Secure Design Considerations for Embedded Systems

by

Hunter Thorington

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfilment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
December 12, 2015

Keywords: embedded system, encryption, security, attack, microcontroller

Abstract

A guide is presented for embedded system designers that details secure design considerations for modern embedded systems. The guide focuses on adapting traditional software design practices to include adapted methodologies for defining, evaluating and producing effective solutions to security problems. Adapted threat modeling and analysis techniques are provided as a framework upon which to further critique, analyze and implement other security techniques as solutions to security problems. Simple implementations and analysis are provided for three encryption algorithms as examples. The encryption algorithms are demonstrated as only one of many solutions available to designers for secure development.

Acknowledgements

To my wife and family:

I want to thank each of you for the abundant love and encouragement.

I could not have completed my degree without you seeing me through.

Thank you!

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AES	Advanced Encryption Standard
BSI	Federal Office for Information Security, Germany
CAN	Controller Area Network
CPU	Central Processing Unit
DOS	Denial of Service
GPIO	General Purpose Input and Output
I/O	Input and Output
I2C	Inter-Integrated Circuit
IOT	Internet of Things
IP	Intellectual Property
JTAG	Joint Test Action Group Interface
PLC	Programmable Logic Controller
RSA	Ron Rivest, Adi Shamirand, Leonard Adleman
SDL	Secure Development Lifecycle
SPI	Serial Peripheral Interface
TEA	Tiny Encryption Algorithm
UART	Universal Asynchronous Receiver/Transmitter

Chapter 1

Introduction

Embedded systems are an integral part of our society today. Every aspect of daily life is somehow dependent on an embedded system. Utilities from power and water to gas and internet all rely heavily on a vast network of sensors and specialized embedded systems to keep our world running. No matter how simple or complex these systems may be, they are links in an interconnected web holding up our way of life. A problem with any single link can wreak havoc. No matter what the system, be it a large server running thousands of calculations per minute promoting stock trades, to a simple servo circuit designed to shut a valve or close a relay, unexpected behavior can cause massive problems and be used by an attacker to induce harm. In today's world not everyone follows a moral code of "do unto others as you would have them do unto you." The attackers, often called hackers, are people who work each day not for the greater good, but seek to induce harm on others, our way of life and the infrastructure. For this reason, the security of embedded systems is vitally important to protect our daily lives. Dependence on these systems, has grown beyond simple convenience, and has reached a level of required sustainability. To put it another way, failure of these systems would negatively affect daily operation of our infrastructure and lives. The security of embedded systems is often lumped in with computer glitches, internet viruses and email scams. While at a high level, many of the same security schemes and models apply, at a low level this way of thinking leaves many areas of the embedded system vulnerable and their security needs misunderstood. As the widespread use of embedded systems grows, so does their connection to the Internet of Things (IoT). The Internet of Things is the network of software and hardware devices collecting and sharing data with each other by means of a network infrastructure. The classification of embedded systems as IoT devices often leads to traditional software security practices being applied as solutions for embedded systems. This practice ignores many of the differences in application and function of embedded systems and further opens them to attack. This thesis will present necessary adaptations to traditional software practices that enhance the success of security solutions for embedded systems.

The responsibility to secure embedded systems falls to each and every individual involved in the life cycle of an embedded system. This life cycle begins in the research, feasibility, and conceptualization phases and carries through to prototyping and manufacturing, continuing until the product has reached end of life. The long life of an embedded system makes security very difficult, and each phase in a product's life is vulnerable to attack. Unlike most software, embedded systems receive updates less often, enhancing the need to ensure strong security from the start.

Ensuring the security of an embedded system to protect our nation's infrastructure and personal properties from attack is paramount for engineers developing today. The security of our infrastructures and the embedded systems that control their vital operation is not taken seriously enough by the market today. The reliance on these systems for daily life, has created the need for new security solutions to be developed to ensure these types of systems are robust against attacks. There are many avenues to securing embedded systems, including physical security, security training through education, encryption and real-time monitoring, to name a few. It is commonly said that the best security is the security that one will use or implement. Unfortunately, encryption is often the elephant in the room and carries with it a negative connotation. This connotation is evident no matter what stage of planning, design or execution. Encryption is one of the most easily achieved, and secure means to protect an embedded system. In the eyes of both managers and engineers alike, it often brings along false synonyms and stereotypes, such as encryption requires more time, money, head-ache and complexity. These costs often drive designers to rule out encryption before evaluating its benefits.

One could easily get lost in long mathematical formulas, time complexity calculations, and best and worst case analyses, but these topics are largely untouched by practicing engineers developing encryption modules for embedded systems. Of equal importance, but less complexity, is the application of secure design methodologies, paired with application of simple encryption algorithms or readily available encryption modules developed in open source or available for purchase as intellectual property (IP).

The overarching goal of this thesis is to provide engineers with logical, real-world application examples, paired with simple performance metrics to facilitate the evaluation of encryption algorithms for possible adoption in embedded designs. It encourages designing all embedded systems, connected to the IoT or not, with basic and robust security to prevent against cyber-attacks. The examples and metrics demonstrated in this thesis help engineers evaluate the many tradeoffs associated with implementing embedded security models as part of project specification, design and implementation.

Thesis Contributions

This thesis furthers the understanding and application of encryption algorithms for the design of embedded systems. The contributions made by this thesis are as follows:

1. Discussion and analysis of performance and design tradeoffs associated with implementing encryption in the design of embedded systems.
2. Provide embedded designers with analytical characterizations of simple encryption algorithms as implemented for direct application in modern 32-bit microcontrollers, developed using the C programming language.
3. Discuss and remove barriers and road blocks embedded engineers face when selecting, implementing and researching encryption algorithms for use in memory constrained and/or low power devices. Discussions include, but are not limited to removing initial assessment doubt as to the difficulty of implementing algorithms in low cost systems and a discussion on performance tradeoffs for embedded systems running encryption in real time. Performance tradeoffs regarding memory requirements vs level of security and speed vs level of security are discussed.
4. Provide a comprehensive analysis of secure design practices and methodologies. In addition based on this analysis, provide practical applications and adjustments to secure software design philosophies for the embedded system.

Profile of a Typical Embedded System

An embedded system is any device or collection of devices that is controlled by low-level software or firmware to accomplish a finite set of tasks. The term “embed” finds its roots in the Latin prefix im- meaning to put in or into. The term embedded therefore describes a computer system that is dedicated to performing specific functions within the context of a larger system. It is a single portion of a complete electrical or mechanical device. For clarity of discussion and to provide examples, the most common embedded system consists of three components:

1. Microcontroller: This device is brain of the system, and generally controls all functions of the system by communicating and directing operation of the system as a whole.
2. Input and output (I/O) devices: Input devices such as sensors, wireless radios and inter-circuit buses bring in data for the microcontroller analyze. The microcontroller processes this information making logic decisions on the available data in order to direct operations of output devices. Output devices such GPIOs enable the microcontroller to control peripherals and output data from the system in the form of a physical response, such as switching a relay or digital response in the form of data distributed to another system for the purpose of further processing or connecting the embedded device to a larger system.
3. Storage: At minimum, an embedded system must have some type of memory to store program code. Most embedded systems also have data storage.

The Internet of Things, a conglomerate or network of embedded systems with a method of connecting to each other such as Wi-Fi, hardwire network, Bluetooth etc. is the most active development branch in this field. Of the devices in the IoT, the “embedded sensor” is most common and often attacked point. The embedded sensor is the backbone of any larger system. The sensor is the point at which raw data enters the system. This raw data is what drives the software controlling the system. An embedded sensor is often in the control path for actuators, motors, alarms etc. and as such has great impact on the function of the system. The sensors are often exposed to the environment and are some of the most vulnerable parts of an embedded system to attack, due to lack of both physical and software security. Ensuring the security of an embedded system both IoT connected and not, must follow a fundamental set of design considerations. These design considerations are discussed later in the thesis.

Hardware Description

At the heart of a typical embedded system, as mentioned above is a microcontroller. The microcontroller acts as the brain of system. A microcontroller typically comprises a generic 8-32bit CPU, program storage, data storage, general purpose input and output (GPIO) and communications functions (I2C, SPI, UART, IP etc.). Each component of the hardware is susceptible to attack and security measures must be taken to ensure that each is protected. The microcontroller gathers all the data from the environment though sensors, user input, network connections etc. The microcontroller then processes all this information using its firmware written by the designer. Once processed, the microcontroller performs its tasks based on the data it collected. This process is repeated, often in cycles or via interrupts, billions of times over the life of a typical embedded system. The highly repeatable actions and design of embedded systems often makes them prime targets for attack precisely because their actions are predictable.

Software/Firmware Description

The software in an embedded system is most correctly categorized as firmware. Firmware is the set of read only instructions residing in flash memory of an embedded CPU. These instructions are rarely changed during the life cycle of a system. Firmware often has a simple structure due to resource constraints. The purpose of firmware is to parse and interpret the data entering the embedded system, transform it into a usable form, and make decisions based on a pre-defined set of rules or algorithms. Firmware must also output data, through its peripherals, to the rest of the system in a meaningful way, so as to further the operation of the system as a whole. Due to resource constraints and specific application requirements, many designers often focus on meeting design requirements and neglect security. This leaves firmware, sensitive data and the collective embedded system vulnerable to attacks by hackers. This thesis presents software design practices and methodologies that can protect embedded systems from such attacks by moving security design earlier into the development cycle.

Chapter 2

Notable Cyber Attacks Involving Embedded Systems

Cyber security breaches are reported in just about every newspaper and magazine today. The majority of recent cyber-attacks revolve around credit cards and email. These types of attacks are considered data breaches or software breaches. A data breach is the intentional or accidental dissemination of private information to untrusted entities. In these types of attacks, the attacker seeks only to damage a person's dignity or finances, rather than to inflict physical harm on a person or organization. Data breaches are consistent with the public perception of a hacker. The over generalization of a man and his computer stealing credit cards is what most people associate with when they hear of social security or credit card numbers being leaked. Attacks and security breaches of embedded systems are more likely to cause real physical harm to people and infrastructure. One could draw the analogy that credit cards are attacked by hackers and embedded systems are attacked by terrorists. The public's general view of a terrorist is one who attacks in order to cause as much harm as possible. These are the types of consequences exploitation of embedded systems can have. Researchers and governments have been warning about the security of our nation's infrastructure for years. Many people believe creating strong passwords and keeping your credit card number private are the keys to a secure world. Holding this type of narrowly defined notion, regarding what information needs to be protected, restricts the scope of threats applicable to embedded systems. To understand embedded security, one must look beyond securing only information that can be used for monetary gain, as is the case for most forms of identity and intellectual property theft.

The nation's cyber security policy is the first step to changing the understanding of the nation's need for new security measures. In Executive Order 13636, President Barack Obama lays out new directions for cybersecurity and one of the top priorities is "Protecting the country's critical infrastructure — our most important information systems — from cyber threats".[1] While this is one of the top priorities, it receives little attention from engineers and developers because it has been seldom demonstrated that compromising these systems pose a threat to our lives or generate monetary gain for attackers. There have been only two widely confirmed instances of physical damage caused by embedded systems, Stuxnet in June 2010 and the unnamed German steel mill attack. The attacks are unique because they relied heavily on the exploiting embedded systems. The attacks caused embedded systems to perform as designed, but in a manner that produced malicious results.

