB dst	Swap bytes	-	-	-	-
SXT dst	Extend sign	0	*	*	*
TST(.B)†	$dst + 0xFFFF + 1$	0	*	*	1
XOR(.B) src,dst	$src \oplus dst \rightarrow dst$	*	*	*	*

In this chapter, we contribute to the WSN community with a first ever example of how an attacker can obtain cryptographic keys from a capture mote through reverse engineering. We have analyzed the disassembled version of the binary that was dumped from the captured End-Device mote that is part of L1-Secure system. All End-Devices that are part of the L1-Secure STMS have a copy of the network wide shared key. If a network wide shared key has been pre-deployed then the security of any WSN environment will be broken through reverse engineering. With that being said, let us now discuss the technical details of how a cryptographic key can be obtained from an application that is using encryption at the physical layer.

In Chapter 4, we discussed how the power-up/reset interrupt handler always points to address 0x4000 for its interrupt service routine. The reason this address holds true for all applications is because the starting address of all MSP430 applications is 0x4000. Furthermore, the 0x4000 address is also the starting address of main memory or "user code". Therefore, the reverse engineering process will begin at address 0x4000 and execution flow will continue from this address. We have grouped the instructions into blocks as multiple instructions achieve one overall function. First, the instruction blocks along with a description of each instruction is presented. The preceding paragraph describes the overall function and its importance. Since our contribution is to investigate how an "attacker" can obtain a copy of the cryptographic keys, only the instructions that lead to finding the keys will be highlighted.

Address	Op-codes	Instruction	Description
4000:	55 42 20 01	mov.b &0x0120, r5	;Watchdog-timer.byte (0x00) $\rightarrow$ r5
4004:	35 d0 08 5a	bis #23048, r5	;$0x5a08 \vee r5 \rightarrow r5$
4008:	82 45 06 13	mov r5, &0x1306	;$0x5a08 \rightarrow \&0x1306$

The first block of instructions are used to establish watchdog timer peripheral support and are found in every application. The instructions at address 0x4000 moves the low byte found in the watchdog timer register (WDTCTL). The watchdog timer peripheral is protected from direct access by a user code and only the WDTCTL register (0x120) can be used to get access to the watchdog timer peripheral. The instruction reads the lower byte of WDTCTL register, which at start up has the default value of 0x6900. The instruction at address 0x4004 performs a 0x5a08 OR 0x0000 operation resulting in the value 0x5a08 to be stored in register r15. Lastly, the instruction at address 0x4008 moves the value in register r5 (0x5a08) to the address 0x1306 located in the RAM region. The purpose of the first block of instructions is to initiate the watchdog timer peripheral support in order to perform or prevent application restarts. All applications are required to start the watchdog time support in order for any application to properly function.

Address	Op-codes	Instruction	Description
400c:	31 40 00 39	mov #14592, r1	;$0x3900 \rightarrow r1$

The second block of instructions are used to initiate the stack pointer by moving the immediate value 0x3900 to register r1. The stack pointer is initiated to address 0x3900 which points to the top of the RAM region in flash memory (refer to Figure 5.1).

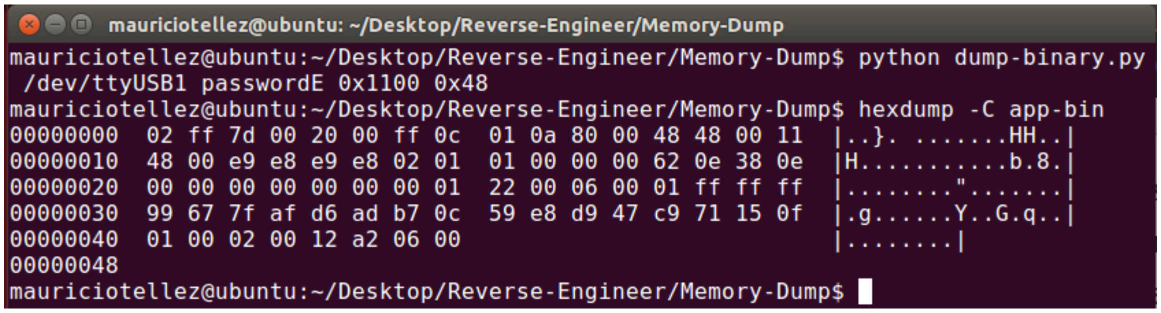
Address	Op-codes	Instruction	Description
4010:	3f 40 48 00	mov #72, r15	;$0x0048 \rightarrow r15$
4014:	0f 93	tst r15	;$r15 + 0xFFFF + 1$
4016:	08 24	jz $+18	;Jump to 0x4028 if Z is set
4018:	92 42 06 13	mov &0x1306, &0x0120	;$\&0x1306 \rightarrow \&0x120$
401c:	20 01		;Part of previous instruction
401e:	2f 83	decd r15	;$r15 - 2 \rightarrow r15$
4020:	9f 4f a6 7c	mov 31910(r15),4352(r15)	;$0x7ca6(r15) \rightarrow 0x1100(r15)$
4024:	00 11		;Part of previous instruction
4026:	f8 23	jnz $-14	;Jump to 0x4018 if Z not set

The third block of instructions are used to copy the content from the .data section into the RAM region. The instruction at address 0x4010 moves the immediate value 0x48 to register r15. The register r15 is used as an offset to copy data from the main memory region to the RAM region. The instructions at address 0x4014 and 0x4016 check to see if the value in register r15 is 0x0 and if it is jumps to address 0x4028. We know the value in register r15 is 0x48; therefore, the program counter or register r0 points to address 0x4018 after executing the TST and JZ instructions. The only time the TST and JZ instructions will hold true is if the .data section of the binary is empty. The instruction at address 0x4018 moves the value at address 0x1306 to the WDTCTL register (0x120). From the first block of instructions we know the value stored at address 0x1306 is 0x5a08; therefore, the value stored in WDCCTL register is 0x5a08. To interpret the value 0x5a08 of the WDCCTL register requires splitting the word into two bytes: the high byte (0x5a) and low byte (0x08). The high byte is used to send the write password 0x5a to watchdog timer peripheral in order to grant access to indirect access to the user code. The low byte specifies the type of action requested to the watchdog timer peripheral by the user code. Each bit of the low byte represent a watchdog timer mode that is set when the byte is written to the watchdog timer peripheral. In this particular case, the watchdog timer is set to the WDTCNTCL mode which clears the reset counter value to 0 to prevent the application to reset. The next instructions at address 0x401e decrements the value stored in register r15 by two bytes; therefore, the new value in register r15 is 0x46. The instruction at address 0x4020 is the most important instruction of the third block because the instruction is copying the data found at address 0x7ca6(offset) to address 0x1100(offset). The last instruction at address 0x4026 jumps to address 0x4018 if the zero bit is not set. The the zero bit will be set by the DECD instruction once the value in register r15 has reached zero. In summary, the instructions at address 0x4018 to 0x4026 are a loop that will copy the data found in the main memory region to the RAM region. Figure 5.4 shows what the instructions in the loop are essentially accomplishing.

0x7ca6(0x0) or 0x7ca6	→	0x1100(0x0) or 0x1100
0x7ca6(0x2) or 0x7ca8	→	0x1100(0x2) or 0x1102
0x7ca6(0x4) or 0x7caa	→	0x1100(0x2) or 0x1104
	...	
0x7ca6(0x42) or 0x7ca8	→	0x1100(0x42) or 0x1142
0x7ca6(0x44) or 0x7cea	→	0x1100(0x44) or 0x1144
0x7ca6(0x46) or 0x7cec	→	0x1100(0x46) or 0x1146

Figure 5.4: An example of how main memory data is copied into RAM.

Up to this point, we know that there was a chunk of data in main memory that was copied to the address range of 0x1100 to 0x1148. The address of 0x1100 is the starting address of RAM; therefore, we know that the chunk of data was copied to the first 72 (0x48) bytes of the RAM region. Investigating the data copied to RAM was of our interest because our ultimate goal was to obtain a copy of the cryptographic keys. Unlike registers that are "variables" internal to the MCU, all variables used in an application are stored in RAM at start-up time[82]. Since it is a good coding practice to use variables to hold data, then all applications that use encryption will most likely have a variable that will be initialized to the cryptographic key. With that being said, as seen in Figure 5.5, we used our dump-binary.py script to dump the data that was copied to RAM.

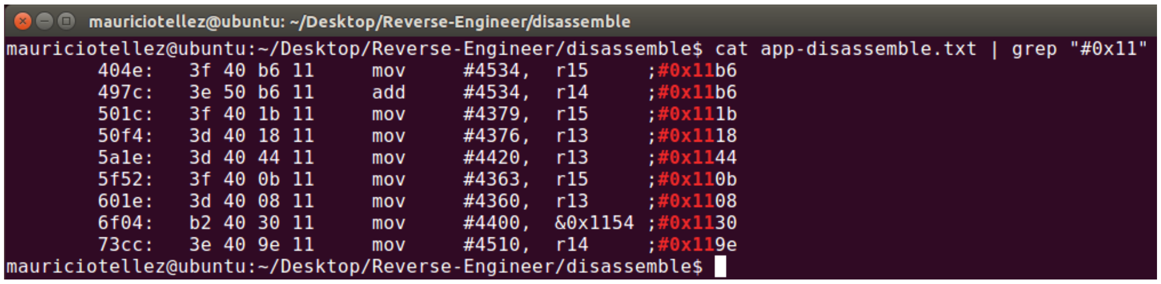

mauriciotellez@ubuntu: ~/Desktop/Reverse-Engineer/Memory-Dump

```
mauriciotellez@ubuntu:~/Desktop/Reverse-Engineer/Memory-Dump$ python dump-binary.py
 /dev/ttyUSB1 passwordE 0x1100 0x48
mauriciotellez@ubuntu:~/Desktop/Reverse-Engineer/Memory-Dump$ hexdump -C app-bin
00000000  02 ff 7d 00 20 00 ff 0c  01 0a 80 00 48 48 00 11  |..}. .......HH..|
00000010  48 00 e9 e8 e9 e8 02 01  01 00 00 00 62 0e 38 0e  |H...........b.8.|
00000020  00 00 00 00 00 00 00 01  22 00 06 00 01 ff ff ff  |........".......|
00000030  99 67 7f af d6 ad b7 0c  59 e8 d9 47 c9 71 15 0f  |.g......Y..G.q..|
00000040  01 00 02 00 12 a2 06 00                           |........|
00000048
mauriciotellez@ubuntu:~/Desktop/Reverse-Engineer/Memory-Dump$ 
```

Figure 5.5: The output of dumping the data that was copied to RAM during the third block of instructions.

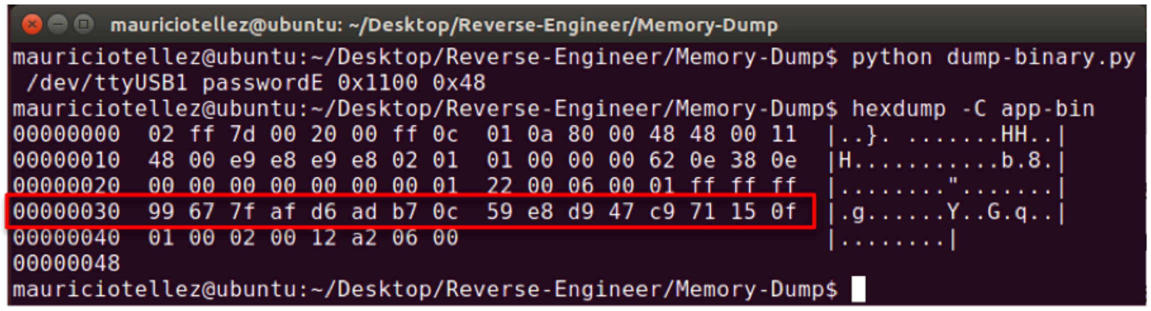
The 72 copied bytes found at the beginning of the RAM region match the original 72-bytes found in main memory from address 0x7ca6 to 0x7cef. From the 72-bytes that were dumped from the starting application of RAM we were not 100% sure which bytes were the cryptographic key bytes. However, without having to continue the tedious disassembly analysis, an attacker may be able to take shortcuts to verify which of the 72-bytes are the cryptographic bytes. In Chapter 3, we demonstrated how a mote can be used to capture packets over a WSN. Using the PPPSniffer results we were aware that the application was using only the physical layer IEEE802.15.4 protocol; therefore, highly suggesting that hardware encryption was used. Since we know that the TelosB capture more uses the CC2420 radio chip, which has a built inline AES-128 encryption, then we know the key size has to be 128-bits or 16-bytes. Therefore, our final goal was to conclude what 16 consecutive bytes of the 72-bytes found in memory was the cryptographic key.

We know that the user code in memory referenced bytes found at address range 0x1100 to 0x1148 in RAM. Since time is of essence to an attacker, we used the cat and grep linux commands to find instructions that made references to addresses found in the range of 0x1100 to 0x1148. Figure 5.6 shows six instructions that reference the bytes at locations 0x111b, 0x1118, 0x1144, 0x110b, 0x1108 and 0x1130. Since registers are a word in size then the instructions that move 2 bytes from RAM to registers was not of our interest; therefore, we ignored 0x111b, 0x1118, 0x1144, 0x110b, and 0x1108 RAM address. We were left with only one valid instruction that moved the immediate value 4400 (0x1130) to address 0x1154. Therefore, the address 0x1154 pointed to the beginning of the cryptographic key, which was located at address 0x1130. Figure 5.7 shows the 16-bytes of the cryptographic key or the network wide shared key used with the L1-Secure system. There is high possibility that the private keys for the [44][45][19][20][46][42][48][49] proposals are also stored in RAM. Furthermore, we are more confident that the symmetric keys for the [11][14][10][13][15][12][16][42][17][7][8][18] proposals are stored in RAM.


mauriciotellez@ubuntu: ~/Desktop/Reverse-Engineer/disassemble

```
mauriciotellez@ubuntu:~/Desktop/Reverse-Engineer/disassemble$ cat app-disassemble.txt | grep "#0x11"
    404e:   3f 40 b6 11     mov     #4534,  r15     ;#0x11b6
    497c:   3e 50 b6 11     add     #4534,  r14     ;#0x11b6
    501c:   3f 40 1b 11     mov     #4379,  r15     ;#0x111b
    50f4:   3d 40 18 11     mov     #4376,  r13     ;#0x1118
    5a1e:   3d 40 44 11     mov     #4420,  r13     ;#0x1144
    5f52:   3f 40 0b 11     mov     #4363,  r15     ;#0x110b
    601e:   3d 40 08 11     mov     #4360,  r13     ;#0x1108
    6f04:   b2 40 30 11     mov     #4400,  &0x1154 ;#0x1130
    73cc:   3e 40 9e 11     mov     #4510,  r14     ;#0x119e
mauriciotellez@ubuntu:~/Desktop/Reverse-Engineer/disassemble$
```

Figure 5.6: The addresses of all the instruction that reference the RAM region (0x11##).



```
mauriciotellez@ubuntu: ~/Desktop/Reverse-Engineer/Memory-Dump
mauriciotellez@ubuntu:~/Desktop/Reverse-Engineer/Memory-Dump$ python dump-binary.py
 /dev/ttyUSB1 passwordE 0x1100 0x48
mauriciotellez@ubuntu:~/Desktop/Reverse-Engineer/Memory-Dump$ hexdump -C app-bin
00000000  02 ff 7d 00 20 00 ff 0c  01 0a 80 00 48 48 00 11  |..}. .......HH..|
00000010  48 00 e9 e8 e9 e8 02 01  01 00 00 00 62 0e 38 0e  |H...........b.8.|
00000020  00 00 00 00 00 00 00 01  22 00 06 00 01 ff ff ff  |........".......|
00000030  99 67 7f af d6 ad b7 0c  59 e8 d9 47 c9 71 15 0f  |.g......Y..G.q..|
00000040  01 00 02 00 12 a2 06 00                           |........|
00000048
mauriciotellez@ubuntu:~/Desktop/Reverse-Engineer/Memory-Dump$
```

Figure 5.7: The network wide shared key found through reverse engineering.

5.3 Chapter Summary

In summary, we have contributed to the WSN community with a step-by-step process of reverse engineering MSP430 applications found in a capture sensor nodes. In particular, we investigated the process of dumping MSP430 applications found in flash memory as well as converting the dumped binary to a format that can be reverse engineered using a Linux machine. The results collected during our investigations provide the WSN community with detailed characteristics of binaries found in MSP430 flash memory. In addition, we investigated the difficulty of reverse engineering MSP430 applications to obtain a copy of cryptographic keys found in a capture node. We have presented the reader with a point of reference guide to assist in future MSP430 application disassembly processes. To our knowledge, our reverse engineering example is the first of its kind that demonstrates the feasibility of obtaining cryptographic keys from a secure WSN. Chapter 5 can serve as a starting point to future papers that concentrate in the reverse engineering MSP430 applications .

Chapter 6

Protecting MPS430 Firmware With Secure-BSL

In this chapter, we contribute to the WSN community with a better approach in protecting firmware found in MSP430 flash memory. Our approach supports two levels of security. The first security level consists of one-factor authentication in order to grant full access to the MPS430. The second security level consists of two-factor authentication in order to grant full access to the MSP430. The two levels of security have been implemented on top of the original bootstrap loader software and our new Secure-BSL software is available to the WSN community through our Github project. This chapter first discusses how the original BSL code works then goes into discussing the security design of our Secure-BSL software and the implementation of the Secure-BSL.

6.1 Original-BSL Code

As stated in Chapter 4, the BSL code allows users or developers to communicate with the MSP430 through a USB interface. Once the TelosB mote is plugged into a USB port, a python script (tos-bsl.py) is used to communicate with the MSP430 chip. The python script uses the serial python library in order to establish and support communication between the computer and the MSP430 chip. Furthermore, to support USB serial communication the MSP430 chip stores BSL code to respond to any commands sent by the computer. The BSL code is stored in a secure location (0x0C00 - 0x0FEF) that is write protected to avoid any bypass techniques if the password is unknown. Figure 6.1 shows a diagram of the BSL commands that are supported by the MSP430. The commands in green represent the commands that are allowed without the MSP430-BSL passwords.

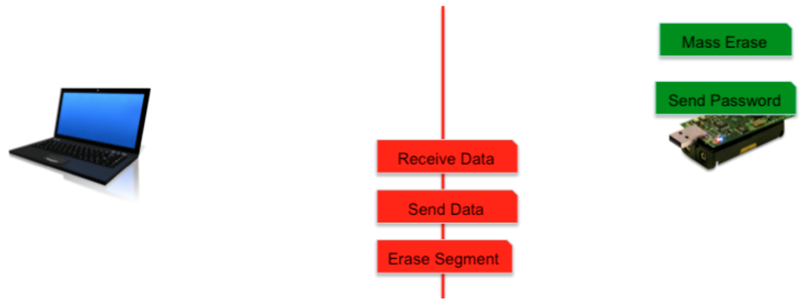


Figure 6.1: The BSL commands supported by the MSP430 without the password.

