Control System Data Integrity using a Variable-round Message Authentication Code with an Elliptic Curve Key Exchange Protocol

By

Kenneth Alan Fischer

Control System Data Integrity using a Variable-round Message Authentication Code with an Elliptic Curve Key Exchange Protocol

By

Kenneth Alan Fischer

Dissertation
Submitted to Department of Electrical and Computer Engineering
College of Engineering
Villanova University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

In

Computer Engineering

November, 2017

Villanova, Pennsylvania

Control System Data Integrity using a Variable-round Message Authentication Code with an Elliptic Curve Key Exchange Protocol

By

Kenneth Alan Fischer

Approved: ______________________________

Dr. Richard Perry
Associate Professor, Department of Electrical and Computer Engineering
Primary Advisor

Approved: ______________________________

Dr. Bijan Mobasseri
Chair, Department of Electrical and Computer Engineering

Approved: ______________________________

Dr. Gary A. Gabriele
Drosdick Endowed Dean of Engineering, College of Engineering

A copy of this dissertation is available for research purposes
At Falvey Memorial Library.

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at Villanova University, and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Associate Dean for Graduate Studies and Research of the College of Engineering when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

ACKNOWLEDGEMENTS

This dissertation is the result of my Ph.D studies at Villanova University, which took place over a period of over 6 years while I worked full time at the Naval Surface Warfare Center, Philadelphia Division (NSWCPD). First, I would like to thank my advisor, Dr. Richard Perry and the members of my PhD committee for sharing their time, experience, and wisdom during my research. I also thank my supervisors, subordinates, and colleagues at NSWCPD, particularly within the Cybersecure Machinery Control Systems and Networks Department, who took the time to listen to my ideas and share with me their own insights and experiences in the field of Control System, Network, and Cybersecurity Engineering. Particular thanks go out to Michael Iacovelli, Jeff Cohen, and Paul Gucciardi for their mentorship, ideas, challenges, and assistance in testing the implementation. I also want to thank the wonderful engineers at Rockwell Automation, particularly Nick Yiantsos, for their assistance and loans of equipment and software.

I worked on this dissertation over a period of several years, often under severe stress from working full time and trying to raise a family. I want to thank my parents, Alan and Debi Marteney, my mentors Dr. Gary and Mindy Salkind, and again Dr. Perry for the assistance and encouragement during some of the more “trying” moments. There were a few times I definitely was about to crack under the pressure! I also want to thank Dr. Anderson and Dr. Mastro in particular for some of their own “war stories” that they shared. The humor that helped ease some of my own internal frustrations when I struggled to figure something out.

Lastly and most importantly, I would like to thank my wife, Ana Fischer, for all the extra work she did in taking care of me and our four children while I pursued this work.

There were many long nights where she would sit with me as I would stare at my white board trying to solve some issue, or at a computer screen wondering why something wasn’t working properly. Throughout this work she joked that she should be getting an honorary degree for having to listen to me talk thru things out loud over the years. I’ve lost count of the number of times she tried to stop the children from running around the house causing all sorts of mayhem so I could focus. She usually failed, but with small children you cannot expect much.

DEDICATION

I dedicate this dissertation to my children,

Whose insatiable curiosity in the world

Inspires me to learn daily

And to the men and women of our armed forces

Who deserve our best

As they defend us around the world

And finally to Congregation Beth Yeshua

For helping me to remember the things that truly matter in life

TABLE OF CONTENTS

Section Page

STATEMENT BY AUTHOR 3

ACKNOWLEDGEMENTS 4

DEDICATION 6

TABLE OF CONTENTS 7

TABLE OF FIGURES 13

ABSTRACT 16

CHAPTER 1: INTRODUCTION 17

1.1 Overview 17

1.2 Background 19

1.3 Fundamental Objectives 22

1.3.1 NIST SGiP 23

1.4 Traditional Solutions for Information / Corporate Systems 25

1.4.1 Symmetric-key Cryptography 26

1.4.2 Public-key Cryptography 27

1.5 Industry Solutions 34

CHAPTER 2: ELLIPTIC CURVE CRYPTOGRAPHY 36

2.1 Background 36

2.2 Mathematical Foundations 37

2.2.1 Finite Fields 37

2.2.2 Elliptic Curves 41

2.2.3 Projective Coordinates 45

2.2.4 Point Multiplication ... 47

2.3 Domain Parameters ... 48

2.3.1 Prime Field Elliptic Curves ... 49

2.3.2 Binary Field Elliptic Curves ... 49

2.3.3 Standardized Versus Random Curves ... 50

2.4 Known Attack Mechanisms against ECC ... 51

2.4.1 Naïve Method ... 51

2.4.2 Pohlig-Hellman Attack ... 52

2.4.3 Pollard's rho Attack ... 52

2.4.4 Index-Calculus Attacks ... 53

2.4.5 Isomorphism Attacks ... 53

2.5 Cryptographic Protocols Useful for Control Systems ... 53

2.5.1 Key Generation ... 54

2.5.2 Elliptic Curve Digital Signature Algorithm (ECDSA) ... 56

2.5.3 Supported Secure Hash Algorithms ... 58

2.6 Comparing RSA Signatures to ECDSA ... 59

2.7 Edward's Curves ... 62

2.7.1 Key Generation ... 63

2.7.2 Signature Generation ... 64

2.7.3 Signature Verification ... 65

2.7.4 Special Curves ... 66

CHAPTER 3: TECHNICAL CONSTRAINTS AND SOLUTIONS ... 68

3.1 Introduction ... 68

3.2 Random Number Generation 68

3.3 BigIntegers 70

3.4 Processing Speed 74

3.5 Multicast 79

3.6 Lack of Proper Time Synchronization 84

3.7 Message Structures and Send Rates 85

3.8 Proof of Concept Implementation Overview 86

CHAPTER 4: VMAC 89

4.1 Introduction 89

4.2 Cryptographic Details 89

4.3 Attack Vectors 93

4.3.1 Length Extension Attacks 94

4.3.2 Collision Attacks 95

4.3.3 Key Reversal Attacks 96

4.4 Implementation Details 100

4.4.1 Outgoing Messages 102

4.4.2 Incoming Messages 104

CHAPTER 5: COMPLEX MATH OPERATIONS 108

5.1 Introduction 108

5.2 Custom Data Types 109

5.2.1 C DataType 110

5.2.2 D Data Type 110

5.2.3 E Data Type 110

5.2.4 POINT Data Type ... 111

5.3 Sequencers ... 111

5.4 Temp Data ... 113

5.5 AOIs ... 113

5.5.1 B256_MODPOW ... 114

5.5.2 B256_MODINV ... 115

5.5.3 B256_MODL ... 116

5.5.4 POINT_MUL ... 116

5.5.5 POINT_ENC ... 118

5.5.6 POINT_DEC ... 118

5.5.7 SHA512_SEQ ... 119

5.5.8 Ed25519_SIGN ... 120

5.5.9 Ed25519_VERIFY ... 122

5.5.10 PRNG ... 123

CHAPTER 6: KEY EXCHANGE PROTOCOL ... 126

6.1 Introduction ... 126

6.2 KEP Message Structure ... 131

6.3 Listener ... 133

6.4 Processor ... 135

6.4.1 State 0: Power On ... 137

6.4.2 State 1: Server Check ... 139

6.4.3 State 2: Ks’ Req Check ... 141

6.4.4 State 3: EdDH Keys ... 142

6.4.5 State 4: Sign message ... 144

6.4.6 State 5: Check for new message ... 146

6.4.7 State 6: Gen Kp, Encrypt Ke ... 148

6.4.8 State 7: Create new Ks’ ... 149

6.4.9 State 8: Set Ks = Ks’ ... 150

6.4.10 State 9: Gen Kp, Decrypt Ke ... 151

6.5 Typical Operation ... 152

CHAPTER 7: RESULTS AND FUTURE WORK ... 158

APPENDIX A. VMAC ANALYSIS ... 167

A.1 Testing SHA-256 ... 167

A.2 Cracking 16-Rounds ... 170

APPENDIX B. GRAPHICAL VMAC IMPLEMENTATION ... 174

B.1 Controller Organizer ... 174

B.2 VO_NODE1 ... 175

B.2 VI_NODE2 ... 176

APPENDIX C. SOURCE CODE ... 179

C.1 Custom Data Types ... 179

C.2 Add-On Instructions ... 179

C.3 VMAC and IO_Mapping ... 179

C.4 KEP Program ... 179

APPENDIX D. SEQUENCER CHARTS ... 180

D.1 Ed25519_SIGN ... 180

D.2 Ed25519_VERIFY ... 181

D.3 B256_MODINV 182

D.4 B256_MODL 183

D.5 B256_MODPOW 184

D.6 POINT_DEC 185

D.7 POINT_ENC 186

D.8 POINT_MUL 187

D.9 PRNG 188

D.10 SHA512_SEQ 188

APPENDIX E. KEY SCRIPTS 189

E.1 keygen.py 189

TABLE OF FIGURES

Figure 1: Binary Finite Field Reduction Polynomials 40

Figure 2: Sample Elliptic Curves [10] 41

Figure 3: Geometric Representation of Point Addition and Point Doubling [10] 42

Figure 4: Group Law for E($\mathbb{F}_p$): $y^2=x^3+ax+b$, char(K) $\neq 2,3$ [10] 44

Figure 5: Group Law for non-supersingular E($\mathbb{F}_{2^m}$): $y^2+xy=x^3+ax^2+b$ [10] 44

Figure 6: Group Law for supersingular E($\mathbb{F}_{2^m}$): $y^2+cy=x^3+ax+b$ [10] 45

Figure 7: Operation Counts on $y^2 = x^3 - 3x+b$ [10] 47

Figure 8: ECC vs RSA Comparable Key Sizes (in bits) [29] 60

Figure 9: ECC vs RSA Key Generation [29] 60

Figure 10: ECC vs RSA Signature Generation [29] 60

Figure 11: ECC vs RSA Signature Verification [29] 60

Figure 12: EdDSA Key Generation 64

Figure 13: EdDSA Signature Generation 65

Figure 14: EdDSA Signature Verification 66

Figure 15: TRNG 9815 Device 69

Figure 16: VMAC Add-On Instruction 91

Figure 17: VMAC Timing Results (in ms) 93

Figure 18: SHA256 Reduced Round Test Results 96

Figure 19: SHA Intermediary Hash Values 97

Figure 20: B256_MODPOW Add-On Instruction 114

Figure 21: B256_MODINV Add-On Instruction 115

Figure 22: B256_MODL Add-On Instruction 116

Figure 23: POINT_MUL Add-On Instruction... 117

Figure 24: POINT_ENC Add-On Instruction... 118

Figure 25: POINT_DEC Add-On Instruction... 119

Figure 26: SHA512_SEQ Add-On Instruction.. 119

Figure 27: Ed25510_SIGN Add-On Instruction.. 121

Figure 28: Ed25519_VERIFY Add-On Instruction... 123

Figure 29: PRNG Add-On Instruction... 124

Figure 30: KEP Message Flags.. 133

Figure 31: Processor State Machine Diagram 136

Figure 32: Processor State 0 Sequence Chart 138

Figure 33: Processor State 1 Sequence Chart 139

Figure 34: Processor State 2 Sequence Chart 141

Figure 35: Processor State 3 Sequence Chart 143

Figure 36: Processor State 4 Sequence Chart 145

Figure 37: Processor State 5 Sequence Chart 147

Figure 38: Processor State 6 Sequence Chart 149

Figure 39: Processor State 7 Sequence Chart 150

Figure 40: Processor State 8 Sequence Chart 151

Figure 41: Processor State 9 Sequence Chart 152

Figure 42: LEFT: A generating Ks’ and Ks, RIGHT: B thru D requesting Ks’..... 153

Figure 43: LEFT: A in Idle, RIGHT: B thru D in Idle but Exchange in Progress.. 154

Figure 44: LEFT: A encrypting Ks’, RIGHT: B thru D decrypting Ks’ 155

Figure 45: Client Has Ks and Decrypting new Ks’ 156

Figure 46: RIGHT: Client updating Ks = Ks’ 156

Figure 47: Client or Server in Idle Cycle with Ks = Ks’ 157

ABSTRACT

The challenge of securing industrial control systems is significant, and the need to provide authentication and verification services for control system commands and data has been well established. There is currently a lack of options for the control systems engineer, and the options that do exist mainly involve purchasing additional hardware. Additionally, these solutions disrupt the use of some real time communications products such as the Profinet/iMap combination used on Siemens Programmable Logic Controllers (PLCs).

There are a number of challenges to implementing control system data authentication and verification in a PLC, including performing mathematical operations on BigIntegers; generating and distributing keys; generating cryptographically secure hash values; implementing random number generation; and ensuring that the operations can be performed without impacting normal operation / scan times. In this work a generic solution to these problems is presented along with a proof-of-concept implementation written for Rockwell ControlLogix PLCs and tested using 1756-L83 controllers.

The solution is based on two new algorithms: Variable-round Message Authentication Code (VMAC) and the Key Exchange Protocol (KEP). VMAC is based on SHA256 and uses a symmetric key to generate message codes for data authentication and verification. KEP is a protocol that allows PLCs to securely generate and distribute the symmetric key used in VMAC. KEP is capable of being configured in a multiple tree configurations to increase the efficiency of key distribution, but also provides for redundancy in case the root node is taken offline. KEP was shown to have an average scan time impact of 10ms with a maximum impact of 20ms during a key exchange.

CHAPTER 1: INTRODUCTION

1.1 Overview

This document will present the challenges facing industrial control systems today, particularly in protecting against Man-in-the-Middle (MITM) attacks that can severely compromise a system. Control systems are used for a wide variety of electrical and mechanical systems, such as gas turbine generators, electrical distribution, producing food and drug products, and cooling for critical machinery such as a nuclear reactor. Therefore a compromise of the control system can have immediate impacts including damage to machinery, damage to critical infrastructure, and injury/death to operators of equipment or individuals who are dependent on the continual operation of the equipment. This document will present a new cryptographic system for providing control system data authentication and verification.

At the heart of this system are two new cryptographic algorithms developed by the author which are designed to work within the constraints of a control system, such as the inability to perform BigInteger calculations, the inability to generate random numbers, and the need for 24x7 operation over a period of years. The algorithms are:

1. Variable-round Message Authentication Code (VMAC)
2. A control system Key Exchange Protocol (KEP)

Note that the VMAC algorithm proposed here is not to be confused with the VMAC algorithm proposed by Ted Krovetz and Wei Dai in April 2007 [38], which shares a similar name but is based on an entirely different construction to meet a different set of requirements.

The fundamental mathematical theory and security is based on existing cryptographic algorithms such as the Secure Hash Algorithms (SHA), the Hash Message Authentication Code (HMAC) algorithms, the Elliptic Curve Diffie-Hellman Key Exchange algorithm, and a unique variant of an Elliptic Curve Cryptography (ECC) algorithm known as the Edward's Curve Digital Signature Algorithm (EdDSA). Each of these algorithms in of themselves is insufficient for use in an industrial control system for a range of reasons that will be described in this document.

Chapter 1 of this document provides introductory and background information for this work. Chapter 2 describes the mathematical theories and benefits of Elliptic Curve Cryptography which are necessary for understanding KEP, ending with a description of Edward's curves and the unique benefits they provide in addressing the challenges described in Chapter 3. Chapter 3 describes the limitations of industrial control systems in designing a data authentication and verification scheme and provides an overview of the proposed solution. This section also provides a high level overview of the solutions to these limitations that were developed as part of this work and defined in detail in the later sections. It concludes by describing the "proof of concept" implementation that was developed as part of this work. Chapter 4 describes the VMAC algorithm and provides details of the VMAC portion of the "proof of concept" implementation. Chapter 5 expands on the BigInteger and processing speed limitations presented in Chapter 3 by defining the various instructions that were developed as part of this work to perform complex operations such as modular arithmetic and point multiplication. Ultimately the concepts presented in Chapters 4 and 5 form the backbone required to understand the KEP algorithm presented in Chapter 6, which dynamically creates the key used by the

VMAC algorithm described in Chapter 4. Chapter 7 concludes by presenting information on testing results and describing areas of future work.

1.2 Background

Increasing demands in all sectors of an industrial society have led to an ever increasing need for more sophisticated controls and monitoring equipment and software. Control systems, once consisting of simple transmitters and relays, have evolved into complex systems containing dozens of controllers communicating with each other, each containing tens of thousands of lines of code, for even the simplest processes. Complex Human-Machine Interface (HMI) mechanisms designed to give system owners and operators enhanced capabilities to remotely operate, maintain, and troubleshoot equipment are being developed and deployed. At the core of most modern control systems is the Programmable Logic Controller (PLC), a device whose power lies in the ability of a Control System Engineer to quickly and easily implement complex control schemes at minimal cost. As a result, PLCs (originally designed to replace relay panels) have become prevalent in virtually every industrial environment from pharmaceutical plants to electrical power distribution systems.

Automation using PLCs and other embedded devices has become more prevalent in recent years. Virtually every aspect of our nation’s critical infrastructure is controlled using these devices, most prevalently in the areas of electrical generation, oil and gas production, transportation systems, water and water treatment systems, food production systems, chemical production systems, and heating systems. Baggage handling systems, for example, are increasingly being controlled by PLCs to provide automated distribution at luggage at airports and other mass transit facilities. The new “Smart Grid”, as it commonly called, will be characterized by a two-way flow of electricity and information creating a widely distributed energy network. The control system required to support this energy network will be of an unheard of scale, the design of which will introduce significant challenges never before addressed.

The use of PLCs and embedded controllers has grown exponentially in Department of Defense systems, especially in the U.S. Navy. The USS Arleigh Burke-class destroyers (DDG 51) first commissioned in 1991 uses a VME based control system with approximately 4,000 I/O points. By comparison, the new USS Zumwalt (DDG 1000) class destroyers use a PLC based control system with approximately 37,000 I/O points and roughly 1/3rd the crew. In related efforts, the US Navy has been rapidly migrating to ship designs with propulsion, auxiliary, and weapons systems with significantly higher energy requirements than in the past. Both classes of ships also use a wide range of various embedded controllers produced by different Original Equipment Manufacturers (OEMs) that must be integrated together in order to fulfill the ship’s missions.

Implementing control systems on a large, highly integrated scale introduces significant challenges partly because control system networks were not designed with security being primarily in mind. Historically, control system networks were designed to be completely physically isolated from other networks and therefore securing those control system networks seemed unnecessary. Instead, control system networks were designed to have maximum throughput with minimal to nonexistent data loss. In recent years though control systems have gradually been getting connected to the Internet, mostly via corporate network systems, in order to meet business and maintenance requirements. In order to secure networks, IT administrators have been applying traditional security measures in order to prevent attackers from gaining access to the corporate networks thus protecting control system networks. The mentality was one of "security by obscurity", since it was thought that PLCs were only capable of running a specific type of software that was not susceptible to viruses and other types of malware.

This mentality was eventually shattered by the rise of Advanced Persistent Threats (APTs) and customized malware such as Stuxnet that was specifically designed to target PLCs and other industrial control devices.