Stuxnet

Stuxnet was the first verified case of a computer worm used to attack an embedded system [2]. A computer worm is a specially designed virus that is not reliant on a secondary means of transportation to its host; worms replicate themselves. The Stuxnet worm is a piece of software designed to attack a specific brand and model of programmable logic controller (PLC). PLCs are widely used in manufacturing to control many types of equipment. The PLC is one of the most basic embedded systems, operating on a basic "see-this, do-that" type of algorithm. They are often connected to a network, making them targets of attack from anywhere in the world. PLCs are commonly found on assembly lines and often are used to perform repetitive tasks or high speed tasks. A typical PLC consists of analog or digital inputs, outputs, usually relays, and a small simple software control program. At its most basic level the PLC takes input, processes it, and then controls the output accordingly. Little or no security exists in most PLCs today. The Stuxnet worm affected a common Siemens PLC. The target was the Iranian Nuclear Program. The Stuxnet software is maliciously installed into the flash memory of a specific Siemens PLC via a network. The worm changes the RPM of connected motors and hides the change from the user by displaying false motor speeds to the system. These small changes accelerated the failure rate of centrifuges in the Natanz nuclear facility in Iran. The Stuxnet worm infected an estimated 58% of computers in Iran. While the Stuxnet attack merely slowed the productivity of the centrifuges, one can easily extend the possibilities for the worm to cause serious damage by turning off temperature sensors or shutting valves unexpectedly. The Stuxnet worm is a prime example of the vulnerability of embedded systems and the seriousness of the effects an exploit invoked upon them can be.

German Steel Mill

The incident at an unnamed German steel mill has been heavily redacted by the German government, but some information does exist in reports by the German Federal Office for Information Security (BSI). In a report released by the BSI, a story familiar to Stuxnet is confirmed by the German government [3]. The report explains that an unknown hacker was able to penetrate the mill's business network infrastructure through an email attack and then gain access to the computer network controlling production equipment. With access to this system the attacker was able to control a blast furnace's values, rendering the system "unable to shut down" [BSI].The result was significant damage to the blast furnace. While there are few details about this attack, the message is clear. First, the Internet of Things has connected vulnerable embedded systems to the outside world more readily exposing them to attack. Second, attacks on embedded systems strictly for the purpose of damaging infrastructure are possible and must be viewed as valid threats. The German mill attack demonstrates that unsecure and vulnerable embedded systems in every branch of infrastructure, including both public and private companies can be the target of attacks even when monetary gains cannot be considered a motive.

Chapter 3

Pillars of a Secure Embedded System

A common safety guideline, known as the “three point rule”, requires that when a person is climbing, a ladder for example, three points of contact should always be kept while moving. To ensure the security of an embedded system, designers must likewise study three points to ensure that an embedded system can be effectively secured. The three pillars are (1) Modeling and Analysis, (2) Secure Development Lifecycle and (3) Education. The three pillars presented below are the foundation of any embedded projected intended to produce a secure product.

1. Threat Analysis and Modeling

In order to ensure security for embedded systems over the wide range of potential attacks from the highly sophisticated to the simple, an entire ecosystem must be protected. An embedded system undergoes many stages throughout its development cycle including design, manufacturing, end user deployment and firmware upgrading. In order to ensure a secure system, care must be taken to secure each stage. Any unsecure point in the ecosystem can introduce a security risk, not only to the end user but to the developer. Threat Analysis is the first step toward any attempt to secure a system. The term “threat analysis” refers to the organized, systematic process of examining an embedded systems’ points of target and sources from which it will be attacked. Before any design decisions are made regarding methods of protection from attack, such as the use of encryption for data protection and integrity, a thorough threat analysis must be performed. The chart in Figure 1 demonstrates a proposed iterative threat modeling flow for an embedded system. The system design must first be reviewed to determine points of weakness and attack, followed by a detailed review of security requirements and objectives. Then modeling must occur for each security requirement, taking into account the three points of view of an attack scenario. Both the modeling of specific security requirements and system objects are then iteratively evaluated until a high level of certainty is reached that the model developed provides adequate security against the identified threats and no new vulnerabilities have been introduced.

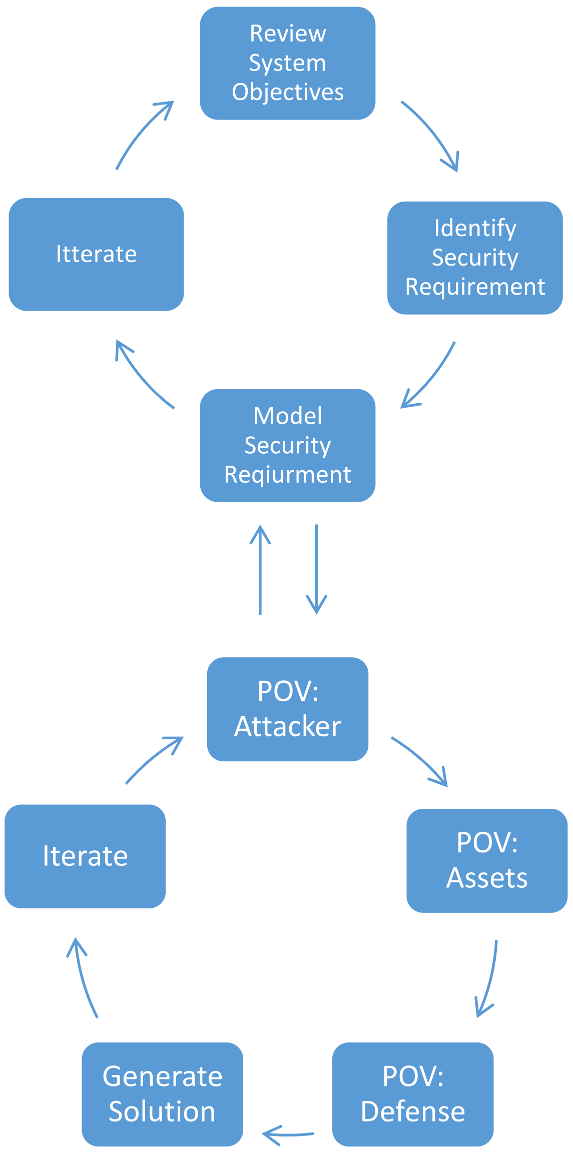

Figure 1. Threat Modeling Process

Attacker-Centric Threat Modeling

Threat modeling that is attacker-centric takes on the point of view of the attacker. This type of modeling is the most abstract of the three types of threat modeling. Why? This is because the best way to sum up attacker-centric threat modeling is to say: “Get inside the head of the attacker!” The attacker is studied to produce a profile that is used to understand the attacker’s motives and capabilities. The attacker centric threat model is heavily reliant on attack tree diagrams, projected thinking, and profiling, which often lead to some ambiguity, but ultimately provide guidance towards appropriate attacker mitigation methods.

Asset-Centric Threat Modeling

Asset-Centric modeling can be quite limited in the embedded world; nonetheless it is still very important. Assets in an embedded system can be defined as the information that passes through the system from component to component, or entering and leaving the system via the Internet of Things. Bus communications, flash memory contents, etc. are all assets modeled by the asset-centric modeling method. To model for asset-centric threats, the developer starts with the asset itself. Assets in embedded systems can also include the peripherals connected to the embedded system. Devices such as sensors, actuators, motors, relays, etc. are all assets to the embedded system because they bring in, or distribute out information valuable to the operation of the entire system. These devices are often used by both the recipients of an attack and the attacker to characterize the effectives or response of an attack. Assets often connote a sense of security, because designers view these as internal to the system and therefore not exposed to attack. The assets in traditional software design are trusted as reliable data sources and often the context of the data received from these assets is not analyzed for potential compromises. Asset-centric modeling should focus on modeling devices internal to the embedded system that send or receive data, whose integrity cannot be directly confirmed. This is achieved by creating contextual or situationally secure states of operation for a system. Secure software design must analyze all available data to ensure these systems are not comprised and evaluate the system as whole to decide whether trusted data can truly be trusted. Excessive resets, non-default configurations, monitored values above or below thresholds limits, excessive memory reads and writes, and other device statistics can be analyzed and monitored to gather situational awareness about the state of an embedded system. The asset-centric modeling should iterate on this data to generate a figure of certainty describing the level of trust in data. When this figure is determined to be outside of an acceptable range, data cannot be trusted even if no attack is detected. A simple example of this type of situational awareness is found in many modern desktop computers that have a mechanism to detect when the case is open and subsequently do not allow the user to boot the computer. The biggest security weakness of trusted assets is that compromised assets often act in a manner consistent with non-compromised assets. This is used by attackers to introduce nefarious data, code, or functionality into a system that the system views as valid. Assets of the system can easily be turned into assets for an attacker. If they are not included in threat models, and normal system operation is not characterized to determine acceptable variances, these assets become security weaknesses.

Attack Tree Diagrams

Attack Trees are a methodical graphical way to lay out potential threats against the embedded system. Each attack tree represents a unique threat to the system. The root node of each tree is a goal, either of the attacker or the developer, and leaf nodes are ways or paths to be followed in order to achieve that goal. Some attack tree models support additional levels of complexity by allowing each leaf to hold a value representing the likelihood that that event will occur. This ranking system can be used to filter a very large attack tree to show only the highest priority attacks. The nodes of an attack tree can be connected as either AND scenarios or OR scenarios. In the AND scenario the goal root node is achieved if all leaf nodes are traversed. Likewise in the OR scenario only a subset of the leave nodes must be traversed to achieve the root node's goal.

STRIDE Model adapted for Embedded Systems

The SRIDE model was developed by Microsoft as a model for application security. This model is highly accepted in the software realm as a tool for guiding secure software design. It can easily be extended to embedded systems. This thesis extends the scope of the STRIDE model to included embedded systems. STRIDE is an acronym that describes the six threats to a system.

1. Spoofing
2. Tampering
3. Repudiation
4. Information Disclosure
5. Denial of Service Attacks
6. Elevation of privilege

Purpose of STRIDE

The STRIDE model should be used as a guide for discussions in developing threat trees. The model categorizes the types of attacks developers need to protect against. By progressing through the STRIDE model, each type of threat can be discussed and organized into the six categories for later dissection and implementation. The model gives the developers a sense of scale to easily determine how much time it will take to develop the needed security measures for the embedded system. The following sections describe the six threats as seen from the point of view of an embedded system.

Spoofing

In layman's terms, spoofing is the act of one embedded system or device claiming to be, or acting like, an entity that it is not. Spoofing varies widely in the Internet of Things, from e-mail spoofing that a person gets in their inbox every day to IP address spoofing. Spoofing in the context of embedded systems can take one of two forms. First, hardware spoofing. Flash memory chips can be removed and replaced with alternative chips with new application code. In addition, sensors and other I/O peripherals can be emulated and or replaced with attacker modified models. The second form is protocol spoofing. Protocol spoofing is the introduction of an outside source or receiver onto a bus that listens to bus traffic (a bus scanner) or transmits on the bus. Embedded systems are especially susceptible to protocol spoofing because they use bit-level protocols with litte or no protection. Protocols like I2C, SPI, CAN etc. are highly susceptible to eavesdropping and the bus can be hijacked at any time and spoofed data injected on the bus.