Figure 6.2 shows a diagram of the BSL commands that are supported once the correct MSP430-BSL password is provided. As you can see from the diagram, full access (all the commands) are supported by the MSP430 once the correct MSP430-BSL password is provided. Therefore, if an attacker does not know the correct MSP430-BSL password then they will not have full access to the MSP430 chip.

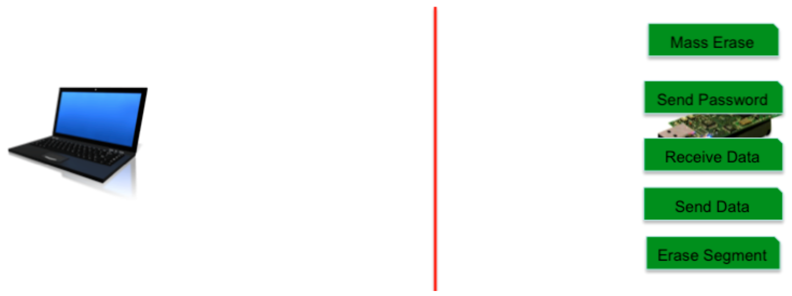


Figure 6.2: The BSL commands supported by the MSP430 with the correct password.

6.2 Secure-BSL Design

As we discussed in Chapter 4, the problem of the current MSP430-BSL password lies in the fact that it is not random. We proved that in the best case scenario the MSP430-BSL password can be brute forced in two days. More importantly, we have lowered the brute force time to a matter of days rather than years. Also, we have proved that the MSP430-BSL password is predictable because of the IVT entries not being random. In particular, we have seen a correlation between the number of unused interrupts and the brute force time (refer to Chapter 4). With that being said, the MSP430-BSL password is strong if and only if the brute force times are impractical.

One approach that we considered but did not pursue was the use of hash functions to take the hash of the main.ihex file and use the results as the MSP430-BSL password. Since flashing already requires a binary as an input to the bootstrap loader software then one would assume that hashing would be a plausible solution. For example, if we took the SHA256 of a MSP430 binary and used the 256-bit (32 bytes) hash values as the password (IVT values) then the password would be random. Each byte of a binary would influence the output of the SHA256 hash function and would be a unique random password per application. Meaning, if the same binary was flashed into multiple motes then all the motes that have the same binary have the same password. Although this is a quick and efficient solution, taking the hash of a binary and using the hash value as the password does not work with the MSP430 MCU. The problem with using the hashing approach is the fact that the MSP430 MCU uses the address range of the IVT as the address space to store the password.

The address space of the IVT and the address space for the password is the same; therefore, storing the hash value on the IVT address space makes the mote malfunction. Modifying the IVT to use random values prevent the application to properly start because the IVT does not have the actual addresses of the ISRs. For example, the Power-Up interrupt needs to know where the start address of the application is, which is 0x4000, and if the value at this interrupt is random 0x1234 then the sensor application will never properly start. Future MCU should designate an address space to solely store the BSL password if hashing is to be used as the password. Hashing the binary to use as the MSP430-BSL password is not a good solution; therefore, a different approach was required.

The approach we implemented allowed us to randomize the password and still maintain proper functionality of the interrupt handlers. Our Secure-BSL implementation is similar to the one presented by Becher et al. [27], where the IVT was randomized without affecting the overall functionality of the IVT. The first level of security consists of randomizing the values stored in the IVT address space and used branch instructions to maintain proper functionality of MSP430 applications. The Secure-BSL varies from Bechers' solution because we contributed a second level of security by using two-factor authentication to restrict further access to the MSP430. The second level of security consists of using a user defined password to encrypt the password file generated by the first level of security. The rest of the this section discusses the design approach of the Secure-BSL software.

6.2.1 Security Level One

The first security level of the Secure-BSL software improved the MSP430-BSL password by randomizing the values stored in the IVT. Similar to the original bootstrap loader software (tos-bsl) the Secure-BSL takes as input the main.ihex or the sensor application to flash. Figure 6.3 shows the input and output of the Secure-BSL software using the security level of one. What varies from the Secure-BSL compared to the original tos-bsl is the output generated by the Secure-BSL. The output of the Secure-BSL is a password file to be used for future access to the MSP430 MCU after flashing. Where as the original tos-bsl software did not output a password file since the actual main.ihex file has a copy of the IVT values or the MSP430-BSL password. Since the Secure-BSL software generated random IVT values it must output the new IVT values that are independent of the IVT values found in main.ihex file. Note if the password file generated by the Secure-BSL is lost then future access to the MSP430 is no longer supported. In order to regain control of the MSP430 then the MCU must be mass erased and reset to factory settings.

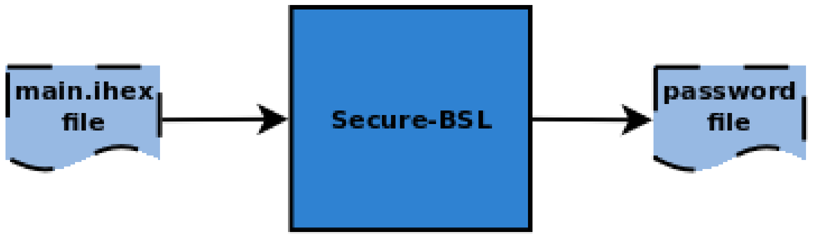


Figure 6.3: Input and output of the Secure-BSL one-factor security level.

Inside the Secure-BSL, it allows the IVT to be randomized and maintain proper functionality of the MSP430 application. The design of the Secure-BSL using security level one consists of two main components: the Unused Address List Generator and the Random Secure Password Generator. Figure 6.4 shows the process of using the main.ihex file to generate the random IVT values and output to a password file. The Unused Address List Generator component generates a list of unused addresses based on the size of the firmware to be flashed. The Random Secure Password Generator generates random IVT entries to be used as new IVT values, thus producing a random MSP430-BSL password.

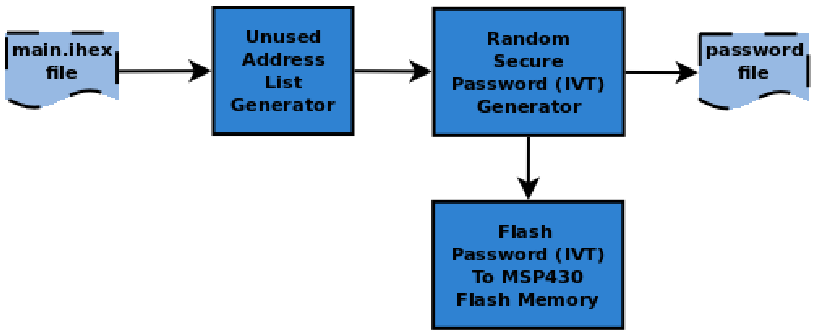

main.ihex file
Unused Address List Generator
Random Secure Password (IVT) Generator
password file
Flash Password (IVT) To MSP430 Flash Memory

Figure 6.4: The high level components of the Secure-BSL security level one.

The purpose of the Unused Address List Generator is to generate all possible addresses that can be used as new IVT values, or in other words new ISR addresses. In order to generate a list of unused address the Secure-BSL calculates the range of addresses that are not used based on the size of the binary found in the main.ihex file. For example, on average the size of a MSP430 binary is 15,840 (0x3DE0) bytes in size (refer to Table A.4). Since main memory on the MSP430 starts at address 0x4000, the average application ends at the address $0x3DE0 + 0x4000$ or 0x7DE0. We know that the IVT begins at address 0xFFE0; therefore, the range of unused addresses are from 0x7DE0 to 0xFFE0. Figure 6.5 shows the memory layout of the former example. The end address is the same for all MSP430 applications; however, the start address varies depending on the size of the binary being flashed into the MSP430 flash memory.

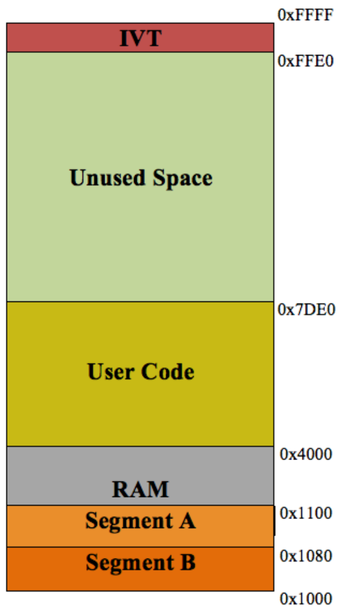


Figure 6.5: A sample of how MSP430 flash memory looks like after flashing firmware.

Once the unused addresses range is calculated, the Unused Address List Generator creates a list of all possible ISR addresses to be used as new random IVT entries. Figure 6.6 shows a flow chart of the Unused Address List Generator design. First, the generator creates an empty list (addr-list) that holds all the possible ISR addresses to be used with the second component of the Secure-BSL security level one. Next, the generator iterates beginning at the start address of the unused memory space and terminates at the end address of the unused memory space. Each iteration appends the current start address value to the end of the addr-list and increments the start address by 4. The start address is incremented by four because each possible ISR address has to be able to store 4-bytes of op-codes. Using our previous example, the first address appended to the addr-list was 0x7DE0, the second address appended was $0x7DE0 + 0x4$ or 0x7DE4, the third address appended was 0x7DE8 and so on. Addresses have to be skipped by 4-bytes because the 4-bytes are used to store the branch instruction op-codes to prevent the MSP430 application from malfunctioning. The design approach of using the unused address space allows the Secure-BSL software to add randomness to the MSP430-BSL password. The unused memory space makes it possible to maintain information about the ISR by storing interrupt handler addresses in the unused space. Therefore, randomizing the IVT address space did not affect the functionality of the sensor application because the original values stored on the IVT were saved at a different addresses.

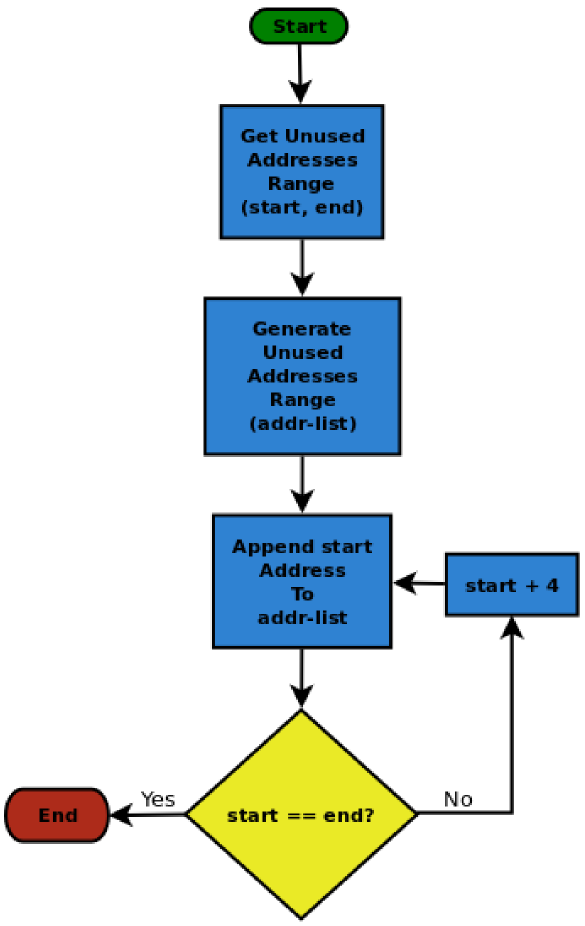

Start
Get Unused Addresses Range (start, end)
Generate Unused Addresses Range (addr-list)
Append start Address To addr-list
start + 4
start == end?
Yes
No
End

Figure 6.6: The flow chart describes the process of generating a list of possible passwords (IVT values).

The second component and last phase of the Secure-BSL was generating a random password using the Random Secure Password Generator. Figure 6.7 shows a flow chart of the Random Secure Password Generator design. The password generator uses the output of the Unused Address List Generator (addr-list) as a pool of word values that can be used as new interrupt handler addresses. Since the IVT consists of 16 words then the Get IVT Entries operation generates a list (ISR-list) that holds all 16 original ISRs found in the IVT of the main.ihex file. The generator iterates through each item in the ISR-list to create a new interrupt handler by randomly choosing an address from the addr-list. The detailed process of creating a randomly chosen interrupt handler address and maintaining the original ISR address consists of five phases. The first phase creates the op-codes to branch to the original ISR address in order to maintain information about the original IVT. The second phase shuffles the items in the addr-list in order to create a new random interrupt handler address. The third phase pops the last item of the shuffled addr-list to use as the new interrupt handler address. The fourth phase uses the op-codes generated in the first phase and programs the op-codes to the new interrupt handler address. The fifth and final phase is to update the IVT to use the new interrupt handler address, which creates a random IVT (password).

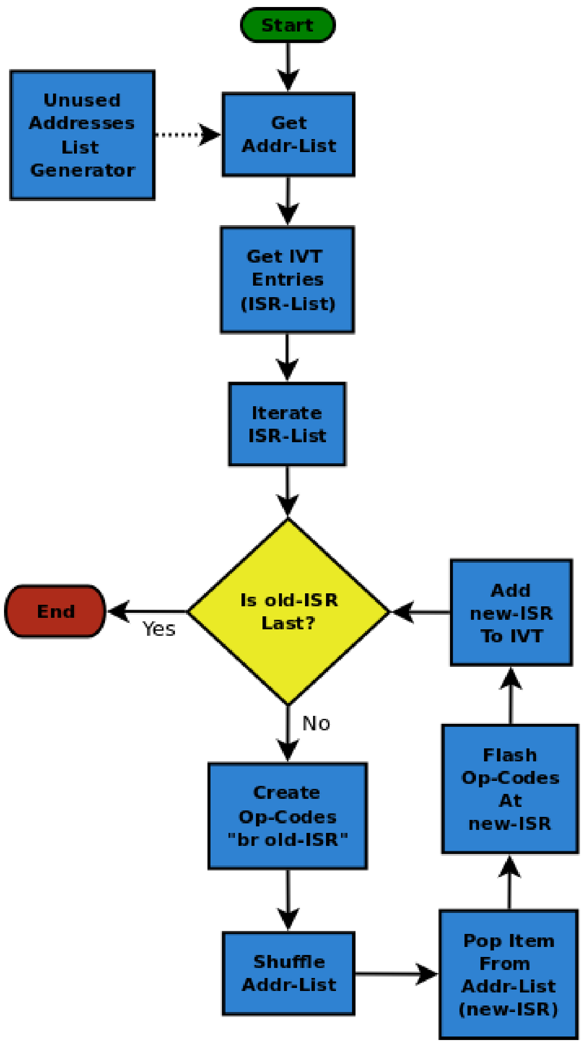

Start
Unused Addresses List Generator
Get Addr-List
Get IVT Entries (ISR-List)
Iterate ISR-List
Is old-ISR Last?
End
Yes
No
Add new-ISR To IVT
Flash Op-Codes At new-ISR
Create Op-Codes "br old-ISR"
Shuffle Addr-List
Pop Item From Addr-List (new-ISR)

Figure 6.7: The flow chart describes the process of generating a secure random password (IVT values).

The security level one of the Secure-BSL made it possible to increase the brute force times of the MSP430-BSL password and better protect the MSP430 MCU. Furthermore, our design assures the overall security of a WSN by decreasing the likelihood of each sensor node having the same password. The use of a randomization methodology increases brute force times for each sensor node that has a MSP430 chip. Even if the binary is the same size, the generated password will most likely be unique using our Secure-BSL software. With that being said, the security level one increases the brute force times for each mote on a WSN, even if the motes have the same MSP430 application in flash memory. To get full access to a flashed MSP430 mote users will be required to have a copy of the password file that was generated when the mote was originally programmed. Figure 6.3 shows how the password file is generated when using the security level one.

6.2.2 Security Level Two

The second security level of the Secure-BSL software further protects the MSP430 MCU by encrypting the password file generated by the security level one. In other words, the security level two is built on top of the security level one in order to provide a two-factor authentication when attempting to gain full access to the MSP430 MCU. Figure 6.8 shows the input required by the main.ihex file as well as a user defined passphrase. The output is an encrypted version of the password file generated by the security level one.

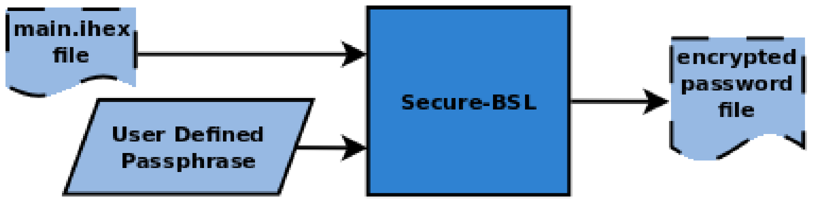


Figure 6.8: Input and output for the Secure-BSL two-factor security level

Figure 6.9 shows the three components of the Secure-BSL security level two feature. The functionality of the first two components are identical to the previously discussed security level one design. Unlike security level one, the security level two adds an additional operation that encrypts the password file that is used for future MSP430 MCU access. The AES-256 Encryption operation encrypts the password file using a user defined passphrase. The additional encryption operations allows the support of a two-factor authentication to grant full access to the MSP430 MCU.

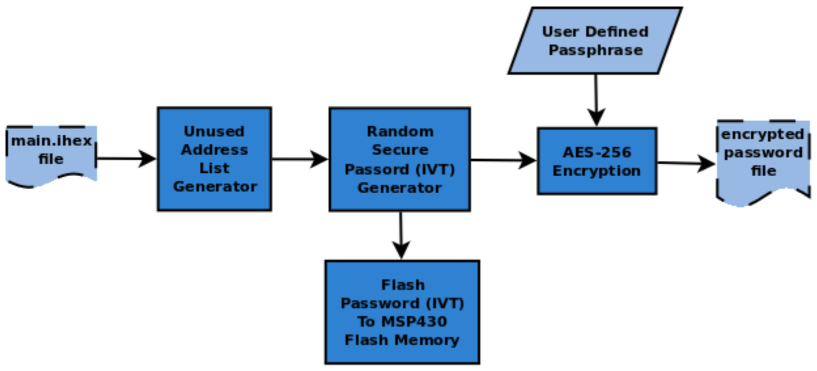

User Defined Passphrase
main.ihex file
Unused Address List Generator
Random Secure Passord (IVT) Generator
AES-256 Encryption
encrypted password file
Flash Password (IVT) To MSP430 Flash Memory

Figure 6.9: The high level components of the Secure-BSL two-factor security level

Figure 6.10 shows the process of gaining full access to a MSP430 MCU using security level two. The AES-256 Decryption operation takes the user defined password and the encrypted password file as input and outputs of the decrypted version of the password file. The decrypted data will be sent to the MSP430 MCU to check if the correct password has been sent by the user in order to grant full access. The two-factor authentication adds an additional security to the MSP430 by only allowing users that have the password file and the passphrase to gain full access to the MSP430 MCU.