Control systems do have some features providing security, mainly in the areas of protecting the running software on a PLC to prevent an attacker from changing the algorithms running on the PLC that provide the automation. This has improved significantly in the wake of Stuxnet. However control systems still remain extremely vulnerable to Man-in-the-Middle attacks where an attacker issues false commands to a system causing it to take actions that should normally not be taken. The most common is to shut down or start up machinery systems in such a way that will compromise the systems and lead to a range of consequences including property damage and injury/death to human beings. Attackers have also been studying the various protocols used by control systems to communicate internally, such as Profinet and Modbus, and found that they are relatively trivial to spoof. Control systems inherently have little to no data authentication and validation capabilities, typically placing all the restrictions in a Human-Machine Interface (HMI). However if an attacker were to go around the HMI and spoof a packet then all the control system security protections would be bypassed, allowing an attacker to easily issue false commands to the control system.

There are some lightweight cryptographic algorithms that exist such as Photon [34] which are specifically targeted towards more limited devices. However these lightweight algorithms are only useful for small data messages in the 5-100 byte range. Control systems typically send messages closer to 250-1500 bytes at an interval of every 50-250ms depending on the application. Therefore a new approach to solving control system message authentication is required that accounts for the limitations of the equipment and the size and frequency of the data messages, and is robust enough to meet failover and redundancy requirements of control systems. The approach must allow for multiple vendors of different equipment to be able to integrate their systems into a comprehensive approach that is both platform and protocol independent.

1.3 Fundamental Objectives

Within the field of cryptography, there are multiple solutions providing various degrees of secure communication. In order to be effectively used to establish secure communications these solutions have the following fundamental objectives:

- Confidentiality – ensuring that the data can only be read by those authorized to see it
- Integrity – ensuring that the data has not been modified by unauthorized means
- Availability – ensuring that the data can be read when necessary to meet performance requirements

- Data Origin Authentication – ensuring that the data supposedly sent by a source actually originated with that source
- Entity Authentication – ensuring that an entity participating in a data transfer is who it claims to be
- Non-repudiation – ensuring that a source of data is unable to later deny sending the data

Information / Corporate systems are concerned with meeting each of the above objectives. Control systems are also equally concerned with the above objectives, with confidentiality to a significantly lesser degree, but also have unique requirements not present in information systems. When an information system receives a piece of data through insecure means, it can disregard the information with reasonably low risk. Control systems, on the other hand, need to make critical decisions with the information at hand. If the data received is insecure, the control system is placed in a position of having to make critical decisions about the operation of real world machinery without knowing which decision to take. Unfortunately, the control system will be regularly in the position where it must take some critical action or shut down the equipment, with each scenario resulting in possible equipment damage and injury/death to personnel operating that equipment.

1.3.1 NIST SGiP

In response to the number of concerns related to the Smart Grid and Cyber Security, NIST established the Smart Grid Interoperability Panel (SGiP) Cyber Security Working Group which published NISTIR 7628 (2010) [1]. This document decomposed the various kinds of communications that would be prevalent in a full international Smart Grid system into a number of categories such as “Category 10 – Interface between Control Systems and Non-Control / Corporate Systems”. SGiP then identifies the unique security requirements for each of these categories, focusing on the three areas of confidentiality, integrity, and availability.

Most, but not all of the categories identified by SGiP are directly or indirectly applicable to control systems (some that have little to no bearing such as categories 13 through 18 are not shown here) operating in the Smart Grid and are shown in the list below:

- Category 1: Interface between control systems and equipment with high availability, and with compute and/or bandwidth constraints
- Category 2: Interface between control systems and equipment without high availability, but with compute and / or bandwidth constraints
- Category 3: Interface between control systems and equipment with high availability, without compute or bandwidth constraints
- Category 4: Interface between control systems and equipment without high availability, without compute or bandwidth constraints
- Category 5: Interface between control systems within the same organization
- Category 6: Interface between control systems in different organizations
- Category 10: Interface between control systems and non-control / corporate systems
- Category 12: Interface between sensor networks and control systems
- Category 19: Interface between operations decision support systems

- Category 20: Interface between engineering / maintenance systems and control equipment
- Category 21: Interface between control systems and their vendors for standard maintenance and service
- Category 22: Interface between security / network / system management consoles and all networks and systems

In reviewing the categories, it becomes obvious that all of them have significant overlap with next generation integrated power system efforts as well as industrial control systems in general. On looking through the requirements of these categories as identified by SGiP, it is seen that the primary concern in these categories is that of data integrity and authentication. Data encryption can be useful in some circumstances, but it is not as critical as the other two requirements.

1.4 Traditional Solutions for Information / Corporate Systems

While traditional solutions for Information / Corporate Systems will not be feasible for implementation in Control Systems due to the different requirements and architectures, it is important to establish an understanding of current solutions used in Information Systems. There are essentially two main categories of cryptographic solutions, symmetric-key cryptography and public-key cryptography.

1.4.1 Symmetric-key Cryptography

Symmetric-key Cryptography includes schemes such as the Data Encryption Standard (DES) (now obsolete), RC4, and the Advanced Encryption Standard (AES) to achieve confidentiality. They may also be used with a message authentication code (MAC) algorithm such as HMAC to achieve data integrity and data origin authentication. In a typical symmetric-key cryptography scheme two parties already share a secret key k that has been communicated to the parties by some other means (typically a physical secure channel such as a trusted courier, or by using a public-key cryptography scheme to negotiate a shared secret key). Party A wishing to transmit to B uses one of the previously mentioned schemes to compute a ciphertext $c = ENC_k(m)$ to be sent to B. B then receives the message and uses the same k (and knowing the same scheme used to encrypt m used by A) to recover the plaintext message $m = DEC_k(c)$.

If data integrity and data origin authentication are desired, then the same principles apply however instead of encrypting the message m into ciphertext c a tag t is first computed where $t =MAC_k(m)$ of the plaintext message using a MAC algorithm (of which there are many) and the key. The plaintext message and the tag are both transmitted, and the receiver can use the plaintext message to compute its own tag t'. If $t = t'$ then the receiver can accept the message as having originated from the source.

While symmetric-key cryptography can be very efficient, the key distribution and key management problems tend to render it ineffective for large scale systems communicating to multiple partners. In a network of N entities, each entity may have to maintain keying material with each of the other N-1 entitites. Some symmetric-key systems attempt to alleviate this problem by using an online trusted third party that distributes the keys as required, however for control systems this creates a single critical point of failure that will be unacceptable as control systems become more and more distributed and de-centralized. Additionally, while key distribution in symmetric-key cryptography may be possible through a physical courier on a ship it will not be practical for large scale systems such as the Smart Grid.

1.4.2 Public-key Cryptography

Public-key cryptography began in 1975 [35] to address the aforementioned limitations in symmetric-key cryptography. Unlike symmetric-key schemes, public-key schemes require the keying material that is exchanged to only be authentic, but not secret. Additionally, instead of each pair of entities sharing a secret key, each entity selects a single pair of keys *(e, d)* consisting of a *public key e* and a related *private key d*. The entity keeps the private key a secret from all other entities and shares the public key with all other entities. The keys are mathematically related but it is computationally infeasible to determine the private key solely from knowledge of the public key. Deriving the private key from the public key is equivalent to solving a computational problem that is believed to be intractable.

1.4.2.1 RSA

The most commonly used public-key cryptography scheme is RSA, named after its inventors Rivest, Shamir, and Adleman [3]. It was first proposed in 1977 shortly after the discovery of public-key cryptography. In RSA, the public key consists of a pair of integers *(n, e)* where *n* is the modulus. The modulus is a product of two randomly generated (and secret) primes p and q which are of the same bitlength. Algorithm 1 below shows how to generate an RSA key pair. RSA encryption and signature schemes use the fact that $m^{ed} = m \ (mod \ n)$. Algorithm 2 and Algorithm 3 show how basic RSA encryption and decryption work respectively. The hardness in breaking RSA is based on the integer factorization problem, i.e. determining the secret primes p and q from the public key for large values of bitlength l.

The RSA signature generation and signature verification algorithms are shown in Algorithm 4 and Algorithm 5. As in all signature schemes, the signer first generates a cryptographic hash H which acts in a similar manner as the tag in symmetric-key encryption. The signer then generates the signature and transmits the message m along with the signature s to a verifying party.

In order to increase the efficiency of RSA, smaller exponents can be selected. In practice, the most commonly chosen values are $e = 3$ and $e = 65537$ for encryption and signature verification [3]. Note that there is no known attack against using small public exponents as long as proper padding is used. Decryption and signature generation always use the exponent d (the private key) which is the same bitlength as n. Thus RSA encryption and signature verification with small values of e are significantly faster than RSA decryption and signature generation.

Algorithm 1. [3] – Generating RSA Key Pair

```
 INPUT:  bitlength $l$
OUTPUT: RSA public key $(n, e)$ and private key $d$
1.  Randomly select two primes $p$ and $q$ of the same bitlength $l / 2$
```

```
2. Compute $n = pq$ and $\Phi = (p-1)(q-1)$
3. Select an arbitrary integer $e$ with $1 < e < \Phi$ and $gcd(e, \Phi) = 1$
4. Compute the integer $d$ satisfying $1 < d < \Phi$ and $ed \equiv 1 \pmod{\Phi}$
5. Return (n, e, d)
```

Algorithm 2. [3] – RSA Encryption

```
INPUT: RSA public key $(n, e)$, plaintext $m \in [0, n-1]$
OUTPUT: Ciphertext $c$
1. Compute $c = m^e \bmod n$
2. Return ($c$ )
```

Algorithm 3. [3] – RSA Decryption

```
INPUT: RSA public key $(n, e)$, RSA private key $d$, ciphertext $c$
OUTPUT: Plaintext $m$
1. Compute $m = c^d \bmod n$
2. Return ($m$ )
```

Algorithm 4. [3] – RSA Signature Generation

```
INPUT: RSA public key $(n, e)$, RSA private key $d$, message $m$
OUTPUT: Signature $s$
1. Compute $h = H(m)$ where $H$ is a cryptographic hash function
2. Compute $s = h^d \bmod n$
3. Return ($s$ )
```

Algorithm 5. [3] – RSA Signature Verification

INPUT: RSA public key *(n, e)*, message *m*, signature *s*

OUTPUT: Acceptance or rejection of the signature

1. Compute $h = H(m)$ where H is the same cryptographic hash function used by the signing party
2. Compute $h' = s^e \bmod n$
3. If $h = h'$ then accept the signature, else reject

1.4.2.2 Digital Signature Algorithm

In 1976 Diffie and Hellman proposed developing a key agreement protocol based on the discrete logarithm problem (DLP) [2], which like the integer factorization problem used in RSA is computationally infeasible to solve. Discrete logarithms are group-theoretic analogues of ordinary logarithms. For example, an ordinary logarithm $log_a(b)$ is a solution of the equation $a^x = b$ for x. In a discrete logarithm, you have a group G which consists of a range of integer values from 0 to n-1. If a and b are elements in the group then a solution of x of the equation $a^x = b$ is called a discrete logarithm to the base a of b in the group G. In a discrete logarithm public-key cryptography system a key pair is associated with a set of domain parameters (p, q, g). Algorithm 6 shows how these domain parameters are generated, and Algorithm 7 shows how to generate corresponding key pairs.

In 1984 ElGamal described discrete logarithm public-key encryption and signature schemes, and since then many different variants have been proposed leading up to the establishment of the Digital Signature Algorithm (DSA) [2]. DSA was proposed in August 1991 by the U.S. National Institute of Standards and Technology (NIST) and was specified in a U.S. Government Federal Information Processing Standard (FIPS 186), adopted in 1993. A minor revision was issued in 1996 as FIPS 186-1, which was expanded further in 2000 as FIPS 186-2 and again in 2009 as FIPS 186-3 [4]. Algorithm 8 and Algorithm 9 shown below give the procedures respectively for DSA signature generation and verification.

Algorithm 6. [2] – Discrete Logarithm Domain Parameter Generation

```
 INPUT:  Parameters $l$ and $t$
OUTPUT: Discrete logarithm domain parameters $(p, q, g)$
1. Select a $t$-bit prime $q$ and an $l$-bit prime $p$ such that $q$ divides $p-1$
2. Select an element $g$ of order $q$
      a. Select arbitrary $h \epsilon [1, p-1]$ and compute $g = h^{(p-1)/q} \bmod p$
      b. If $g = 1$ then repeat 2.a.
3. Return (p, q, g)
```

Algorithm 7. [2] – Discrete Logarithm Key Pair Generation

```
 INPUT:  Discrete logarithm domain parameters $(p, q, g)$
OUTPUT: Public key $y$ and private key $x$
1. Select $x \epsilon_{R} [1, q-1]$
2. Compute $y = g^{x} \bmod p$
3. Return $(y, x)$
```

Algorithm 8. [2] – DSA Signature Generation

```
INPUT: Discrete logarithm domain parameters $(p, q, g)$, private key $x$, message $m$
OUTPUT: Signature $(r, s)$
1. Select $k \in_R [1, q-1]$
2. Compute $T = g^k \bmod p$
3. Compute $r = T \bmod q$, if $r = 0$ then go to step 1
4. Compute $h = H(m)$, where $H$ is a cryptographic hash function
5. Compute $s = k^{-1}(h+xr) \bmod q$, if $s = 0$ then go to step 1
6. Return $(r, s)$
```

Algorithm 9. [2] – DSA Signature Verification

```
INPUT: Discrete logarithm domain parameters $(p, q, g)$, public key $y$, message $m$, signature $(r, s)$
OUTPUT: Acceptance or rejection of the signature
1. Verify that $r$ and $s$ are integers in the interval $[1, q-1]$, if either verification fails then reject the signature
2. Compute $h = H(m)$, where $H$ is the same cryptographic hash function used by the signing party
3. Compute $w = s^{-1} \bmod q$
4. Compute $u_1 = hw \bmod q$ and $u_2 = rw \bmod q$
5. Compute $T = g^{u1} y^{u2} \bmod p$
6. Compute $r' = T \bmod q$
```

7. If r' = r then accept the signature, else reject

1.4.2.3 Limitations Using Public-Key Cryptography

In cryptography, the security of an algorithm cannot exceed its key length (measured in bits) since any algorithm can be cracked by brute force. A key therefore should be sufficiently large enough such that a brute force attack is infeasible – i.e. it would take too long to execute. If there is some indicator that an attack may exist to feasibly break a key for a particular algorithm in an efficient manner for some bit length, then the size of the key is increased to provide additional security. The key size to security level ratio is not the same for all categories of algorithms.

As of 2003 [5] RSA Security claims that 1024-bit RSA keys are equivalent in strength to 80-bit symmetric keys, 2048-bit RSA keys to 112-bit symmetric keys and 3072-bit RSA keys to 128-bit symmetric keys. RSA claims that 2048-bit keys are sufficient until 2030. An RSA key length of 3072 bits should be used if security is required beyond 2030. NIST key management guidelines further suggest that 15360-bit RSA keys are equivalent in strength to 256-bit symmetric keys [5]. These key lengths, while implementable in Information / Corporate systems, are infeasible in Control Systems where processing power and data storage is limited. Therefore an alternative public-key algorithm is needed that provides the benefits of algorithms such as RSA and DSA without the excessive key lengths required by these algorithms.

1.5 Industry Solutions

In the last year PLC and embedded controller vendors have begun to offer a range of various proprietary solutions for data authentication and verification. These solutions typically rely on proprietary algorithms designed to work solely within the vendor's product sphere and do not integrate with 3rd party devices. Since the algorithms are proprietary it is not possible to evaluate them from a security perspective.

1.6 Summary

Control systems are a major component of daily life, and the safe operation of these systems is necessary, however they are radically different than traditional IT systems and require special considerations. For example, the use of the RSA algorithm in public key cryptography is common when purchasing products online, but control systems are unable to manage keys of this size and the processors are generally too slow to perform the necessary math in a reasonable time frame. Symmetric-key cryptography will also not work due to the challenges in managing key distribution. The use of Elliptic Curve Cryptography has offered a number of opportunities such as smaller key sizes that can be leveraged in this work and is described in the following section. The recent introduction of Edward's Curves offers further opportunities which are just starting to be leveraged in traditional IT systems, with RFC8032 being published in January of this year [27].

There are a number of challenges and constraints that need to be addressed when designing a system to provide control system data integrity. This work describes the challenges in Chapter 3 in detail and begins to outline the solutions to these challenges. These solutions are further refined in Chapters 4 thru 6 with the introduction of two new algorithms, VMAC and KEP, which provide a communication protocol independent solution for providing control system data integrity.

CHAPTER 2: ELLIPTIC CURVE CRYPTOGRAPHY

2.1 Background

Elliptic curve public key cryptosystems were first independently proposed by V.S. Miller (1985) [6] and by N. Koblitz (1987) [7]. They have only recently begun to be used in commercial systems, and adoption has been slow. This is primarily due to concerns about intellectual property, as a number of optimizations and special algorithms used to increase efficiency have been patented in recent years. Despite these concerns, elliptic curve cryptography (ECC) has grown resulting in its inclusion in standards by accredited standards organizations such as ANSI (American National Standards Institute) [8, 9], IEEE (Institute of Electrical and Electronics Engineers) [10], ISO (International Standards Organization [11, 12], and NIST (National Institute of Standards and Technology [13].

The most prominent group for the standardization and propagation of ECC technology is SECG (Standards for Efficient Cryptography Group) [14]. They have published numerous and detailed works on the subject, including documents on how to implement ECC and on recommended elliptic curve domain parameters [15, 16]. The SECG consists of a number of organizations including NIST and key industrial partners such as VISA, Fujitsu, and Certicom. Certicom, which is a wholly owned subsidiary of Research in Motion (RIM), is the main industrial leader in ECC, with over 350 patents and patents pending worldwide covering key aspects of the technology [17].

In order to promote the use of ECC technology, NIST has licensed 26 patents held by Certicom with the right to grant sublicenses for free to industrial vendors for developing products used for protecting national security information [1]. NIST has also identified a subset of key ECC technologies for use in Smart Grid and related applications, such as the Elliptic Curve Digital Signature Algorithm as part of its NSA Suite B collection of approved encryption, key exchange, digital signature, and hashing protocols. It is also worth noting that ECC implementation strategies based on the fundamental algorithms of ECC, which were published prior to filing dates of many patents can be found in the IETF Memo "Fundamental Elliptic Curve Cryptography Algorithms." [18]

2.2 Mathematical Foundations

This section presents an overview of the mathematical techniques and concepts required for an intermediary level of understanding of elliptic curve cryptography. This material is sufficient for engineering purposes to develop ECC systems using standardized existing mathematic implementations and standardized elliptic curve domain parameters. The works of Koblitz [7], Miller [6], Hankerson et al. [2], and the SECG [15] can be referred to for more advanced mathematical concepts that may be helpful should the need arise for development of new implementations or the use of random elliptic curve domain parameters.

2.2.1 Finite Fields

A finite field $\mathbb{F}_q{}^m$ consists of a finite set of objects called field elements together with the description of two operations – addition and multiplication – that can be performed on pairs of field elements. Subtraction and division within a finite field are defined in terms of an additive inverse and multiplicative inverse, respectively. In ECC there are two kinds of fields that are primarily used: prime finite fields $\mathbb{F}_p$ with $q=p$ and $m=1$, with q being prime; and binary fields $\mathbb{F}_{2^m}$ where $q=2$ for some m ≥ 1. A third type of field less commonly used is known is Optimal Extension Fields (OEF). The general idea in OEFs is to select values of q and m, along with a reduction polynomial to more closely match underlying hardware characteristics [2]. At this time there are no recommended implementations of ECC by SECG that utilize OEFs, and therefore they are only mentioned here for completeness.