Tampering

Embedded systems are also highly susceptible to tampering. Tampering is the act of an attacker damaging or disabling a portion of an embedded system. This often means physically tampering with sensors or other peripherals in order to inject real, but unexpected and potentially malicious, data into the embedded system. Attackers can also attack hardware subsystems such as fans, temperature sensors, power supplies etc. in order to facilitate the needed criteria for their attack. This criteria can in the form of a reset, over/under temperature situation, power failure, system lockup, etc. that puts the system into a state where an attack is not protected against. The attacker can modify sensor readings, voltage supplies and other seemingly unconnected entities in order to force the system into an undesired state, in which an attack is easier to assert. This can lead to or cause damage to a system, while the attacker was never required to modify code, break passwords, or steal sensitive data. On the other hand, tampering may not always result in damage to a system. Simply tampering in order to disable a system temporarily can allow an attacker the needed window of time to launch a more complex attack without triggering alarms or suspicion.

Repudiation

Repudiation, by definition, is the denial of the truth or validity of something. In embedded systems, repudiation is the most difficult of the six STRIDE attacks to prepare for. In traditional software applications, when a violation occurs or inter-module communication fails, a log file is usually created to store not only user keystrokes, but commands, timestamps etc. In the embedded world, developers are usually bound by speed and/or resource constraints and these advanced logging methods are often out of the question. This allows the user to perform almost any command or tampering and then reset the device to clear the evidence. In the embedded world, this threat is generally handled by trying to limit effects once invalid actions are detected. While logging of the current state may not be realizable, due to constraints, well designed shut down procedures, lock out states and error handling can be used to ensure security when the possibility of attack is detected.

Information Disclosure

Information disclosure comes in four forms in the embedded system. First, disclosure by protocol. A protocol, by definition, systematically structures data for easy transport and display. The orderly nature of protocols also makes them easily deciphered by attackers. Using unencrypted protocols is the most common method for an attacker to find information about the system. Developers should remember that the same data sheet used to develop the system is also available to the attacker. Bitmasks, binary strings, and bit patterns become decidedly less arbitrary and unique when these tools are also available to the attacker. Second, disclosure by physical copying. Programs and flash data can be easily read by an attacker by removing flash memory or connecting via developer interfaces, such as JTAG, allowing an attacker direct access to the internal elements of a system. Any information stored in flash memory should be assumed to be readable by an attacker; therefore measures must be taken to ensure that sensitive data is stored using encrypted flash or software encryption. Third, disclosure via the IoT. Embedded systems with an internet connection are inherently more vulnerable to information disclosure than non-connected systems. Developers of these systems must also follow secure design practices for standard internet protocols. Lastly, disclosure by source code. The source code is the most valuable piece of information available to the attacker. With the source code, the attacker can completely control every aspect of the embedded system.

Denial of Service

Denial of Service (DOS) attacks render embedded systems unusable, or very usable, depending on how one looks at it. The purpose of a denial of service attack is to render a system unable to respond. Herein lies the dual nature of DOS attacks. For example, imagine a large building with a card reader at every door. A single compromised card reader, might cause someone to use another door, while on the other hand, when every card reader is compromised, the entire building becomes locked down. By this, 'very usable' becomes clear, the attacker has now severely restricted the movement of personnel in or out of the building, potentially as a pretext to a larger or more complex attack scheme. Embedded systems have a unique role in the DOS attack realm. If an embedded system is connected to the IOT, it is both vulnerable to DOS attacks and can be used to generate DOS attacks. When an embedded system is subject to a DOS attack, it can shut down unexpectedly or cease to respond to user commands. This can result in unintended operation or the inability to stop operation in an emergency. On the other hand, a compromised embedded system can be used as part of a larger attack to launch a DOS attack on another system (as in the card reader example). Both of these are situations must be planned for by an embedded systems developer. The developer must understand what affect a compromised device has on the larger collective network. Questions such as: "If a single device fails does it bring down an entire system?" or "Does a compromised asset expose other areas to an attacker?" can help guide the design.

Escalation of Privilege

Escalation of Privilege for embedded systems can be viewed in one of two ways. First, at the lowest level, operating systems, software locks, interrupts and many bus protocols rely on priority levels to control the flow and response to information. The priority levels can be manipulated, to perform many unintended tasks. For, example elevating the priority of an interrupt can place a system into a state of continuous interrupt, starting a self-inflicted DOS. Likewise lowering a priority could hide or mask an important interrupt from being handled, such as a warning from a temperature sensor. The second way to view escalation of privilege is from the viewpoint of an already compromised system. Attackers often target simple systems first, then rely upon other devices in the network implicitly trusting the data distributed by the previously uncompromised device. By this method, the attacker gains credibility in the ecosystem. Malicious actions then invoked by an attacker, may be overlooked by other devices or systems, because they are cloaked under the guise of a trusted device.

Application of Encryption to Threats

Each of the threats discussed above can be partially mitigated by designing encryption into embedded systems. The focus of this thesis is encryption as tool, but not directly focused toward any one threat or threat solution. Encryption is widely applicable and therefore is a first step to protecting against the wide range of attacks out there. While often used as a catch all or first line of defense, it should not be considered a full security solution by itself and must be paired with hardware security solutions, secure development and good education as discussed further in the next sections.

2. Secure Development Lifecycle:

A secure development lifecycle covers the entire life time of an embedded system from initial planning stages to end of life. There are five phases in a secure development lifecycle. They are (1) Training (2) Design (3) Development (4) Production (5) De-commission. Figure 2 demonstrates a traditional waterfall style software development cycle consisting of design, development and production stages. To this traditional path, security training is added, that focuses on phase specific secure practices that designers must be educated on. This training is project specific, and must be continually refined as new threats emerge. This thesis should be viewed as a resource for designers in the design and development stages. It can be considered a tool in the training and education phase used to gain understanding and guide research and development cycles that incorporate encryption as a security tool. The secure development lifecycle also adds a secure upgrade and de-commission phase often left out of non-secure project life cycles.

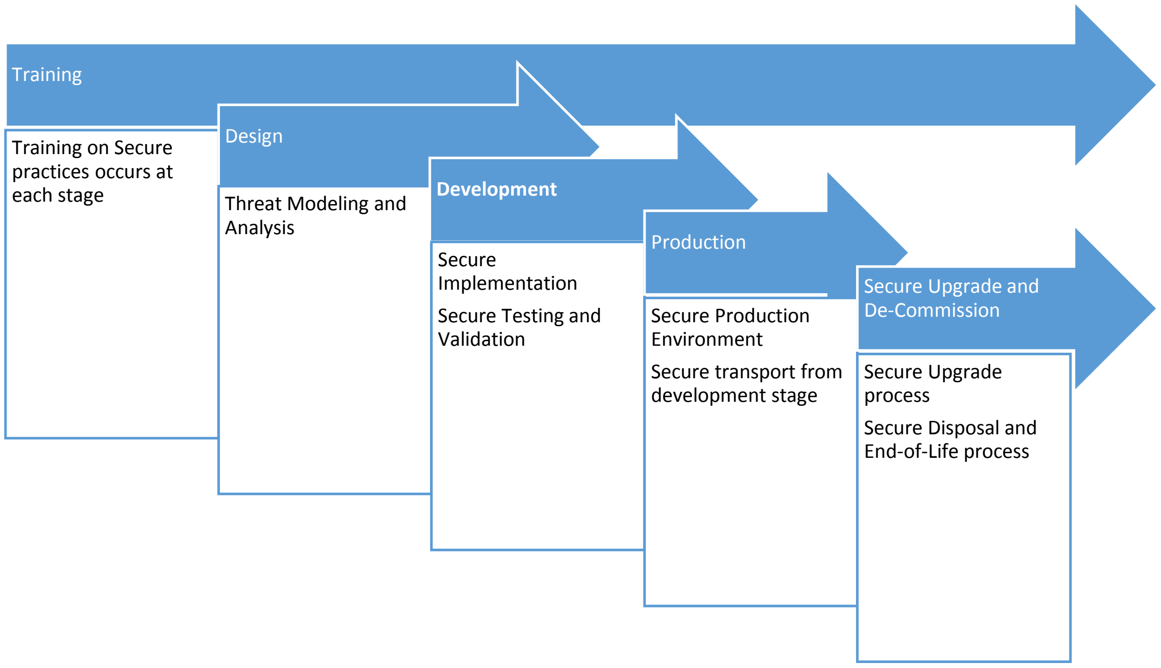
Training
Training on Secure practices occurs at each stage
Design
Threat Modeling and Analysis
Development
Secure Implementation
Secure Testing and Validation
Production
Secure Production Environment
Secure transport from development stage
Secure Upgrade and De-Commission
Secure Upgrade process
Secure Disposal and End-of-Life process

Figure 2. Secure Development Lifecycle

Training

Training is by far the most important element of a secure development lifecycle (SDL). The development lifecycle of any project exposes the system to many hands un-trained in security. Therefore, it is in the best interest of parties requiring security in a product to ensure the entire lifecycle is protected. Each phase of the lifecycle requires specific training for personnel involved in that phase. A secure development lifecycle, by default, requires that every stage, every person, every decision be made with security in mind.

Design

The design stage is generally understood as the secure part of the development lifecycle. This fact hinges on one very important criteria, which is secure intellectual property. During the design phase of any product, there can be countless diagrams, flow charts, code examples, emails, and other documents generated by the design team. Every one of these documents must be considered intellectual property and training must be provided to each team member on how to ensure these documents are kept private. Care must be taken to ensure that traditional design methods do not make information accessible to attackers.

Development

The development stage is often where security breaches can be traced to. The development stage is all about source code and coding decisions. To put it plain and simple, source code must be kept in a secure location. Source code is the golden ticket to any hacker. It gives an attacker a perfect map of the system they are attacking, both exposing security techniques, and security holes. This security is generally ensured by using source and version control on a secure server properly managed by IT personnel.

Production

To ensure security, the production stage must be completely separated from the development stage. All intellectual property (source code, etc.) must be secured in this stage. Encrypted bit streams should be used to ensure secure programming. The production stage should never have access to raw source code. Encryption at this stage allows the product to be handled in a much less secure factory environment, especially if this stage is outsourced.

Secure Upgrade and De-commission

A secure upgrade progress starts in the design stage. After the device leaves the manufacturer, it is open for attack by all. A secure upgrade process requires secure key storage and the use of encrypted bit files. A secure upgrade process should never expose the original source code or any part of the system to attack. It is important to ensure that regression testing has been performed to ensure that old firmware does not introduce new security flaws into a product.

Secure de-commission is the hardest aspect of the life cycle to control. It is more often than not unreasonable to assume a device will be returned and properly disposed of at the end of its life. For practical purposes, efforts should focus on preventing misuse of a product once it has reached end of life. This includes continued protection of intellectual property, proper de-commissioning of production equipment and properly designed operational expectations of the devices in failure and compromised modes of operation.

Application

The type of analysis and self-education this thesis provides directly reinforces the need for enhanced training before and during the secure development lifecycle. For a secure design to be effective, designers must research and analyze the performance and security of each element of an embedded system. This thesis provides, at minimum, a scope of understanding upon which is needed to effectively modify existing design practices of a development lifecycle to include secure elements. The secure element discussed here is encryption but the frame work allow for the inclusion of other security tools such a secure hardware devices into an embedded design.

3. Education

Leonardo da Vinici one said “learning never exhausts the mind”. The reason the third pillar of security is education is because security is an ever-changing challenge. Threats to embedded systems are constantly evolving. As hardware changes, new threats are introduced and old threats are removed. Any plan for security that does not involve a continued and iterative path of education is a weak plan for security. The education here applies to not only designers, but anyone involved in the product lifecycle. Education is by far the most valuable tool a system can have as a defense. Place an embedded system in the hands of an educated designer, educated user, and educated environment and security becomes a tool, not a hindrance.