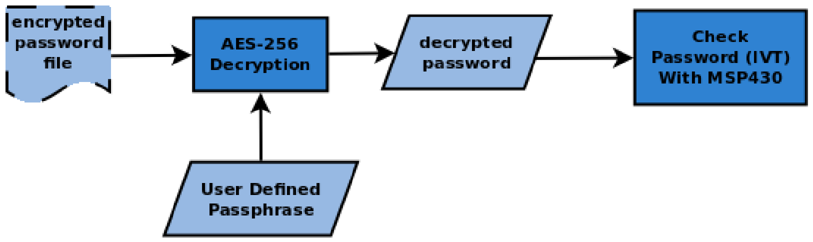


Figure 6.10: The high level execution flow when dumping data from flash memory using two-factor security level

Although the MSP430 Programming with BSL Manual is wrong by stating that ”access to the MSP430 memory through the BSL is protected against misuse by a user-defined password”[26]. We have been able to contribute to the MSP430 family with functional features of allowing users to specify a password to protect flash memory. Our contribution of adding a two-factor authentication strengthens the security of the MSP430 by requiring the user to enter two forms of authentication in order to have full access to the MCU. Even if an attacker is able to gain a copy of the password file they will not be able to gain full access to the MSP430 MCU without knowing the passphrase to decrypt the file. Using AES-256 makes it harder for the attacker to brute force the passphrase in order to obtain the values stored in the password file.

6.3 Secure-BSL Implementation

The implementation of our Secure-BSL software is built on top of the original bootstrap loader (tos-bsl) software. Figure 6.11 shows the UML diagram of the Secure-BSL with the yellow boxes representing our contributions. The UML diagram only displays the components of the Secure-BSL that are responsible for flashing firmware and extracting firmware. However, the complete Secure-BSL source code can be found in our Github project[75]. The Secure-BSL software has six classes, each of which contribute a unique operation to the program. First, the main class is responsible for collecting user parameters and initiating proper bootstrap loader functionality based on user input. The Memory class is responsible for preparing data to be flashed into memory by parsing binary data into segments. The Segment class provides the segment structure to represent flash memory and it is used by the Memory class. The AES256 class is responsible for providing the security level two functionality. The BootStrapLoader class is a core component of the Secure-BSL as it provides the operations to interact/communicate with the MSP430 bootstrap loader. Lastly, the LowLevel class is responsible for exchanging information between the MSP430 MCU and the computer running the Secure-BSL software. The rest of this chapter discusses in detail the implementation of the two main operations supported by the Secure-BSL: flashing firmware and extracting firmware. In particular, we highlight how our implementation assures the security of firmware through the use of our two levels of security.

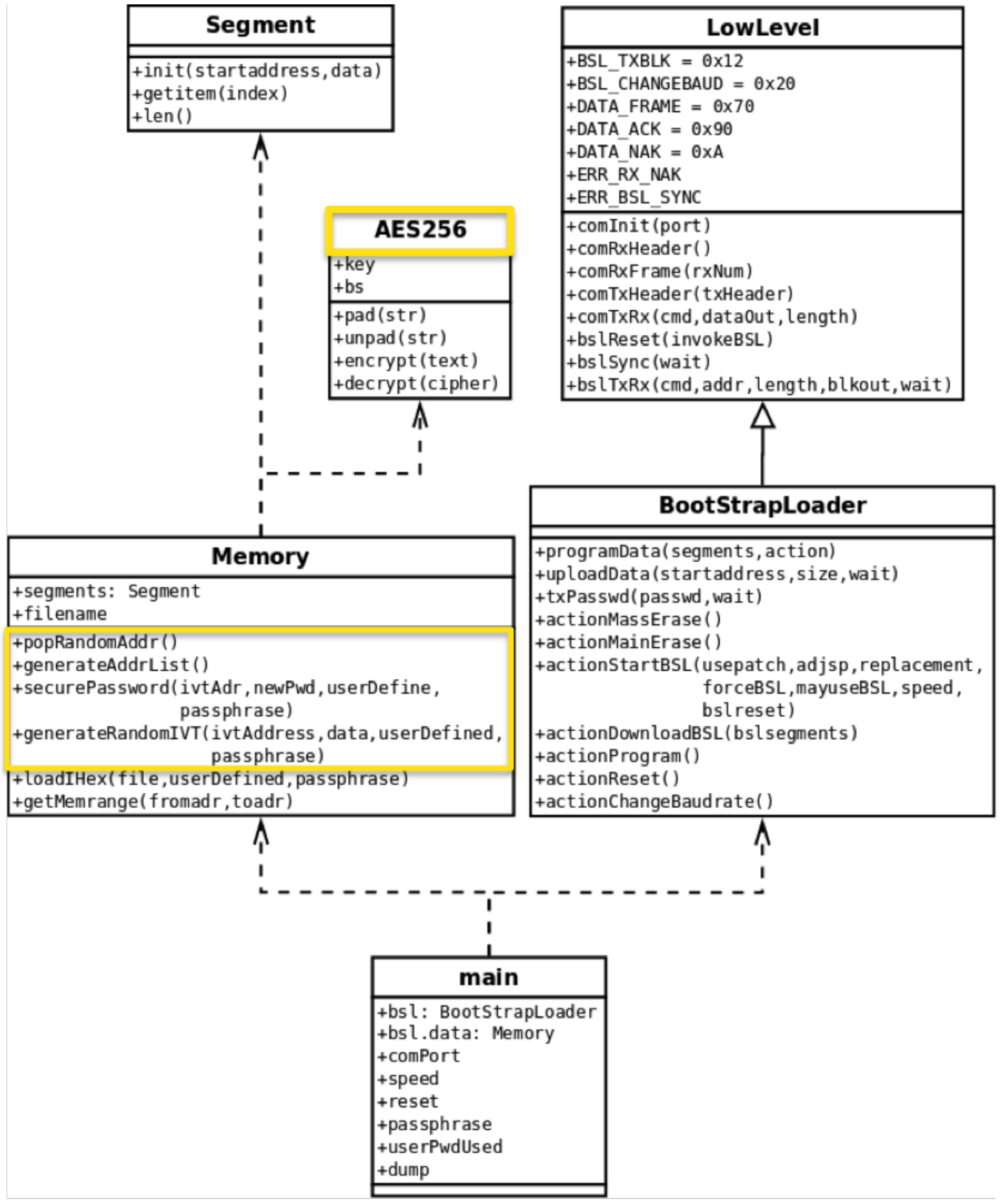


Figure 6.11: The UML diagram of the important components of the Secure-BSL software

Before discussing in detail our implementation, we must first highlight how the Secure-BSL works when flashing firmware into the MSP430 MCU. Figure 6.15 shows the flags used when flashing firmware into the MSP430 MCU. Below are the descriptions of each flag used with the Secure-BSL software when flashing firmware into the MSP430 MCU.

- –telosb: specifies that the sensor device connected is a TeloB mote.
- -c: specifies the port number where the TelosB mote is connected (e.g. /dev/ttyUSB0, /dev/ttyUSB1)
- -r: reset the TelosB mote
- -e: mass erase data found in TelosB flash memory
- -l: specifies the Intel Hex file type of the program
- -p: specifies the program to flash

Using the –telosb flag sets the appropriate settings, such as baud rate speeds before initiating any BootStrapLoader operations. The -c sets the connection port to the USB port where the TelosB mote is plugged in. The -r flag forces the TelosB to reset after the firmware has been flashed. The -e erases all data found in the TelosB mote; in other words, the -e flag erases all segments found in the MSP430 flash memory. The -I lets the program know that the binary to flash is in a ASCII text representation. The -p specifies the file that contains the binary of the sensor application to flash into the MSP430 flash memory.

Refer to the UML diagram for any clarifications as we explain the process of how flashing firmware works. The main class collects all the flags specified by a user when running the Secure-BSL software. The flags are used to set the appropriate setting before initiating communication with the mote, or in our case the MSP430 MCU. Essentially, the Secure-BSL is composed of two core components: the BootStrapLoader class and the Memory class. The BootStrapLoader class is responsible for exchanging information between the computer and the MSP430 using commands. For example, the txPasswd command is used to the check the password before granting full access to the MSP430 MCU. If the password is correct then other commands are granted, such as the actionProgram, programData or uploadData. The actionProgram command flashes data stored in a list of segments to the MSP430 main memory. The programData command allows the user to write N number of bytes to the flash memory without the need to re-flash the whole MCU. The uploadData command allows the user to extract data from firmware and is triggered if the -u flag and -s flag are used. Figure

6.17 shows that the -u flag specifies the start address in flash memory and the -s flag specifies the number of bytes to extract from flash memory. Other interesting commands that are supported prior to providing the password are the actionMassErase and the actionReset. These two commands do not allow you to extract firmware found in flash memory because the actionReset command will just reset the mote and the actionMassErase will erase all the data in the mote. The actionMassErase will also erase the password (IVT values) and set it to the default of 16 0xFF values. The BootStrapLoader class inherits the LowLevel class since the LowLevel class has the operations of communicating directly with the mote. The LowLevel class uses the python serial library to communicate directly with the MSP430 MCU. To exchange information between the MSP430 and the computer the serial library is used to read and write frames between the two devices. For example, to send data from the computer to the MSP430 MCU a frame is created with the DATA_FRAME (0x70) stored in the header section. Or if data is sent from the MSP430 to the computer, the frame will have the DATA_ACK command in the header section. There are various commands that are supported by the MSP430 that we do not list because the purpose of this chapter is to explain our Secure-BSL implementation. With that being said, let us now discuss our implementation, which was implemented on top of the second core components of the Secure-BSL, the Memory class.

As we can see from the UML diagram there are two components highlighted in yellow that were added to the original tos-bsl in order to implement our Secure-BSL software. The first component was added to the Memory class to implement the first security level of the Secure-BSL. The second component was added as an additional class used by the Memory class to implement the second level of security. First, we discuss the security level one implementation and then conclude by discussing the security level two implementation.

6.3.1 Security Level One Implementation

As previously stated, the Memory class was responsible for parsing the binary found in the main.ihex file into segments. Using -I flag with the Secure-BSL software sets the file type to ihex and invoked the loadIHex operation from the Memory class. The loadIHex operation was responsible for reading the data in the main.ihex file and parsing the data into a list of segments. All bytes found in the main.ihex file were transferred into the list of segments except for the IVT bytes. The loadIHex operation used the generateRandomIVT operation to generate random IVT values before adding the IVT bytes into the list of segments. The generateRandomIVT operation was responsible for generating new interrupt handler addresses, saving the old ISR, and updating the IVT values. To generate new interrupt handler addresses at random ,the generateRandomIVT operation used the generateAddrList operation and the popRandomAddr operation. To update the IVT values, the generateRandomIVT operation used the securePassword operation. The pseudo code for the generateRandomIVT operation is as follows:

```
generateAddrList()
Create empty list newIVT
FOR eachItem IN oldIVT
    Create branch opcodes 30 40 eachItem
    newISR = popRandomAddress()
    Append opcodes newISR: 30 40 eachItem
    Append newISR to newIVT list
updateNewPassword(newIVT)
```

The newIVT list held the new IVT values that were selected at random from the list of the unused address. The opcodes 30 40 represented the branch instruction op codes for the MSP430 instruction set. A branch instruction was created for every entry on the old IVT and a randomly selected unused address was picked as the new IVT entry. The list of segments was updated accordingly and the new randomly chosen interrupt handler address was added to the new IVT.

The generateAddrList operation was responsible for generating a list of unused address that could be used as a new interrupt handler address. The start of the unused address space was calculated by adding the size of the binary with the starting address of main memory. The end of the unused address space was always the starting address of the IVT, which was always 0xFFE0. The start address of main memory may have varied depending on the version of the MSP430 MCU (refer to Figure 5.1). The unused addresses that were added to the list had to be able to hold 4-bytes of data (the branch opcodes). The pseudo code for the generateAddrList operation is as follows:

```
Calculate start address $start = sizeOfBinary + 0x4000$
FOR address IN range (start to 0xffe0, skip every 4)
    Append address to Unused-Address-list
```

After the list of unused addresses were generated, the popRandomAddr operation allowed to randomly pop one on the list. Therefore, the popRandomAddr always returned an unused address at random, and by removing the item from the list we avoided using the same address for two different interrupt handlers. The pseudo code for the popRandomAddr operation is as follows:

```
Shuffle Unused-Address-list
Pop address from Unused-Address-list
```

The securePassword operation is discussed in the following section because it interlinks with the security level two.

6.3.2 Security Level Two Implementation

The security level two feature of the Secure-BSL is built on top of the security level one implementation. The securePassword operation implemented most of the functionality of the security level two but also implemented part of the security level one. The responsibilities of the securePassword operation was to add the new IVT to the list of segments and output the new IVT to the user. The new IVT was added to the list of segments and was later flashed into the MSP430 main memory using the BootStrapLoader class. The new IVT was also written to the password file; however, this was where the operation varied depending on the security level. The new IVT values were written to the password file unencrypted if the security level one was used. The new IVT values were written to the password file encrypted if the security level two was used. The -t flag specified to the Secure-BSL software to enable a two-factor authentication by using a user defined passphrase. The pseudo code for the securePassword operation is as follows:

```
Append $newPwd$ to list of segments
Convert $newPwd$ to hex $hexPwd$
IF passphrase is used
    Create AES256 object $encryptPWD$
    Encrypt $hexPwd$ $cipherPwd = encryptPWD.encrypt(hexPwd)$
    Write $cipherPwd$ to $password$ file
ELSE
    Write $hexPwd$ to $password$ file
```

The AES256 class was used to encrypt the password file using a user defined password. Figure 6.12 shows the parameters passed to the AES256 class in order to encrypt the password file. The IV used was generated using the Random python library package, which guaranteed randomness with the password and achieved semantic security. The password file contained either the encrypted or decrypted version of the new IVT depending on the security level used.

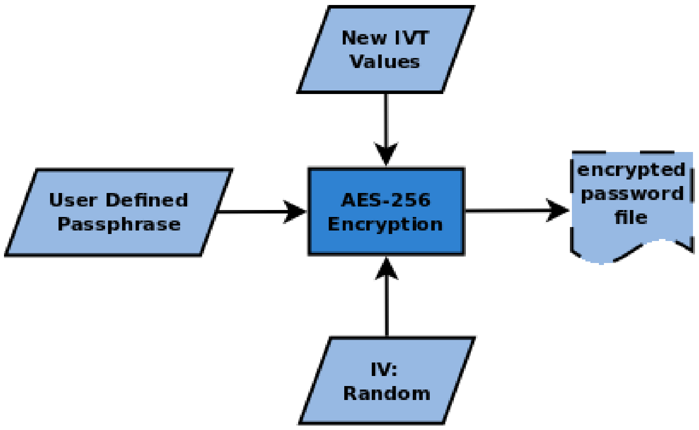


Figure 6.12: The input and output of the AES 256 encryption.

The last component of the Secure-BSL was the process of extracting binary data found in the MSP430 flash memory. The operation to extract data from flash memory was handled by the main class by first checking if the user had requested a memory dump. Figure 6.21 shows the flags used to initiate a memory dump using the Secure-BSL software. The -u and -s will specified the range of memory to extract from the MSP430 flash memory. The -P flag specifies the password file to use which either held an encrypted or decrypted version of the IVT. The -t specified the user passphrase to use in order to decrypt the password file before sending the IVT values to the mote. Therefore, the main class was responsible for decrypting the password file if a user defined passphrase had been used to encrypt the password file. The pseudo code for a memory dump operation is as follows:

```
IF dump firmware requested
Read password file $pwdBytes$
    IF passphrase is used
        Create AES256 object $decryptPWD$
        Decrypt $pwdBytes$ $pwdBytes = decryptPWD.decrypt(pwdBytes)$
        IF decryption error occurs
            THEN wrong password
    FOR $byte$ IN $pwdBytes$
        convert $byte$ to $character$
        add $character$ to $password$
Send $password$ to MSP430
```

If the password file had not been encrypted then the values stored in the password file were converted to characters before sending them to the MSP430. The MSP430 checked to see if the correct password had been sent before granting full access the MCU. Figure 6.13 shows the parameters that were passed to the AES256 class in order to decrypt the file.

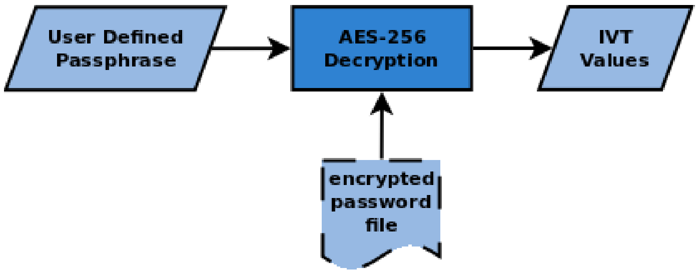
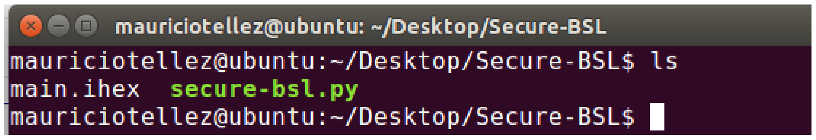


Figure 6.13: The input and output of the AES 256 decryption.

6.4 Secure-BSL Testing

We created a Secure-BSL testing environment by creating a directory that held the main.ihex file and the secure-bsl.py file. Figure 6.14 shows the testing environment for our Secure-BSL software. The main.ihex file is the ASCII text representation of the binary created from the STMS L1-Secure Coordinator application. However, any TinyOS applications can be used to obtain a main.ihex file to be used as an example. The secure-bsl.py file is our Secure-BSL software and can be dowloaded from our Github project[75].