Equations involving finite fields do not explicitly denote the *mod p* operation, but it is understood to be implicit.

2.2.1.1 Prime Finite Fields [15]

Elements in a prime finite field $\mathbb{F}_p$ should be represented by the set of integers:

$$\{0, 1, \ldots, p-1\}$$

Operations on prime finite fields are defined as follows:

- Addition: If $a, b \in F_p$, then $a + b = r$ in F_p, where $r \in [0, p-1]$ is the remainder when the integer $a + b$ is divided by p.
- Multiplication: If $a, b \in F_p$, then $ab = s$ in F_p where $s \in [0, p-1]$ is the remainder when the integer ab is divided by p.
- Additive inverse: If $a \in F_p$, then the additive inverse ($-a$) of a in F_p is the unique solution to the equation $a + x \equiv 0 \bmod p$.
- Multiplicative inverse: If $a \in F_p$, $a \neq 0$, then the multiplicative inverse a^{-1} of a in F_p is the unique solution to the equation $ax \equiv 1 \bmod p$.

In order to increase efficiency and to facilitate interoperability, prime finite fields using the NIST primes should be use. These finite fields have:

$[\log_2 p] \in \{192, 224, 256, 384, 521\}$

Except for 521, *p* is aligned with word size to increase efficiency in computation and communication. 521 is an anomaly that is often included to align with the U.S. government's recommended elliptic curve domain parameters.

2.2.1.2 Binary Finite Fields [15]

Elements of a binary finite field $\mathbb{F}_{2^m}$ should be represented by the set of binary polynomials of degree *m-1* or less:

$$\{a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \ldots + a_1x + a_0 : a_i \in \{0,1\}\}$$

and an irreducible polynomial *f(x)*. Operations on binary finite fields are defined as follows:

- Addition: If $a = a_{m-1}x^{m-1}+a_{m-2}x^{m-2}+\ldots+ a_0$, $b = b_{m-1}x^{m-1}+b_{m-2}x^{m-2}+\ldots+ b_0 \in \mathbb{F}_{2^m}$, then $a + b = r$ in $\mathbb{F}_{2^m}$ where $r = r_{m-1}x^{m-1}+r_{m-2}x^{m-2}+\ldots+ r_0$ with $r_i \equiv a_i +b_i \mod 2$
- Multiplication: If $a = a_{m-1}x^{m-1}+a_{m-2}x^{m-2}+\ldots+ a_0$, $b = b_{m-1}x^{m-1}+b_{m-2}x^{m-2}+\ldots+ b_0 \in \mathbb{F}_{2^m}$, then $ab = s$ in $\mathbb{F}_{2^m}$ where $s = s_{m-1}x^{m-1}+s_{m-2}x^{m-2}+\ldots+ s_0$ is the remainder when the polynomial *ab* is divided by *f(x)* with all coefficient arithmetic performed modulo 2.
- Additive inverse: If $a \in \mathbb{F}_{2^m}$, then the additive inverse (-*a*) of *a* in $\mathbb{F}_{2^m}$ is the unique solution to the equation $a + x \equiv 0$ in $\mathbb{F}_{2^m}$.

- Multiplicative inverse: If $a \epsilon \mathbb{F}_{2^m}$, $a \neq 0,$ then the multiplicative inverse a^{-1} of a in $\mathbb{F}_{2^m}$ is the unique solution to the equation $ax \equiv 1$ in $\mathbb{F}_{2^m}$.

In order to increase efficiency and interoperability, the characteristic binary finite fields used should have:

$$m \epsilon \{163, 233, 239, 283, 409, 571]$$

These fields were chosen in order to construct a suitable Koblitz curve whose order is 2 or 4 times a prime over $\mathbb{F}_{2^m}$. The field with $m = 239$ is an anomaly shown here because it has already been widely used in practice. The field with $m = 283$ is an anomaly that is often included to align with the U.S. government's recommended elliptic curve domain parameters.

Multiplication should be performed using one of the irreducible binary polynomials of degree m in the figure below. These polynomials enable efficient calculation of field operations, except for the polynomial with $m = 239$ which is an anomaly shown here because it has been widely deployed.

Field	Reduction Polynomial(s)
$\mathbb{F}_{2^{163}}$	$f(x) = x^{163}+x^7+x^6+x^3+1$
$\mathbb{F}_{2^{233}}$	$f(x) = x^{233}+x^{74}+1$
$\mathbb{F}_{2^{239}}$	$f(x) = x^{239}+x^{36}+1$ or $x^{239}+x^{158}+1$
$\mathbb{F}_{2^{283}}$	$f(x) = x^{283}+x^{12}+x^7+x^5+1$
$\mathbb{F}_{2^{409}}$	$f(x) = x^{409}+x^{87}+1$
$\mathbb{F}_{2^{571}}$	$f(x) = x^{571}+x^{10}+x^5+x^2+1$

Figure 1: Binary Finite Field Reduction Polynomials

2.2.2 Elliptic Curves

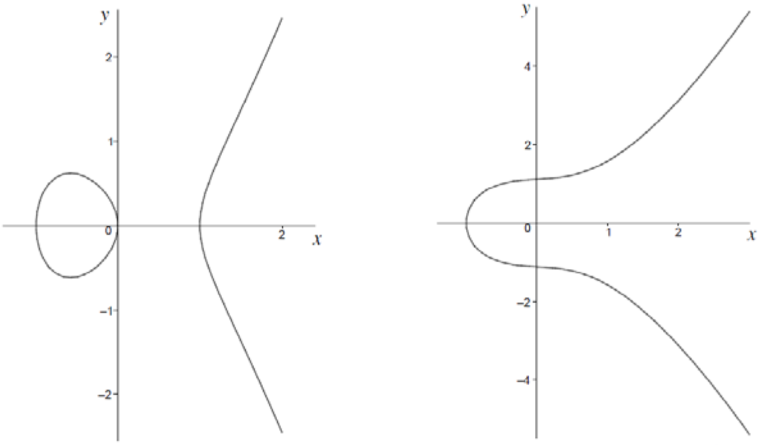
Elliptic curves are most commonly shown in the form of the simplified Weierstrass equation in the form of:

$$y^2 = x^3 + ax + b$$

where

$$4a^3 + 27b^2 \neq 0$$

This condition is critical to ensure that the elliptic curve is "smooth", i.e. that there are no points at which the curve has two or more distinct tangent lines. The curves shown in the figure below illustrate examples of elliptic curves satisfying this condition.



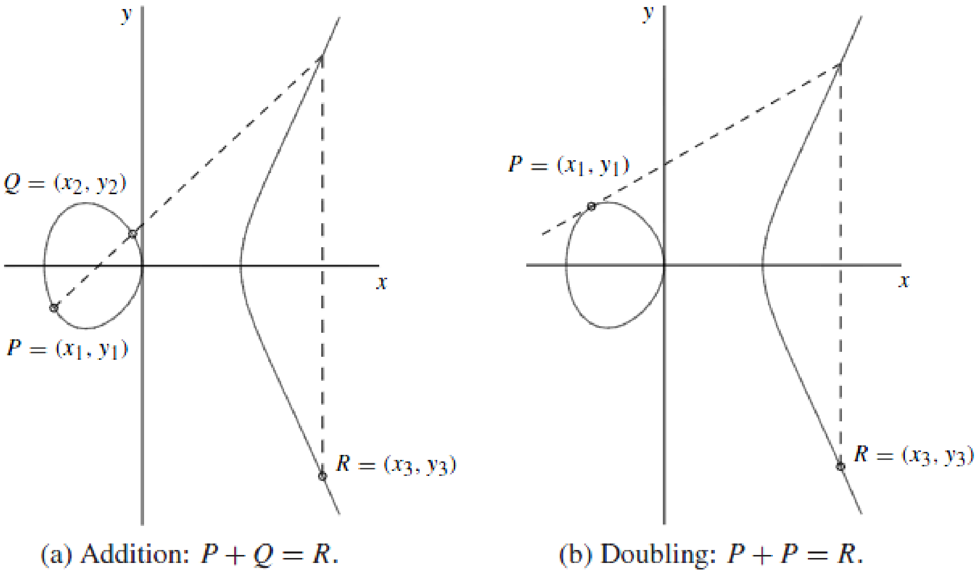
(a) $E_1 : y^2 = x^3 - x$

(b) $E_2 : y^2 = x^3 + \frac{1}{4}x + \frac{5}{4}$

Figure 2: Sample Elliptic Curves [2]

The security of ECC is based on the elliptic curve discrete logarithm problem (ECDLP), which arises when elliptic curves are used over finite fields. The ECDLP is [2]: given an elliptic curve E defined over a finite field $\mathbb{F}_q$, a point $P \epsilon E(\mathbb{F}_q)$ of order n, and a point $Q \epsilon <P>$, find the integer $l \epsilon [0, n\text{-}1]$ such that $Q = lP$, where $<P>$ is the subgroup generated by P. The integer l is called the discrete logarithm of Q to the base P, denoted $l = log_P Q$. The elliptic curve domain parameters for cryptographic schemes should be carefully chosen in order to resist all known attacks on the ECDLP. However, since the methods for computing solutions to the ECDLP are much less efficient then methods used for computing solutions to integer factorization (used in RSA) ECC can provide the same level of security as RSA with smaller key lengths, and ECC scales much better at higher levels of security than RSA.

When an elliptic curve E is defined over a field (call it K) there exist rules for adding two points in $E(K)$ to give a third point in $E(K)$. This operation is commonly known as point addition. Furthermore, there also exist rules for doubling a point as to obtain another point, an operation commonly known as point doubling. The figure below shows a geometric representation of both of these rules.



(a) Addition: $P + Q = R$.

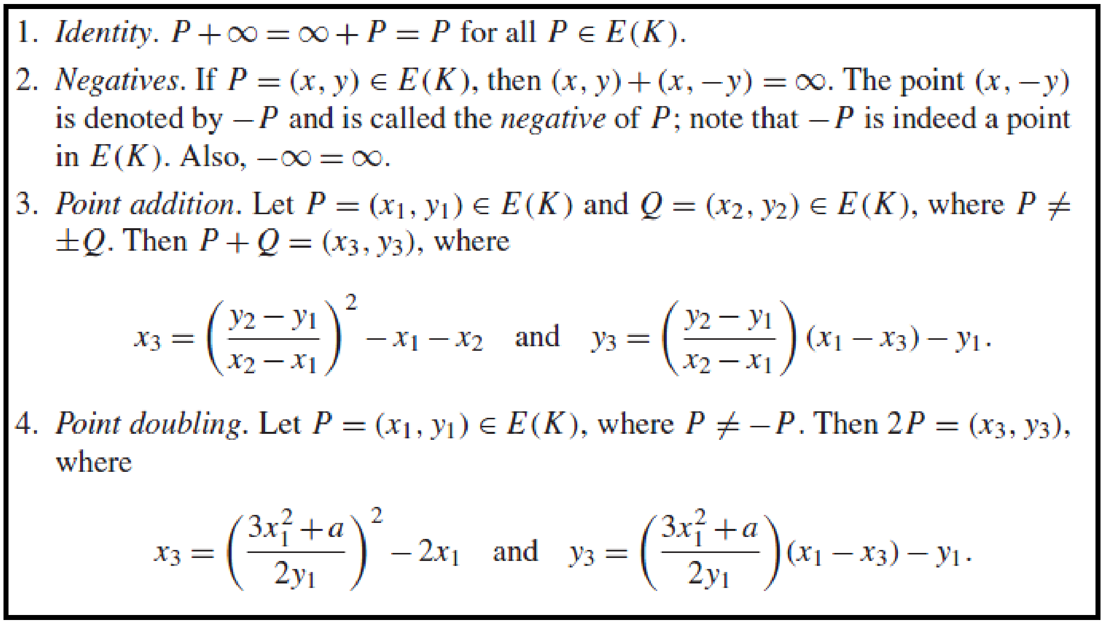
(b) Doubling: $P + P = R$.

Figure 3: Geometric Representation of Point Addition and Point Doubling [2]

Algebraic formulas for these operations can be derived from the geometric representation. The exact formulas themselves (the group law) will vary depending on whether you are using a simplified Weierstrass form or the complete form. They will also vary depending on the characteristic *q* of the underlying field [2]. We consider these cases:

- The characteristic of the underlying field *K* is not 2 or 3 (e.g. $K = \mathbb{F}_p$ where $p > 3$ is a prime)
- The curve *E* is non-supersingular of the form over $K = \mathbb{F}_{2^m}$
- The curve *E* is supersingular of the form over $K = \mathbb{F}_{2^m}$

The easiest group law to understand is for that of the simplified Weierstrass form for $char(K)\neq 2,3$, shown in Figure 4. Group laws for the simplified Weierstrass form for $char(K)=2$ are shown in Figure 5 and Figure 6 for non-supersingular and supersingular curves respectively.



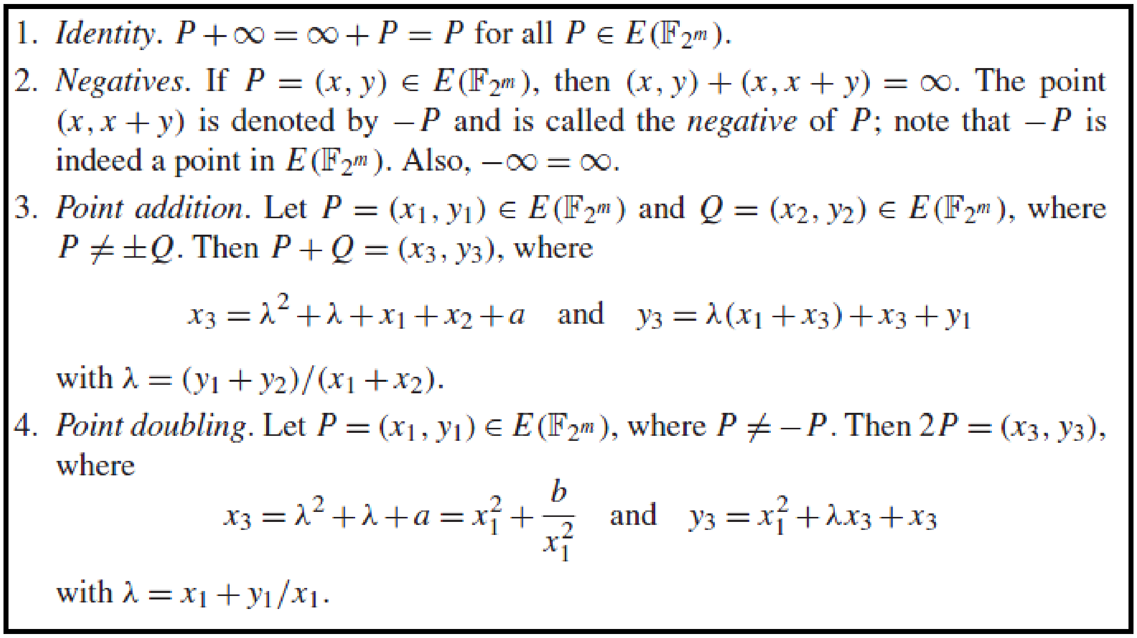
1. *Identity*. $P+\infty = \infty + P = P$ for all $P \in E(K)$.
2. *Negatives*. If $P = (x, y) \in E(K)$, then $(x, y) + (x, -y) = \infty$. The point $(x, -y)$ is denoted by $-P$ and is called the *negative* of P; note that $-P$ is indeed a point in $E(K)$. Also, $-\infty = \infty$.
3. *Point addition*. Let $P = (x_1, y_1) \in E(K)$ and $Q = (x_2, y_2) \in E(K)$, where $P \neq \pm Q$. Then $P + Q = (x_3, y_3)$, where

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \quad \text{and} \quad y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1.$$

4. *Point doubling*. Let $P = (x_1, y_1) \in E(K)$, where $P \neq -P$. Then $2P = (x_3, y_3)$, where

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \quad \text{and} \quad y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1.$$

Figure 4: Group Law for $E(\mathbb{F}_p)$*:* $y^2=x^3+ax+b$*, char(K)* $\neq 2,3$ *[2]*



1. *Identity*. $P+\infty = \infty + P = P$ for all $P \in E(\mathbb{F}_{2^m})$.
2. *Negatives*. If $P = (x, y) \in E(\mathbb{F}_{2^m})$, then $(x, y) + (x, x + y) = \infty$. The point $(x, x + y)$ is denoted by $-P$ and is called the *negative* of P; note that $-P$ is indeed a point in $E(\mathbb{F}_{2^m})$. Also, $-\infty = \infty$.
3. *Point addition*. Let $P = (x_1, y_1) \in E(\mathbb{F}_{2^m})$ and $Q = (x_2, y_2) \in E(\mathbb{F}_{2^m})$, where $P \neq \pm Q$. Then $P + Q = (x_3, y_3)$, where

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \quad \text{and} \quad y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$

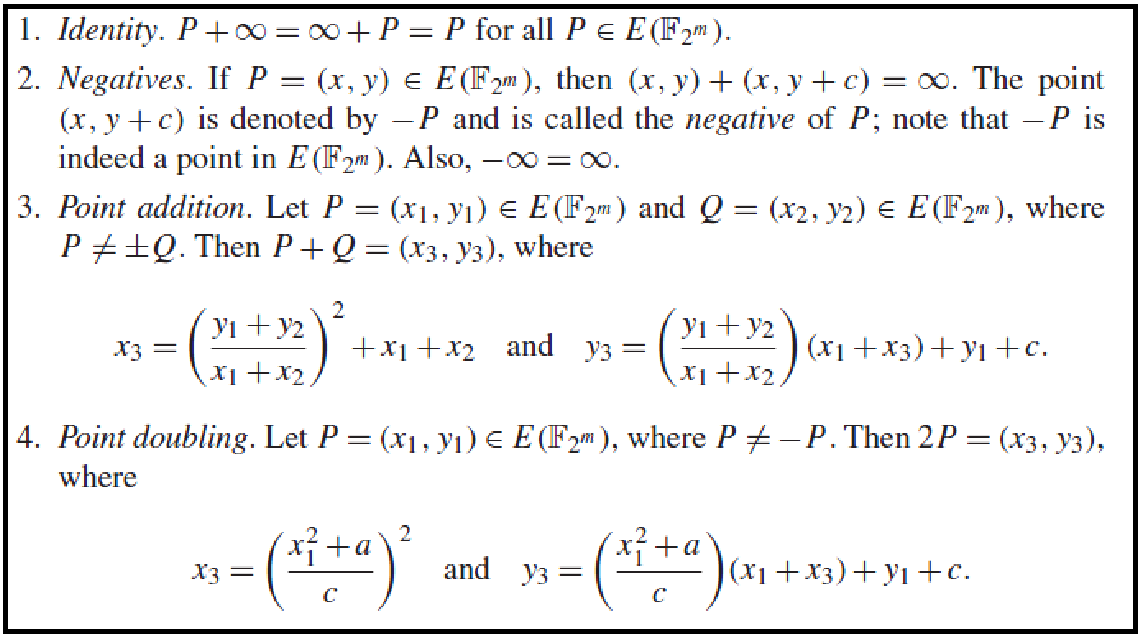
with $\lambda = (y_1 + y_2)/(x_1 + x_2)$.