Chapter 4

Encryption for the embedded system

Encryption in embedded systems serves three primary purposes. First and foremost it protects intellectual property. Secondly, it protects sensitive data. Lastly, it protects against un-authorized use. The six threats to embedded systems, spoofing, tampering, repudiation, information disclosure, denial of service and elevation of privilege, are specific examples of attacks exploiting one or more of the purposes of encryption. Encryption is only one tool for combating these attacks, but unique in the fact that it is the only tool that can be used to combat all threats. The cross attack effectiveness of encryption allows for the ideas and analyses presented in this thesis not only to be applied to a wide range of threats but also to create a framework for studying other tools and characterizing their effectiveness against an attack. The first two purposes of encryption are quite familiar. Intellectual property is often highly protected. Take, for example, the Coke recipe stored in a highly secure vault in Atlanta, GA. The intellectual property of an embedded system is most often its collective source code. Sensitive data is often data stored on the system, such as encryption keys or usernames and passwords. These first two purposes of encryption have obvious gains and clear consequences when not implemented. These consequences are that competitors can steal intellectual property and reproduce products, and thieves can steal sensitive data. Encryption as a form of anti-tampering or system protection takes on a less clear role, with often undefined consequences. Designers focus heavily on IP protection and data security without looking at how a device could be exploited. Encryption for this purpose protects against the system being used for nefarious purposes. This is the new frontier for hackers and designers must prepare for it.

In this chapter, three encryption algorithms are presented as tools designers can use to secure embedded systems. The algorithms' operations, as well as their security and performance are discussed. Each section is focused on a specific algorithm's strengths, weaknesses and performance. Closing the chapter is a comparison of the algorithms and a discussion of the recommended application for each of the algorithms.

Discussion of Algorithm Types

The two main types of encryption algorithms for embedded use are symmetric and asymmetric algorithms. Each algorithm type has its advantages and disadvantages. Symmetric and asymmetric algorithm architectures are discussed below.

Symmetric Algorithms

Symmetric algorithms date back as far as 1900 BC, beginning with substitution ciphers, wherein A is replaced by A + delta, and so forth. Symmetric algorithms encrypt and decrypt data using the same key or secret. This is most commonly embodied by performing repeated rounds of a mathematically invertible operation, such as XOR, add, subtract, multiply or divide, whereupon in decryption, the same round is repeated using the same key, but with the inverse mathematical operation. The simplest symmetric cipher is the substitution cipher. Using the alphabet as an example, a secret key is defined as “shift by 3”. In this case the word CAT is encrypted as cipher text as FDW. To decrypt, the letters are shifted or substituted back and the word CAT is retrieved as the plain text. Likewise, no matter how complex the operation on the data or encryption method is, a similar pattern is followed for many symmetric algorithms. The secret key, which can be any defined as a sequence or number, is applied to data, transforming it into cipher text. To unencrypt the cipher text the same key must be used. This poses the problem of distributing the key securely to the receiver and is the biggest drawback to symmetric key algorithms.

Asymmetric Algorithms

Asymmetric algorithms, in which encryption is comprised of both secret on non-secret portions, is relatively new, beginning with the Diffie Hellman algorithm in 1976 [4]. Asymmetric algorithms encrypt and decrypt data using a key pair. A key pair contains one private key and one public key. Asymmetric algorithms, known as public key encryption algorithms currently do not have known time-efficient solutions for cracking. These algorithms rely on large mathematical computations on prime numbers or elliptic curves, requiring enhanced processing power, for both encryption and decryption. Unlike symmetric algorithms, a public key can be shared in plain text without necessarily using a secure exchange. This also allows the receiver to accept encrypted messages from anyone with a public key - a many-to-1 relationship. The high security of asymmetric algorithms relies on the inability to generate the private key from the public key in a computationally feasible amount of time. Higher security comes at a cost. Increased encryption and decryption time, and complex key generation routines are common in asymmetric algorithms. Often these algorithms are used to provide a means of secure key exchange for symmetric algorithms. This removes the disadvantage of symmetric algorithms, requiring a key exchange, but retains the speed of symmetric algorithms.

Encryption Cost and Performance Factors

Embedded encryption is bound by many cost vs. performance trade-offs. The costs discussed in this thesis are (1) algorithm size (measured in ROM and RAM usage bytes), (2) energy consumption and (3) data integrity. The performance factors discussed in this thesis are: (1) encryption time, (2) decryption time and (3) brute force security. The algorithms characterized in the next three sections focus on these cost and performance metrics. These metrics provide a way to compare the algorithms despite their widely different implementations and complexity. Similar metrics should be developed or studied by designers of encryption modules to determine the security scheme that maximizes performance while reducing costs. It is suggested that algorithm size has been given somewhat lesser weight than energy consumption for most embedded systems. This is especially true for embedded systems that operate on battery power. In fact, in the course of researching for this thesis, 8-bit microcontrollers were considered as a target for simple encryption techniques adapted for use with single byte operands, but the cost in dollars of 32-bit systems has been reduced so drastically that it is more cost effective to use 32-bit platforms for research and development projects. In addition, many 32-bit platforms have been designed from the ground up for low-power performance. Low-power optimized processors are outfitted with low-power modes, allowing designers tweak out long battery lives. These advances for 32-bit platforms have allowed designers to move to 32-bit solutions, while still retaining the low power consumption of 8-bit microcontrollers. As for the performance factors, encryption time and the metric of brute force security remain of equal weight and both weigh heavily into designers' decisions.

Data Integrity and Encryption

Data integrity and encryption are not one in the same. It is a misconception that encryption provides data integrity. Encryption does not increase the quality of transmitted data or raise the likelihood that transmitted data will be received exactly as it was sent. Traditional error detection schemes such as cyclic redundancy check (CRC) or digital signatures must still be used. For completeness, data integrity is viewed as a cost in this thesis, because the effect of errors in an encrypted data set is often larger amounts of corrupt data than had the same error occurred in an unencrypted data set. The loss of data integrity in an encrypted data set can result in a single error being propagated forward or the corruption of the entire data set. To ensure the integrity of encrypted data a cryptographic hash function must be used to compute a hash on the data, which is separate step requiring separate tools and time. This topic is left for future research.

Research Platform

The research tools for this thesis were provided by Cypress Semiconductor through Cypress University Alliance. More information about this program can be found at http://www.cypress.com/university-alliance. Provided for the research were two Cypress Programmable System-on-Chip Solutions. The kits were model CY8CKIT-5-PSoC 5 Low Power Development Kits. The CPU is a 32-bit ARM Cortex-M3. The kit offered a great development platform, for this research integrating power monitoring tools, a breadboard and extensive I/0, including a LCD. All code development was done using the Cypress PSOC Creator software.

Analysis of the Tiny Encryption Algorithm (TEA)

Description

The Tiny Encryption Algorithm (TEA) is a symmetric encryption algorithm designed for high speed and low memory usage. The algorithm makes use of a Feistel type structure for the manipulation of data blocks for encryption. TEA was proposed by Wheeler et al. in 1994 as an algorithm designed to “get security with simplicity” [5]. The algorithm relies on the functional building blocks of most microcontrollers and only uses shift, add, subtract and multiply instructions to manipulate data. These operations are practically universal and are supported across most programming languages and handled directly in hardware on most microcontrollers. The use of common instructions, which most microcontrollers have optimized, allows for high speed to be assured across devices and platform architectures.

Feistel Structure

The Feistel structure is performed in rounds. The Feistel structure is displayed as a invertible mathematical function *F(P,K)* which represents a sequence of *n* Feistel rounds. The input data vector P, is the plain text data set. It is divided equally into two blocks $(Pl_, Pr)$. K, the encryption key is divided into n sub-keys, $K_0 \ldots K_{n-1}$, one for each round. A round function *R(P,K)*, is used to manipulate one half of the input data at a time. This function is an invertible mathematical function. For each P, Feistel Structure computes $F(P_, K)$:

```
$F(P, K) =$
        $for\ (i = 0\ to\ n)\ compute$
        {
                $Pl_{i+1} = Pr_i$
                $Pr_{i+1} = Pl_i\ XOR\ R(Pr_{i,} K_i)$
        }
```

After n rounds, P is transformed to cipher text, C, i.e. $[Pl_0, Pr_0] \rightarrow [Cl, Cr]\ where$ $Cl = Pl_{n+1}\ and\ Cr = Pr_{n+1}$.

For decryption, the Feistel structure uses the inverse of R, R', to re-compute the plain text.

```
$F(C,K) =$

        $for\ (i = n\ to\ 0)\ compute$
        {
            $Cr_i = Cl_{i+1}$
            $Cl_i = Cr_{i+1}\ XOR\ C(Pl_{i+1,}K_i)'$
        }
```

After *n* rounds, *C* is transformed to plain text, P, i.e$[Cl_0, Cr_0] \rightarrow [Pl, Pr]$ $where$ $Pl = Cl_{n+1}$ and $Pr = Cr_{n+1}$.

The two Feistel rounds are identical, apart from an inversion of the sub key K order and data order. The sub key order for decryption is reversed, i.e. the sub key blocks are processed from K_n down to K_1 . Likewise the data blocks, C, are processed from Cn down to C_1. The Feistel structure allows for significant code reuse and results in both performance and memory savings.

Encryption

The encryption routine provided by Wheeler et al. is written for the C programming language. Data is provided in v[0] and v[1], 32-bit word vectors. The 128-bit encryption key is provided in 32-bit words, K[0] – K[3]. Figure 3 demonstrates the C routine used to implement the TEA algorithm for this thesis. The input n is the cycle count, used as a security scale factor. Delta, is a constant, suggest by the designers to ensure variability of the cipher text when single bit changes occur in the plain text. Finally, v is a vector representing the data to encryption and k a vector key to encrypt the data.

```
void encrypt (uint32_t* v, uint32_t * k)
{
    uint32_t y=v[0],z=v[1], sum=0,                      /* set up */
    delta=0x9e3779b9 ,                                  /* a key schedule constant */
    n=32 ;                                              /* number of rounds */
    while (n-->0)
    {                                                   /* basic cycle start */
        sum += delta ;
        y += ((z<<4)+k[0]) ^ (z+sum) ^ ((z>>5)+k[1]) ;  /*Feistel Rounds */
        z += ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3]) ;
    }                                                   /* end cycle */
v[0]=y ; v[1]=z ;
}
```

Figure 3. TEA Encryption Routine

The key, K, is 128 bits, split into four 32-bit sub key vectors K[0], K[1], K[2], K[3], where K[0] is the most significant 32 bits of K. The plain text D, 64 bits, is divided into two equal halves, D[0] and D[1], where D[0] is the most significant 32 bits of D. The keys and data split at word boundaries and are recombined as bitwise concatenations. Each half, in turn, is then used to seed the other half for the Feistel round. The delta constant used here, and specified by the designers, has no security significance, other than assurance that the data is effectively transformed. The TEA algorithm uses a running sum, modulo 32 bits, instead of the XOR of the Feistel structure. This adaptation allows for improved speed and applicability across programming languages and is acceptable by its ability to be inverted mathematically. The shifts are performed bitwise, with zeros shifted into vacant bit locations. Figure 4 demonstrates the logical structure of the TEA encryption module.