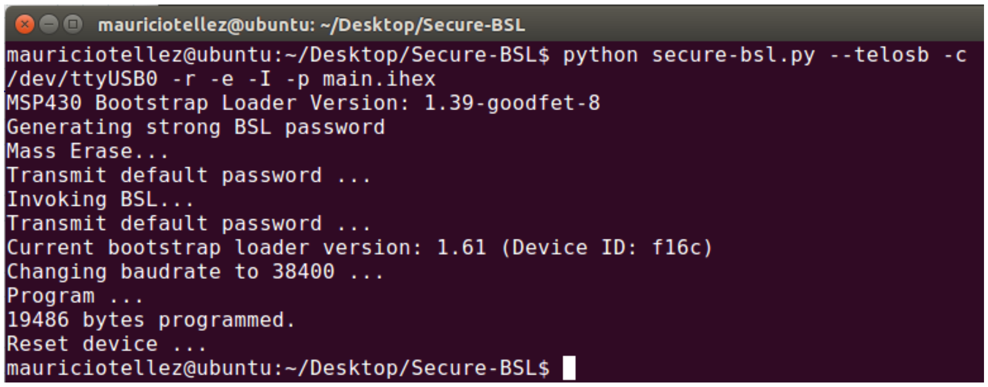


```
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$ ls
main.ihex  secure-bsl.py
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$ 
```

Figure 6.14: The Secure-BSL testing environment consisting of the main.ihex file and secure-bsl.py file

6.4.1 Security Level One Tests

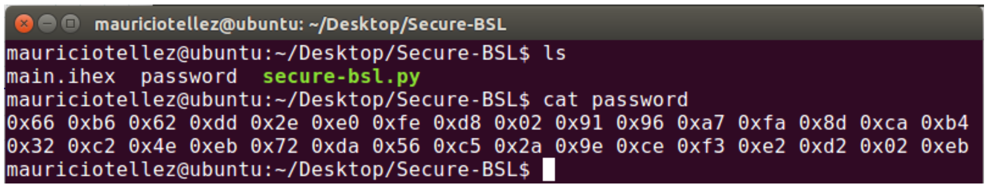
The first test performed was to check if flashing an application to the MSP430 properly worked when using the security level one. Figure 6.15 shows the flags used to flash firmware using our Secure-BSL software. After programming the mote with the L1-Secure Coordinator application we performed the same steps as discussed in Chapter 3.3 to check if the STMS L1-Secure WSN worked properly. The results show that using the Secure-BSL software to flash firmware does not affect the functionality of STMS. Therefore, we prove that our solution of randomizing the IVT and using branch instructions does maintain the functionality of MSP430 application.



```
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$ python secure-bsl.py --telosb -c
/dev/ttyUSB0 -r -e -I -p main.ihex
MSP430 Bootstrap Loader Version: 1.39-goodfet-8
Generating strong BSL password
Mass Erase...
Transmit default password ...
Invoking BSL...
Transmit default password ...
Current bootstrap loader version: 1.61 (Device ID: f16c)
Changing baudrate to 38400 ...
Program ...
19486 bytes programmed.
Reset device ...
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$
```

Figure 6.15: An example of using the Secure-BSL security level one to flash firmware into the MSP430

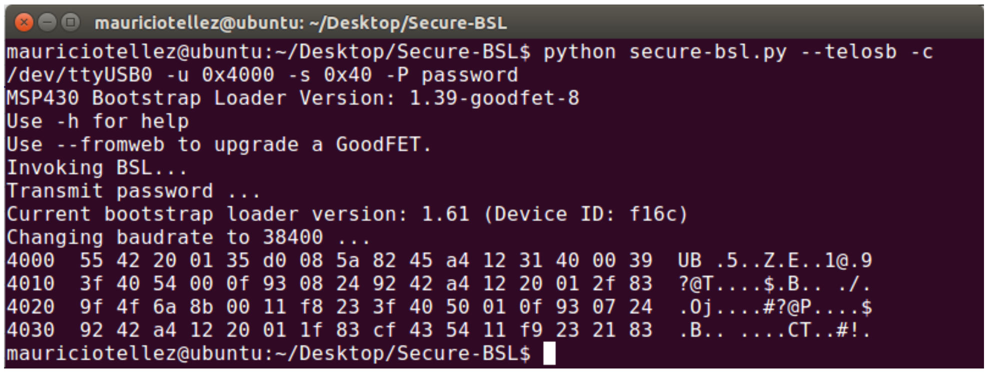
After flashing the firmware into the MSP430 MCU, the Secure-BSL software generated a password file that contained the new IVT values or the new password. Figure 6.16 shows the content of the password file generated by the secure-bsl.py script. The values found in the password file were randomly generated by the Secure-BSL software and were used for future access to the MSP430 MCU.



```
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$ ls
main.ihex  password  secure-bsl.py
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$ cat password
0x66 0xb6 0x62 0xdd 0x2e 0xe0 0xfe 0xd8 0x02 0x91 0x96 0xa7 0xfa 0x8d 0xca 0xb4
0x32 0xc2 0x4e 0xeb 0x72 0xda 0x56 0xc5 0x2a 0x9e 0xce 0xf3 0xe2 0xd2 0x02 0xeb
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$
```

Figure 6.16: The content found in the password file generated by the Secure-BSL software

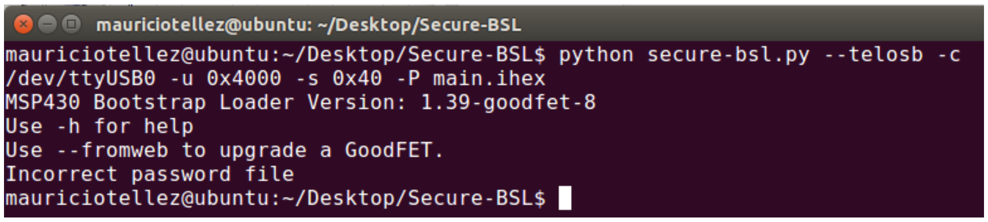
The second test performed was to check if extracting firmware using the correct password file properly worked. Figure 6.17 shows the parameters used to extract firmware found in flash memory using our Secure-BSL software. The -P flag specified the password file to use, in which we provided the correct password file that was generated during test one. The results show a successful flash memory dump of 64-bytes starting at address 0x4000.



```
mauriciotellez@ubuntu:~/Desktop/Secure-BSL
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$ python secure-bsl.py --telosb -c
/dev/ttyUSB0 -u 0x4000 -s 0x40 -P password
MSP430 Bootstrap Loader Version: 1.39-goodfet-8
Use -h for help
Use --fromweb to upgrade a GoodFET.
Invoking BSL...
Transmit password ...
Current bootstrap loader version: 1.61 (Device ID: f16c)
Changing baudrate to 38400 ...
4000  55 42 20 01 35 d0 08 5a 82 45 a4 12 31 40 00 39  UB .5..Z.E..1@.9
4010  3f 40 54 00 0f 93 08 24 92 42 a4 12 20 01 2f 83  ?@T....$.B.. ./.
4020  9f 4f 6a 8b 00 11 f8 23 3f 40 50 01 0f 93 07 24  .Oj....#?@P....$
4030  92 42 a4 12 20 01 1f 83 cf 43 54 11 f9 23 21 83  .B.. ....CT..#!.
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$
```

Figure 6.17: An example of extracting firmware found in flash memory using the correct password file

The third test performed was to check the security of extracting firmware using an incorrect password file. Figure 6.18 shows the results of using the incorrect password file to extract firmware. We specified to use the main.ihex file, which contained the original values of the IVT before they were randomized using our Secure-BSL software. The results show that without having the correct password file, full access to the mote was restricted even with a copy of the original main.ihex file.

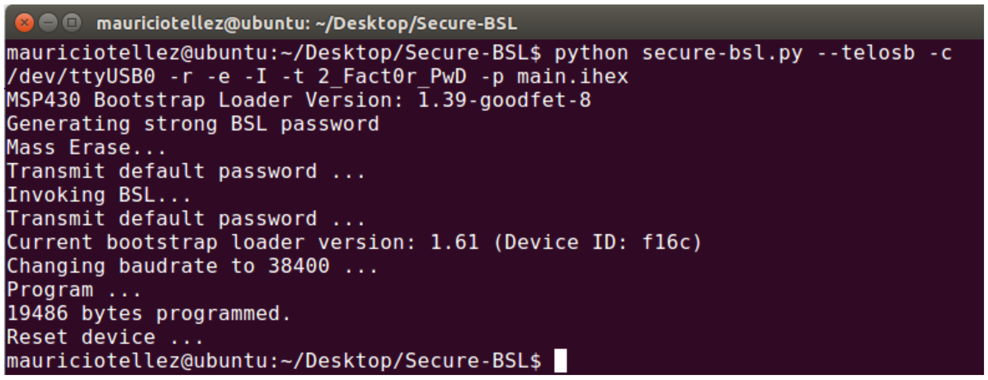


```
mauriciotellez@ubuntu:~/Desktop/Secure-BSL
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$ python secure-bsl.py --telosb -c
/dev/ttyUSB0 -u 0x4000 -s 0x40 -P main.ihex
MSP430 Bootstrap Loader Version: 1.39-goodfet-8
Use -h for help
Use --fromweb to upgrade a GoodFET.
Incorrect password file
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$
```

Figure 6.18: An example of extracting firmware found in flash memory using the incorrect password file

6.4.2 Security Level Two Tests

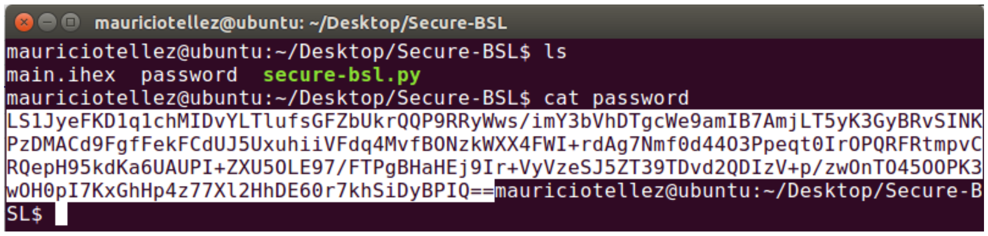
The fourth test performed was to check if flashing an application to the MSP430 properly works when using the security level two. Figure 6.19 shows the additional flag (-t) used to flash the program using the security level two. The -t flag allowed the developer to specify a user defined passphrase to be used to protect the password file. Since the actual bytes flashed into the MSP430 memory were not affected by the security level two there was no need to check if the functionality of STMS is maintained. We have already shown from test one that the L1-Secure application properly worked when using the Secure-BSL software.



```
mauriciotellez@ubuntu: ~/Desktop/Secure-BSL
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$ python secure-bsl.py --telosb -c
/dev/ttyUSB0 -r -e -I -t 2_Fact0r_PwD -p main.ihex
MSP430 Bootstrap Loader Version: 1.39-goodfet-8
Generating strong BSL password
Mass Erase...
Transmit default password ...
Invoking BSL...
Transmit default password ...
Current bootstrap loader version: 1.61 (Device ID: f16c)
Changing baudrate to 38400 ...
Program ...
19486 bytes programmed.
Reset device ...
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$
```

Figure 6.19: An example of using the Secure-BSL security level two to flash firmware into the MSP430

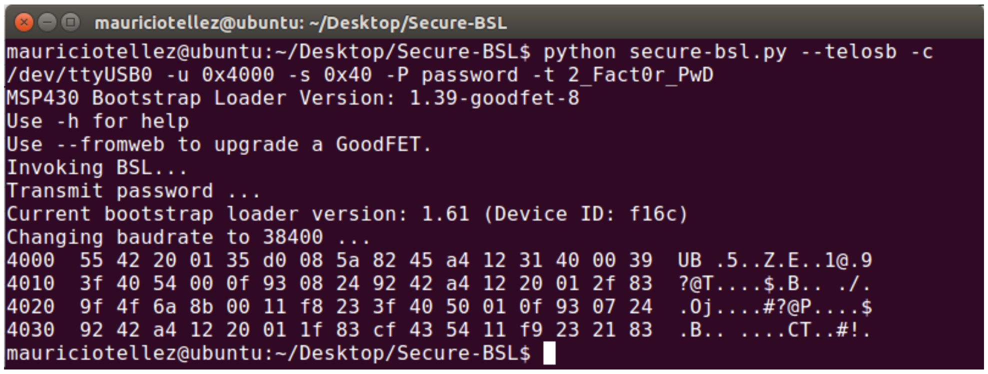
Flashing firmware using security level two generated a password file that contained an encrypted version of the new IVT values. Figure 6.20 shows the content of the password file highlighted in white. The values found in the password file were encrypted using AES-256 and the user defined passphrase was used as the key.



```
mauriciotellez@ubuntu: ~/Desktop/Secure-BSL
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$ ls
main.ihex  password  secure-bsl.py
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$ cat password
LS1JyeFKD1q1chMIDvYLTlufsGFZbUkrQQP9RRyWws/imY3bVhDTgcWe9amIB7AmjLT5yK3GyBRvSINK
PzDMACd9FgfFekFCdUJ5UxuhiiVFdq4MvfBONzkWXX4FWI+rdAg7Nmf0d4403Ppeqt0IrOPQRFRtmpvC
RQepH95kdKa6UAUPI+ZXU50LE97/FTPgBHaHEj9Ir+VyVzeSJ5ZT39TDvd2QDIzV+p/zw0nT0450OPK3
wOH0pI7KxGhHp4z77Xl2HhDE60r7khSiDyBPIQ==mauriciotellez@ubuntu:~/Desktop/Secure-B
SL$
```

Figure 6.20: The encrypted content found in the password file generated by the Secure-BSL software

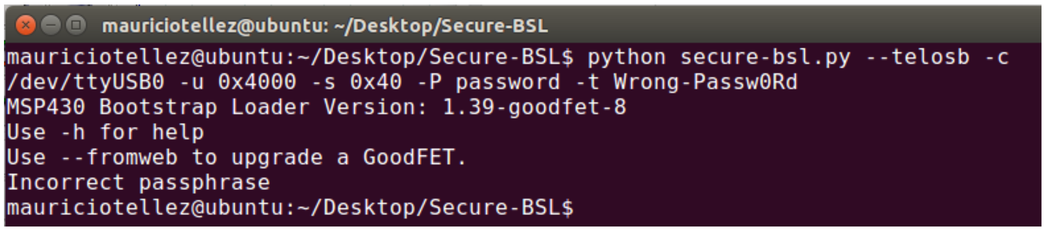
The fifth test performed was to check if extracting firmware using the correct password file and the correct passphrase properly worked. Figure 6.21 shows the additional (-t) parameter to use to extract firmware found in flash memory. We provided the correct encrypted password file that was generated during test four and the correct user defined passphrase. The results show a successful flash memory dump of 64-bytes starting at address 0x4000.



```
mauriciotellez@ubuntu: ~/Desktop/Secure-BSL
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$ python secure-bsl.py --telosb -c
/dev/ttyUSB0 -u 0x4000 -s 0x40 -P password -t 2_Fact0r_PwD
MSP430 Bootstrap Loader Version: 1.39-goodfet-8
Use -h for help
Use --fromweb to upgrade a GoodFET.
Invoking BSL...
Transmit password ...
Current bootstrap loader version: 1.61 (Device ID: f16c)
Changing baudrate to 38400 ...
4000  55 42 20 01 35 d0 08 5a 82 45 a4 12 31 40 00 39  UB .5..Z.E..1@.9
4010  3f 40 54 00 0f 93 08 24 92 42 a4 12 20 01 2f 83  ?@T....$.B.. ./.
4020  9f 4f 6a 8b 00 11 f8 23 3f 40 50 01 0f 93 07 24  .Oj....#?@P....$
4030  92 42 a4 12 20 01 1f 83 cf 43 54 11 f9 23 21 83  .B.. ....CT..#!.
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$ 
```

Figure 6.21: An example of extracting firmware found in flash memory using the correct password file and the correct passphrase

The sixth test performed was to check the security of extracting firmware using an incorrect passphrase. Figure 6.22 shows the results of using the incorrect passphrase to decrypt the password file. We specified to Secure-BSL software to use a different passphrase that was used to encrypt the password file. The results show that without having the correct passphrase an attacker would not be able to access the MSP430 MCU even after stealing a copy of the password file.



```
mauriciotellez@ubuntu: ~/Desktop/Secure-BSL
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$ python secure-bsl.py --telosb -c
/dev/ttyUSB0 -u 0x4000 -s 0x40 -P password -t Wrong-Passw0Rd
MSP430 Bootstrap Loader Version: 1.39-goodfet-8
Use -h for help
Use --fromweb to upgrade a GoodFET.
Incorrect passphrase
mauriciotellez@ubuntu:~/Desktop/Secure-BSL$
```

Figure 6.22: An example of extracting firmware found in flash memory using the incorrect passphrase

6.5 Chapter Summary

In summary, we have contributed to the WSN community with a working solution to the problem of using a weak MSP430-BSL password. We investigated possible solutions to improve the MSP430-BSL password used to protect firmware found in flash memory. Our results show that using the hash of the binary breaks the overall functionality of the MSP430 application. Instead, we investigated and used a randomization approach where the values of the IVT were replaced with random unused addresses. The use of branch instructions to save the original IVT entries guarantees that our solution does not break the overall functionality of an MSP430 application. We presented the reader with our design approach in fixing the weak MSP430-BSL password and implemented a working version by building on top of the default bootstrap loader software (tos-bsl). We tested our Secure-BSL software and provided a fully functionality MSP430 bootstrap loader software to the MSP430 family community. Chapter 6 serves as a guideline on how the Secure-BSL software works as well as contributes proof that our Secure-BSL software truly does secure data found in the MSP430 MCU.

Chapter 7

Results and Evaluations of Secure-BSL

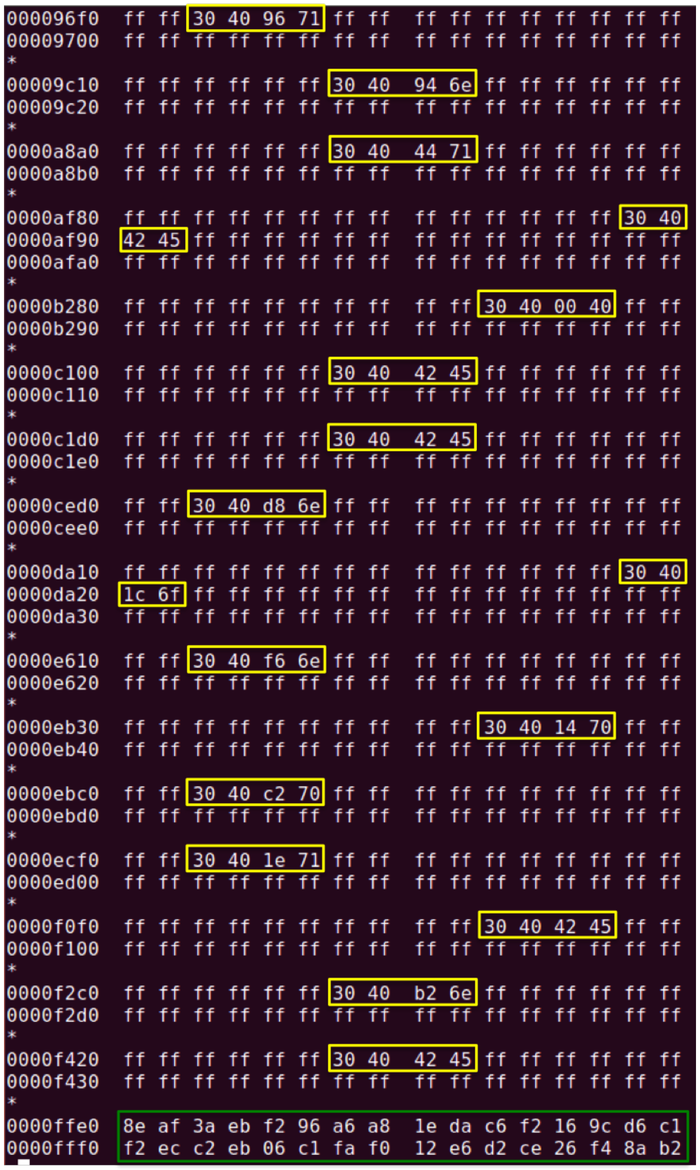
In this chapter, we evaluate the security strength of the improved MSP430-BSL password generated by our Secure-BSL software. We evaluate the memory overhead and computational overhead of using our Secure-BSL software. Next, we prove that our Secure-BSL software provides security enhancements in protecting the MSP430 MCU from feature mote capture attacks. More importantly we contribute to the WSN community investigation results that analyze the strength of the improved MSP430-BSL password. First, we investigate any patterns found in the randomized version of the MSP430-BSL password. Second, we investigate worst case scenarios to calculate the brute force time to break our secure MSP430-BSL password. Lastly, we investigate the security strength of using the two-factor authentication and the likelihood of an attacker successfully accessing firmware.