4. *Point doubling*. Let $P = (x_1, y_1) \in E(\mathbb{F}_{2^m})$, where $P \neq -P$. Then $2P = (x_3, y_3)$, where

$$x_3 = \lambda^2 + \lambda + a = x_1^2 + \frac{b}{x_1^2} \quad \text{and} \quad y_3 = x_1^2 + \lambda x_3 + x_3$$

with $\lambda = x_1 + y_1/x_1$.

Figure 5: Group Law for non-supersingular $E(\mathbb{F}_{2^m})$*:* $y^2+xy=x^3+ax^2+b$ *[2]*



1. *Identity*. $P+\infty=\infty+P=P$ for all $P\in E(\mathbb{F}_{2^m})$.
2. *Negatives*. If $P=(x,y)\in E(\mathbb{F}_{2^m})$, then $(x,y)+(x,y+c)=\infty$. The point $(x,y+c)$ is denoted by $-P$ and is called the *negative* of P; note that $-P$ is indeed a point in $E(\mathbb{F}_{2^m})$. Also, $-\infty=\infty$.
3. *Point addition*. Let $P=(x_1,y_1)\in E(\mathbb{F}_{2^m})$ and $Q=(x_2,y_2)\in E(\mathbb{F}_{2^m})$, where $P\neq\pm Q$. Then $P+Q=(x_3,y_3)$, where

$$x_3=\left(\frac{y_1+y_2}{x_1+x_2}\right)^2+x_1+x_2 \quad\text{and}\quad y_3=\left(\frac{y_1+y_2}{x_1+x_2}\right)(x_1+x_3)+y_1+c.$$

4. *Point doubling*. Let $P=(x_1,y_1)\in E(\mathbb{F}_{2^m})$, where $P\neq -P$. Then $2P=(x_3,y_3)$, where

$$x_3=\left(\frac{x_1^2+a}{c}\right)^2 \quad\text{and}\quad y_3=\left(\frac{x_1^2+a}{c}\right)(x_1+x_3)+y_1+c.$$

Figure 6: Group Law for supersingular E($\mathbb{F}_{2^m}$*):* $y^2+cy=x^3+ax+b$ *[2]*

2.2.3 Projective Coordinates

The group laws shown above illustrate that the formulas for point addition and point doubling require field inversions and field multiplications. These are complex operations for the very large fields typically used in cryptographic applications. If inversion in a field *K* is significantly more expensive than multiplication (and it typically has a cost of roughly 80 field multiplications [2]), then the use of a technique known as projective coordinates may be advantageous to use.

Projective coordinates essentially work by defining an equivalence relationship between a field *K* and a set $K^3\backslash\{0,0,0\}$. The relationship is obtained by replacing *x* with X/Z^c and *y* with Y/Z^d, and clearing the denominators. We end up with a 1-1 relationship between the affine points that lie on *E* and the projective points on *E*. There are a number of different versions of projective coordinates, with varying values of *c* and *d*.

In the “standard projective coordinates” *c* and *d* are both set to one. Another form of projective coordinates known as “Jacobian coordinates” sets *c=2* and *d=3*. This changes the simplified Weierstrass equation from:

$$y^2 = x^3 + ax + b$$

to the projective form:

$$Y^2 = X^3 + aXZ^4 + bZ^6$$

The result of this change allows a new group law to be formed in which point doubling can be computed using six field squarings and four field multiplications [2]. The use of field inversions is now no longer required. Algorithms also exist to perform point multiplication between points in different coordinate systems, such as affine and Jacobian. Jacobian coordinates yield the fastest point doubling, while mixed Jacobian-affine coordinates yield the fastest point addition.

A third type of coordinate system is “Chudnovsky coordinates”. In Chudnovsky coordinates Jacobian coordinates $(X:Y:Z)$ are represented as $(X:Y:Z:Z^2:Z^3)$. There are some point multiplication algorithms that make use of the redundancy in Chudnovsky coordinates and use mixed Jacobian-Chudnovsky and mixed Chudnovsky-affine coordinates for point addition. The figure below gives some example operation counts for using projective coordinates in point addition. In the figure *A* represents affine coordinates, *P* represents standard projective coordinates, *J* represents Jacobian coordinates, and *C* represents Chudnovsky coordinates. The mathematical operations of field inversion, field multiplication, and field squaring are representated as *I*, *M*, and *S* respectively.

Doubling		General addition		Mixed coordinates	
2A → A	1 I, 2M, 2S	A + A → A	1 I, 2M, 1 S	J + A → J	8M, 3S
2P → P	7M, 3S	P + P → P	12 M, 2 S	J + C → J	11M, 3S
2J → J	4M, 4S	J + J → J	12 M, 4 S	C + A → C	8M, 3S
2C → C	5M, 4S	C + C → C	11 M, 3 S		

Figure 7: Operation Counts on $y^2 = x^3 - 3x+b$ *[2]*

2.2.4 Point Multiplication

In cryptographic applications point multiplication (the computation of *kP* where *P* is a point on the curve and *k* is an integer) dominates the execution time of ECC schemes. There are three cases where point multiplication occurs:

- *kP* where precomputation must be online
- *kP* for *P* known in advance and precomputation may be offline
- *kP* + *lQ* where only the precomputation for *P* may be done offline

The last two cases are motivated by the Elliptic Curve Digital Signature Algorithm (ECDSA), where signature generation requires a calculation *kP* where *P* is fixed, and signature verification requires a calculation *kP* + *lQ* where *P* is fixed and *Q* is known a priori.

There are a number of mathematical techniques that can be used in order to increase the efficiency of point multiplications. Some methods, such the “sliding-window methods”, require that extra memory be available. Additionally, if the point *P* is fixed and some storage is available, then the point multiplication *kP* can be accelerated by pre-computing some of the data dependent on *P* using a type of fixed-base windowing method such as that proposed by Brickell, Gordon, McCurley, and Wilson [2]. Shamir’s

Trick is yet another method used specifically to speed up the calculation of $kP + lQ$ by performing simultaneous multiple point multiplication [2].

2.3 Domain Parameters

As stated previously, the elliptic curve domain parameters for cryptographic schemes should be carefully chosen in order to resist all known attacks on the ECDLP. In general, for elliptic curves over a finite field $\mathbb{F}_{q^m}$, the following domain parameters are required to be specified:

$$D = (q, FR, S, a, b, P, n, h)$$

Where:

q – field order

FR – field representation

S – seed, used if the elliptic curve was generated randomly

a & *b* – coefficients in the field $\mathbb{F}_{q^m}$ that define the equation over the field

P – the base point $P=(x_p, y_p) \in \mathbb{F}_{q^m}$ that has prime order

n – the order of P

h – the cofactor h=#E($\mathbb{F}_{q^m}$) / n

This section describes the domain parameters needed to generate curves for the prime and binary finite fields used in ECC. We then go on to discuss the use of standardized special curves and the generation of new random curves, discussing the pros and cons of each.

2.3.1 Prime Field Elliptic Curves

For elliptic curve domain parameters over $\mathbb{F}_p$ the domain parameters are the sextuple:

$$D = (p, a, b, P, n, h)$$

They consist of an integer *p* specifying the finite field along with certain general domain parameters defined above. Elliptic curve domain parameters over $\mathbb{F}_p$ precisely specify an elliptic curve and a base point. This is necessary to define public-key cryptography schemes based on ECC [16]. If the elliptic curve domain parameters are verifiably random than they should be accompanied by the seed value *S* from which they are derived [16].

2.3.2 Binary Field Elliptic Curves

For elliptic curve domain parameters over $\mathbb{F}_{2^m}$ the domain parameters are the septuple:

$$D = (m, f(x), a, b, P, n, h)$$

They consist of an integer *m* specifying the finite field $\mathbb{F}_{2^m}$, an irreducible binary polynomial *f(x)* of degree *m* specifying the representation of $\mathbb{F}_{2^m}$, along with certain general domain parameters defined above. Elliptic curve domain parameters over $\mathbb{F}_{2^m}$ precisely specify an elliptic curve and a base point. This is necessary to define public-key cryptography schemes based on ECC [16]. If the elliptic curve domain parameters are verifiably random than they should be accompanied by the seed value *S* from which they are derived [16].

2.3.3 Standardized Versus Random Curves

In order to increase efficiency of cryptographic implementations and to prevent all known attacks, various standardized domain parameters have been developed for elliptic curves over both prime and finite fields. These standardized, or “special”, curves have been published by the SECG [16] and are recommended by NIST for use in U.S. government applications. However, in order to guard against future attacks against these curves one might decide to generate a new curve randomly but that has a validation process that proves the new curve resists all known attacks on the ECDLP. Fortunately algorithms exist to accomplish this very task [2]. The conventional wisdom of ECC has been, as described by Koblitz [19]:

- For greatest security choose parameters as randomly as possible
- It is safest to choose the defining equation to have random coefficients
- It is okay to use special curves for reasons of efficiency if you insist, however that choice may one day come back to bite you

Recent work on isogenies in elliptic curve cryptography has shown that there are various scenarios in which a special curve is better than a random curve. Isogenies, simply put, allow one to transport the discrete logarithm problem from one curve to another. It is “random self-reducible” within a set of endomorphism classes with small conductor gaps. Work in this area has shown that we need to assume that some version of a Weil Descent attack or another approach someday will lead to a faster-than-sqrt attack on a small but non-negligible portion of random curves [19].

It is unknown at this time whether random curves are truly more secure than special curves. Therefore, for control systems for the Smart Grid and other next generation integrated power system following the NIST recommendation seems to be the most prudent.

2.4 Known Attack Mechanisms against ECC

This section presents a basic overview of the theory behind various attacks against ECC, focusing more on the implications of these attack methods and the countermeasures to these attacks. Attacks against ECC focus on finding ways to solve the ECDLP in sub-exponential time. It should be noted that using ECC technologies such as the Elliptic Curve Digital Signature Algorithm (ECDSA) using any of the SECG recommended elliptic curve domain parameters [16] will provide protection against all known attacks (i.e. render these attacks computationally infeasible).

2.4.1 Naïve Method

The most naïve method for solving the ECDLP is to perform an exhaustive search where one computes the sequence of points *1P, 2P, 3P,…lP* until *Q* is encountered. On average this will take *n/2* steps. Therefore the naïve method can be circumvented by selecting elliptic curve domain parameters with *n* being sufficiently large to represent an infeasible number of calculations (e.g. $n = 2^{80}$) [2]. Therefore other methods of solving the ECDLP must be sought.

The best general-purpose attack known on the ECDLP is the combination of the Pohlig-Hellman algorithm and Pollard's rho algorithm [2]. Even these attacks can have an exponential running time depending on the selection of the domain parameters. However, it should be noted that there exists no mathematical proof that there does not exist an efficient algorithm for solving the ECDLP. Some evidence for the intractability of the ECDLP does exist and researchers have been studying the problem extensively since 1985 when it was first proposed [2].

2.4.2 Pohlig-Hellman Attack

The Pohlig-Hellman attack uses an algorithm that reduces the computation of $l = log_p Q$ to the computation of discrete logarithms in the prime order subgroups of $<P>$. Therefore in order to maximize resistance to the attack domain parameters should be selected such that the order n of P is divisible by a large prime so that the subgroup field is large.

2.4.3 Pollard's rho Attack

The idea of Pollard's rho attack is to find distinct pairs (c', d') and (c'', d'') of integers modulo n such that:

$$c'P + d'Q = c''P + d''Q$$

Hence $l = log_p Q$ can be obtained by computing

$$L = (c'-c'')(d'-d'')^{-1} \; mod \; n$$

This attack on its own takes roughly the same expected time as the naïve method but has negligible storage requirements [2]. There are multiple ways of speeding up this attack, including methods of parallelizing the attack to allow multiple processors to work together to solve an ECDLP instance in which the speedup is linear to the number of processors used. The processors also do not have to communicate to each other and need only limited communications to a central server.

2.4.4 Index-Calculus Attacks

Index-calculus algorithms are the most powerful methods known for computing discrete logarithms in groups such as the multiplicative group of a finite field. The question that naturally arises is whether these algorithms can be used to solve the ECDLP in sub-exponential time. The problem for the ECDLP is that no one knows yet how to efficiently lift points in $E(F_p)$ to $E(Q)$. Additionally, it has been proven under some reasonable assumptions that the number of points of the small height required for these algorithms is extremely small so that only an insignificant proportion of the points can be lifted. Therefore, so far no one has found an index-calculus approach that yields a general subexponential-time (or better) algorithm for the ECDLP [2].

2.4.5 Isomorphism Attacks

Isomorphism attacks essentially try to reduce the ECDLP to the DLP in groups for which subexponential-time (or faster) algorithms are known. Consequently the ECDLP for curves on which an isomorphism attack are found can be efficiently solved. Weil and Tate pairing attacks and Weil descent attacks are examples of isomorphism attacks [2].

2.5 Cryptographic Protocols Useful for Control Systems

As discussed in section 1.3.1 the primary need for control systems is to verify data integrity and authentication. This need is fulfilled in corporate / non-control systems through the use of the Digital Signature Algorithm discussed above. However the use of this algorithm is infeasible for control systems. Elliptic curves offer us an alternative path through the use of the Elliptic Curve Digital Signature Algorithm (ECDSA). There are also a number of other alternative elliptic curve signature schemes, such as Elliptic Curve ElGamal Signatures (ECES) and Abbreviated ECES Signatures (AECES). The subsections below detail the algorithm, beginning with generating private and public keys for use in ECDSA.

2.5.1 Key Generation

ECC key pairs are associated with the particular elliptic curve domain parameters used in the generation of the key pair. The public key is a randomly selected point Q in the group $<P>$ generated by P. The private key that corresponds to the public key is the solution to the ECDLP $d = log_p Q$. The entity that is generating the key pair must have the assurance that the domain parameters are valid (i.e. resistant to all known attacks), and the association between the domain parameters and the public key must be verifiable by all entities in the communication.

In non-control / corporate systems this would normally be done by a certification authority that generates a certificate attesting to the association between a public key and its domain parameters. Large scale control systems such as the Smart Grid will need to perform the same function on some level. For smaller control systems, such as those planned for use on US Navy ships for NGIPS, this association can be achieved by context (i.e. all entities in the system use the same domain parameters).

Algorithm 10 below illustrates how to generate an ECC key pair assuming valid domain parameters. It is critical that the number *d* generated be random, as in the likelihood that any particular value of *d* would be chosen over any other value is so small that an adversary is unable to narrow down the search space for *d*. This is akin to the idea that one should not select a password that includes their spouse's name.

Algorithm 10. [2] – Generating ECC Key Pair

```
INPUT: Domain Parameters D = (q, FR, S, a, b, G, n, h)
OUTPUT: Public key $Q$, Private key $d$
1. Randomly select $d \epsilon_R [1, n-1]$
2. Compute $Q = dP$
3. Return (Q, d)
```

Entities that receive a public key *Q* and a set of associated domain parameters will need to validate the public key to ensure that the private key actually exists and that the keys lie on the curve. Failure to perform public key validation could allow an attacker to try to get you to use the invalid public key in such a way that information about your private key could be revealed. Algorithm 11 illustrates how to perform the required validation.

Algorithm 11. [2] – ECC Public Key Validation

```
INPUT: Domain Parameters $D = (q, FR, S, a, b, G, n, h)$, public key $Q$
OUTPUT: Acceptance or rejection of the validity of $Q$
1. Verify that $Q \neq \infty$
```

2. Verify that x_Q and y_Q are properly represented elements of $\mathbb{F}_q$ (i.e. integers in the interval *[0, q-1]* if the field is prime, and bit strings of length *m* bits if the field is a binary field of order 2^m)
3. Verify that *Q* satisfies the elliptic curve equation defined by *a* and *b*
4. Verify that $nQ = \infty$
5. If any verification fails then return invalid, else return valid

Note that the check is step 4 involves an expensive point multiplication. Faster methods do exist for certain curves. For example, if the cofactor *h* of a prime field curve is equal to 1 (which is usually the case in practice and for all of the SECG recommend prime field curves [16]) then successful completion of the checks in steps 1 through 3 imply that $nQ = \infty$ [2].

2.5.2 Elliptic Curve Digital Signature Algorithm (ECDSA)

Algorithm 12 and Algorithm 13 below define how to generate and verify ECDSA signatures, respectively. In these algorithms, *H* denotes some cryptographic hash function whose outputs have bitlength no more than that of *n*. If this condition is not satisfied though, the outputs of *H* can be truncated.

ECDSA uses a per-message secret *k* that if discovered by an adversary can be used to recover the private key since:

$$d = r^{-1}(ks-e) \bmod n \qquad \text{where } e = H(m)$$

Furthermore it has been shown that if an adversary obtains even a few consecutive bits of the secret *k* then the adversary can easily compute the private key. It is therefore of utmost importance that *k* be randomly and securely generated, securely stored, and securely destroyed after it has been used. The reason why *k* should be generated randomly is to help ensure that *k* does not repeat. If the same per-message secret *k* was used to generate ECDSA signatures *(r, s1)* and *(r, s2)* on two messages *m1* and *m2* then if $s1 \neq s2$ (which with overwhelming probability they will not be equal) it can be shown that:

$$k \equiv (s_1 - s_2)^{-1}(e_1 - e_2) \bmod n \quad where\ e_1 = H(m_1)\ and\ e_2 = H(m_2) \qquad [10]$$

Thus an adversary could determine *k* and then use it to determine the private key *d*.

Algorithm 12. [2] – ECDSA Signature Generation

```
INPUT:  Domain Parameters $D = (q, FR, S, a, b, P, n, h)$, private key $d$, message $m$
OUTPUT: Signature $(r, s)$
1. Randomly select $k \epsilon_R [1, n-1]$
2. Compute $kP = (x_1, y_1)$ and convert $x_1$ to an integer $\overline{x_1}$
3. Compute $r = \overline{x_1} \bmod n$ and if $r = 0$ go to step 1
4. Compute $e = H(m)$
5. Compute $s = k^{-1}(e + dr) \bmod n$ and if $s = 0$ go to step 1
6. Return $(r, s)$
```

Algorithm 13. [2] – ECDSA Signature Verification

INPUT: Domain Parameters *D = (q, FR, S, a, b, P, n, h)*, public key *Q*, message *m*, signature *(r, s)*

OUTPUT: Acceptance or rejection of the signature

1. Verify that r and s are integers in the interval $[1, n\text{-}1]$, if any verification fails then reject the signature
2. Compute $e = H(m)$
3. Compute $\mathrm{w} = \mathrm{s}^{-1} \bmod \mathrm{n}$
4. Compute $\mathrm{u}_1 = \mathrm{ew} \bmod \mathrm{n}$ and $\mathrm{u}_2 = \mathrm{rw} \bmod \mathrm{n}$
5. Compute $\mathrm{X} = \mathrm{u}_1\mathrm{P} + \mathrm{u}_2\mathrm{Q}$
6. If $X = \infty$ then reject the signature
7. Convert the x-coordinate x_1 of X to an integer $\overline{x_1}$; compute $v = \overline{x_1} \ mod\ n$
8. If $v = r$ then accept the signature, else reject

2.5.3 Supported Secure Hash Algorithms

Cryptographic hash functions are used in many applications within ECC, including verifiably random curve and base point generators, key derivation functions, and ECDSA. According to the SECG [16] supported hash functions for ECC are:

- SHA-1
- SHA-224
- SHA-256
- SHA-384
- SHA-512

On October 2, 2012 NIST concluded a competition for a new SHA-3 algorithm, selecting Keccak as the winner [36]. Future versions of SECG standards are likely to allow use of the new SHA-3.