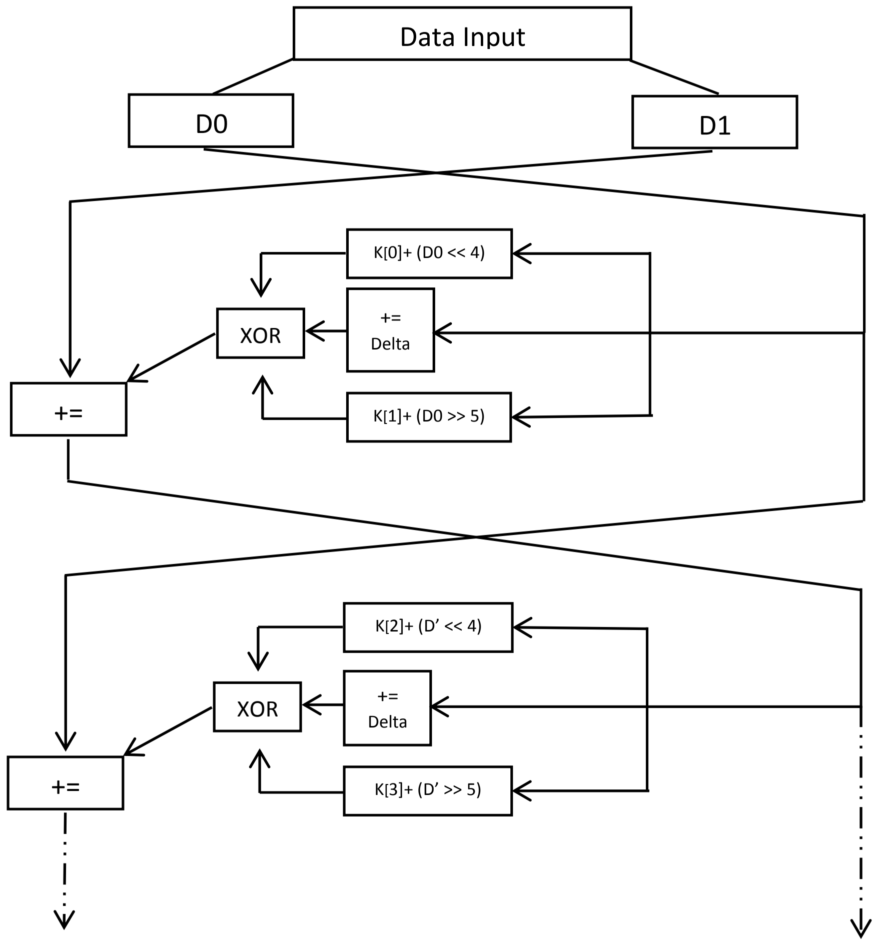

Figure 4. TEA Encryption Data Flow for Single Round

Decryption

The Decryption routine is displayed in Figure 5. The decryption routine is formed by processing the input block, using the crypto keys in the inverse order and applying the inverse of the running total, i.e. the running subtraction, modulo 32 bits.

```
void decrypt (uint32_t* v, uint32_t * k)

{
    uint32_t y=v[0],z=v[1], sum=0,                    /* set up */
    delta=0x9e3779b9 ,                                 /* a key schedule constant */
    n=32 ;

    sum = delta * 32;
    while (n-->0)

    {                                                  /* basic cycle start */
        sum -= delta ;
        y -= ((z<<4)+k[2]) ^ (z+sum) ^ ((z>>5)+k[3]) ;
        z -= ((y<<4)+k[0]) ^ (y+sum) ^ ((y>>5)+k[1]) ;
    }                                                  /* end cycle */
v[0]=y ; v[1]=z ;

}
```

Figure 5. TEA Decryption Routine

Performance and Cost Metrics

The TEA algorithm performance and cost metrics were gathered for a constant data set size of 1024 bytes. The CPU clock speed was varied from 4-24 Mhz. Over this range, the encryption and decryption time of TEA was captured and the power costs were monitored. Table 1 demonstrates the fast performance of the TEA algorithm, with respect to other more complicated block ciphers such as AES discussed in the next section; Table 4 also shows energy related costs. Notably, of the three algorithms profiled, TEA has the most symmetric encrypt and decrypt times, averaging only a 0.55ms difference between the encrypt and decrypt paths. This figure is regardless of clock rate, resulting in a predicable delta between encryption and decryption cycle times. This allows the necessary buffers between encryption and decryption modules to remain fixed size, regardless of clock speed. An advantage of this fact is that the clock rate can be adjusted to meet low power needs, without modifying buffer size.

Performance			**Cost**		
Clock (Mhz)	Encrypt Time(s)	Decrypt Time(s)	Avg. I (mA)	Power(w)	Energy (J)
24	0.01864	0.01812	0.013387	0.0441771	0.00082346
12	0.03743	0.03681	0.00729	0.024057	0.00090045
8	0.05665	0.05608	0.005264	0.0173712	0.00098408
6	0.07473	0.07404	0.004248	0.0140184	0.0010476
4	0.11206	0.11163	0.003312	0.0109296	0.00122477

Table 1. Performance metrics gathered on the TEA Algorithm for a data set size 1024 bytes on a Cypress CY8C58LP.

Security Analysis

TEA suffers from the existence of equivalent keys. Each key has four equivalent keys that produce the same cipher text when applied to the same plain text. This results in an effective key length of n-2 bits. For a 128-bit key, the effective key length is 126 bits. This results in a minor loss of security. A more significant security concern is that, some research has shown that the TEA algorithm can be broken with 2^{23} attempts using a reduced set of keys [6]. This complexity makes a brute force attack reasonable. The implication of a 2^{23}brute force time complexity on a 128-bit key is projected to 64 and 256-bit keys sizes in Table 2, using calculations provided by Vikram Reddy Andem [7]. Key sizes below 64-bits are not projected, because their security level is too weak for practical use. It should be noted that TEA should not be used as a hash algorithm due to the existence of equivalent keys.

Key Size	**Key Permutations**	**Projected Security**
1	1	-
2	1	-
4	4	-
8	64	-
16	16384	-
32	8.38E+06	-
64	7.03E+13	2.00E+11
128	9.67E+24	2.00E+23
256	9.35E+49	2.00E+45

Table 2. Measure of TEA Security vs Key Size

Analysis of the Advanced Encryption Standard (AES) Algorithm

Description

The AES algorithm was designated a standard by the U.S. National Institute of Standards and Technology (NIST) in 2001 [8]. AES is based on the cipher proposed by Rijndael, known as the Rijndael cipher [9]. Like TEA, AES is a symmetric key cipher and uses the same key for encryption and decryption. Unlike TEA, AES uses a substitution-permutation network, the mathematics of which are not within the scope of this thesis. The key sizes specified are 128, 192 and 256-bit keys. The algorithm's round count is specified by key size and is 10, 12 and 14 rounds respectively. AES 128 is characterized in this thesis. The analysis structure can be extended to other key sizes.

S-box

The substitution box, or s-box is the basis of the AES algorithm operation. The s-box serves as a lookup table, with which both key bytes and data bytes are transformed during the operation of the AES algorithm. This lookup table replaces the traditional mathematical operations of other ciphers with fast-performing matrix operations. The S-box is defined as the multiplicative inverse of a number defined as Rijndael's finite field. The field is represented by the equation $GF(2^8) = \frac{GF(2)[x]}{(x^8+x^4+x^3+x+1)}$. The s-box's generation is beyond the scope of this thesis. The s-box is readily available in many formats for implementation in the most common architectures. Both the s-box and the inverse s-box are usually stored in RAM to ensure fast access times.

Algorithm Operation

AES operates in rounds, like a traditional block cipher. For AES-128, ten rounds are performed on a 128-bit block. All rounds are performed the same, except the final round. This applies to the final round for all keys sizes. The input data block D is called the input state array. The state array has the form of a 4x4 matrix, as shown in Figure 6. The input D is divided into 16 single bytes. Each column is called a word.

```
Input_State = {  b0, b4, b8, b12,
                 b1, b5, b9,  b13,
                 b2, b6, b10, b14,
                 b3, b7, b11, b15 }
```

Figure 6. AES State Array

A single round of AES comprises four steps. The four steps are explained next.

Step 1. Round Key Expansion

In each round the 128-bit key must be expanded into a key schedule. The key schedule is the data set representing the expanded original key, which the algorithm uses for encryption and decryption. The original 128-bit key is placed into a 4x4 key state array in the same manner as the input state. The 4 words, or 4 columns, of the key state are expanded into a key schedule. For a 128-bit key, the key schedule is 4x44 array.

The Key Expansion Algorithm

The key expansion algorithm is a designed to ensure that single bit changes in the input key produce a key schedule expansion that results in a new round key for more than one round. The key expansion for a 128-bit key produces a 4x44=176 byte round key. AES uses a round constant defined as a 32-bit word. This word's lower three bytes are always zero and the upper byte is 2^i, where i is the round number. This is usually a separate function denoted as "rcon(i)". This constant performs the same role as the delta constant in TEA and prevents back to back rounds from producing the same cipher value. The key expansion routine is best implemented as multiple smaller functions to ensure proper operation. The expansion steps are detailed below.

1. Expand the four key state words into the first 16 bytes of the round key, as they exist in the original key
2. rcon = rcon(1)
3. Until 40 words are expanded do:
    a. Word 1
        i. Create a new word *temp* initialized to the previous four bytes of the round key
        ii. Rotate word 8 bits left
        iii. Substitute each byte using the s-box
        iv. Apply rcon(i++); to most significant word
        v. XOR *temp* with the first word of the previous 16-byte expansion
        vi. *Temp* now holds the next four bytes of the round key

b. Words 2-4
    i. Create a new word *temp*, initialized to the previous 4 bytes of the round key
    ii. XOR temp with the word 16 bytes previous in the updated round key from step a.
    iii. Append word to round key and repeat

The round key has been expanded. A sample expanded key would be as follows:

Input Key:

0x00000000000000000000000000000000

Expanded Key Schedule:

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
62 63 63 63 62 63 63 63 62 63 63 63 62 63 63 63
9B 98 98 C9 F9 FB FB AA 9B 98 98 C9 F9 FB FB AA
90 97 34 50 69 6C CF FA F2 F4 57 33 0B 0F AC 99
EE 06 DA 7B 87 6A 15 81 75 9E 42 B2 7E 91 EE 2B
7F 2E 2B 88 F8 44 3E 09 8D DA 7C BB F3 4B 92 90
EC 61 4B 85 14 25 75 8C 99 FF 09 37 6A B4 9B A7
21 75 17 87 35 50 62 0B AC AF 6B 3C C6 1B F0 9B
0E F9 03 33 3B A9 61 38 97 06 0A 04 51 1D FA 9F
B1 D4 D8 E2 8A 7D B9 DA 1D 7B B3 DE 4C 66 49 41
B4 EF 5B CB 3E 92 E2 11 23 E9 51 CF 6F 8F 18 8E
```

Step 2. Add Round Key

This step occurs once for each 128-bit block and is performed before any encryption rounds. In this step the bytes of the input state array are XORed with the first 16 bytes of the key. This is performed in index order, i.e.