7.1 Resource Overhead Evaluation

The memory overhead and computational overhead of randomizing the MSP430-BSL password is not significant. The security enhancements obtained from using the Secure-BSL outweighs the overhead added to the MSP430 MCU. We evaluated the memory overhead of adding additional opcodes used to implement the branch instructions. Next we evaluated the computational over head of using the branch instructions to maintain the original IVT functionality.

The memory overhead generated by the Secure-BSL depended on the number of interrupts found in the IVT. We know that the MSP430 supported 16 interrupts; therefore, we used this number to calculate the number of extra bytes added to memory. Figure 7.1 shows a sample of the extra bytes added to main memory when flashing an application with the Secure-BSL software. The yellow boxes show the 4-bytes opcodes used to implement the branch instructions and maintain proper functionality of the IVT. For example, the last 2-bytes in the IVT highlighted in green show the hex values 8a b2. The hex values correspond to the interrupt handler address 0xb28a (little endian) found at IVT entry 15. The values found at address 0xb28a are 4-byte opcodes 30 40 00 44, which translates to the following instruction: br 0x4000. The branch instruction found at address 0xb28a is used to maintain proper functionality and points to the entry address of the application. With that being said, a total of 4-bytes are needed to maintain the original 16 ISR addresses found in the IVT. The total memory overhead generated by the Secure-BSL is 64 (7.1) bytes. The 64 bytes of code added a minor memory overhead to the overall MSP430 flash memory.

$$16 \quad interrupts \times 4 \quad bytes \qquad or \qquad 64 \quad bytes \tag{7.1}$$



```
000096f0  ff ff 30 40 96 71 ff ff  ff ff ff ff ff ff ff ff
00009700  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
*
00009c10  ff ff ff ff ff ff 30 40  94 6e ff ff ff ff ff ff
00009c20  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
*
0000a8a0  ff ff ff ff ff ff 30 40  44 71 ff ff ff ff ff ff
0000a8b0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
*
0000af80  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff 30 40
0000af90  42 45 ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
0000afa0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
*
0000b280  ff ff ff ff ff ff ff ff  ff ff 30 40 00 40 ff ff
0000b290  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
*
0000c100  ff ff ff ff ff ff 30 40  42 45 ff ff ff ff ff ff
0000c110  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
*
0000c1d0  ff ff ff ff ff ff 30 40  42 45 ff ff ff ff ff ff
0000c1e0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
*
0000ced0  ff ff 30 40 d8 6e ff ff  ff ff ff ff ff ff ff ff
0000cee0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
*
0000da10  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff 30 40
0000da20  1c 6f ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
0000da30  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
*
0000e610  ff ff 30 40 f6 6e ff ff  ff ff ff ff ff ff ff ff
0000e620  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
*
0000eb30  ff ff ff ff ff ff ff ff  ff ff 30 40 14 70 ff ff
0000eb40  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
*
0000ebc0  ff ff 30 40 c2 70 ff ff  ff ff ff ff ff ff ff ff
0000ebd0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
*
0000ecf0  ff ff 30 40 1e 71 ff ff  ff ff ff ff ff ff ff ff
0000ed00  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
*
0000f0f0  ff ff ff ff ff ff ff ff  ff ff 30 40 42 45 ff ff
0000f100  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
*
0000f2c0  ff ff ff ff ff ff 30 40  b2 6e ff ff ff ff ff ff
0000f2d0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
*
0000f420  ff ff ff ff ff ff 30 40  42 45 ff ff ff ff ff ff
0000f430  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
*
0000ffe0  8e af 3a eb f2 96 a6 a8  1e da c6 f2 16 9c d6 c1
0000fff0  f2 ec c2 eb 06 c1 fa f0  12 e6 d2 ce 26 f4 8a b2
```

Figure 7.1: A sample of memory after flashing an application using secure-BSL.

To calculate the computational overhead required investigating the number of clock cycles needed to execute the branch instruction. As discussed in Chapter 5.2, the branch instruction was an emulated instruction that was replaced with a core instruction (refer to Table 5.3). Since a branch instruction is an unconditional jump to an address anywhere in memory then the emulated BR instruction is replaced with the JMP core instruction[60]. Therefore, the number of clock cycles required to execute the JMP instruction was the same as the number of clock cycles required to execute the branch instruction. Using the MSP430 Users Guide manual[60] we determined that it requires 2 clock cycles to execute a JMP instruction. With that being said, our random IVT approach had a computation overhead of 2 clock cycles per interrupt event. For example, our random IVT approach required an additional 2 clock cycles to handle a Power-Up interrupt and successfully start a MSP430 application. The additional 2 clock cycles were only applied whenever an interrupt occurred, which was less frequent than the actual user code execution. Since the MSP430, when powered by a USB, handles 7 million clock cycles per second, an additional 2 clock cycles is a very small computational overhead. The additional 64-bytes of code and 2 clock cycles per interrupt did not affect the overall MSP430 application. Instead, the security posture of the MCU was enhanced at little to no cost.

7.2 Password Strength Investigation Results

Using the same approach discussed in Chapter 4.3, we generated password samples using a modified version of the generate-pwds.sh script. The modified version used the Secure-BSL software to generate the MSP430-BSL passwords for both the security levels (one-factor and two-factor). The modified script and all other scripts used to analyze the Secure-BSL passwords can be downloaded from our Github project[75]. In addition, the output of all of the scripts and any analysis not included in Appendix B can be found in our Github project. The application's sample size is 93 and was obtained from the TinyOS /apps directory. The covert-endianess.py script was used to convert the 32-bytes password to the corresponding IVT entries (little endian) for each of the 93 applications. The results showed that the IVT entries generated by the Secure-BSL software were indeed random and every IVT entry was unique. Unlike the original bootstrap loader (tos-bsl), our Secure-BSL was able to generate unique IVT entries for every request made to flash firmware. For instance, we saw in Figure 4.4 how the tos-bsl software had duplicates at IVT entries 0, 10, 11, and 14. Duplicate IVT entries are the results of applications not using all the interrupts supported by the MSP430. Furthermore, we established that the Power-Up interrupt at IVT entry 15 was the same for all 93 applications because it always points to the entry point of the application. Our Secure-BSL approach improved the security of the MSP430-BSL password by randomizing the values stored in these unused, or default, interrupts. Table B.1 shows results of generating random interrupt handler addresses for IVT entries 0, 10, 11, 14, and 15 for all 93 applications. The results show that not a single entry in the IVT were duplicates and holds true for all other 11 interrupts not shown in Table B.1. We only presented five of the sixteen interrupts as an example; however, the complete set of IVT entries for all 93 applications can be found in our Github project[75]. In addition, to double check that all of the IVT entries are unique we ran the IVT-duplicates.py script on our applications sample. Our results, shown on Table B.2, show that not a single application has a duplicate IVT entry. Every single IVT entry is unique for all applications; hence, we see the number 16 in the IVT entries column.

In Chapter 4.3, we demonstrated how the MSP430-BSL password can be brute forced in a matter of days by sorting the ISR address found in the IVT. The graph in Figure 4.5 showed how sorting the ISR addresses produced patterns than could be used to predict future MSP430-BSL passwords. For example, the red line in Figure 7.2 represents the pattern that is persistent between 13 applications that have eleven unused interrupts. The pattern is based on the IVT position after sorting the ISR address found in each IVT entry. The pattern makes the password predictable because we prove that future applications with 11 unused interrupts have the following sorted pattern: 15, 0, 10, 11, 14, 7, 3, 1, 8, 9, 2, 4, 6, 5, 13, and 12. Unlike the tos-bsl software, our Secure-BSL software truly enforces the security of the MSP430-BSL password by randomizing the values found in the IVT. In this chapter, we refer to the IVT generated by the Secure-BSL as ”secure IVT” and the IVT generated by the tos-bsl as ”original IVT”. The bars in Figure 7.2 represent the IVT positions after sorting the interrupt handler addresses found in the secure IVT entries. The same 13 applications that were used to collect the data for the line graph were used to collect the data for the bar graphs. The data can be found in Table B.3; however, the graph presents a visual representation of the randomness found in the secure IVT entries. For example, for the 13 applications examined, the first address of the sorted interrupt handlers corresponds to the following secure IVT positions: 12, 5, 9, 0, 11, 7, 2, 1, 6, 5, 1, 13, and 15, where the first address of the sorted ISRs for all 13 applications correspond to the original IVT entry 15. We can conclude that even if the entry address 0x4000 was always stored in IVT entry 15, our Secure-BSL software was capable of randomizing the value stored in the secure IVT entry 15 while maintaining sensor application functionality.

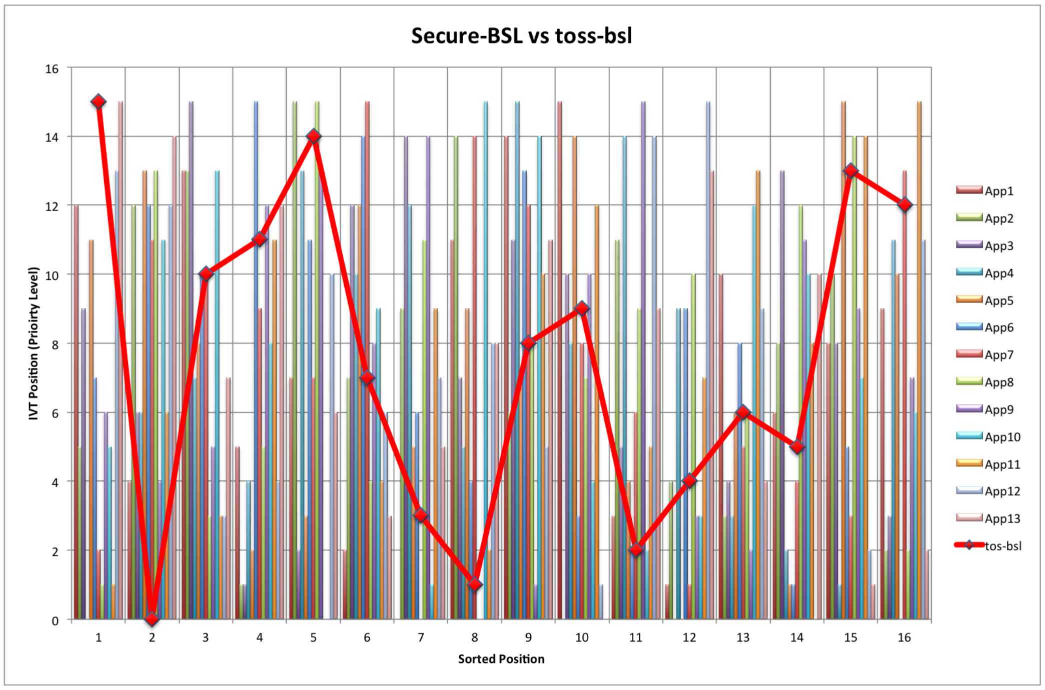


Figure 7.2: Comparing the results of the applications with 11 duplicates when using the tos-bsl vs the Secure-BSL .

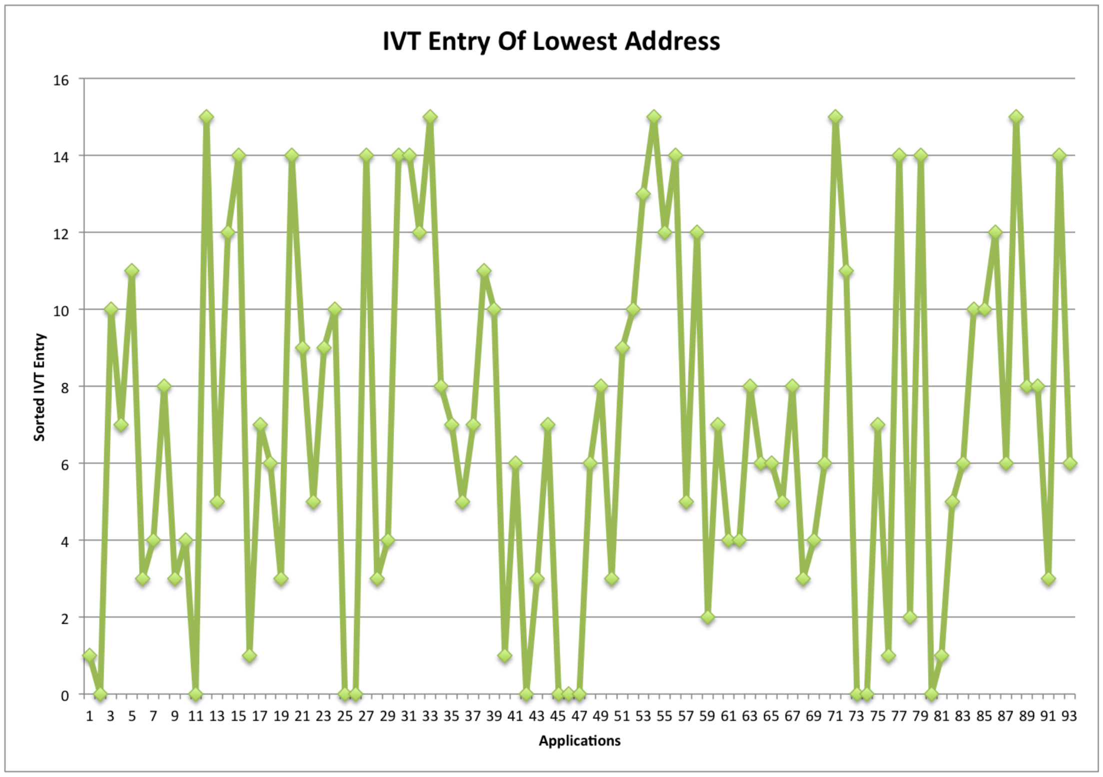


Figure 7.3: Results of the first IVT entry when sorting the IVT addresses.

To get an in-depth analysis of the level of randomness based on brute force times we analyzed the rest of the applications in our sample to investigate for any predictable patterns. If any patterns were found in the sorted version of the secure IVT entries then our approach would have practical brute force times. For all the applications in our sample we investigated the first address (lowest address) found in the sorted version of the secure IVT entries. Figure 7.3 shows the priority level corresponding to the lowest address found in the secure IVT of each application. The results show that the Secure-BSL software generated true randomness because IVT entries that correspond with the lowest addresses were unpredictable. For example, for the first ten applications the IVT entry that corresponded to the lowest address were as follows: 1, 0, 10, 7, 11, 3, 4, 8, and 3. Unlike the original IVT version where the entry was always 15, our Secure-BSL did not provide practical patterns for future password predictions. For the sake of completeness, Figure 7.4 shows the priority levels corresponding to the sorted address found in the secure IVT of each application. The numbers around the circle represent the applications and the numbers in the inner circle represent the IVT entry. The circle graph shows that there was not a single line that created a perfect circle, which would indicate the IVT entry being constant between all 93 applications. We can conclude that there was not a single indicator of patterns when using our Secure-BSL; thus, generating unpredictable MSP430-BSL passwords.

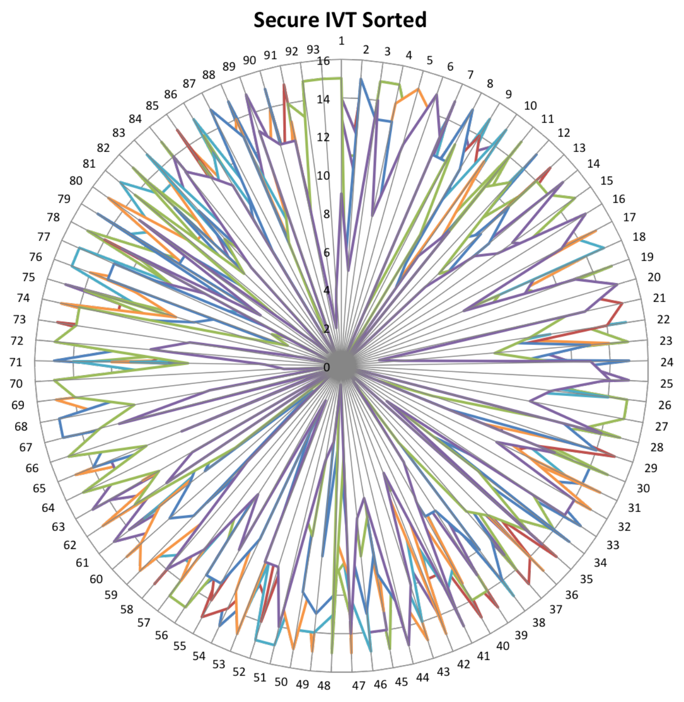


Figure 7.4: Results of all IVT entries when sorting the IVT addresses.

In Chapter 4.3, we demonstrated that the key to reducing the brute force time to a matter of days lies behind the fact that there were significant patterns found by taking the difference between sorted addresses found in the original IVT entries. Therefore, it was important to investigate whether our secure IVT values lacked of this MSP430-BSL password security flaw. We used IVT-Entry-Diff.py script to take the difference of the sorted addresses found in the secure IVT entries for each application. Figure 7.5 shows a comparison of the MSP430-BSL password for the Blink application using the Secure-BSL (in green) vs the tos-bsl (in red). The tos-bsl results show that taking the difference of the sorted addresses significantly decreased the brute force time because it only required to brute force 2 of the 16 original IVT entries. Furthermore, the possible combinations for these 2 IVT entries were in the address range of $0x4000 + 900$ or 0x4380 to $0x4000 + 1500$ or 0x45DC. Where as the Secure-BSL results show that taking the difference of the sorted addresses did not provide patterns for potential brute force time reduction. The difference between sorted addresses for the Secure-BSL were not consistent and varied between addresses at different IVT entries. For example, the difference between the second lowest address and the lowest address was roughly 4700 bytes. The differences between sorted addresses for the Blink application were in the range of 300 up to 8100-bytes and were random between addresses. Impractical brute for times still holds true for the Secure-BSL approach even after taking the difference between sorted addresses found in the secure IVT entries. The results of taking the difference between sorted addresses remain random for all 93 applications found in our sample.