The security level associated with a hash function depends on its application. Collision resistance is generally needed for computing message digests in ECDSA, and where collision resistance is needed the security level is at most half the output length (in bits) of the hash function. Testing has shown that SHA-1 provides less than 80 bits of collision resistance [15] and therefore should be used with ECDSA only when providing backwards compatibility.

2.6 Comparing RSA Signatures to ECDSA

It has already been stated that ECDSA offers security equivalent to RSA using much smaller key sizes which can lead to increased efficiency. Figure 8 below shows a chart of comparable key sizes for equivalent levels of security. Figure 9 through Figure 11 below show published literature execution times for ECDSA and RSA algorithms for key generation, signature generation, and signature verification.

These times were taken from tests performed on an Intel Pentium 4 2.0 GHz machine with 512MB of RAM, on a 100KB text file used as a message [20]. The authors used the RSA Crypto++ Library 5.1™ and EC Borzoi 1.02 in their work. As discussed previously though, the architecture for control system components such as PLCs is radically different than that of an x86 architecture, and therefore these timings only provide a very basic indication of what the performance of ECC might look like in control system applications.

Symmetric	ECC	RSA
80	163	1024
112	233	2240
128	283	3072
192	409	7680
256	571	15360

Figure 8: ECC vs RSA Comparable Key Sizes (in bits) [20]

Key Length		Time (s)	
ECC	RSA	ECC	RSA
163	1024	0.08	0.16
233	2240	0.18	7.47
283	3072	0.27	9.80
409	7680	0.64	133.90
571	15360	1.44	679.06

Figure 9: ECC vs RSA Key Generation [20]

Key Length		Time (s)	
ECC	RSA	ECC	RSA
163	1024	0.15	0.01
233	2240	0.34	0.15
283	3072	0.59	0.21
409	7680	1.18	1.53
571	15360	3.07	9.20

Figure 10: ECC vs RSA Signature Generation [20]

Key Length		Time (s)	
ECC	RSA	ECC	RSA
163	1024	0.23	0.01
233	2240	0.51	0.01
283	3072	0.86	0.01
409	7680	1.80	0.01
571	15360	4.53	0.03

Figure 11: ECC vs RSA Signature Verification [20]

The results show that ECC outperforms RSA significantly in key generation time, and performs signature generation faster than RSA for higher key sizes. RSA outperforms ECC in signature verification significantly for all key sizes. The times appear to show that RSA signature verification time is fairly independent of key size and for practical purposes this is true, however this is really just due to the resolution at which testing was performed (for example RSA signature verification at 7680 bit key size should be approximately 0.008 seconds while signature verification at 15360 bit key size should be approximately 0.032 seconds). ECC signature verification grows linearly with an increase in key size, however the times show that RSA significantly outperforms ECC in this area. Signature verification is therefore of particular concern in looking at implementing ECC signature algorithms for control systems. At stronger levels of security with larger key sizes, ECDSA will outperform RSA for the total message transmission (including both signature generation and verification) since ECC signature verification timing scales linearly while RSA signature generation timing scales exponentially (due to the exponential increase in key sizes) for equivalent levels of security.

A variant of ECDSA, known as the Elliptic Curve Korean Certificate-based Digital Signature Algorithm (EC-KCDSA), is computationally more efficient than ECDSA. In EC-KCDSA the signer's private key is an integer $d \in_R [1, n\text{-}1]$ as is in ECDSA, but the public key is instead $Q= d^{-1}P$ (instead of dP). This allows for the design of signature generation and verification procedures that do not require performing modular inversion and therefore could potentially be more applicable in meeting control system needs. EC-

KCDSA has been proven secure under the assumptions that the discrete logarithm problem is intractable and that the hash function is a random function.

An alternative variant of ECDSA, proposed by Antipa et al (2005) [21], involves reconstructing the ephermeral elliptic curve point *R* from the signature component *r*. In other words one converts the ECDSA signature *(r, s)* over some message *m* to a new ECDSA* signature *(R, s)*. Antipa et al provide a general procedure for this change which accepts the ECDSA signature as an input, performs the reconstruction/conversion, and returns either acceptance or rejection of the signature. This speeds up ECDSA signature verification by 35-40% at the cost of only a small number of bits appended to traditional ECDSA signatures. Unfortunately, the EC-KCDSA algorithm and the ECDSA* algorithms are non-compliant with any of the existing ECDSA standards.

2.7 Edward's Curves

In 2007 Harold Edwards introduced a new form for elliptic curves that is birationally equivalent to an elliptic curve in the Weierstrass form described above. Edwards showed that all elliptic curves over number fields could be transformed to the shape $x^2 + y^2 = c^2 (1+x^2y^2)$ with $(0,c)$ as a neutral element [24]. This work was further expanded upon by Daniel Bernstein and Tanja Lange in [25] to demonstrate that the elliptic curve "Curve25519", which had previously set speed records for single-scalar multiplication, could be transformed to Edwards curves over the same field. Their work showed that this transformation would become the new speed leader for multi-scalar multiplication. This work was further developed in 2011 to create the Edwards-curve Digital Signature Algorithm (EdDSA) [27].

EdDSA promoted a number of benefits over ECDSA, including enhanced speed in signature generation and batch verification, while at the same time providing a 128-bit security level. However the primary advantages of EdDSA with respect to this work are as follows [26]:

1. Foolproof session keys: Signatures are generated deterministically, so while key generation consumes new randomness the generation of signatures does not.
2. Collision resilence: Hash-function collisions do not break this system. This adds a layer of defense against the possibility of weakness in the selected hash function.
3. Small signatures and keys: Signatures fit into 64 bytes and public keys consume only 32 bytes. The signatures and keys are actually compressed versions of longer signatures.

The details of EdDSA implementation are described in [27], however the sections below provide an overview of key generation, signature generation, and signature verification.

2.7.1 Key Generation

The figure below shows how key generation works in EdDSA. A random scalar is created which becomes the private key. This value is then hashed which helps the security of the key in the event that there a few bits of missing entropy in the generation of the key. The hash is then split into two parts, with the left half of the hash going through a series of bit manipulations to become the private scalar and the right half being saved. The private scalar is then multiplied by a base point through a point multiplication operation to become the public key.

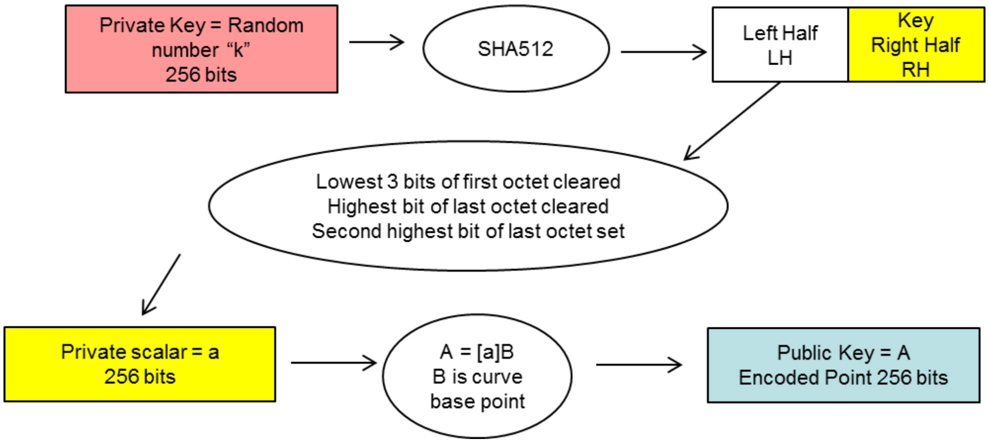


Figure 12: EdDSA Key Generation

Note that in Figure 12 the private key is run through the hash each time signature creation and verification is required. It is possible to simply save the "Key Right Half" and "Private Scalar" values, which doubles the amount of storage memory required to save the private key but improves overall timing efficiency.

2.7.2 Signature Generation

Figure 13 shows how signatures are generated in EdDSA. The "Private Key Right Half" is concatenated with the message and run through a hash, the results of which are intepreted as a scalar. The scalar then goes through a **mod L** operation to become **r**, where **L** represents the order of the curve. The value **r * B** is then encoded as a point **R** on an Edwards-curve, which is then concatenated with the message and the public key, and then hashed together to create a new scalar. That scalar also goes thru a **mod L** operation in order to become **k**. The scalar **k** is then multiplied by the "Private Key Scalar", added to the scalar **r**, and then goes through a **mod L** operation to become a scalar **S**. The point **R** and the scalar **S** together make up the EdDSA signature.

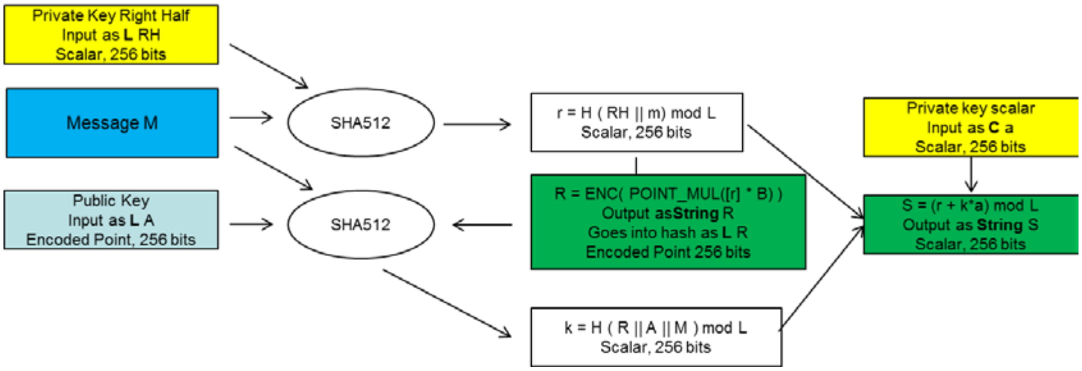


Figure 13: EdDSA Signature Generation

2.7.3 Signature Verification

Figure 14 below shows how signatures are verified in EdDSA. The operations are similar to Signature Generation, except that the Public Key and the Signature **R** values are both points on an Edwards-curve represented by strings that must be decoded back into points. These points are processed via two different point multiplication operations, which makes signature verification more computationally expensive than signature generation. Ultimately verification performs a check on the two different sides of the equation to ensure that the values are equal. This check is done by encoding the two different sides of the equation, both of which are points, into strings and then comparing the encoded strings. This encoding is critical because the two different points (x,y) are usually represented in extended coordinates (X, Y, Z, T), and a straight comparison between different (X, Y, Z, T) values is not an accurate way to compare the results and determine if a signature is valid or not.

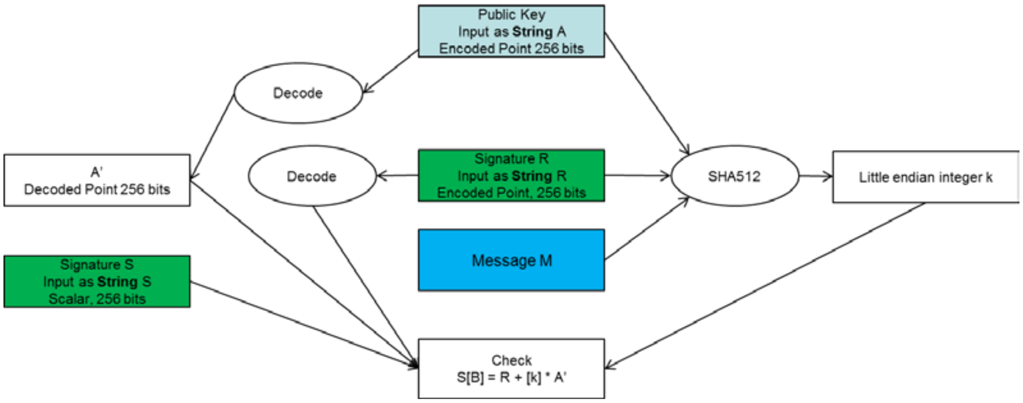


Figure 14: EdDSA Signature Verification

2.7.4 Special Curves

Reference [27] describes two special curves that have been developed for EdDSA. The first is known as Ed25519, which is named after the prime number **p** defining the underlying finite field (in this case 2^{255}-19). This curve is fully defined in [27] and is intended to operate at around the 128-bit security level. The second curve is Ed448, again named after the prime number **p** defining the underlying finite field (in this case $2^{448} – 2^{224} – 1$). Ed448 is provides security at around the 224-bit security level and is intended for applications which require higher security and have reduced performance requirements. Ed448 is sometimes known as “Goldilocks” or “Ed448-Goldilocks”, and is especially useful when there is a desire to hedge against analytical attacks on elliptic curves.

For control system applications, however, performance is the primary requirement as long as security can be provided at the 128-bit security level, and reasonable projections of classical computing capabilities have concluded that this curve is perfectly safe [27].

Therefore the Ed25519 curve was chosen for this work. Additionally there are variants of all of the EdDSA schemes known as PureEdDSA and HashEdDSA. The difference between them is that PureEdDSA requires two passes over the input while HashEdDSA only requires one, however PureEdDSA provides security even if it is possible for someone to compute a collision for the hash function. Therefore it was decided to use the PureEdDSA variant of Ed25519 (hereafter just known as Ed25519 vice the term Ed25519ph use to designate the hash variant). This decision does not impact the timing per scan cycle, but does increase the total time that it takes to compute and verify the digital signatures.

CHAPTER 3: TECHNICAL CONSTRAINTS AND SOLUTIONS

3.1 Introduction

There are a number of problems with implementing cryptography in Industrial Control Systems (ICS), such as the inability to natively perform 64 bit operations. This section presents the most significant challenges the author faced in developing a control system cryptography solution. It concludes with an overview of the solution and the proof of concept implementation, which will be further defined in the following chapters.

3.2 Random Number Generation

In cryptography, the ability to generate true random numbers is crucial and failure to do so will result in a vulnerable system. In the prototype, every message was digitally signed before being sent to a controller using the Elliptic Curve Digital Signature Algorithm (ECDSA). ECDSA requires a random number to be generated for each signature in order for the algorithm to be secure. While it is possible to use a pseudo-random number generator (PRNG) for ECDSA, the PRNG has to be initialized with a true random seed which is kept secret. The seed itself is partially consumed with each use of the PRNG, eventually weakening to the point that the seed must be reinitialized to a new true random number to maintain security. Since PLCs send a message anywhere from 20 to 50 times a second, and each message in the prototype requires a digital signature which consumes randomness, the seed would be too quickly consumed to provide long term security.

Unfortunately, the creation of a true random number generator (TRNG) is a non-trivial task which usually relies on some kind of quantum effect. The most common sources for TRNGs include radioactive sources, quantum effects in semiconductors, and quantum effects in photon polarization detection. Sources that do not rely on quantum effects usually rely on “human error” by asking a human to perform a task such as move a mouse around randomly and calculate the deltas in time when the human moves the mouse in a different direction. Software such as Veracrypt, for example, relies on this “human error” approach.

At this time for most applications the only true reliable method of obtaining random numbers is to install an external hardware-based random number generator. There are a number of products on the market today that make the claim to be TRNG (evaluation of such claims is beyond the scope of this paper). However most of those devices were designed with a more standard computer in mind and are incompatible for use with a PLC. An example is the TRNG9815 device commercially available at www.trng98.com and shown in Figure 15 below. This device is based upon a Zeener diode noise source which is then amplified to be read by a PC.

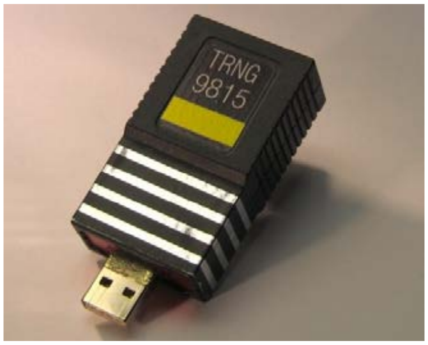


Figure 15: TRNG 9815 Device

One problem with this device, and others like it, is that they usually rely on a USB connection to a PC and external driver software to function properly. ICS devices like PLCs do not have the capability to read USB devices let alone install driver software. Additionally, many control engineers consider the presence of USB ports on ICS devices to be a significant security risk.

The good news is that it is possible for ICS vendors to create a hardware based TRNG using existing technologies with relative ease. Therefore, while a commercial product is not available at this time to the best of the author's knowledge, it is likely one will be available in the near future. The bad news though, is that it takes time for a TRNG to collect enough quantum data to generate a random number long enough to be cryptographically usable. Given the high rate of PLC messages it is doubtful a TRNG will be available in the near term that is fast enough for per-message authentication. We will have to continue to use a PRNG with a TRNG to update the PRNG's seed value.

Therefore this work proposes a solution that will significantly decrease our need for random numbers from potentially 380+ million random bits a day to 256 random bits every couple of months. This would allow us to use PRNGs to meet our application needs in the short run and ultimately loosen the design constraints for an eventual hardware based TRNG that can be used on a PLC.

3.3 BigIntegers

There are a range of existing cryptographic algorithms used in Information Technology (IT) systems that can provide authentication and validation for data streams and packets. Open source libraries such as OpenSSL provide readily accessible implementations of these algorithms that can be included in software packages to provide data authentication and verification services for almost every application. Unfortunately none of these libraries are appropriate for use by PLCs and embedded controllers as they run a unique software language known as “ladder logic” that is based on old electrical relay diagrams.

Ladder logic offers a number of benefits for control systems, primarily the benefit of being easy to understand and implement. Additionally it offers the ability to watch the code execute “live” to debug without the need of more advanced features such as breakpoints which are common when performing debugging of a higher level language. In many ways implementing a solution in ladder logic is much more similar to a hardware based solution than a software based solution. Ultimately this means that the PLCs can run for years without stopping, and it is not uncommon to find industrial control systems that have been running non-stop for 10, 15, or even 20 years.

The downside is that these controllers are unable to use existing implementations of cryptographic algorithms such as those found in the OpenSSL library. Even basic operations such as declaring an unsigned integer and a standard “for loop” are dramatically more difficult than in a language like C/C++. This becomes especially critical when looking at concepts such as BigIntegers which are essential for performing cryptographic functions.

A BigInteger is a data type that represents an arbitrarily large integer whose value has no upper or lower bound. This is distinguished from a DINT or even a LINT, which has an upper bound limited to the number of bits present in the data type (32 and 64 respectively). In RSA, a common cryptographic algorithm used to secure a wide range of systems, BigIntegers on the order of 2048 to 8192 bits are used. The use of ECC helps reduce the scale of this problem by allowing us to use smaller BigIntegers to provide solutions of equivalent strength at reduced bit sizes. For example, to provide security at the 128-bit level (the common security level for data considered "secret") RSA BigIntegers need to contain 3072 bits according to NIST SP 800-57 [5]. ECC can provide us the same security using BigIntegers of 256 bits, which dramatically improves the scope of the problem but does not in itself provide us a solution to perform 256-bit math in a PLC.

Representing a 256-bit BigInteger can be done a number of ways, with the most obvious being to simply take an array of 8 32-bit DINTs. The problem with this approach is that when you perform operations such as addition and multiplication you end up having to propagate the carries, and if you use an array of 8 32-bit DINTs you have to perform a lot of work to propagate the carries which turns into a very long sequence of bit operations. Bernstein on his blog [23] states "The standard NIST P-256 reduction procedure becomes even more painful if integers aren't represented in radix 2^{32} (or 2^{16} or 2^{8}): the word shuffling required for T,S1,S2,S3,S4,D1,D2,D3,D4 then turns into a long sequence of bit manipulations. The reason this is important is that radix 2^{32} isn't the best way to carry out big-integer arithmetic on most CPUs. Even on CPUs where the largest multiplier size is exactly 32 bits, it's almost always better to use a radix smaller than 2^{32}, so that carries can be delayed."