```
k = 1
for(i = 1 to 4)
    for(j = 1 to 4)
        input_state[i,j] XOR key[k++]
```

Step 3. Encryption Rounds

Step 3a. Substitute Bytes

This step is a non-linear substitution of the input state with the s-box. The substitution is performed via a lookup table. For example, the input_state(2,2) is replaced with s-box(2,2).

Step 3b. Shift Rows

In the shift rows step, the first row of the state is not modified. Then state rows 2, 3 and 4 are circularly shifted logically left by 1, 2 and 3 bytes, respectively.

Step 3c. Mix Columns

Each byte in the state is operated on by a function of all the bytes in the containing column. The function bytes, by row, are as follows:

*state(0,x) = 2*state(0,x) XOR 3*state(1,x) XOR state(2,x) XOR state(3,x)*

*state(0,x) = state(0,x) XOR 2*state(1,x) XOR 3*state(2,x) XOR state(3,x)*

*state(0,x) = state(0,x) XOR state(1,x) XOR 2*state(2,x) XOR 3*state(3,x)*

*state(0,x) = 3*state(0,x) XOR state(1,x) XOR state(2,x) XOR 2*state(3,x)*

Step 4c. Add Round Key

In this step the bytes of the input state array are XORed with the first 16 bytes of the key. This is performed in index order of bytes, i.e. input_state[i] XOR key[i]. This is the same step as two.

Step 4. Final Round

The final round is a repeat of step three, with the mix columns routine removed. Therefore the final round, Step 4, is comprise a repeat of the steps detailed above. The steps performed are:

4a. Substitute Bytes

4b. Shift Rows

4c. Add Round Key

AES Algorithm Steps Summary

1. Key Expansion
2. Add Round Key
3. Rounds
    a. Substitute Bytes
    b. Shift Rows
    c. Mix Columns
    d. Add Round Key
4. Final Round
    a. Substitute Bytes
    b. Shift Rows
    c. Add Round Key

The operation of AES is summarized in the high level flow chart in Figure 7. This figure uses matrix state notation for input plaintext, output cipher text, the key, and key schedule.

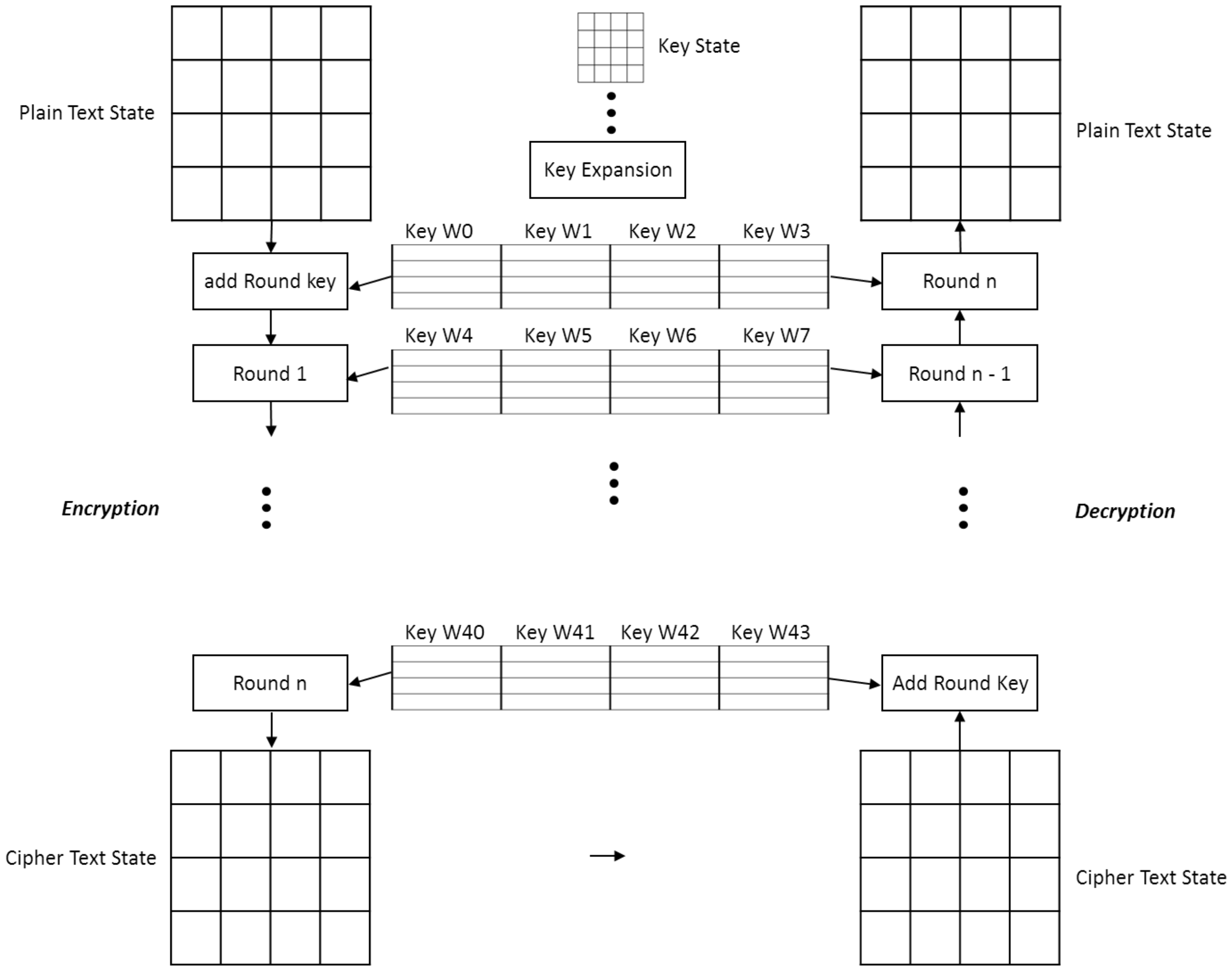

Figure 7. AES High Level Operation

<u>Performance and Cost Metrics</u>

The AES algorithm developed showed good speed performance as a block cipher, operating at approximately one half the speed of the TEA implementation. Table 3 lists the performance characteristics and cost the implemented AES algorithm. The average encryption time for 1024 bytes was 36ms. The data demonstrates that, unlike TEA, the difference between encryption and decryption times is not constant and linearly increases with respect to clock rate. The code size for AES is significant, at 344 bytes of ROM for the support and main algorithm routine and two 256-byte blocks of ROM for the forward and reverse s-boxes. The algorithm also requires 68 bytes of RAM, or 244 bytes if the key schedule is stored in RAM. The real performance gains come from the increased security, with 128-bit key brute force attacks requiring 3.4E38 combinations.

Performance			**Cost**		
Clock (Mhz)	Encrypt Time(s)	Decrypt Time(s)	Avg. I (mA)	Power(w)	Energy (J)
24	0.036782	0.023939	0.012731	0.0420123	0.0015453
12	0.072354	0.047718	0.007206	0.0237798	0.00172056
8	0.109221	0.074144	0.005306	0.0175098	0.00191244
6	0.145631	0.095262	0.004355	0.0143715	0.00209294
4	0.219004	0.142983	0.003404	0.0112332	0.00246012

Table 3. Performance metrics gathered on the AES Algorithm for a data set size 1024 bytes.

Analysis of the RSA Encryption Algorithm

<u>Description</u>

The RSA encryption algorithm was developed by Rivest, Shamir, and Adelman (RSA) in 1978 [10]. RSA is an asymmetric public key algorithm. The encryption key is public and can be shared without compromising the integrity of data. The decryption key is considered secret and must be protected in the same manner as symmetric keys. The RSA algorithm works by first generating a public key, seeded by two very large prime numbers. The key is shared with anyone wanting to encrypt data to be sent to the receiver. The receiver, which retains the large prime number used to generate the key, uses these numbers to produce a decryption key. The security of RSA is directly related to the ability of modern computers to factor the product of very large prime numbers in reasonable time. The RSA algorithm, at currently computational efficiencies, is immune to a brute force attack for all key sizes, with a 1024 bit key requiring greater than 10^{200} operations to attempt all possibilities. Other research has shown achievable factorability of primes up to 768-bit keys [11]. This research suggests that 2048-bit keys, and greater should be used to provide security for timespans of more than 20 years. The security and prime factorization problem is outside the scope of this research and a designer can reasonably trust published reports of RSA's security.

The following sections detail the operation of RSA.

Algorithm Operation

Key Generation

As discussed above, RSA uses both a public and a private key derived from two large prime numbers p and q. The process of key generation is detailed below.

Step 1: Prime Selection

The prime integers p and q should be pseudo-randomly generated prime numbers with different lengths. For this thesis, prime integers were generated and verified using a brute force primality test. For a 1024-bit key the average time was greater than 30 seconds. No optimization was attempted for this method, although many methods exists that can significantly reduce verification time.

Brute force Primality Verification:

```
for(i = 1; i < n; i++)
{
        if(n % i == 0) return false;
}
return true;
```

Step 2: Compute Key Length

The key length of the private and public keys is computed as $n = p*q$. The key length is used in step 5.

Step 3: Compute ϕ

ϕ is computed using Euler's totient function: $\phi = (p-1)*(q-1)$. This value should be kept secret and is used in calculating the private key.

Step 4: e Selection

The integer e is selected with the constraint $1 < e < \phi$, and $\gcd(e, \phi) = 1$. This value is the modular exponentiation factor used to produce the cipher and plain text. This value is the public exponent.

Step 5: Compute d

The value d, the modular multiplicative inverse of e, is the private exponent. The private exponent, d, is solved for by calculating: d*e = 1 (mod ϕ(n)). In code, this can be implemented as:

```
d = 1;
while(temp != 1)
{
        temp = (d*e)*mod(phi);
}
d = temp- 1;
```

Step 6: Key Generation

Finally, the public key is represented as *public(n,e)*. The cipher text can be calculated as $\mathrm{C(m)} = m^e \, mod(n)$.

In review, the private and public values needed for key generation, encryption and decryption are summarized in Table 4. Public values are those needed for encryption, but that can be shared freely allowing anyone to encrypt a message. Private values are those needed for decryption and must be kept secret by the decrypting party.

RSA Cryptosystem		
Integer	Public	Private
P and Q		▲
N (modulus)	▲	
E (public key exponent)	▲	
D (private key exponent)		▲
Φ		▲

Table 4. RSA Cryptosystem Integer Summary

Encryption

The transmitter uses the public key, *public(n,e)* to encrypt the data. The data P, plain text, is first transformed to a message m, with the following constraint: $0 \leq m < n$ and $\gcd(m, n) = 1$. This transformation takes place in the context of a pre-defined padding scheme. The padding scheme is used to prevent the m value from being mathematically structural and introduces randomness to the encryption message, ensuring encryptions produce semantically secure results. The padding scheme is not considered a secure element of the algorithm, as both parties must know the padding scheme. It is conjectured that additional security can be achieve with a one-way padding scheme, in which the inverse of the padding scheme can be kept secret. Following transformation of the plaintext P, into integer m, the cipher C is computed as: $\mathrm{C(m)} = m^e \, mod(n)$.

Decryption

The receiver use the private key *private(n,d)* to decrypt the cipher text C. Using $\mathrm{P(m)} = C^d \, mod(n)$, the result, m, is then transformed using the inverse of the padding scheme used for encryption.

Performance Metrics

The performance metrics for RSA are based on the assumption that the necessary keys are pre-computed. The initialization time to compute these keys is significant, but is not considered a factor in steady state data transfer. After the key computation has occurred, data can be encrypted using they calculated key as the data is transmitted, i.e. during steady state data transfer, without introducing significant delay. The RSA algorithm demonstrates that asymmetric algorithms have a higher time cost than symmetric algorithms for encryption and decryption, yet still produce linear results as a function of clock speed. The power consumption is also significantly increased. These results are summarized in Table 5. The encryption time trends show that RSA has data rates five times slower than those of faster symmetric algorithms. The difference between encryption and decryption module time requirements demonstrates a trend similar to that of TEA in which the encryption time – decryption time delta is a constant.

The performance of key generation cannot be ignored all together. Key verification time is non-trivial, as discussed greater than 30 seconds on the hardware used for this research without optimization. With optimization, sub-second generations can be achieved. These results were not tested and are left for further analysis.

Performance			**Cost**		
Clock (Mhz)	Encrypt Time(s)	Decrypt Time(s)	Average Current (mA)	Power(w)	Energy (J)
24	0.10091	0.08654	0.012416	0.040973	0.004135
12	0.20217	0.18764	0.007027	0.023189	0.004688
8	0.30281	0.28818	0.005183	0.017104	0.005179
6	0.40394	0.38921	0.004263	0.014068	0.005683
4	0.60533	0.52021	0.003454	0.011398	0.0069

Table 5. Performance metrics gathered on the RSA Algorithm for a data set size 1024 bytes.

Conclusions

RSA is a highly secure algorithm for data exchange. It is the only viable choice, of the algorithms discussed, for secure data transfer when a secure key exchange cannot be guaranteed. The enhanced security comes at the cost of significantly slower real-time encryption and decryption as well as substantial overhead and initialization for key generation. RSA is quite efficient in terms of memory requirements as discussed in the next section, if the key generation and verification logic is removed and only encryption and decryption routines are considered. This would be the case in a one way sensor network, in which data is transmitted back to a server and client nodes only need to know the public key.

Comparison of TEA, RSA, and AES Encryption Algorithms

Security

Level of security is the most important performance characteristic of any algorithm, yet is often the hardest to quantify. The two algorithm types, symmetric and asymmetric, serve different purposes in encryption. High data rates are best served by symmetric algorithms, while ultimate security is best served by asymmetric algorithms. Comparing their security, in the most simplistic terms, leads to a conclusion that symmetric algorithms are more secure and asymmetric algorithm are less secure. In reality, designers must realize that the resources needed to implement asymmetric algorithms, such as RSA, are not realistic in every situation. The highest level of security is not always achievable. In order to evaluate whether an algorithm provides a suitable level of security, one can look at the resources required to reach a stated security level objective. In the sections below, more quantifiable benchmarks than a simple ‘brute force security level’ are presented to help the designer narrow which algorithms best serve the interests of a specific design.

Metrics such as key size and encryption speed can be used to rule out the practical implementation of the most secure algorithms. Using metrics such as block encryption time, power consumption, and memory requirements, can narrow the available algorithms to a few choice algorithms. Using the various scale factors of each algorithm, such as TEA's number of Feistel rounds, or the key length of AES, the selected algorithms' levels of security can be maximized by finding the optimum set of scale factors that provides the most security within the resource constraints.

Encryption and Decryption Time

Comparing TEA, AES, and RSA, the results demonstrated similar performance for the two symmetric algorithms TEA and AES. In all cases, RSA took significantly longer to transform the data. Encryption for all algorithms took slightly longer that the decryption routines. To generate the plots in Figures 8 and 9, below, the clock speed was varied from 4 – 24 Mhz and the time to encrypt 1024 bytes was measured. The difference in speed was most apparent at lower clock speeds. If higher clock speeds can be support by architecture and power requirements the higher security of AES and RSA can be easily achieved with minimal delays.

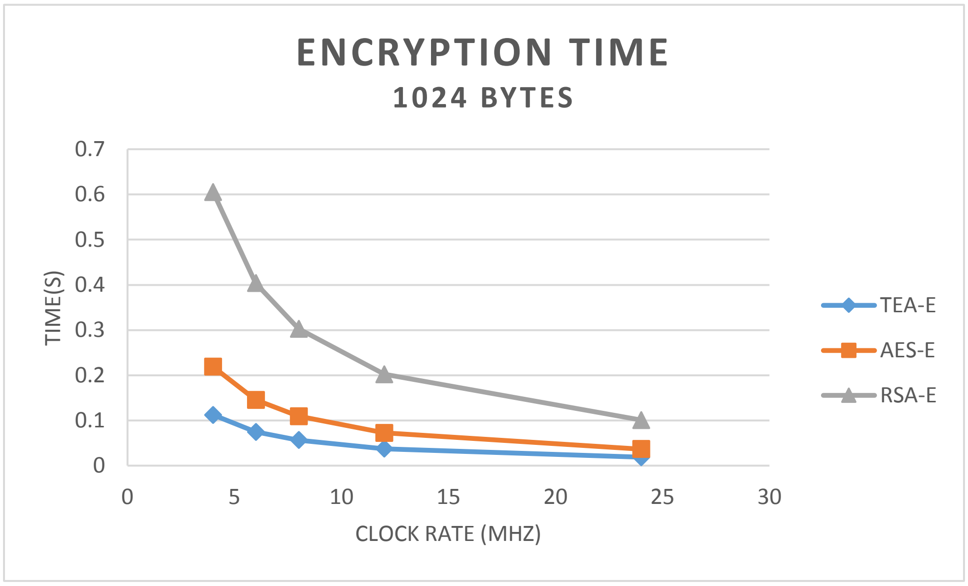

Figure 8. Comparison of Encryption Times

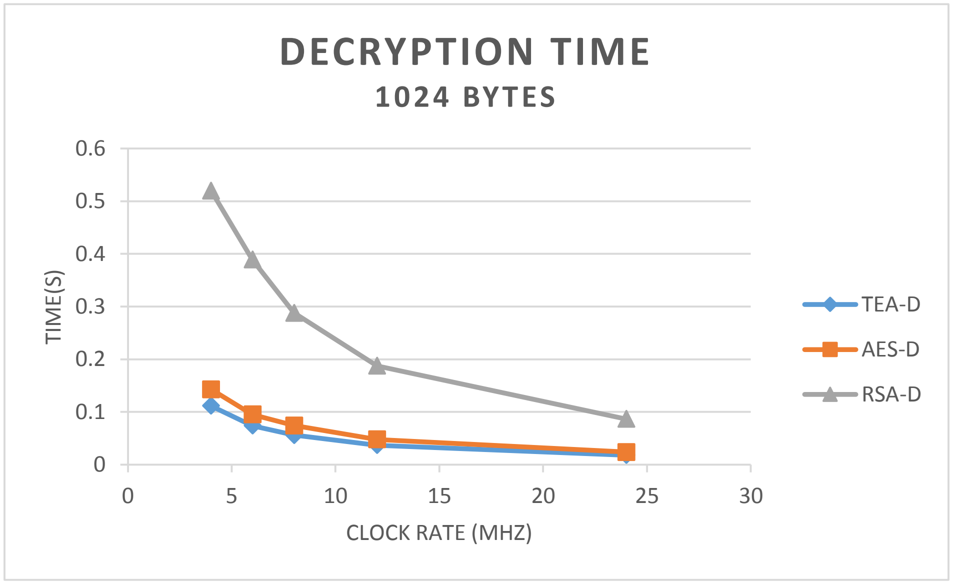

Figure 9. Comparison of Decryption Times

For small data blocks, less than 1024 bytes, the time required can be considered similar. As the data size increases, the time required to implement RSA becomes much larger. Figure 10 shows the encryption times for each algorithm, for plain text block sizes 1024 to 8192 bytes. The trends show a highly predictable linear increase for all the algorithms profiled. This trend makes it easy to predict encryption times and extends to predicting decryption times based on previous analysis of the algorithms' encryption – decryption delta factor. The data also shows that, as data block size increases AES and TEA provide a much more efficient solution for encryption and decryption than RSA. For large block sizes, 8192 bytes and greater, time becomes the largest factor to consider, possibly outweighing the security benefits.

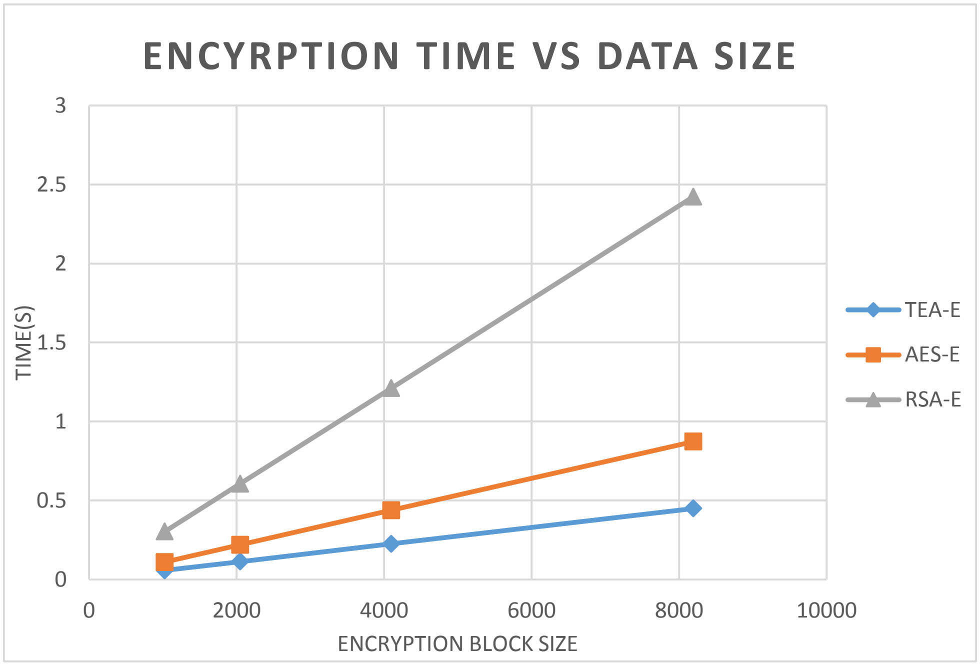

Figure 10. Encryption Time vs. Block Size

To enhance the security of TEA or AES, RSA can be used to transmit private encryption keys. Following a secure key exchange the faster symmetric algorithms can be used for transferring large data sets efficiently. In real world scenarios, the increased encryption time of RSA might not justify the security level increase over AES. One might be better served to increase the key size of AES to achieve the same level of security.

Power Consumption

Power analysis of each of the three algorithms demonstrates an advantage to running at higher clock speeds for battery operated devices. As illustrated in Figure 11, power consumption is higher, increased speed reduces encryption/decryption time, resulting in overall less energy used. The use of processor sleep and low power states can extend and realize these gains. Low power and sleep states should be used any time interrupt or time-driven operation is feasible. The power consumption and energy statistics are for the encryption routines only, and are only measured for the duration of encryption. Without using low power or sleep states, faster clock speeds could result in increased energy consumption if the processor runs continuously. For these test, low power states of the PSOC were used immediately before and immediately after the encryption routine.

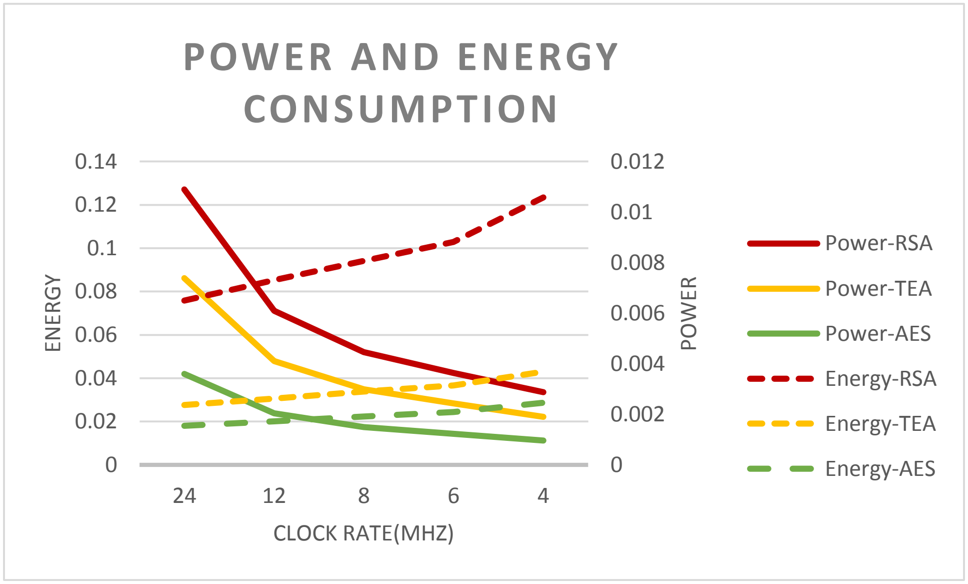

Figure 11. Power vs Energy Comparison

Memory Requirements