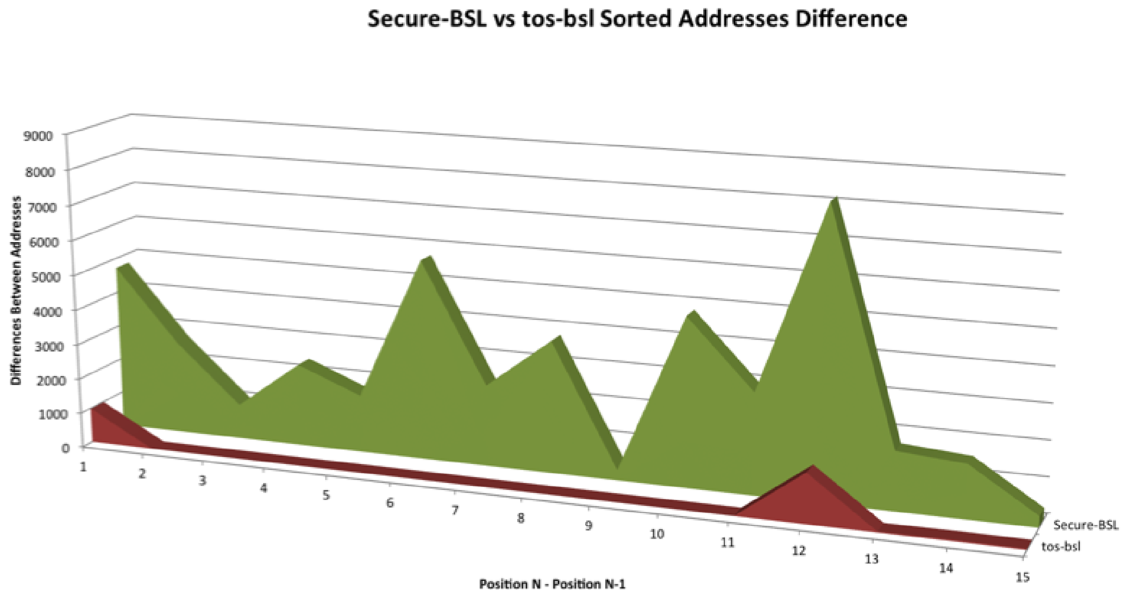


Figure 7.5: Comparison of taking the difference of the Blink application using Secure-BSL vs tos-bsl.

Our investigation results show that the MSP430-BSL password generated by our Secure-BSL software significantly increases brute force times. Even if an attacker is able to generate password samples using the Secure-BSL, they are still not able to deduct a concrete pattern to use for future password predictions. Furthermore, even if they generate password samples for the same application they are still not able to deduct a concrete pattern because the Secure-BSL is independent of the bytes found in a binary. Every time a new password is generated the output is always different than the previous result. The Secure-BSL software selects a random and unused address found in main memory. Therefore, the only factor that affects the generated password is the binary size found in flash memory.

Since the generated MSP430-BSL password depended on the unused space found in the main memory, we investigated the correlation between unused space and brute force time. We investigated the worst case scenario where the unused space was big enough to implement the Securel-BSL randomization approach. In order for our Secure-BSL approach to work, there had to be at least 64-bytes (4-bytes per interrupt) of unused space available in order to maintain application functionality. Therefore, the following assumptions can be made for our worst case scenario.

- Attacker knows the end address of the unused space: 0xFFE0 (IVT Start Address)
- Attacker knows the start address of the unused space: 0xFFA0 (0xFFE0 - 64)
- Attacker knows memory layout: Addressing space (even numbers only)

Since code access is always performed on even addresses then out of the 64-bytes we only care about the even numbers; therefore, reducing the set of addresses to 32. The address set N looks as follows:

N={0xFFA0, 0xFFA2, 0xFFA4, ... , 0xFFDA, 0xFFDC , 0xFFDE}

The set N has 32 items that correspond to possible addresses that could be found in the random IVT generated by the Secure-BSL. Since there were a total of 16 interrupts in the IVT we calculated the number of possible combinations of 16 items (addresses) from set N. We used the permutation formula (7.2) shown below to calculate the number of possible permutations of r items from set n where the order of the items matter.

$$\frac{n!}{(n-r)!} \tag{7.2}$$

In our worst case scenario the value for N is 32 and the value for R is 16 (7.3).

$$\frac{32!}{(32-16)!} \qquad or \qquad 1.25 \times 10^{22} \tag{7.3}$$

The total possible IVT value combinations that can be generated from a set of 32 possible addresses is 12,576,278,705,767,096,320,000 (7.3). We know that our TelosB mote was capable of checking 2280 passwords per second at a baud rate 38400 (refer to Figure 4.3). Therefore, to brute force our approach of using a randomized IVT in the worst case scenario with an address space of 32 would take 1.04×10^{13} (7.4) years! We can conclude that in the worst case scenario where the binary is $(0xFFE0 - 64) - 0x4000$ or 0xBFA0 (49056) bytes in size, it is not practical to brute force. Reducing the set of addresses to 32 still does not make it practical to brute force because of the 16 interrupts addresses required in each permutation.

$$((((1.25 \times 10^{22} \div 2280) \div 60) \div 24) \div 365) \qquad \mathit{or} \qquad 1.04 \times 10^{13} \tag{7.4}$$

We further examined the worst case scenario where the attacker was aware of how our implementation worked. For instance, the attacker was able to get a copy of this thesis and examined Chapter 6 to get information on how the Secure-BSL was implemented. In particular, the attacker was interested in knowing that the Secure-BSL used 4-bytes to store the opcodes used to maintain the application functionality. By knowing that 4-bytes were stored at every address the attacker could further reduce the set of addresses to $32 \div 2$, or 16. Therefore, in the worst case scenario the attacker had all the addresses found in the random IVT; however, the order of the addresses or the IVT entries were unknown. The attacker would have to brute force all possible combinations of the 16 addresses that were part of the random IVT but the order was unknown. The total number of combinations for 16 addresses is 16! or 20,922,789,888,000. With that being said, to brute force the worst case scenario where all the addresses are known but the order is unknown it would take 17,459 (7.5) years! Our results show that even in the worst case scenario where there are only 16 addresses to combine and check it is still not practical to brute force. Adding randomization to the MSP430-BSL password made it impossible or impractical to brute force.

$$((((2.09 \times 10^{13} \div 2280) \div 60) \div 24) \div 365) \qquad \mathit{or} \qquad 17459 \tag{7.5}$$

We would also like to take the time to point out that data stored in memory was in little endian; therefore, when brute forcing the addresses they must be converted to little endian. For example, using our worst case scenario example one of the permutations is as follows:

0xFFD4, 0xFFB0, 0xFFC4, 0xFFCC, 0xFFA8, 0xFFA0, 0xFFB4, 0xFFD8,
0xFFAC, 0xFFC0, 0xFFD0, 0xFFA4, 0xFFDC, 0xFFBC, 0xFFC8, 0xFFB8

The IVT values are converted to get the corresponding password bytes that is sent to the MSP430 to check the password. The converted IVT values that are sent to the mote looks as follows:

D4 FF B0 FF C4 FF CC FF A8 FF A0 FF B4 FF D8 FF

AC FF C0 FF D0 FF A4 FF DC FF BC FF C8 FF B8 FF

With that being said, although the password was 32-bytes, we only needed to permutate 16 addresses since the MSP430 used the IVT address space to also store the password.

Our last evaluation consisted of examining the security strength of using the two-factor authentication. Since two-factor authentication does not protect access to the actual MSP430 MCU then if the attacker is able to somehow brute force the MSP430-BSL password the two-factor is also broken. However, we have demonstrated that even in the worst case scenario it would take 17,459 years to brute our secure MSP430-BSL password. The two-factor authentication protects the password file generated by the Secure-BSL. The password file generated by Secure-BSL has a copy of the IVT values found in flash memory and is used to authenticate users for future access to the MSP430 MCU. Without the password file, future access to the MSP430 is restricted and requires mass erasing flash memory to regain control of the MCU. With that being said, if an attacker is able to obtain a copy of the password file they are guaranteed full access to the MSP430 MCU. The two factor authentication is used to encrypt the file using a user defined passphrase. The Secure-BSL requires users to enter an 8-character long password if they decide to use the security level of two features. With a copy of the password file, the attacker can perform an offline attack to brute the password used to encrypt the password file. Table 7.1 shows how long it would take to brute force the password depending on the characters used in the passphrase[79].

Table 7.1: Password cracking time for passphrase of 8 characters long.

Character Types	Time
Numerals	10 Seconds
Alphabet	348 minutes
Alphabet (Lower and Upper)	62 days
Alphabet (Lower and Upper) and Numerals	253 Days
Alphabet (Lower and Upper), Numerals and Symbols	23 Years

The calculated times are based on a dual processor PC capable of guessing 10,000,000 passwords per second[79]. Enforcing the user to use 8-character long passwords better secures the encrypted file; however, it is up to the user for the desired level of security. Ideally, the user should use a password that has combinations of alphabetical letters, numerals and symbols, which would take roughly 23 years to brute on a dual processor PC. The strength passphrase is out of our hands because only the user will be able to dictate the level of security for the password file. Keep in mind, that if a user forgets the password it is equivalent to losing the password file. The user has to mass erase the MSP430 in order to regain access to the MCU. Accessing the MSP430 MCU after flashing could become a hassle when using two-factor; however, most of the sensor devices once deployed are not accessed until the device itself starts malfunctioning.

7.3 Chapter Summary

In summary, we evaluated the memory overhead and computational overhead of using our Secure-BSL software to protect the MSP430 MCU. Our results showed that there is only a memory overhead of 64-bytes and a computational overhead of 2 clock cycles when an interrupt occurs. Next, we investigated the security posture of the randomized MSP430-BSL password generated by our Secure-BSL software. We examined the passwords for true randomness by checking for any patterns after sorting the IVT addresses. Furthermore, we took the differences between addresses to check or any consistent patterns that can be used for future password predictions. The results proved that the MSP430-BSL password is truly random and that a unique password is guaranteed even if binaries are the same. We continued our analysis by checking for worst case scenarios where an attacker knows the range of addresses used to generate the new IVT values. The results show that even if the attacker is able to lower the set of addresses to 16 it still takes 17,459 years to try all possible permutations of 16. Lastly, we investigated the worst case scenario where an attacker is able to steal the encrypted password file and brute force the passphrase to decrypt the file. We concluded that as long as the user uses a strong passphrase of 8 characters long with a combination of the alphabet (Lower and Upper), numerals and symbols it would take 23 years in a dual processor PC. Chapter 7 has demonstrated that using our Secure-BSL software truly protects future access to the MSP430 MCU.

Chapter 8

Conclusion and Future Work

In this thesis we have evaluated and improved the security of WSNs by contributing a secure approach to restricting physical access to the MSP430 micro-controllers units. In Chapter 2, we discussed background information on various research topics for WSNs. We discussed various types of sensor applications and provided examples of related WSNs that remotely monitor the temperature of indoor environments. We evaluated various proposed cipher protocols for WSNs and concluded that hardware implemented ciphers are the best choice for WSNs. In addition, we mentioned various types of key management protocols and concluded that ECC is the best choice for implementing PKC in WSNs. We highlighted on different algorithms proposed to mitigate the leaking of secret information when nodes are captured. Lastly, we discussed related research of node tampering attacks and shown the possibilities of obtaining information stored the internals of MSP430.

In Chapter 3, we discussed the design, implementation and testing results of our STMS. First, we discussed our design approach of using a star topology, where the central node is a Coordinator and the end nodes are the End-Device's collection of temperature levels. Our implementation allows to encrypt the communication between the Coordinator and the End-Devices using a network wide shared key. Furthermore, we use the AES inline encryption built in to the CC2420 RF chips of our TelosB motes. Using the PPPSniffer application and Wireshark, we were able to confirm that the communication within our STMS was encrypted. We used STMS as our WSN experimental environment to evaluate the security of the network.

In Chapter 4, we discussed our approach of breaking the MSP430-BSL password in a matter of days. We highlighted the two methods of programming the MSP430 and proposed that BSL programming is the most effective way. We analyzed the 32-bytes (256-bit) IVT or the "BSL password" used to protect access to the MSP430. We evaluated password samples generated from 93 TinyOS applications to check for password patterns between applications. Our results showed that BSL password does indeed have patterns that can be used to predict future BSL passwords. The brute force times depends on the number of unused interrupts found in a sensor application we analyzed. If the number of unused interrupts are 11, then the brute force time would be roughly

2 days. If the the number of unused interrupts is 7 then the brute force time would be roughly 80 days. Using our password pattern technique we have been able to reduce the brute force time from years to a matter of days.

In Chapter 5, we demonstrated how attackers can reverse engineer MSP430 applications found in flash memory. We provided a step-by-step example of how an attacker can find cryptographic keys stored in RAM. First, we presented the memory layout of the MSP430 and highlighted the important regions in memory. We the provided instructions on how to disassemble dumped MSP430 applications by using the msp430-objcopy and the msp430-objdump tools. We provided the reader a quick reference guide on the instruction set supported by the MSP430. Lastly, we provided techniques that can be used effectively to reverse engineer MSP430 applications with the goal of reducing the time to find cryptographic keys.

In Chapter 6, we contributed our solution of improving the password used to protect BSL access to the MSP430 chips. We discussed our design of randomizing the IVT values by replacing them with unused addresses. The randomizing of the IVT values and still maintaining functionality of the MSP430 application was achieved by using branch instructions to save the original ISRs. We implemented our approach on top default bootstrap loader software (tos-bsl) and made our Secure-BSL implementation open source [75]. In addition to increasing the brute force times for the MSP430-BSL password, our Secure-BSL software supported two factor authentication by using a user defined passphrase. Our test results show that the brute force times for the MSP430-BSL password is impractical.

In Chapter 7, we evaluated our Secure-BSL implementation by checking the overhead added to the application as well as the level of security of the random generated passwords. Our results showed that there was only a memory overhead of 64-bytes and a computational overhead of 2 clock cycles when an interrupt occurred. The results show that in the worst case scenario, if the attacker is able to lower the set of addresses to brute force to 16, it still takes 17,459 years to try all possible permutations of 16. We also investigated the worst case scenario where an attacker is able to steal the encrypted password file and brute force to decrypt the file. We concluded that as long as the user uses a strong passphrase of 8 characters long with a combination of the alphabet (lower and upper case), numerals and symbols it would take 23 years in a dual processor PC.

We have contributed to the WSN community with additional security investigation results. First, this thesis provides an implemented secure temperature monitoring system that collects temperature levels in a environment. Second, this thesis demonstrates a brute force attack on a capture node to be practical and the password that protects access to the MSP320 MCU can be broken in a matter of days. Third, to our knowledge, we have provided the first example of reverse engineering MSP430 applications to obtain copies of cryptographic keys. Fourth, we have contributed the Secure-BSL software that improved the password used to protect access to the MSP430 MCU. Lastly, our evaluation results show the significant improvements of protecting MSP430 MCU from node capture attacks.

Future work directions focus on evaluating other popular micro-controller units to test their level of tamper resistance. In particular, we are interested in evaluating the ARM, AVR and PIC micro-controllers [2]. We want to investigate whether any of the various micro-controllers have the same BSL password vulnerable as well as investigate any other tampering techniques that can be used to access the MCU internals. Lastly, we would like to increase our password sample size to thousands or even million samples in order to find more password patterns

Appendix A

Script Results For tos-bsl

Table A.1: Time Results of Checking The Correct MSP430-BSL Password

	1	**200**	**400**	**600**	**800**	**1000**
Trial 1	0.06s	12.58s	25.15s	37.80s	50.44s	63.04s
Trial 2	0.06s	12.64s	25.30s	38.03s	51.67s	64.45s
Trial 3	0.06s	12.91s	25.70s	38.67s	51.58s	64.56s
Trial 4	0.06s	12.87s	25.78s	38.62s	51.54s	64.43s
Trial 5	0.06s	12.94s	25.81s	38.67s	51.53s	64.28s
Trial 6	0.06s	12.90s	25.77s	38.65s	51.49s	64.48s
Trial 7	0.07s	12.88s	25.77s	38.71s	51.63s	64.32s
Trial 8	0.07s	12.92s	25.78s	38.64s	51.35s	64.40s
Trial 9	0.06s	12.84s	25.80s	38.53s	51.60s	64.17s
Trial 10	0.06s	12.92s	25.79s	38.70s	51.65s	64.39s
Averages	0.06s	12.84s	25.67s	38.50s	51.45s	64.25s

Table A.2: Time Results of Checking The Incorrect MSP430-BSL Password

	1	**200**	**400**	**600**	**800**	**1000**
Trial 1	0.06s	12.58s	25.21s	37.74s	51.78s	63.47s
Trial 2	0.06s	12.86s	25.78s	38.73s	51.55s	64.57s
Trial 3	0.06s	12.95s	25.71s	38.60s	51.55s	64.50s
Trial 4	0.06s	12.86s	25.86s	38.59s	51.66s	64.45s
Trial 5	0.06s	12.86s	25.71s	38.66s	51.59s	64.50s
Trial 6	0.07s	12.92s	25.79s	38.74s	51.65s	64.46s
Trial 7	0.09s	12.85s	25.79s	38.54s	51.55s	64.34s
Trial 8	0.06s	12.85s	25.72s	38.60s	51.46s	64.47s
Trial 9	0.06s	12.79s	25.72s	38.61s	51.47s	64.27s
Trial 10	0.06s	12.91s	25.68s	38.60s	51.43s	64.41s
Averages	0.07s	12.84s	25.70s	38.54s	51.57s	64.34s

Table A.3: Time Results of Checking The MSP430-BSL Passwords Using a Baud Rate of 38400

	1	**200**	**400**	**600**	**800**	**1000**
Trial 1	0.02s	5.18s	10.29s	15.57s	20.70s	25.77s
Trial 2	0.03s	5.18s	10.30s	15.41s	20.56s	25.79s
Trial 3	0.03s	5.15s	10.35s	15.45s	20.59s	25.81s
Trial 4	0.03s	5.10s	10.35s	15.43s	20.77s	25.93s
Trial 5	0.03s	5.11s	10.35s	15.48s	20.67s	25.84s
Trial 6	0.03s	5.19s	10.37s	15.55s	20.74s	25.67s
Trial 7	0.03s	5.19s	10.29s	15.53s	20.72s	25.76s
Trial 8	0.03s	5.21s	10.36s	15.46s	20.81s	25.84s
Trial 9	0.03s	5.22s	10.29s	15.51s	20.65s	25.80s
Trial 10	0.02s	5.17s	10.27s	15.51s	20.59s	25.83s
Averages	0.03s	5.17s	10.32s	15.49s	20.68s	25.80s

APPS	**Size (b)**	**IVT Entries**	**Duplicates**
tinyos-main-apps-UDPEcho.ihex	41830	12	5
tinyos-main-apps-TCPEcho.ihex	35304	10	7
beacon-enabled-TestGTS-coordinator.ihex	30830	10	7
beacon-enabled-TestMultihop-router.ihex	29668	10	7
tests-deluge-Basestation.ihex	29070	13	4
tests-deluge-GoldenImage.ihex	28480	13	4
tests-deluge-SerialBlink.ihex	27356	13	4
apps-tests-TestNetwork.ihex	27216	13	4
tutorials-LowPowerSensing-Sampler.ihex	27034	10	7
apps-tests-TestNetworkLpl.ihex	27008	13	4
STMS-Coordinator-src.ihex	26702	12	5
STMS-Coordinator-PWD-Test-src.ihex	26460	12	5
tinyos-main-apps-MultihopOscilloscope.ihex	26324	13	4
TestMultihop-pancoord.ihex	25952	10	7
TestGTS-device.ihex	24714	10	7
TestAssociate-coordinator.ihex	24348	10	7
TestMultihop-device.ihex	24178	10	7
TestAssociate-device.ihex	24042	10	7
tests-deluge-Blink.ihex	24030	11	6
TestData-coordinator.ihex	23748	10	7
TestData-device.ihex	23520	10	7
tinyos-main-apps-MultihopOscilloscopeLqi.ihex	23426	13	4