Rockwell Automation has a data type in the RSLogix5000 series called LINT, which is a 64-bit integer. However, the mathematical operations ADD, SUB, MUL, and others which are usable on a 32-bit DINT data type cannot be used with the LINT.

Rockwell does provide a series of Add-On Instructions (AOIs) that provide this functionality. These AOIs essentially work by breaking the 64-bit LINT into 3 32-bit DINTs, performing the math operation on the three different pieces and then reducing the entire array of DINTs back into one LINT.

Following a similar approach we determined that using twenty-two (22) 12-bit pieces stored in DINT is the most efficient method for representing a 256-bit value for the following reasons:

1. Splitting the value into 32-bit pieces significantly increases the number of bit manipulations which decreases code efficiency
2. Splitting the value into 16-bit pieces works for addition and subtraction, but when you perform multiplication the result in the intermediate steps would have 32-bit pieces which would have to be added together introducing carries which then require complex code to handle

The result is that splitting the value into 12-bit pieces is the largest size that takes up the least amount of space without introducing significant code complexity. In particular, this allows you to be able to multiply two sums without having to perform a reduction until the end of the multiplication. Smaller pieces increases the size of the internal loops of the operations which increases overall run time. Thus twenty-two (22) 12-bit pieces is the optimal way for storing 256-bit BigIntegers on a PLC or on any embedded controller device operating with a 32-bit processor.

3.4 Processing Speed

The single biggest challenge in implementing cryptography in a PLC is the speed of the processor. Over the years PLC processing speeds have increased dramatically, to the point that control system engineers rarely have to even consider the possibility that their application might actually run so slow on a PLC that it would fault the controller. On average, the PLC programs developed in our offices have scan times between 20ms and 100ms for fairly large applications processing hundreds to thousands of I/O points. In general, it is the opinion of the author that a control system program must have a scan time at least twice as fast as the fastest response rate of a controlled physical I/O device (i.e a valve or pump). This typically translates to a required scan time of less than 250ms, and ideally less than 100ms. Additionally, we have found that scan times of greater than 500ms will actually cause a Rockwell 1756-L8x series processor (the latest available at the time of writing) to have a major fault.

Modern computers have multi-core processors that run in the GHz range and typically do not have real-time performance requirements (those that do can always have dedicated cryptographic modules). PLCs on the other hand have strong real-time performance requirements and have processors that run in the MHz range due to the lack of active cooling. To illustrate this challenge a basic SHA-512 algorithm written completely in ladder logic was developed. Running that algorithm in RSEmulate on a standard PC running an Intel 2500K processor at 4.2 GHz for 104 bytes of data gave a scan time of 2ms. Running that same algorithm on a 1756-L8x series PLC gave a scan time of 26ms, over an order of magnitude higher. While this may not seem like a lot, consider that operations used in cryptography such as point multiplication and modular exponentiation are much more complex. Such operations could easily consume the entire available scan time of a PLC and leave no real time to actually perform control work. Therefore a solution is required that would allow us to perform these complex operations without impacting scan time.

As an example, look at the case of an elliptic curve point multiplication, which is the process of repeatedly adding a point along an elliptic curve to itself. This process creates a trapdoor function, and the security of ECC is based on the intractability of determining a multiplier **n** from the equation **Q** = **n*****P** where **Q** and **P** are given points on the curve.

There are a number of approaches in performing point multiplication, such as the sliding-window method and Montgomery ladder. If you follow the approach in RFC 8032 [27] then a single point addition requires nine 256-bit multiplications, four 256-bit additions, and four 256-bit subtractions. Each of these operations must be repeated for each bit in the multiplier in order to perform the point multiplication at constant time and avoid a timing side channel attack.

The coding of such an algorithm into ladder logic is a challenge in of itself, but even the most efficient implementation will not be able to run on the PLC. This is because, as stated above, there is an already known optimally efficient way to store the 256-bit value, which results in a known optimal number of standard ladder logic math instructions such as ADD and MUL to perform a basic B256_ADD and B256_MUL.

For example, just looking at the number of standard RSLogix5000 MUL instructions in one point multiplication gives the following result:

1 POINT_MUL = (8 B256_MUL + 1 POINT_ADD) per bit in n

1 POINT_ADD = 9 B256_MUL

1 B256_MUL = 484 MUL

*1 POINT_MUL = [(8 + 9) * 484] * 256 = 2,106,368 MUL*

So essentially one point multiplication results in over two million ladder logic MUL instructions alone, not including the additions, subtractions, shifts, and other loop instructions required. The result is such a massive number of instructions that if you attempt to run them all in one scan of a PLC it will negatively impact the overall scan time and potentially even fault out the controller. This problem is compounded further by the fact that the Ed25519 Digital Signature Algorithm and the ECDH algorithm require multiple point multiplications, although not all at the same time.

In order to improve the execution rate of the processor per scan we looked at two trade space areas. The first “classic” trade was memory space for time. Memory is much more limited in a PLC than in a typical PC environment, with the PLC memory size typically in the 2-8MB range, but some exceptions that have up to 32MB. However most cryptographic algorithms require a relatively small memory space, and the typically PLC application will only require 2-3 MB of memory for a controller with approximately 500 I/O points. As a result there is a significant opportunity to design a cryptographic system that trades memory space for improved processing time.

The main way space for time trading is used in this design is the way the private key is stored (discussed further in Chapter 6). For an Ed25519 Digital Signature Algorithm the private key is normally stored as a 256 bit value, which is then hashed to generate a 512 bit value. Half of the bits from the hash are used as part of the key, while the remaining 256 bits are then manipulated to form a private scalar. The details of this will be discussed in more depth later, but the point is that an initial 256 bit key ultimately turns into two 256-bit parts via hashing and bit manipulations. Therefore you can save some processing time and skip the hashing and bit manipulations if you double the private key storage space from 256-bits to 512-bits.

The second trade space is a “trade longer total execution time for shorter scan times”. Using this approach the operation of sophisticated mathematical operations such as point multiplication are performed by breaking the operation into a series of sequential pieces, so that only a small portion of the total algorithm is run in any given scan. This is similar to a standard sequencer used in a batch control process, where the sequencer executes a set of commands in each step and waits for a feedback from the process indicating that step is complete before executing the next step. In the case of point multiplication, the most obvious solution is to break up the algorithm so that each point addition is performed on a separate scan. The sequencer waits for a confirmation that the point addition is completed, stores the result in a temporary variable, and then uses that result in the next scan for the next point addition. It should be noted that this approach also utilizes the idea of trading space for time, since it takes additional memory to hold the logic that controls the sequencing and execution of the different pieces of the operation.

Using this approach a point multiplication AOI was developed that can run on a Rockwell 1756-L83 processor with only a 10ms impact to total scan time. The same approach was used on all the other different pieces of the Ed25519 digital signature algorithm such as SHA-512. Combining all the pieces into a master sequencer produces a digital signature implementation capable of running on a PLC with a scan time of impact of less than 10ms. There are downsides to this approach. The first is that the total time to produce a digital signature is significantly longer. Depending on the processor and the amount of control code that has to run in a scan (that is not related to cryptography) the entire operation could take a couple of minutes to complete. Testing on the point multiplication showed that it took less than a minute to complete on a 1756-L8x series processor. An entire Ed25519 digital signature took ~5 minutes to complete on a Rockwell 1756-L8x series processor versus approximately 37 seconds on a Intel 2500K Quad Core Processor overclocked to 4.2 GHz running a generic and non-optimized python implementation. Regardless, the system itself will be able to run without impact during this time. The message traffic will continue to use the old key until negotiation and verification of the new key is complete. The process can also be performed during scheduled maintenance windows to reduce overall risk.

The second potential problem is that extending this algorithm across multiple scans can potentially make the algorithm vulnerable to side channel attacks. To protect against these attacks the following steps must be taken:

1. Ensure that the coding of the algorithm prevents a timing attack
2. Severely restrict users from being able to access and go online to the PLC
3. Use built in features of the PLC products to prevent users from being able to read and write to the tags involved in the cryptographic operations

The final problem is that this approach is not usable for an HMAC implementation because an HMAC must be calculated in its entirety on each scan. Otherwise you will not be able to perform data authentication for each and every message to and from the controller. Therefore an alternative approach must be used for HMACs, which is discussed in the following section.

3.5 Multicast

Historically control system data has been communicated between devices via unicast messages. However as systems have grown and become more redundant the use of multicast technologies has increased dramatically, particularly when having a large number of different controllers talk to a large number of consoles. The benefits of multicast are numerous for a networking perspective, particularly in being able to add nodes to an already established infrastructure. From a controls engineering perspective, this means that additional consoles can be added to a system without any impact to the end controllers, significantly reducing the testing and validation efforts.

From a security perspective, however, multicast presents some unique challenges. The cryptographic algorithms that exist today are designed for end-to-end security between a pair of devices. Security between groups of devices typically relies on the network infrastructure itself acting as a middle man to provide the security between the various end points. For example, in a normal unicast security mechanism between 2 or more nodes each node would have to send a unique unicast packet to each of the nodes that is secured via some cryptographic algorithm. In a normal multicast security scenario a node sends a secured packet to a switch, with the cryptographic algorithm securing the data only between the node and the switch. The switch then needs to execute a separate cryptographic algorithm to secure the packet being sent from the switch to a receiving node, repeating the process for each individual node. The result is that the most significant part of the security burden is placed on the network itself, and the network itself must be considered a trusted agent. If the network is somehow compromised, then all nodes communicating via that network can be compromised as well.

Therefore a security solution must be designed that allows for multicast messages but assumes that the network that is transmitting the data is hostile. Digitally signing each packet from each node avoids problems with hostile networks, since the nodes only need to know the public keys of each other. However as discussed previously, the limitations of processing power prevent this solution from being feasible and thus we require some kind of symmetric-key based solution such as an HMAC. This means that the set of multicasting nodes needs to somehow securely generate and share a symmetric key with each other.

One potential solution involves a modified ECDH key exchange, where a pair or pairs of nodes perform an ECDH key exchange but then interpret the result as a new scalar to use for a key exchange with another node or group of nodes. An example of this is shown below for just three nodes:

Step 1, Node 1: $\boldsymbol{q1*B = Q1}$

Step 2, Node 2: $\boldsymbol{q2*B = Q2}$

Step 3, Node 1: $\boldsymbol{q1*Q2= q12}$*, interpret point as a little-endian scalar*

Step 4, Node 1: $\boldsymbol{q12*B = Q12}$

Step 5, Node 2: $\boldsymbol{q2*Q1 = q12}$*, interpret point as a little-endian scalar*

Step 6, Node 2: $\boldsymbol{q12*B = Q12}$

Step 7, Node 3: $\boldsymbol{q3*B = Q3}$, $\boldsymbol{q3*Q12 = Q123}$

Step 8, Node 1: $\boldsymbol{q12*Q3 = Q123}$

Step 9, Node 2: ***q12*Q3 = Q123***

Ultimately the complexity of this process and the time it takes to perform increases linearly as the number of nodes increases, assuming everything works perfectly and all nodes are fully operational. Work proposed by Rodeh et al [32] follows this general approach and improves the overall efficiency to *Θ(log n)* as well as proposes alternative 2-round and 3-round solutions. However the reality is that this process will have a high error rate since each node must perfectly perform the mathematical operations in a precise sequence in order for the group to arrive at the correct symmetric key. If an event occurs that shuts down a node, such as a power loss, the computation of the symmetric key could be easily compromised. This means that significant error handling logic would be required to dynamically route the generation of the symmetric key in the event of a node failure. Additionally a process would need to be implemented to add a node back into the group when the node's failure is resolved (i.e. power is restored). Ultimately the result would be an algorithm with a greater than linear increase in processing time and complexity as the number of nodes increase.

Most secure multicast schemes follow some variant of the Iolus Framework for Secure Multicasting [30]. The framework essentially establishes a secure distribution tree, designed to communicate a symmetric key between the nodes which is used to secure the multicast data. It introduces the concept of different security subgroups, with each group having a Group Security Controller (GSC) which manages the top level subgroup and a Group Security Intermediary (GSI) which manages each of the other subgroups. Both GSCs and GSIs are known as Group Security Agents (GSA). The idea is that these entities form subgroups which then work together to deliver the multicast traffic to all of the subgroups in the overall group. At the root of the tree is the GSC which is ultimately responsible for the entire group. Typically these kinds of frameworks use a symmetric key called Kgroup to encrypt some new Kgroupnew, and then transmit the new key.

In many ways this approach is similar to the Group Key Management Protocol described in RFC 4046 [33], but since it is a tree it is potentially more scalable since all of the nodes will not have a reason to contact the root GSC at the same time which can cause an out-of-sync implosion. However, one of the issues with Iolus is that a node that wishes to join the group would have to first locate a GSA, particularly in a scenario where a GSA might be dynamic. Solutions that have been proposed to solve this problem primarily involve the node performing some kind of lookup against a directory service, however devices like PLCs do not have the processing power to perform such a lookup. The framework also requires that when a member joins or leaves a group a rekeying of the entire group will need to take place.

Some proposals such as found in [31] involve creating a virtual binary tree from the root node and only require rekeying between the end node and the path towards the root node which improves the overall efficiency of the algorithm to *O(log n)*. In general the primary problem that these works are attempting to solve is how to keep the group secure such that an old node can't just access data within the group without properly joining the group, and to effectively boot the node out of the group. As a result significant overhead must be assigned towards joining and leaving the group to ensure security before the nodes can talk. For an information technology system this makes sense, however for operational systems such as control systems this presents significant problems. A control system needs to be able to communicate as fast as possible when it powers on, and delays in that could have unintended operational consequences. Additionally, control system nodes do not just go on and offline like a typical information system node. The control systems are generally fairly static, and changes to them have to go thru rigorous testing processes for Installation Qualifications (IQ), Operational Qualifications (OQ), and finally Process Qualifications (PQ). Therefore if a node appears to be joining and leaving a group rapidly it is likely due to either equipment or power failure and control system operators will want to restore operation of the node as fast as possible.

The larger problem with the Iolus framework is that it does not account for damage to the network, which could cause partitioning and prevent nodes from being able to contact a GSA even though they are able to communicate with other nodes. This is especially important for military applications, but in general all critical infrastructure systems need to be robust enough to handle damage from all sources including natural disasters. The problem is mitigated to some degree in that the nodes would currently have a shared key and would be able to continue to talk for a limited time until the partitioned network is restored. However since new keys in the Iolus framework are generated and transmitted using the current symmetric key the shelf life of the keys is limited. The Iolus framework even includes an expiration for the keys in the framework, requiring that nodes reach out to a GSA to get an updated key.

The main purpose of the frequent key exchanges is to deal with a scenario where a node leaves a multicast group. The idea is to change the key once a minute or so to avoid having to change the key each time a node leaves the multicast group. This helps ensure that when a node leaves it must renegotiate with a GSA to rejoin the group, which can take some time. For a control system, however, you want the exact opposite approach. Control system nodes do not “come on and off” the network frequently, except in damage or power loss scenarios. Therefore when the node is restored you want to let it be able to rejoin the group as quickly as possible.

Finally, none of these proposals assume any kind of prior trust relationship between the various nodes. This makes sense when considering applications like video conferencing since the creation and membership of the group will vary wildly. However for control systems the creation and group membership will be fairly static. An algorithm tailored for control systems requires some kind of prior trust relationship between the nodes to ensure that only intended nodes join the group. Therefore some kind of public-private key pair and key infrastructure must be put into place so that a vendor can allow a different vendor to have his control system equipment be capable of joining the established group.

3.6 Lack of Proper Time Synchronization

Even when a message is authenticated by a cryptographic algorithm, it is still potentially vulnerable to a replay attack, where an attacker saves a copy of a valid message and then plays it again at a later time to cause systems to perform undesirable behavior. Solving this problem usually involves embedding a timestamp into the message, such that any message that comes in which is older than the last message that was received is immediately rejected as a possible replay attack. Normally this mechanism requires all nodes to have proper time synchronization, which is typically required by an IT system anyway to meet some other business or auditing requirement.

Control systems typically do not have a requirement for time synchronization, and typically do not even have clocks set to accurate times. Most commonly a PLC's clock is set whenever a program is downloaded into it based on the time of the PC that was used to download the program. In large control systems where multiple vendors are working to groom the system it is likely that the clocks of the vendors are not synchronized, and as a result the control system nodes are unsynchronized as well. Most PLC vendors do have a way of allowing a user to configure the PLC to be synchronized to an NTP server, but the problem with that approach is that now you are reliant on your NTP server securely and accurately updating the clocks of the controllers to avoid a replay attack. Therefore an attacker could first compromise the NTP server, roll back the clocks of the PLCs, and then use replay attacks in order to compromise the system.

3.7 Message Structures and Send Rates

One of the more unique differences between control system components and standard IT components is the frequency and how messages are generated and sent. A typical PC application that communicates across a network is capable of performing some kind of logic, generating the message, sending the message, and then waiting for a response. The application can then receive the response, process additional logic, and then generate a new message, potentially with a different structure than the first message. An example of this is the Transport Control Protocol (TCP) itself, which generates and sends messages such as SYN, ACK, and FIN that are used to establish a connection.

A PLC, and control system components in general, have a radically different approach. A PLC is typically not capable of dynamically changing the message structures without significant intervention on the part of a developer, and even then there are many control system components which have no such capability. This means that if a message is required to transmit or process a piece of data, such as a public Diffie-Hellman value, that message structure will always carry that piece of data even if it is not relevant at that time. Additionally the sending and receiving of messages is typically asynchronous of the logic that is generating the contents of the message. This means that even in something like a ECDH key exchange the public values used in the key exchange will continue to be transmitted by the controllers at a regular interval long after the key exchange has been completed. Even if the logic were to “zero-out” the value when the key exchange is completed, the bits that held the values would still be part of the message structure and would still be transmitted each and every scan.

3.8 Proof of Concept Implementation Overview

This section describes the high level concept of the design and the proof of concept implementation, all of which will be elaborated in more detail in the chapters below. The design consists of two parts. The first part is the Variable-round Message Authentication Code (VMAC) algorithm, in which control system data between nodes is authenticated using a shared secret key known by all of the nodes. Details of the VMAC are described in Chapter 4, but at a high level the VMAC is a variant of an HMAC-SHA256 algorithm but designed to run fast enough on a controller. Messages authenticated by VMAC can be communicated using multicast or unicast schemes.