In Table 6, the RAM/ROM requirements for each algorithm are listed. RSA is the most efficient algorithm for memory usage, followed by TEA, and then AES. The small code size of RSA makes it attractive as an add-on algorithm for secure key exchange of private keys. AES is by far the largest algorithm, due to the lookup tables required. This cost must be weighed against its high speed and high security. TEA and RSA also do not require large stack allocations during operation because they do not require any key expansions. While AES modules could be side-loaded from flash memory when needed, applications needing AES should be aware of the large amount of temporary space needed for the key expansion. It should be noted that a combined encrypt and decrypt routine for AES was used in this work to allow code sharing and reduce memory usage.

	Encrypt Routine		**Decrypt Routine**	
	ROM	Stack	ROM	Stack
TEA	172	29	178	29
AES	856	244	856	244
RSA	30	16	26	16

Table 6. Memory Requirements in bytes for TEA, AES, and RSA algorithms

Conclusions

This section has detailed the costs, performance characteristics and operational steps of three common encryption algorithms used in embedded systems today. This analysis was performed as a demonstration of the types of analyses that designers should perform when researching security tools to be implemented for embedded designs. Following a secure design cycle that includes both modeling of attacks and characterization of solutions, such as the encryption algorithms presents above, provides high assurance that the chosen security solution effectively protects against the modeled attack. Careful study of algorithm function and operation ensures that no new weaknesses are introduced by incorrect implementation. Performing modeling and solution characterization in an iterative manner can enhance the confidence in the security solution developed.

Chapter 5

Application Scenario for Embedded Encryption Algorithms

Design

A simple application scenario was developed to provide context for the performance characteristics and research discussed in Chapter 4. The design represents a simulated IoT network of nodes. A high level diagram of the design is shown in Figure 12. The design represents a sensor network in which a collection node receives measurements from the environment and transmits these measurements, securely, to another node for processing. Two PSOC Developer boards were used for the nodes. The first node used a simple electret microphone to capture tones. This node is called the encryption unit. The second developer board was used to replay the tones on a piezo speaker, the decryption unit. The nodes communicated via a simulated wireless link. The link simulates a wireless link such as Bluetooth, Wifi, or RF. The link was implemented as an I2C interface for simplicity and debugging.

Test

A tone was played into the capture microphone, this tone was converted by an analog to digital converter into a data stream for encryption. The stream was encrypted, sent across the I2C bus, and then decrypted in decryption unit. From the decrypted stream the tone's frequency was detected. The appropriate tone was then output to the piezo speaker using PWM. The test was performed for multiple tones, using each encryption algorithm, TEA, AES, and RSA. The buffer size for each algorithm was different in order to accommodate the block size needed for each algorithm. The buffer sizes were 8 bytes, 16 bytes, and 8 bytes, respectively. This buffer represents the initial amount of data need to perform a single run of the encryption or decryption routine. The buffer size is at minimum equal the block size each algorithm.

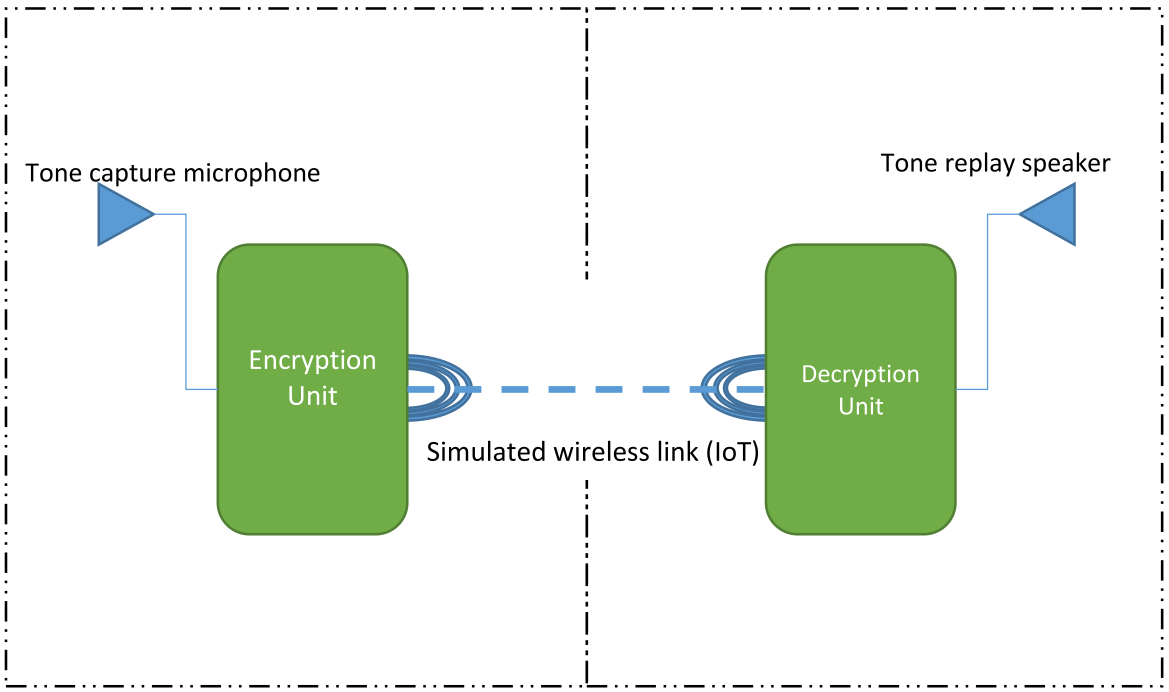

Figure 12. Application scenario setup

<u>Performance</u>

The test results are displayed in Table 7. The achieved data transfer speeds were consistent with the data presented in Chapter 4. RSA was the slowest algorithm, achieving a transfer rate of 81.83kbps. AES was three times faster with a demonstrated transfer rate of 222.7kbps. Finally, TEA was projected to achieve a transfer rate of 440kbps, but the I2C bus on the PSOC boards was unable to reach this speed limiting the actual transfer rate to 375.5kbps. The results demonstrate the various time impacts the different algorithms would have on transmitting data from node to node. The data shows that AES suffers from high initial delay due to the algorithm's large buffer size. This would negatively affect the performance of AES in a multi-hop type environment. Using RSA would result in significant delays if data was transmitted, verified, then retransmitted over many hops due to a large transfer time. TEA was the best performing algorithm for this type of test, and had minimal buffering delay. Its simplicity, and speed allowed it to outperform the other algorithms while providing an acceptable level of security of such an application. For this application, each algorithm was configured for the currently accepted level of security that is not easily broken by attack capabilities. These configurations are TEA-128, AES-128, and RSA-1024. If higher levels of security are desired it is up to the designer to evaluate the performance trade-offs of using larger key sizes.

Algorithm	**Transfer Rate**	**Time to Transfer 1MB**	**Initial Buffering Delay**
TEA	375.5kbps	22.34s	0.145us
AES	222.7kbps	35.9s	1.14ms
RSA	81.83kbps	102.5s	0.788us

Table 7. Transfer Rates for Application Scenario

Chapter 6

Final Conclusions

This thesis has provided designers with a guide for evaluating and characterizing data encryption as tools to protect against embedded attacks. Encryption is a valuable tool for protecting against spoofing, tampering, repudiation, information loss, DOS attacks, and privilege escalation. Encryption is the first line of defense in protecting our nation’s embedded infrastructure from being attacked from within by the systems that control its operation. The three algorithms profiled in this thesis, TEA, AES, and RSA, all serve unique purposes in the encryption field. TEA excels in its speed and simplicity and allows designers to quickly implement encryption in the most restrained resource environments. AES provides military-level security to designers, but proper implementation requires good research of algorithm operation and implementation is quite complex. RSA, despite a complex key generation process, provides the highest level of security, requiring no private key exchange to take place. RSA’s slow speed prevents it from being used for time-sensitive applications, but its robust security makes RSA ideal for handshake transactions before data is transmitted using a technique such as AES. The manner in which these characterization and modeling structures have been presented is intended to be a framework for any security solution. The framework is a foundation, discussed with the intention that it can be adapted to fit any security technique in the embedded field. The three pillars of a secure embedded system, (1) threat analysis and modeling, (2) secure development lifecycle and (3) education, provide designers with a path to understand the threats to which embedded systems are exposed. The model allows these threats to be characterized in manner such that clearly defined security solutions can be developed to mediate attack success.

Research in embedded security must evolve as the threats against embedded systems change. Research in the areas of infrastructure security and cyber warfare is increasing and becoming a priority for many groups both public and private. The nation's infrastructure, the largest connected embedded system ever seen, is a prime target for attacks. More research can be done to understand how these systems are vulnerable to attack and what steps need to be taken to more effectively develop secure embedded systems protected against exploitation.

References

[1] "Foreign Policy Cyber Security Executive Order 13636." *US Government,* 12 Feb. 2013.

[2] "The Real Story of Stuxnet." Kushner, David. IEEE Spectrum. 26 Feb. 2013

[3] "Lageberich 2014". Federal Office of Information Security, Germany. 2014. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Lageberichte/Lagebericht2014.pdf?__blob=publicationFile

[4] "New Directions in Cryptography." Diffie, W. and Hellman, M. *IEEE Transactions Information Theory* 22, 644-654, 1976.

[5] "TEA, a Tiny Encryption Algorithm." David J. Wheeler. Roger M. Needham. Cambridge University. England. 1994.

[6] "Related-Key Cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA." *J. Kelsey, B. Schneier, and D. Wagner ICICS '97 Proceedings*, Springer-Verlag, November 1997, pp. 233-246.

[7] "A cryptanalysis of the tiny encryption algorithm," V. R. Andem, Master's thesis, The University of Alabama, 2003.

[8] Federal Information Processing Standards Publication 197. Advanced Encryption Standard (AES). United States National Institute of Standards and Technology (NIKST). November 26, 2001.

[9] "The design of Rijndael: AES-the advanced encryption standard." Daemen, Joan; Rijmen, Vincent; 20113. Springer Science & Business Media.

[10] “A method for obtaining digital signatures and public-key cryptosystems.”R. L. Rivest, A. Shamir, and L. Adleman. 1978. *Commun. ACM* 21, 2 (February 1978), 120-126.

[11] “Factorization of a 768-bit RSA modulus.” Kleinjung, Thorsten et al. 2010.