STMS-EndDevice-src.ihex	23204	10	7
STMS-EndDevice-PWD-Test-src.ihex	23010	10	7
beacon-enabled-TestIndirect-coordinator.ihex	22926	10	7
tkn154-nonbeacon-enabled-TestPromiscuous.ihex	22720	12	5
beacon-enabled-TestIndirect-device.ihex	22716	10	7
beacon-enabled-TestStartSync-coordinator.ihex	22156	10	7
apps-tests-TestTymo.ihex	20670	10	7
nonbeacon-enabled-TestAssociate-device.ihex	20436	10	7
nonbeacon-enabled-TestIndirectData-coordinator.ihex	20348	10	7
nonbeacon-enabled-TestAssociate-coordinator.ihex	20244	10	7
beacon-enabled-TestStartSync-device.ihex	19992	10	7
apps-tests-LinkBench.ihex	19938	10	7
nonbeacon-enabled-TestIndirectData-device.ihex	19802	10	7
nonbeacon-enabled-TestActiveScan-device.ihex	19734	10	7
nonbeacon-enabled-TestActiveScan-coordinator.ihex	19524	10	7
tests-tkn154-packetsniffer.ihex	19418	12	5
tests-TestFtsp-FtspLpl.ihex	19106	10	7
apps-MOE-MOE-BaseStation.ihex	16796	12	5
apps-MOE-MOE-Attack_replay.ihex	16748	12	5
cc2420-TestSecurity-BaseStation.ihex	16716	12	5
tutorials-LowPowerSensing-Base.ihex	16238	12	5
tinyos-main-apps-Oscilloscope.ihex	15830	11	6
apps-tutorials-PacketParrot.ihex	15024	10	7
cc2420-TestSecurity-RadioCountToLeds1.ihex	14546	10	7
tinyos-main-apps-BaseStation.ihex	14462	12	5
tutorials-RssiDemo-RssiBase.ihex	14382	12	5
tutorials-RssiDemo-InterceptBase.ihex	14350	12	5
apps-MOE-MOE-BroadCast.ihex	14312	10	7
apps-MOE-MOE-Attack_inject.ihex	13842	10	7
tinyos-main-apps-PPPSniffer.ihex	13768	10	7
tests-TestFtsp-FtspLplBeaconer.ihex	12796	10	7
tests-cc2420-LplBroadcastPeriodicDelivery.ihex	12678	10	7

tests-cc2420-LplUnicastPeriodicDelivery.ihex	12606	10	7
apps-tests-TestLpl.ihex	12496	10	7
apps-tests-TestSrp.ihex	12248	10	7
apps-tests-RadioStress.ihex	11410	10	7
apps-tests-TestPowerManager.ihex	11400	6	11
tests-cc2420-TestAcks.ihex	11234	10	7
apps-tests-TestTimerSync.ihex	11160	10	7
tutorials-RssiDemo-SendingMote.ihex	10986	10	7
tests-storage-Config.ihex	10916	10	7
apps-tests-TestAM.ihex	10888	10	7
tests-storage-Log.ihex	10848	10	7
tests-arbiters-TestRoundRobinArbiter.ihex	10748	6	11
tests-storage-CircularLog.ihex	10648	10	7
tests-arbiters-TestFcfsArbiter.ihex	10558	6	11
tests-storage-SyncLog.ihex	10192	10	7
apps-tutorials-BlinkConfig.ihex	10142	10	7
tests-msp430-Adc12.ihex	9318	8	9
tests-storage-Block.ihex	9318	10	7
apps-tests-TestEui.ihex	9046	8	9
apps-tutorials-Printf.ihex	8602	8	9
apps-tests-TestPrintf.ihex	8584	8	9
apps-tests-TestAdc.ihex	7642	7	10
tinyos-main-apps-Sense.ihex	6988	7	10
apps-tests-TestScheduler.ihex	6314	6	11
tests-TestLed-LedColor.ihex	5770	8	9
tests-TestLed-MultiLed.ihex	5710	8	9
apps-tests-TestLocalTime.ihex	5416	8	9
tests-msp430-AdcSimple.ihex	4786	7	10
apps-tests-TestSerialPrintf.ihex	4214	8	9
apps-tutorials-SharedResourceDemo.ihex	3282	6	11
tests-telosb-TestUserButton.ihex	3020	8	9
tests-TestLed-MultiLedSingle.ihex	2622	6	11

tests-TestLed-BlinkLed.ihex	2582	6	11
apps-tutorials-BlinkFail.ihex	2558	6	11
tinyos-main-apps-Blink.ihex	2538	6	11
apps-tutorials-BlinkTask.ihex	2458	6	11
apps-tests-TestPowerup.ihex	1378	6	11
tinyos-main-apps-Powerup.ihex	1378	6	11
tinyos-main-apps-Null.ihex	1328	6	11

Table A.4: Correlation Between Code Size vs. Number of Duplicate IVT Entries

Appendix B

Script Results For Secure-BSL

APPS	App Size	P0	P10	P11	P14	P15
UDPEcho.ihex	41830	0xfe0e	0xf15a	0xf792	0xefc6	0xf952
TCPEcho.ihex	35304	0xea3c	0xd764	0xfa10	0xd69c	0xd4bc
TestGTS-coordinator.ihex	30830	0xfb98	0xe558	0xee3c	0xc994	0xe4e4
TestMultihop-router.ihex	29668	0xedce	0xf316	0xd282	0xe626	0xc2b6
deluge-Basestation.ihex	29070	0xf504	0xf048	0xd720	0xfcd4	0xde08
deluge-GoldenImage.ihex	28480	0xe550	0xcd0c	0xc028	0xd9ec	0xcb28
deluge-SerialBlink.ihex	27356	0xfee2	0xf202	0xddca	0xf152	0xbe9a
TestNetwork.ihex	27216	0xae8a	0xf2d2	0xd94a	0xb846	0xde7a
LowPowerSensing-Sampler.ihex	27034	0xdc9a	0xc85e	0xb266	0xd296	0xf6da
TestNetworkLpl.ihex	27008	0xff74	0xb288	0xb1ec	0xcb94	0xaf48
IEEE802154-Coordinator.ihex	26438	0xa93c	0xd888	0xd79c	0xdc20	0xdfc4
MultihopOscilloscope.ihex	26324	0xf70a	0xf0aa	0xe15e	0xbf7e	0xf376
TestMultihop-pancoord.ihex	25952	0xf4c4	0xcde4	0xef50	0xb168	0xd750
TestGTS-device.ihex	24714	0xae26	0xd17e	0xf352	0xff6e	0xe0fe
TestAssociate-coordinator.ihex	24348	0xeae4	0xff1c	0xbbd0	0xa5a0	0xd8ec
TestMultihop-device.ihex	24178	0xeda8	0xbfbc	0xbbb0	0xf8e4	0xc6f4
TestAssociate-device.ihex	24042	0xcad2	0xe1ea	0xfb6a	0xfd96	0xc3de
deluge-Blink.ihex	24030	0xca96	0xe5ca	0xe492	0xb192	0xe46e
TestData-coordinator.ihex	23748	0xfdc4	0xc770	0xe0e8	0xf3a4	0xd96c
TestData-device.ihex	23520	0xf0e2	0xd14a	0xda96	0xa92e	0xbe16
MultihopOscilloscopeLqi.ihex	23426	0x9fda	0xeb62	0xc75e	0xa6f6	0xddf2
IEEE802154-End-Device-src.ihex	23010	0xe8b6	0xcde2	0xe54e	0xd3ba	0xc09a
TestIndirect-coordinator.ihex	22926	0xf644	0xb30c	0xccec	0xfea8	0xa0f8
tkn154-nonTestPromiscuous.ihex	22720	0xbc80	0xa770	0xa308	0xe030	0x9d04
TestIndirect-device.ihex	22716	0xaa5e	0xc262	0xa55a	0xae42	0xd692

TestStartSync-coordinator.ihex	22156	0xc114	0xd5a0	0x9c8c	0xe0bc	0xd068
TestTymo.ihex	20670	0xd2c6	0xc8de	0xf4f2	0x9b7a	0xf506
nonTestAssociate-device.ihex	20436	0xf382	0xc4da	0xda2e	0xe8a2	0xeaf2
nonTestIndirectData-coord.ihex	20348	0x940a	0xb916	0xe202	0xee36	0xa0ce
nonTestAssociate-coord.ihex	20244	0xaf18	0xe04c	0x9d78	0xc8d4	0xaab8
TestStartSync-device.ihex	19992	0xfa38	0x91b0	0xac68	0xb8c0	0x96bc
LinkBench.ihex	19938	0xc528	0xc4c0	0xbb24	0xd908	0xa384
nonTestIndirectDatadevice.ihex	19802	0xb5f4	0x8e28	0xfe38	0xbdc8	0xaba8
nonTestActiveScan-device.ihex	19734	0xb28a	0xa3e6	0xadfa	0xd226	0xfcfe
nonTestActiveScan-coord.ihex	19524	0x7cfc	0xfae4	0xcee8	0xac18	0xad2c
nonTestActiveScan-coord.ihex	19524	0xadca	0xc38a	0xeb5e	0xf93a	0xc9f6
tkn154-packetsniffer.ihex	19418	0xfa66	0xd12e	0x93de	0xb1d2	0xa732
TestFtsp-FtspLpl.ihex	19106	0xcce2	0x9cc6	0xde4e	0xa71e	0xf66e
L1-Secure-Coordinator-src.ihex	18958	0xf70e	0xb82e	0xa3c6	0xaece	0xf86e
MOE-MOE-BaseStation.ihex	16796	0x8cfa	0xb06a	0xa77a	0x8e12	0xc4a6
MOE-MOE-Attack_replay.ihex	16748	0x893e	0xee52	0x907a	0x8c0a	0xeeaa
TestSecurity-BaseStation.ihex	16716	0x9a26	0x8ae2	0x8a36	0xc4b2	0xae46
LowPowerSensing-Base.ihex	16238	0xdfa4	0xa19c	0xa298	0x9648	0x98c8
Oscilloscope.ihex	15830	0xa964	0xbf38	0xa24c	0xa61c	0xae90
L1-Secure-End-Device-src.ihex	15492	0x8f5a	0xead2	0xe1e2	0xad5e	0xa466
PacketParrot.ihex	15024	0xae52	0x9c3e	0xe5e2	0xdaf6	0xdc5e
RadioCountToLeds1.ihex	14546	0xf8aa	0xa992	0x9fea	0xfaf6	0x94c6
BaseStation.ihex	14462	0xa572	0xbeae	0xc7f6	0x80f2	0xacc2
RssiDemo-RssiBase.ihex	14382	0x926e	0xa546	0x9f2e	0x7ebe	0xf98a
RssiDemo-InterceptBase.ihex	14350	0xd666	0xc236	0xb986	0x8a16	0x8336
MOE-MOE-BroadCast.ihex	14312	0xa842	0x7e36	0xb9f6	0xf12e	0x8af2
PPPSniffer.ihex	13768	0xcd4e	0x91fe	0xbaa6	0xa1b2	0xa0f6
FtspLplBeaconer.ihex	12796	0x9a48	0xae5c	0xb5bc	0xa240	0x8c30
BroadcastPeriodDelivery.ihex	12678	0xb060	0x9914	0xfc6c	0xb2c4	0x8394
UnicastPeriodicDelivery.ihex	12606	0x8880	0x80a4	0x881c	0x793c	0x8364
TestLpl.ihex	12496	0xf87c	0xd314	0xaae0	0x8a20	0xdcbc
TestSrp.ihex	12248	0xe374	0xb7d8	0xcba4	0xc29c	0xcc6c

RadioStress.ihex	11410	0xeb0e	0x7bda	0x6d9e	0xc88a	0xcaea
TestPowerManager.ihex	11400	0x97f6	0xee5a	0x6f76	0xc736	0xb4c2
cc2420-TestAcks.ihex	11234	0xd04a	0xaa2e	0xdf2e	0xbfda	0x9002
TestTimerSync.ihex	11160	0xc45a	0xc432	0xa592	0x913a	0xbc1a
RssiDemo-SendingMote.ihex	10986	0xc86c	0xaa84	0xb4b4	0x71e4	0xe064
storage-Config.ihex	10916	0xa694	0xcfe4	0xe2a8	0x7e04	0xf508
TestAM.ihex	10888	0xe026	0x9526	0xc07e	0xa50e	0xd3ce
storage-Log.ihex	10848	0x8c50	0xc744	0x9efc	0x9c08	0xc7bc
TestRoundRobinArbiter.ihex	10748	0xdb7c	0xeef4	0xdab8	0x7bd4	0x6c20
storage-CircularLog.ihex	10648	0x87c0	0xf3f8	0xf8b0	0xf408	0xc4e8
arbiters-TestFcfsArbiter.ihex	10558	0xe87a	0x8e1e	0xfe2e	0xb79a	0xc3ee
storage-SyncLog.ihex	10192	0x96de	0xa656	0xd196	0xb96a	0xc2be
BlinkConfig.ihex	10142	0xb2d6	0xd422	0xb0e2	0xa556	0xaece
msp430-Adc12.ihex	9318	0xa0bc	0x6fdc	0xffdc	0x88a8	0xaae8
storage-Block.ihex	9318	0xfdcc	0xa520	0x9fac	0x9ef8	0xf8d0
TestEui.ihex	9046	0xfcda	0x8ca6	0xc866	0xa4d2	0xe482
Printf.ihex	8602	0x6f34	0x7e3c	0x7bdc	0xf370	0xbd2c
TestPrintf.ihex	8584	0xd9fa	0xfa66	0xd606	0x767a	0xd572
TestAdc.ihex	7642	0x9658	0xd650	0xa9ac	0xcfa0	0xfeac
Sense.ihex	6988	0xd93c	0x6560	0xba54	0x99b0	0x9ba0
TestScheduler.ihex	6314	0x7b82	0xb57a	0x76d6	0xdfba	0xed86
TestLed-LedColor.ihex	5770	0x82c0	0x9e90	0xa824	0xc900	0xacd4
TestLed-MultiLed.ihex	5710	0x638a	0x8c66	0xd402	0xe90a	0xcb22
TestLocalTime.ihex	5416	0xfe46	0xcb7a	0xbfa6	0x9932	0x77e6
msp430-AdcSimple.ihex	4786	0x6e34	0xaab8	0xf628	0x931c	0xbda4
TestSerialPrintf.ihex	4214	0xad34	0xa844	0xaff0	0xc29c	0xc448
SharedResourceDemo.ihex	3282	0x4eae	0x7dce	0xfc8e	0xd0aa	0x8b0a
telosb-TestUserButton.ihex	3020	0xd0dc	0xeeec	0xd900	0xefb8	0xa6a0
TestLed-MultiLedSingle.ihex	2622	0xf42e	0x7262	0xad62	0xb3fe	0x76ae
TestLed-BlinkLed.ihex	2582	0xc52e	0xf9fe	0x54e6	0x9d2a	0xe736
BlinkFail.ihex	2558	0x98b2	0xe59e	0xa806	0x949e	0x6e5e
Blink.ihex	2538	0x9a8a	0x70be	0x6672	0xa396	0x8446

BlinkTask.ihex	2458	0xd2da	0xc6ca	0xbf62	0x95d6	0x73e6
TestPowerup.ihex	1378	0x8964	0xd178	0x91cc	0x9930	0xb718
Powerup.ihex	1378	0xacec	0xbd64	0xda80	0xa700	0xcccc
Null.ihex	1328	0x8fc6	0xb3fa	0x8f1e	0xf9ca	0x6f16

Table B.1: Sample of the values generated for IVT priority levels 0, 10, 11,14, and 15

APPS	**Size (b)**	**IVT Entries**	**Duplicates**
UDPEcho.ihex	41830	16	1
TCPEcho.ihex	35304	16	1
TestGTS-coordinator.ihex	30830	16	1
TestMultihop-router.ihex	29668	16	1
deluge-Basestation.ihex	29070	16	1
deluge-GoldenImage.ihex	28480	16	1
deluge-SerialBlink.ihex	27356	16	1
TestNetwork.ihex	27216	16	1
LowPowerSensing-Sampler.ihex	27034	16	1
TestNetworkLpl.ihex	27008	16	1
IEEE802154-Coordinator.ihex	26438	16	1
MultihopOscilloscope.ihex	26324	16	1
TestMultihop-pancoord.ihex	25952	16	1
TestGTS-device.ihex	24714	16	1
TestAssociate-coordinator.ihex	24348	16	1
TestMultihop-device.ihex	24178	16	1
TestAssociate-device.ihex	24042	16	1
deluge-Blink.ihex	24030	16	1
TestData-coordinator.ihex	23748	16	1
TestData-device.ihex	23520	16	1
MultihopOscilloscopeLqi.ihex	23426	16	1
IEEE802154-End-Device-src.ihex	23010	16	1
TestIndirect-coordinator.ihex	22926	16	1
tkn154-nonTestPromiscuous.ihex	22720	16	1
TestIndirect-device.ihex	22716	16	1

TestStartSync-coordinator.ihex	22156	16	1
TestTymo.ihex	20670	16	1
nonTestAssociate-device.ihex	20436	16	1
nonTestIndirectData-coord.ihex	20348	16	1
nonTestAssociate-coord.ihex	20244	16	1
TestStartSync-device.ihex	19992	16	1
LinkBench.ihex	19938	16	1
nonTestIndirectDatadevice.ihex	19802	16	1
nonTestActiveScan-device.ihex	19734	16	1
nonTestActiveScan-coord.ihex	19524	16	1
nonTestActiveScan-coord.ihex	19524	16	1
tkn154-packetsniffer.ihex	19418	16	1
TestFtsp-FtspLpl.ihex	19106	16	1
L1-Secure-Coordinator-src.ihex	18958	16	1
MOE-MOE-BaseStation.ihex	16796	16	1
MOE-MOE-Attack_replay.ihex	16748	16	1
TestSecurity-BaseStation.ihex	16716	16	1
LowPowerSensing-Base.ihex	16238	16	1
Oscilloscope.ihex	15830	16	1
L1-Secure-End-Device-src.ihex	15492	16	1
PacketParrot.ihex	15024	16	1
RadioCountToLeds1.ihex	14546	16	1
BaseStation.ihex	14462	16	1
RssiDemo-RssiBase.ihex	14382	16	1
RssiDemo-InterceptBase.ihex	14350	16	1
MOE-MOE-BroadCast.ihex	14312	16	1
PPPSniffer.ihex	13768	16	1
FtspLplBeaconer.ihex	12796	16	1
BroadcastPeriodDelivery.ihex	12678	16	1
UnicastPeriodicDelivery.ihex	12606	16	1
TestLpl.ihex	12496	16	1
TestSrp.ihex	12248	16	1