The second part of the design is the Key Exchange Protocol (KEP), which is designed to securely generate and transmit the key that will be used by the VMAC. KEP uses a unicast based control message scheme between the nodes to generate and exchange the VMAC symmetric key **Ks'**, and contains logic to handle errors such as nodes potentially coming on and offline at various times. The KEP messages are not used for control data, which is typically multicast and is authenticated by VMAC. The concept is that KEP will be used to automatically update the VMAC keys in a fashion that provides a bumpless transfer between the keys at a rate that far exceeds the requirement to actually change the keys. In fact KEP could be configured to update the keys in a "non-stop" fashion, meaning that once a new key is generated, shared, and in use by all nodes KEP immediately could start working to generate a new set of keys. Such a rate would likely be excessive though, with a more reasonable rate being closer to once per year, however KEP does provide this capability. Note that throughout this paper there are two KEP symmetric keys mentioned, **Ks'** and **Ks**. **Ks'** is the primary key that is generated and transmitted by KEP, and generally represents the "latest and greatest" key that was generated. **Ks** represents the last instance of **Ks'** that was used by all nodes to generate VMAC messages, and can be thought of as the backup to **Ks'**. **Ks** is ultimately set equal to **Ks'** at completion of the KEP algorithm.

The design uses EdDSA to securely sign and verify the messages used in KEP to distribute the VMAC keys. Since EdDSA only consumes randomness during key generation and not signature generation it is possible to significantly reduce the number of random bits required by generating the EdDSA keys offline. A python script has been developed that will generate and test the keys for ControlLogix (refer to "Appendix E").

The design is protocol independent, meaning that the underlying communication mechanism between the nodes could be Ethernet, Modbus, Profibus, Profinet, EthernetIP, or some other fieldbus protocol variant. A proof of concept was developed using Rockwell Automation’s Studio 5000 software using IEC 61131-3 compliant ladder logic, making it relatively easy to develop a hardware based solution or a solution for a different PLC vendor. Testing between the nodes was done using four RSLogix1756-L83 PLCs using produce/consume tags in a multicast configuration for the IO.

CHAPTER 4: VMAC

4.1 Introduction

This section describes the details of the Variable-round Message Authentication Code (VMAC) algorithm. The algorithm ultimately consists of two parts, the VMAC itself which provides the cryptographic means for providing data authentication and validation, and the VMAC implementation which provides context for error handling and provides security against various forms of replay attacks. This chapter will begin discussing the VMAC algorithm from a cryptographic perspective, and end with implementation details. One point of note is that in theory the VMAC can be used without KEP in order to provide data authentication and verification, assuming that the engineer or design agent organization has control of every node and does not need to integrate with a third party vendor. From a practical perspective this is unlikely, but there are instances where this could be the case.

4.2 Cryptographic Details

VMAC is essentially a variant of an HMAC-SHA256 algorithm which was modified to meet the following requirements:

1. The algorithm must be fast enough to create the message authentication code without negatively impacting overall control system operation.
2. Any adversary with full knowledge of the software, the data, and matching hardware would not be able to produce the message authentication code without knowing the secret session key
3. Provide 128-bit level security (SHA-256 equivalent)

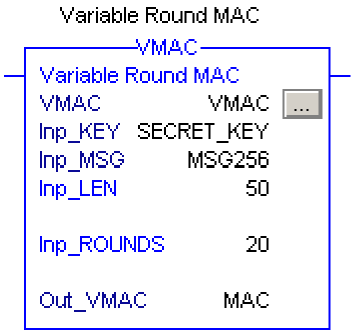
4. For a single bit change in the input produce on average approximately 128 bit changes in the output and no less than 50 bit changes in the output in the worst case (equivalent of SHA-256)
5. Protect against common attacks such as the length extension attack and collision attack

The standard HMAC algorithm for producing values at 128-bit security is HMAC-SHA256 which is a Θ(n) algorithm. This algorithm was implemented and tested as part of this work on a Rockwell 1756-L83 processor. The message size was 200 bytes. Initial results indicated that such an algorithm would have a scan time of 50-100ms when used in production. Considering that you would use two of them for both send and receive the combined scan time impact could reach up to 200ms for only 200 bytes. This would have detrimental impact on overall control system operation. Unfortunately, since the entire operation would have to be performed per message per scan there is no way to practically sequence the code in the same manner as was done for point multiplication or other complex operations. Therefore a new HMAC was designed as a compromise between security and performance using an alternative design approach as compared with the sequencing approach used in performing point multplications described previously.

The standard HMAC-SHA-256 algorithm follows the following construction as defined in RFC 2104 [22] :

$$HMAC(K, m) = H\Big((K' \oplus opad) \,||\, H\big((K' \oplus ipad) \,||\, m) \big)\Big)$$

Where H represents the SHA256 algorithm being used twice, K’ represents the key, and opad and ipad are constant values.



Variable Round MAC

VMAC

Variable Round MAC

VMAC	VMAC
Inp_KEY	SECRET_KEY
Inp_MSG	MSG256
Inp_LEN	50
Inp_ROUNDS	20
Out_VMAC	MAC

Figure 16: VMAC Add-On Instruction

The proposed Variable Round Message Authentication Code (VMAC) shown in Figure 16 was created as a reduced round derivative of SHA-256, keyed in a specific manner to avoid having to add an inner or outer pad and to avoid having to perform a second hash. Specifically the number of internal rounds of SHA-256 is reduced from 64 to between 20 and 64 rounds as defined by the parameter “Inp_ROUNDS” and the key is mixed into the scheduling of the message using the following formulas:

<u>Step 1</u>: Select one of the eight 32-bit portions of the 256-bit key to become the IndexedKey using the algorithm below. Note that if each round from 0 to 63 is designated by the parameter n, then the selected portion of the key is:

KeyIndex = n for n=0..7

KeyIndex = n-5 for n=8..12

KeyIndex = 1 for n=13

KeyIndex = 2 for n=14

KeyIndex = 0 for n=15

KeyIndex = (63-n) mod 8 for n=16..63

IndexedKey = Inp_KEY(KeyIndex)

Note that the **KeyIndex** was specifically chosen to ensure that each of the 256-bits of the output are impacted by each of the 256-bits of the key,

Step 2: Calculate the **RotationIndex** of the key, which causes a different rotation amount to be applied each time an **IndexedKey** value is used:

*RotationIndex =3*KeyIndex + (n>>3)*

The **RotationIndex** ensures that the exact same order of a specific part of the key is never used in more than one round when generating the hash.

Step 3: The key is then mixed into the scheduling process of the message using the following formula:

W[n] = Fo[n] XOR ROTR (RotationIndex,IndexedKey)

Where **Fo** represents the original formula to calculate **W[n]** in the SHA-256. As a result the key is mixed in recursively with the data when n is greater than 15. There is no ipad or opad and the hash algorithm is only used once, which means that if the AOI was run at 64 rounds using a key of all zeros you would get the same output as standard SHA-256. Note that **Fo** for each of the rounds 16 through 63 is a function of 4 previous values and the **IndexedKey**. The custom indexing of the key described in Step1 is necessary to ensure that the final **W** value for 20 rounds or more is a function of all 256 bits of the key.

The following table shows the timing results for processing the algorithm at different byte lengths with varying numbers of rounds on a 1756-L83 processor. These results are typical, as it has been shown in our testing that different processors within the same family might produce slightly different timing results.

Rounds	200 Bytes	400 Bytes	800 Bytes
20	7.3	12.9	23.8
24	9.8	17.1	31.6
32	14.8	25.7	47.6
48	23.9	41.7	76.9
64	33.5	58.7	108.4

Figure 17: VMAC Timing Results (in ms)

4.3 Attack Vectors

In this section VMAC is evaluated against the most common attack vectors for hash based message authentication codes in order to determine possible weaknesses that would make VMAC impractical for control system use from a security perspective. Ultimately it is concluded that the use of VMAC with 20 rounds is sufficiently secure for control system message authentication purposes, with a recommendation of using 24 rounds or more when operating on a single data block (less than 440 bits).

4.3.1 Length Extension Attacks

The most common attack against a hash based message authentication code is the length extension attack. The attack works when you hash a data set that looks like:

$$H\ (\ Key\ ||\ Message\ ) = Hash1$$

You can add more data to the message and generate a valid response (Hash2 below) without knowing the key by taking:

$$H\ (\ Hash1\ ||\ FakeMessage\ ) = Hash2$$

This is because

$$Key\ ||\ Message\ ||\ FakeMessage$$

Is indistinguishable from

$$Key\ ||\ SomeDifferentLongerMessage$$

This assumes that you properly calculate enough padding at the end of the real message before you append the fake data so that you fall on the next block. Thus the construction of most HMACs based on the SHA2 family need to use two hashes that essentially bound the length of the message. Note that SHA3 avoids this limitation by using a sponge construction and avoids blocks altogether.

VMAC is based on the SHA2 family but doesn’t use a second hash function to bound the length of the message. Therefore it would appear that VMAC might be vulnerable to the length extension attack. VMAC avoids this attack vector because it does not prefix the key as part of the first block of the message, it injects the key into every block of the message. Thus adding an extension to the method would not generate the same result because the FakeMessage above would not have been processed in the context of the key. This allows VMAC to avoid the second hash operation altogether and is thus capable of running faster than the HMAC-SHA256 algorithm.

4.3.2 Collision Attacks

The improvement efficiency gained in VMAC by avoiding the second hash helps improve overall timing of the algorithm, but given that the second hash is on a fixed length block the time saved is not significant enough. The bulk of VMAC's timing efficiency is based on reducing the number of internal rounds. However, reducing the number of rounds makes the algorithm weaker than a typical HMAC because it increases the chance of collisions. For starters the first 16 rounds of SHA-256 do not utilize the message expansion contained in the remaining rounds, so doing only 20 rounds means that you are only using 4 rounds of expansion versus 48. Therefore testing was conducted on reduced round SHA-256 to determine how reducing the rounds affected the output result when you change one bit in the input. Python code used to test SHA-256 can be found in "Appendix A.1". Standard NIST messages were used as the input, and the table below gives results for changing each bit in the input sequentially over a range of 512 trials per message per test.

Test	Rounds	Low	High	Average
MSG_0	64	101	152	127.6855
MSG_8	64	103	149	128.1992
MSG_440	64	103	148	127.3574
MSG_0	24	101	147	128.0195
MSG_8	24	110	154	128.4765
MSG_440	24	100	152	128.2656
MSG_0	20	101	150	126.5820
MSG_8	20	96	151	126.9570
MSG_440	20	92	147	127.0371
MSG_0	18	38	149	119.7031
MSG_8	18	44	158	120.2480
MSG_440	18	42	155	120.1914
MSG_0	16	2	149	106.3164
MSG_8	16	2	149	105.8262
MSG_440	16	2	149	105.2089

Figure 18: SHA256 Reduced Round Test Results

The results of the test show that in general you will always have the possibility of changing a large number of bits in the output on average, however you need to use at least 20 rounds to guarantee you will have a significant change every time. Going past 24 rounds you get a negligible increase in returns with a linear increase in time. A number of papers have been published detailing how to create a collision with reduced round SHA-256, some up to 46 rounds. It should be noted that these papers and the analysis done above are using standard SHA-256 without a secret key. VMAC uses the secret key which increases computation complexity, assuming the key is generated securely and randomly.

4.3.3 Key Reversal Attacks

An analysis of the VMAC was performed to determine how difficult it would be to extract the key. The initial assessment involved looking at a single input data block at 16-rounds, which is much simpler since it avoids any of the data bit rotations in the

SHA-256 message schedule. To simplify the analysis the input data that was used did not conform to the SHA-256 padding requirements, which are implemented in VMAC. In order to reverse out a round, first note that at the end of a round the intermediary hash values are available as shown below

```
                H0      H1      H2      H3      H4      H5      H6      H7

t = 0           a       b       c       d       e       f       g       h
...

t = 7           a       b       c       d       e       f       g       h
...

t = N-2         a       b       c       d       e       f       g       h
                 \       \       \       \       \       \       \
                  \       \       \       \       \       \       \
                   \       \       \       \       \       \       \
          T1+T2     \       \       \      +T1      \       \       \
             \       \       \       \       \       \       \       \
              \       \       \       \       \       \       \       \
               \       \       \       \       \       \       \       \
t = N-1         a       b       c       d       e       f       g       h

           +    H0      H1      H2      H3      H4      H5      H6      H7
           ----------------------------------------------------------------
               out0    out1    out2    out3    out4    out5    out6    out7
```

Figure 19: SHA Intermediary Hash Values

The values **H0** through **H7** are known constants, so given out0 through out7, we can subtract the constants and recover values a..h at the end of the 16th (the N-2) round. If we know the values a..h at the 16th round we can recover what the values were at the end of the 15th (N-1) round via the following formulas:

- *Round N-2 a,b,c,e,f,g are equal to Round N-1 b,c,d,f,g,h*
- *Round N-1 T2 can be computed from Round N-2 a,b,c*
- *Round N-1 T1 can be recovered from round N-1 a:*

    - *a = T1 + T2 ➔ T1 = a – T2*
- *Round N-2 d can be recovered from round N-1 T1 and e:*
    - *e =d + T1 ➔ d = e – T1*

If the constant for the round Kt is known (and it is based on the SHA-256 standard) then you can recover round N-2 h from T1:

- *T1= h + ConditionalFunction + Kt + Wt*
    - *h = T1 – ConditionalFunction – Kt – Wt*

where the **ConditionalFunction** in SHA-256 is based on the values of e thru g, which are known per above.

When we look at 16 rounds it turns out that a single bit flip in the input data will cause a corresponding bit flip in **T1**, and based on the change in the carry bit related to that flip we can determine a bit of **h** and the key using the formula:

$$V = T1 - offset = h + W[N-1]$$

where offset includes the previously calculated low-order bits of h, to eliminate carry propagation from those bits in the addition. Details of the analysis are shown in Appendix A.2. It turns out that all of the bits can be easily cracked when using arbitrary chosen input data, finding all but 8 bits of the 256-bit key directly and the remaining 8 bits by exhaustive search. Therefore 16 rounds are not secure, which means that running 20 rounds is really the equivalent of only doing 4 “secure” rounds.

The next assessment was done looking at a single input block at 17-rounds. The idea was similar to the 16-round crack, where if you can determine **h** then you could unwind the hash one more stage and find all of the key bits. At 17 rounds though, the message schedule takes affect which means that **W[16]** is no longer a copied value with a trivial key mix, and the formula expands to:

V = h + (((D XOR E) + X) XOR Y)

where *D = sigma1(Data[14]), E = sigma1(ROTR(Key[2],7)), Y = ROTR(Key[7],23)*

and *X = sum of the Data[9,1,0] and Key[4,1,0] terms*

If we can determine **h** then we could unwind the hash one more stage and find all of the key bits using the method presented in “Appendix A.2”.

Data[9,1,0] affect **h** and **X**, but changes in **D** do not change **h**, **E**, **X**, or **Y**, so flipping bit **i** of **D** will flip bit **i** of **V**. Although the change in bit **i** provides no information itself, the addition with **X** may flip carry bits, and bit **i+1** of ((**D** XOR **E**) + **X**) will change only if bit **i** of **X** is 1. However we can only observe **V**, and the addition with **h** may also flip carry bits. So if bit **i+1** of **V** changes then bit **i** of either **h** or **X** is 1, but we don't know which. Therefore we would have to continue without knowing h by running the approach for cracking 16 rounds for each of the possible 2^{32} values of **h**. Each of those produces 2^{8}

candidate keys so the overall search space is O(2^{40}). For each value of **h**, Crack16 invokes VMAC twice for each of 256-8 = 248 bits, that is 2*248 = 496 invocations. At a scan rate of 100msec, i.e. 10 invocations per second, that would take over 6000 years:

$$496 * ((2^{32})/10) / (60*60*24*365) = 6755.1$$

Extrapolating to N=20 rounds, there would be 4 unknown **h** values associated with each candidate key, so the overall search space is $2^8*(2^{32})^4$ = O(2^{136}). At a scan rate of 100msec that would take over 5.35e32 years.

4.4 Implementation Details

The information and the analysis provided above specifically focused on the underlying mathematics of the VMAC algorithm, as encapsulated by the Add-On Instruction shown in Figure 16. In this section we provide details on how to use the VMAC algorithm in a secure implementation. The secure implementation consists of two parts: the generation of outgoing messages and the processing of incoming messages. The secure implementation also addresses the following concerns:

- Protection against the standard replay attack, where a message between two nodes is saved and then later played again
- Protection against a variant of the replay attack where a message generated by one node intended for a second node is captured and played to a third node

- Determination of the key used in the VMAC algorithm, either **Ks** or **Ks'**, which is changed by the KEP and is used to provide a bumpless transfer during a key change

As stated previously it is possible to use the VMAC algorithm without the KEP, and to only have **Ks'**. However this should only be used in circumstances where the user has complete control of all the nodes in the system and the VMAC key is treated as confidential or secret. It is also still recommended that the key is updated at least once per year, a function provided by KEP.

"APPENDIX B. GRAPHICAL VMAC IMPLEMENTATION" provides sample source code from NODE1 of the proof of concept implementation. In the reference implementation each node generates a VMAC message to each other node thru multicast, however VMAC itself has flexibility for unicast or multicast messages. In a typical Rockwell RSLogix5000 PLC program, a task called "IO_Mapping" is created under the MainTask and is used to map I/O signals to internal signals, with the internal signals being used in a control logic task. The purpose of this logic is to reduce the scope of code changes if it is found during commissioning that something happened with the I/O such that the signal was not in an expected location and that the wiring can't be changed to match the original design. This happens relatively frequently during plant commissioning, often due to simple human error, and sometimes can't be fixed (particularly in hardwired I/O) because cables are not long enough to relocate. Therefore it seemed appropriate that the VMAC messages would also be considered part of the IO Mapping task. Each node in the proof of concept implementation has a "VO_NODE#" part which is the VMAC outgoing messages which are being multicast to each of the other nodes, and there is a “VI_NODE#” part which processes the incoming multicast messages from each node.

4.4.1 Outgoing Messages

Generating outgoing messages is easier than verifying incoming messages, but there are critical steps involved to ensure the message is generated accurately. The most important step is that the VMAC must be generated using a temporary memory space, and then transmitted into the outgoing send buffer due to the messages being sent asynchronously from the logic execution. Otherwise what will happen is that a message will be transmitted in the middle of the VMAC generation process, which will cause the receiving node to get an incorrect VMAC and generate a fault. In the proof of concept implementation, messages were transmitted every 20ms while the scan time of the total IO_Mapping routine was closer to 50ms and the generation of the outgoing message close to 8-9ms of that time. Without processing the VMAC in a buffer prior to transmit an error rate of every few seconds would result, frequently with several invalid VMACs in a row.

The second most important aspect of generating a VMAC is ensuring that the proper “header” data is incorporated into the VMAC data message. The header information is used to prevent the replay attacks described above. The structure of header data is as follows:

- *First 8 bytes: 64-bit Counter*
- *Next 4 bytes: 32-bit Node ID*
- *Next 4 bytes: 32-bit Destination ID*

- *Last 8 bytes: 64-bit Key ID (only used with KEP)*

The first value used in the header is a 64-bit counter, which on controller power up is initialized to the number of microseconds since epoch time. This counter is then incremented by 1 each time a new VMAC is generated, and essentially serves as a timestamp replacement. The controller's time itself does not have to be therefore synchronized to some higher level server, it just needs to be reasonably accurate (within a few minutes) and should never be reset to a time earlier than it was previously set to. However, since the counter is only incremented by 1 for each VMAC generation, and in the proof of concept it takes 8-9 ms to generate a VMAC it means that for each VMAC the counter increases only by 1 where the number of microseconds since epoch time would increase by thousands. This helps ensure that even during a power loss or a fresh download which could cause a disturbance in the clock time the VMAC counter will always be initialized at a value significantly higher than the previous value. This would hold true if even during a power loss the controller's clock does not increase such that when it powers up it still thinks the time is what it was at the last power loss.