RadioStress.ihex	11410	16	1
TestPowerManager.ihex	11400	16	1
cc2420-TestAcks.ihex	11234	16	1
TestTimerSync.ihex	11160	16	1
RssiDemo-SendingMote.ihex	10986	16	1
storage-Config.ihex	10916	16	1
TestAM.ihex	10888	16	1
storage-Log.ihex	10848	16	1
TestRoundRobinArbiter.ihex	10748	16	1
storage-CircularLog.ihex	10648	16	1
arbiters-TestFcfsArbiter.ihex	10558	16	1
storage-SyncLog.ihex	10192	16	1
BlinkConfig.ihex	10142	16	1
msp430-Adc12.ihex	9318	16	1
storage-Block.ihex	9318	16	1
TestEui.ihex	9046	16	1
Printf.ihex	8602	16	1
TestPrintf.ihex	8584	16	1
TestAdc.ihex	7642	16	1
Sense.ihex	6988	16	1
TestScheduler.ihex	6314	16	1
TestLed-LedColor.ihex	5770	16	1
TestLed-MultiLed.ihex	5710	16	1
TestLocalTime.ihex	5416	16	1
msp430-AdcSimple.ihex	4786	16	1
TestSerialPrintf.ihex	4214	16	1
SharedResourceDemo.ihex	3282	16	1
telosb-TestUserButton.ihex	3020	16	1
TestLed-MultiLedSingle.ihex	2622	16	1
TestLed-BlinkLed.ihex	2582	16	1
BlinkFail.ihex	2558	16	1
Blink.ihex	2538	16	1

BlinkTask.ihex	2458	16	1
TestPowerup.ihex	1378	16	1
Powerup.ihex	1378	16	1
Null.ihex	1328	16	1

Table B.2: The number of duplicates found for each application using the Secure-BSL

Pos:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
tos	15	0	10	11	14	7	3	1	8	9	2	4	6	5	13	12
App1	12	4	13	5	7	2	0	11	14	15	3	1	10	6	8	9
App2	5	12	13	1	15	7	9	14	6	0	11	4	3	8	10	2
App3	9	6	15	1	2	12	14	7	11	10	5	0	4	13	8	3
App4	0	6	7	4	13	10	12	5	15	8	14	9	3	2	1	11
App5	11	13	8	2	3	12	5	9	7	14	4	0	6	1	15	10
App6	7	12	10	15	11	14	6	4	13	3	2	9	8	1	5	0
App7	2	11	10	9	7	15	0	14	12	8	6	1	5	4	3	13
App8	1	13	3	5	15	4	11	0	8	7	9	10	6	12	14	2
App9	6	4	5	12	13	8	14	0	1	10	15	3	2	11	9	7
App10	5	11	13	8	0	9	1	15	14	4	2	3	12	10	7	6
App11	1	6	3	11	0	4	9	2	10	12	5	7	13	8	14	15
App12	13	12	3	4	10	6	7	8	5	1	14	15	9	0	2	11
App13	15	14	7	12	6	3	5	8	11	0	9	13	4	10	1	2

Table B.3: The results of Secure-BSL sorted passwords for the same applications that had 11 unused interrupts when using tos-bsl software.

Bibliography

[1] Y. W. Law, J. Doumen, and P. Hartel, "Survey and benchmark of block ciphers for wireless sensor networks," *ACM Transactions on Sensor Networks (TOSN)*, vol. 2, no. 1, pp. 65–93, 2006.

[2] R. Roman, C. Alcaraz, and J. Lopez, "A survey of cryptographic primitives and implementations for hardware-constrained sensor network nodes," *Mobile Networks and Applications*, vol. 12, no. 4, pp. 231–244, 2007.

[3] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer networks*, vol. 38, no. 4, pp. 393–422, 2002.

[4] P. Ganesan, R. Venugopalan, P. Peddabachagari, A. Dean, F. Mueller, and M. Sichitiu, "Analyzing and modeling encryption overhead for sensor network nodes," in *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications.* ACM, 2003, pp. 151–159.

[5] K. O'Flaherty, "Securing the internet of things," *SC Magazine UK*, 2015.

[6] S. McGillicuddy, "Who's in charge here? securing the internet of things," *Information Security Insider Edition*, 2014.

[7] J. Andersen and M. T. Hansen, "Energy bucket: A tool for power profiling and debugging of sensor nodes," in *Sensor Technologies and Applications, 2009. SENSORCOMM'09. Third International Conference on.* IEEE, 2009, pp. 132–138.

[8] F. Busching, A. Figur, D. Schurmann, and L. Wolf, "Utilizing hardware aes encryption for wsns," in *CONFERENCE PAPER*, 2013.

[9] T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel, "A survey of lightweight-cryptography implementations," *IEEE Design & Test of Computers*, no. 6, pp. 522–533, 2007.

[10] D. Jinwala, D. Patel, and K. Dasgupta, "Flexisec: a configurable link layer security architecture for wireless sensor networks," *arXiv preprint arXiv:1203.4697*, 2012.

[11] C. Karlof, N. Sastry, and D. Wagner, "Tinysec: a link layer security architecture for wireless sensor networks," in *Proceedings of the 2nd international conference on Embedded networked sensor systems.* ACM, 2004, pp. 162–175.

[12] B. Kiruthika, R. Ezhilarasie, and A. Umamakeswari, "Implementation of modified rc4 algorithm for wireless sensor networks on cc2431," *Indian Journal of Science and Technology*, vol. 8, no. S9, pp. 198–206, 2015.

[13] L. E. Lighfoot, J. Ren, and T. Li, "An energy efficient link-layer security protocol for wireless sensor networks," in *Electro/Information Technology, 2007 IEEE International Conference on.* IEEE, 2007, pp. 233–238.

[14] M. Luk, G. Mezzour, A. Perrig, and V. Gligor, "Minisec: a secure sensor network communication architecture," in *Proceedings of the 6th international conference on Information processing in sensor networks.* ACM, 2007, pp. 479–488.

[15] A. Perrig, J. Stankovic, and D. Wagner, "Security in wireless sensor networks," *Communications of the ACM*, vol. 47, no. 6, pp. 53–57, 2004.

[16] S. Sciancalepore, G. Piro, G. Boggia, and L. Grieco, "Application of ieee 802.15. 4 security procedures in openwsn protocol stack," *IEEE Standards Education e-Magazine*, vol. 4, no. 2, 2014.

[17] P. Toldo, M. Saloni, and N. Manica, "Aes implementation in tinyos," 2008.

[18] M. T. Hansen, "Asynchronous group key distribution on top of the cc2420 security mechanisms for sensor networks," in *Proceedings of the second ACM conference on Wireless network security.* ACM, 2009, pp. 13–20.

[19] P. Szczechowiak, L. B. Oliveira, M. Scott, M. Collier, and R. Dahab, "Nanoecc: Testing the limits of elliptic curve cryptography in sensor networks," in *Wireless sensor networks.* Springer, 2008, pp. 305–320.

[20] D. J. Malan, M. Welsh, and M. D. Smith, "Implementing public-key infrastructure for sensor networks," *ACM Transactions on Sensor Networks (TOSN)*, vol. 4, no. 4, p. 22, 2008.

[21] A. Liu and P. Ning, "Tinyecc: A configurable library for elliptic curve cryptography in wireless sensor networks," in *Information Processing in Sensor Networks, 2008. IPSN'08. International Conference on.* IEEE, 2008, pp. 245–256.

[22] H. Wang, B. Sheng, and Q. Li, "Elliptic curve cryptography-based access control in sensor networks," *International Journal of Security and Networks*, vol. 1, no. 3-4, pp. 127–137, 2006.

[23] L. Eschenauer and V. D. Gligor, "A key-management scheme for distributed sensor networks," in *Proceedings of the 9th ACM conference on Computer and communications security.* ACM, 2002, pp. 41–47.

[24] H. Chan, A. Perrig, and D. Song, "Random key predistribution schemes for sensor networks," in *Security and Privacy, 2003. Proceedings. 2003 Symposium on.* IEEE, 2003, pp. 197–213.

[25] Y. Xiao, V. K. Rayi, B. Sun, X. Du, F. Hu, and M. Galloway, "A survey of key management schemes in wireless sensor networks," *Computer communications*, vol. 30, no. 11, pp. 2314–2341, 2007.

[26] T. Instruments, "Msp430 programming with the bootloader (bsl)," *TI Application Report SLAU319K*, 2015.

[27] A. Becher, Z. Benenson, and M. Dornseif, *Tampering with motes: Real-world physical attacks on wireless sensor networks.* Springer, 2006.

[28] T. Goodspeed, "A side-channel timing attack of the msp430 bsl," *Black Hat USA*, 2008.

[29] B. Son, Y.-s. Her, and J.-G. Kim, "A design and implementation of forest-fires surveillance system based on wireless sensor networks for south korea mountains," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 6, no. 9, pp. 124–130, 2006.

[30] A. Baggio, "Wireless sensor networks in precision agriculture," in *ACM Workshop on Real-World Wireless Sensor Networks (REALWSN 2005), Stockholm, Sweden.* Citeseer, 2005.

[31] B. Warneke, M. Last, B. Liebowitz, and K. S. Pister, "Smart dust: Communicating with a cubic-millimeter computer," *Computer*, vol. 34, no. 1, pp. 44–51, 2001.

[32] G. J. Pottie and W. J. Kaiser, "Wireless integrated network sensors," *Communications of the ACM*, vol. 43, no. 5, pp. 51–58, 2000.

[33] M. L. McKelvin Jr, M. L. Williams, and N. M. Berry, "Integrated radio frequency identification and wireless sensor network architecture for automated inventory management and tracking applications," in *Proceedings of the 2005 Conference on Diversity in Computing.* ACM, 2005, pp. 44–47.

[34] E. M. Petriu, N. D. Georganas, D. C. Petriu, D. Makrakis, and V. Z. Groza, “Sensor-based information appliances,” *Instrumentation & Measurement Magazine, IEEE*, vol. 3, no. 4, pp. 31–35, 2000.

[35] C. Herring and S. Kaplan, “Component-based software systems for smart environments,” *IEEE Personal Communications*, vol. 7, no. 5, pp. 60–61, 2000.

[36] D. M. Doolin and N. Sitar, “Wireless sensors for wildfire monitoring,” in *Smart Structures and Materials.* International Society for Optics and Photonics, 2005, pp. 477–484.

[37] P. Sachan and A. Saharia, “Civil applications of wireless sensor networks,” *International Journal of Science and Research*, 2015.

[38] V. Boonsawat, J. Ekchamanonta, K. Bumrungkhet, and S. Kittipiyakul, “Xbee wireless sensor networks for temperature monitoring,” in *the second conference on application research and development (ECTI-CARD 2010), Chon Buri, Thailand*, 2010.

[39] Y.-J. Mon, C.-M. Lin, I. J. Rudas *et al.*, “Wireless sensor network (wsn) control for indoor temperature monitoring,” *Acta Polytechnica Hungarica*, vol. 9, no. 6, pp. 17–28, 2012.

[40] B. Risteska Stojkoska, A. Popovska Avramova, and P. Chatzimisios, “Application of wireless sensor networks for indoor temperature regulation,” *International Journal of Distributed Sensor Networks*, vol. 2014, 2014.

[41] ZBOSS, “Zigbee open source stack http://zboss.dsr-wireless.com/,” Trusted Software Development, Tech. Rep., 2013.

[42] I. Mansour, G. Chalhoub, and P. Lafourcade, “Key management in wireless sensor networks.” *Journal of Sensor & Actuator Networks*, vol. 4, no. 3, pp. 251 – 273, 2015.

[43] SmartRF, *Chipcon AS SmartRF CC2420 Preliminary Datasheet (rev 1.2)*, Chipcon, 06 2004.

[44] G. Jolly, M. C. Kuşçu, P. Kokate, and M. Younis, “A low-energy key management protocol for wireless sensor networks,” in *Computers and Communication, 2003.(ISCC 2003). Proceedings. Eighth IEEE International Symposium on.* IEEE, 2003, pp. 335–340.

[45] R. Watro, D. Kong, S.-f. Cuti, C. Gardiner, C. Lynn, and P. Kruus, “Tinypk: securing sensor networks with public key technology,” in *Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks.* ACM, 2004, pp. 59–64.

[46] D. K. Nilsson, T. Roosta, U. Lindqvist, and A. Valdes, “Key management and secure software updates in wireless process control environments,” in *Proceedings of the first ACM conference on Wireless network security.* ACM, 2008, pp. 100–108.

[47] X. Du, Y. Xiao, S. Ci, M. Guizani, and H.-H. Chen, “A routing-driven key management scheme for heterogeneous sensor networks,” in *Communications, 2007. ICC’07. IEEE International Conference on.* IEEE, 2007, pp. 3407–3412.

[48] O. Alfandi, A. Bochem, A. Kellner, C. Göge, and D. Hogrefe, “Secure and authenticated data communication in wireless sensor networks,” *Sensors*, vol. 15, no. 8, pp. 19 560–19 582, 2015.

[49] J. Zhang and V. Varadharajan, “Wireless sensor network key management survey and taxonomy,” *Journal of Network and Computer Applications*, vol. 33, no. 2, pp. 63–75, 2010.

[50] Y. Wang, B. Ramamurthy, and X. Zou, “Keyrev: An efficient key revocation scheme for wireless sensor networks,” in *Communications, 2007. ICC’07. IEEE International Conference on.* IEEE, 2007, pp. 1260–1265.

[51] S. Chattopadhyay and A. K. Turuk, “A scheme for key revocation in wireless sensor networks,” *International Journal on Advanced Computer Engineering and Communication Technology*, vol. 1, no. 2, pp. 16–20, 2012.

[52] I. Krontiris, T. Dimitriou, T. Giannetsos, and M. Mpasoukos, “Intrusion detection of sinkhole attacks in wireless sensor networks,” in *Algorithmic Aspects of Wireless Sensor Networks.* Springer, 2007, pp. 150–161.

[53] B. Sun, L. Osborne, Y. Xiao, and S. Guizani, “Intrusion detection techniques in mobile ad hoc and wireless sensor networks,” *Wireless Communications, IEEE*, vol. 14, no. 5, pp. 56–63, 2007.

[54] C. K. Priya, B. Sathyanarayana, J. Mizuno, T. Kakizaki, S. Takahashi, and S. Kudo, “Energy efficient and dynamic key management scheme for wireless sensor networks,” *IEEE Xplore*, 2007.

[55] H. Soroush, M. Salajegheh, and T. Dimitriou, “Providing transparent security services to sensor networks,” in *Communications, 2007. ICC’07. IEEE International Conference on.* IEEE, 2007, pp. 3431–3436.

[56] J. Deng, C. Hartung, R. Han, and S. Mishra, “A practical study of transitory master key establishment forwireless sensor networks,” in *Security and Privacy for Emerging Areas in Com-*

munications Networks, 2005. SecureComm 2005. First International Conference on. IEEE, 2005, pp. 289–302.

[57] S. P. Skorobogatov, “Semi-invasive attacks: a new approach to hardware security analysis,” Ph.D. dissertation, Citeseer, 2005.

[58] R. Anderson and M. Kuhn, “Low cost attacks on tamper resistant devices,” in *Security Protocols.* Springer, 1997, pp. 125–136.

[59] T. Goodspeed, “Practical attacks against the msp430 bsl,” in *Twenty-Fifth Chaos Communications Congress. Berlin, Germany*, 2008.

[60] *MSP430x1xx Familiy*, Texas Instrument, 2006.

[61] S. P. Skorobogatov and R. J. Anderson, “Optical fault induction attacks,” in *Cryptographic Hardware and Embedded Systems-CHES 2002.* Springer, 2002, pp. 2–12.

[62] *Datasheet SHT1x (SHT10, SHT11, SHT15) Humidity and Temperature Sensor IC*, Sensirion The Sensor Company, December 2011.

[63] MEMSIC, “Powerful sensing solutions,” http://www.memsic.com/, April 2016.

[64] T. Watteyne, A. Mehta, and K. Pister, “Reliability through frequency diversity: why channel hopping makes sense,” in *Proceedings of the 6th ACM symposium on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks.* ACM, 2009, pp. 116–123.

[65] moteiv, *tmote sky - Ultra low power IEEE 802.15.4 compliant wireless sensor module*, Moteiv Corporation, 02 2006.

[66] *MSP430F15x, MSP430F16x, MSP430F161x MIXED SIGNAL MICROCONTROLLER*, Texas Instruments, POST OFFICE BOX 655303 DALLAS, TEXAS 75265, March 2011.

[67] M. O. Farooq and T. Kunz, “Operating systems for wireless sensor networks: A survey,” *Sensors*, vol. 11, no. 6, pp. 5900–5930, 2011.

[68] P. Levis, “https://github.com/tinyos/tinyos-main,” *Standford University*, 2016.

[69] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler, “The nesc language: A holistic approach to networked embedded systems,” in *Acm Sigplan Notices*, vol. 38, no. 5. ACM, 2003, pp. 1–11.

[70] B. Sigg, “Yeti 2-tinyos 2. x eclipse plugin,” Ph.D. dissertation, ETH, Eidgenössische Technische Hochschule Zürich, Department of Computer Science, Distribution Computing Group, 2008.

[71] T. Maas, “A basic nesc editor plugin for tinyos-2.x using eclipse,” https://github.com/tyll/tinyos-2.x-contrib/tree/master/nescdt, 2008.

[72] M. Zukowsky’s, “Tinydt - tinyos plugin for the eclipse platform,” http://tinydt.sourceforge.net/, 2008.

[73] J. Chen, “Make of things,” Mokoversity - Innovation of Things, Tech. Rep., 2015.

[74] *IEEE Standard for Local and metropolitan area network s Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)*, IEEE Standard Association Std., April 2011.

[75] M. Tellez, “Thesis githup project: https://github.com/mtellez/wsn-thesis,” *GIthub*, 2016.

[76] *Flash Memory Controller*, Texas Instrument, POST OFFICE BOX 655303 DALLAS, TEXAS 75265, May 2015.

[77] M. Jenihhin, *MSP430 Teaching.* Tallinn University of Technology, 2009.

[78] J. Paek, “Porting your tinyos-1.x code to tinyos-2.x code,” *University of Southern California*, 2010.

[79] I. Lucas, “Password recovery speeds,” *The Home Computer Security Centre*, 2009.

[80] T. Goodspeed, “msp430static - http://msp430static.sourceforge.net/,” *Source Forge*, 2008.

[81] ——, “Reversing and exploiting wireless sensors,” *Arlington, VA, February*, 2009.

[82] G. Litovsky, “Beginning microcontrollers with the msp430 tutorial,” *Texas Instruments*, 2010.