The 32-bit Node ID and Destination ID are used in the message to indicate the source of the message and the intended recipient of the message. The Destination ID can be a code such as "234" used to indicate a group of nodes, or an IPv4 address that represents a unicast, multicast, or broadcast address. Similarly the Node ID can represent something as simple as a Node number (as used in the proof of concept) to a full IPv4 address. The purpose of these IDs is to ensure that someone does not take a message from one node and send it to a different set of recipients, thus preventing a variant of the replay attack.

The last 64-bits represent the VMAC Key ID, which indicates which **Ks** or **Ks'** was used to generate the VMAC. Note that it is not the value of the key itself, but rather a 64 bit code that could either be the hash of the key or a timestamp of the key. The proof of concept uses a timestamp value where the 64-bits represents the number of microseconds since epoch time at the time the key was generated, however a hash of the key could easily be substituted and used to verify that the correct key was transmitted, and if not then it could trigger a new **Ks'_REQ** in KEP to get a corrected copy of the key. In general, the VMAC Key ID is used by KEP to determine which key, either **Ks** or **Ks'** was used in VMAC.

Details will be provided in the description of KEP, but from a VMAC perspective KEP provides a mechanism by which a node will know if all nodes are reporting that they have the latest and greatest **Ks'**. If any nodes do not have the new **Ks'**, which likely occurs during the middle of key propagation, then VMAC uses **Ks** which represents a backed up copy of the old **Ks'**. Once KEP detects that all nodes have a copy of **Ks'** it instructs the node to use **Ks'** for all outgoing messages and embed the corresponding ID into the VMAC message. This will let a receiving node know whether to use the new key or the old key. Note that KEP will have already determined that the receiving node has a copy of the new key prior to instructing VMAC to use the new key to send the message.

4.4.2 Incoming Messages

The same principles for outgoing messages apply for incoming messages, but in reverse. To begin, an incoming message must first be copied into a buffer for processing since the messaging is asynchronous. Otherwise in the process of verifying one VMAC a new one could be written in which could cause the verification to improperly fail. After the message is copied into a buffer, the Counter is immediately checked to determine Iif it is greater than the last received value. If it is not then the message can be immediately discarded, since it is either an old message that has not yet been updated by the sending controller due to asynchronous processing or it is a replay attack. In general, if a counter is less than the last counter than the message is likely to be a replay attack, but if it is equal it could be just stale data.

The next step is to extract the VMAC Key ID from the message and determine if the VMAC was created using **Ks'** or **Ks**. Under most circumstances these values will be the same, except during the middle of a key update by KEP. KEP will have already determined that the receiving node has a copy of the new **Ks'** before it gets used in a VMAC from the sender node. KEP also has logic in it to monitor the VMAC verifications, and a node waits for all nodes to have successfully sent it a valid VMAC using the new **Ks'** before it sets **Ks** = **Ks'**, essentially discarding the old key and backing up the new key.

Once the correct key is determined the VMAC of the incoming data is calculated and then compared against the VMAC provided by the sending node. If the two values are equal, and the counter is greater than the previous counter, and the Node ID of the sender is correct, and the Destination ID is correct then the VMAC is flagged as "OK". The data for the message is then moved into a "data verified" buffer to be used for process control. Invalid data is not processed. The VMAC implementation for incoming messages includes 5 alarms for each message:

1. Invalid VMAC Alarm – if a node does not receive a valid VMAC message within a set time period then an alarm will be generated (recommend 250ms)
2. Replay Alarm – if a node does not receive a valid VMAC with a Counter greater than the last valid message within a set time period then an alarm will be generated (recommend 250ms), OR if a node receives a VMAC with a Counter less than the last valid message
3. Node Alarm – if the Node ID of the message is not what was expected then trigger an alarm (a 50ms debounce timer is recommended to avoid potential network errors)
4. Destination Alarm – similar to Node Alarm but uses the Destination ID vice the Node ID
5. Invalid Key Alarm – if a node receives a VMAC message with a key ID that does not match **Ks** or **Ks'** then generate an alarm

All alarms are cleared automatically on power up, but to be cleared during operation both the alarm condition itself must be cleared and an acknowledgement from an operator must be provided. Actions that an operator should take depend on the design of the system under question and the acceptable risk profile of that system. The decision tree is therefore beyond the scope of this thesis, however it should be understood that unlike IT systems sometimes it will make sense to continue operating using invalid data even in the presence of a cyber threat. For example, if a machine is providing life support to someone and the option is to either shut down or keep running under risk then it is probably better to keep running because that option *might* cause harm where shutting down will *definitely* cause harm. Control system design is also beyond the scope of this thesis, but it should be noted that a good design will take into account that sometimes a system component might need to run in a “standalone” mode due to equipment damage, plant maintenance, cyber threats, or a number of other reasons.

CHAPTER 5: COMPLEX MATH OPERATIONS

5.1 Introduction

As described in sections 3.3 and 3.4, two of the major limitations in developing a cryptographic system for control systems and embedded controllers is the inability to process BigIntegers natively, along with the overall reduced processing speed. These factors make it challenging to develop a system capable of performing the complex operations necessary to execute the mathematics required in performing the steps necessary for an Ed25519 key generation, signature generation, and signature verification. Note that this section is specific to Ed25519 as an overall proof of concept, but can be extended for Ed448 if security at the 224-bit level is required in the future.

This section describes the advanced Add-On Instructions (AOIs) that were developed as part of this project which were required to perform operations such as point multiplication that are so complex that a controller is unable to perform them in one scan. Therefore a new design approach was used, which was described in section 3.4. Much of the code here is based on work done in [29], which is an implementation of the Ed25519 elliptic curve digital signature scheme using 32-bit integers. The idea behind [29] was to first create an implementation in a higher level language (in this case C) that would be easier to write than in ladder logic. This would allow us to work out how to perform the math operations using 32 bit integers only first as a proof of concept, and then work out how to handle the limitations of processing speed and memory allocation.

The source code for the AOIs can be found at [28] and in Appendix C. The code consists of *.L5K text files which can be imported into a PLC processor or used as a reference to develop a hardware-based solution. This chapter describes the functional purpose of each of these AOIs. It provides the interface that is used in the main ladder logic code, and describes any aspects of their design, coding, or implementation that are uniquely noteworthy to that AOI.

Basic AOI functions like addition and subtraction are not described in this section, but the code for them can be found at [28]. A “basic” AOI is defined as any mathematical function which can be processed fully during one scan of the PLC, which makes it most like the C-based implementation because it does not require a large degree of special handling to work around the limitations of a PLC. As a result the coding is fairly straightforward with the most important aspect, how to handle BigIntegrers, already described in Section 3.3.

5.2 Custom Data Types

This section describes the various data types that were developed as part of this project. From a mathematical perspective these custom data types are invaluable as they represent the 256-bit values in accordance with the principals described in in Section 3.3. It should be noted that PLC ladder logic does have the ability to declare custom data types in a way similar to most other high level langauges, and this was done for the POINT data type. For the C, D, and E data types this was not done because these data types are really just different sized arrays of DINTs that are interpreted in different ways. Creating a custom data type for them in the PLC code would only add an unnecessary overhead with little to no value from an execution or an organizational perspective.

5.2.1 C DataType

A “C” data type is a 256-bit value stored as an array of twenty-two (22) DINTs in the little endian format. The DINTs from 0 to 21 each contain 12 bits of the value, with DINT[21] containing the highest 4-bits. This data type is typically used as the normal way of representing 256-bit value, for all operations except multiplication and exponentiation.

5.2.2 D Data Type

A “D” data type is either a 256-bit value that has been reduced to be within the finite field or a 512-bit value that requires reduction to be within the finite field. It consists of forty-three (43) DINTs in the little endian format. All DINTs contain 12 bits of the value except the highest DINT which contains only 8 bits. This data type is typically used in multiplication, where storage of the result requires a bit space equal to the sum of the bit-size of the two multipliers. This value is normally a temporary value that is reduced to fall within the modulus, thus using only the bottom 22 DINTs and thus becomes equivalent to the C data type.

5.2.3 E Data Type

An “E” data type is a 256-bit value stored in thirty-two (32) DINTs in the little endian format, where each DINT contains one byte of the value. This data type is used during exponentiation operations and is faster than using the C data type. Values stored in the E data type are typically converted back into the C data type.

5.2.4 POINT Data Type

A “POINT” data type consist of four C values that represent an Ed25519 curve point (x,y) stored in the extended coordinate system (X, Y, Z, T). Mathematical operations done on points use the POINT data type, however points are ultimately encoded into hex strings for transmission and storage. These strings are decoded back into the POINT data type when required for operations such as point multiplication and point addition.

5.3 Sequencers

In order to perform the complex mathematical operations such as point multiplication and modular inverse, the operation had to be broken up into a series of smaller steps which could then be executed sequentially. This is the most significant difference between the work done in [29] and what was required for this work in order to perform the complex mathematical operations on limited power devices. A generic sequencer algorithm was developed which would allow for various parts to then be executed in an order defined by the sequencer chart for that operation. The sequencer charts can be found in Appendix D, where the top row of the charts contains the various parts that could be executed by the sequencer. The “x” marks in the chart indicate which of these steps is being executed in which step, and form the command value which is then stored in the program. In many ways this process is similar to the way firmware is developed for embedded processors in common commercial electronic devices such as BluRay players.

Each sequencer contains control logic which looks at the command table and determines which actions need to be taken. The actions are then processed in order, and when completed a “DONE” flag is set. Most of the actions are one shot actions which are completed within one scan. The sequencer determines that all actions are complete by examining a mask of the word containing all of the DONE flags with a word containing all of the commands. When all actions are determined to be complete the sequencer advances to the next step, resets the DONE flags, and extracts a new series of commands. This process continues until all steps in the logic are complete. Note that in most cases the next step in the sequencer is the result of taking the current step and incrementing that step by one, but in some cases loops are formed within the sequencer by setting the next step equal to some previous step until some loop counter **i** (for example) reaches an intended value.

Some of the most complex operations, such as the Ed25519_SIGN operation, contain multiple sequencers within the overall sequencer. Since each sequencer has a SEQCTRL_DN flag which represents all steps in the sequencer having completed execution it is possible to embed one sequencer within another. The host sequencer then watches the SEQCTRL_DN flag of the embedded sequencer and uses that to trigger the command’s DONE flag of the host sequencer. Logic is also provided that initializes the sequencers properly in the event of a power loss or some other interruption that might occur during mid execution. It is also possible to pause the operation of a sequencer mid-execution and then resume it later, provided the memory areas being accessed by the sequencer are not manipulated during the pause.

5.4 Temp Data

One of the features of most high-level software languages such as C++ is that you can create temporary variables in routines and objects that can be used to store and process data. These variables can have descriptive names and be easily discarded (along with the associated memory space) when execution is complete. PLCs and other embedded controllers do not have this capability, since they are ultimately much more similar to hardware than to software. As a result it is necessary to declare and reserve memory space in the ladder logic code explicitly, which can then be used for temporary variables. For clarity these temporary variables have been given names like "Temp_C1", indicating that it is the first "C" type temporary variable used in the program.

Each of the AOIs properly initializes the temporary space, since it is assumed that the space may still have garbage data leftover from a previous instruction. This means that the temporary variables can be used over and over again, with the catch that the same temporary variables (i.e. the same memory space) can't be used by two different instructions at the same time. In many ways this is similar to how standard memory allocation works in a language like C/C++, but more explicit given that a ladder logic implementation is much more similar to hardware than software.

5.5 AOIs

This section describes the AOIs in more detail. In general an AOI format will be a block shown with a blue border. A description of the block will be shown at top. The first field of the block with an input will be the name of the block where a "control" field is added. This is more of a Rockwell Software convention which is used to differentiate between multiple instances of the same AOI. For this section the control block will have the same name as the AOI itself. The AOIs then begin with the temporary variables, followed by the input variables and lastly with the output variables.

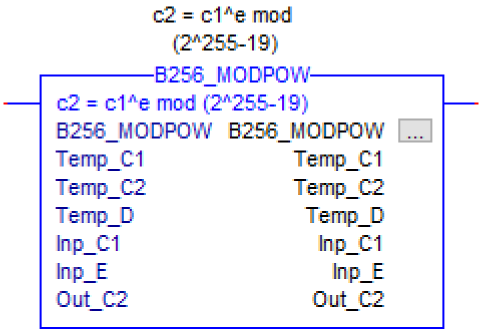
During development it was determined that Rockwell actually provides 3 different types of fields: an input, an output, and an InOut. The input and output fields are fairly limited in the data types that can be assigned to them. The InOut field is extremely flexible, and therefore was used to pass just about every data type except for the most primitive such as an INT or DINT.

5.5.1 B256_MODPOW

The "B256_MODPOW" AOI is an add-on instruction designed to compute the modular exponentiation of a 256-bit number where the modulus is the underlying finite field of the Ed25519 curve (i.e. 2^{255}-19). The mathematical description of the AOI is:

c2 = c1^e mod (2^255-19)

The figure below shows the AOI interface:



c2 = c1^e mod
(2^255-19)

B256_MODPOW

c2 = c1^e mod (2^255-19)

B256_MODPOW	B256_MODPOW
Temp_C1	Temp_C1
Temp_C2	Temp_C2
Temp_D	Temp_D
Inp_C1	Inp_C1
Inp_E	Inp_E
Out_C2	Out_C2

Figure 20: B256_MODPOW Add-On Instruction

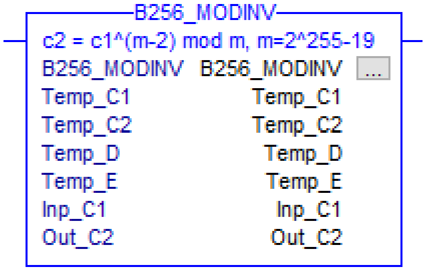
B256_MODPOW works by initializing the temporary variables and then for each bit in the exponent byte performing a multiplication. If the bit in the exponent is a 1 then the multiplication updates the output using input data. If the bit in the exponent is a 0 then the multiplication simply takes the current value and multiplies it by 1. This is done in order to help protect against timing attacks.

5.5.2 B256_MODINV

The “B256_MODINV” AOI is an add-on instruction designed to compute the modular inverse of a 256-bit number where the modulus is the underlying finite field of the Ed25519 curve (i.e. 2^{255}-19). The mathematical description of the AOI is:

$$c2 = c1{}^\wedge(m-2) \bmod m,\ m=2{}^\wedge 255-19$$

The figure below shows the AOI interface:



B256_MODINV
c2 = c1^(m-2) mod m, m=2^255-19
B256_MODINV B256_MODINV ...
Temp_C1 Temp_C1
Temp_C2 Temp_C2
Temp_D Temp_D
Temp_E Temp_E
Inp_C1 Inp_C1
Out_C2 Out_C2

Figure 21: B256_MODINV Add-On Instruction

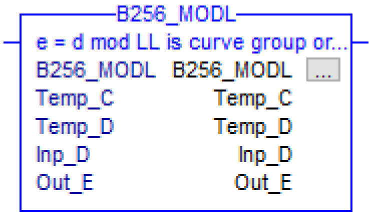
The modular inverse is calculated almost identically to performing the modular exponentiation. The main difference is that the exponent is no longer an input, but instead is set to a predefined value *m-2* which results in the inverse.

5.5.3 B256_MODL

The “B256_MODL” AOI is an add-on instruction designed to compute the modulus of a 256-bit number where the modulus is the group order L (i.e. 2^{252} + 27742317777372353535851937790883648493) of the Ed25519 curve, and to store that result in an E value for future use. The mathematical description of the AOI is:

$$e = d \bmod L$$

The figure below shows the AOI interface:



B256_MODL
e = d mod LL is curve group or...
B256_MODL B256_MODL ...
Temp_C Temp_C
Temp_D Temp_D
Inp_D Inp_D
Out_E Out_E

Figure 22: B256_MODL Add-On Instruction

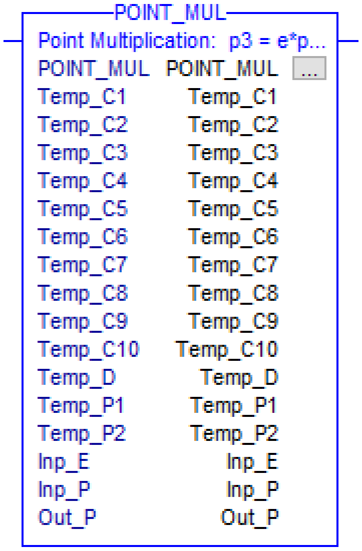
This AOI is used in signature generation and verification for Ed25519 as shown in Section 2.7.

5.5.4 POINT_MUL

The “POINT_MUL” AOI is an add-on instruction designed to compute the result of multiplying a point on an Ed25519 curve with a scalar to compute a new point on the curve. The mathematical description of the AOI is:

$$p3 = e*p1$$

The figure below shows the AOI interface:



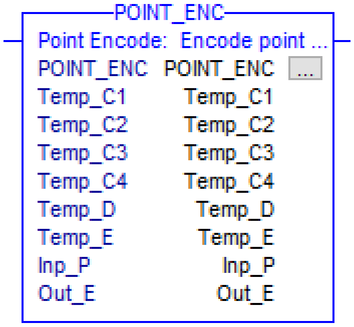
```
POINT_MUL
Point Multiplication:  p3 = e*p...
POINT_MUL   POINT_MUL  ...
Temp_C1       Temp_C1
Temp_C2       Temp_C2
Temp_C3       Temp_C3
Temp_C4       Temp_C4
Temp_C5       Temp_C5
Temp_C6       Temp_C6
Temp_C7       Temp_C7
Temp_C8       Temp_C8
Temp_C9       Temp_C9
Temp_C10     Temp_C10
Temp_D         Temp_D
Temp_P1       Temp_P1
Temp_P2       Temp_P2
Inp_E           Inp_E
Inp_P           Inp_P
Out_P           Out_P
```

Figure 23: POINT_MUL Add-On Instruction

This AOI is used to perform digital signature generation, signature verification, and to generate temporary symmetric key pairs in KEP that are used to encrypt/decrypt **Ks'** to be sent between nodes. Point multiplication is primarily the result of performing successive point additions along a curve, one for each 1 bit in the exponent. The recommendations found in [27] are followed, where the points are represented in extended homogeneous coordinates (X, Y, Z, T) with x = X/Z, y = Y/Z, x*y = T. This operation is usually performed by initializing Z to be 1 and then setting X = x, Y = y, and calculating T. Reference [27] then specifies a generic option for adding two points and a faster implementation for doubling a point. In general the AOI performs point doubling as a specific type of point addition, so this is followed as much as possible. Each point doubling operation saves two 256-bit subtractions, one 256-bit addition, one 256-bit multiplication, and substitutes three 256-bit squares for three 256-bit multiplications (a square is faster than a multiplication). This resulted in about a 15% timing efficiency improvement when run on the RSLogix 1756-L83 series PLCs.

5.5.5 POINT_ENC

The "POINT_ENC" AOI is an add-on instruction designed to take a POINT represented in extended homogenous coordinates (X, Y, Z, T) and encode it as an E value (i.e. an array of 32 bytes). The instruction stores the value in the little endian format, and is often combined with other instructions in KEP to turn the point into a STRING for transmission. This is particularly true when performing the Diffie-Hellman key exchange portions of KEP. The figure below shows the AOI interface:



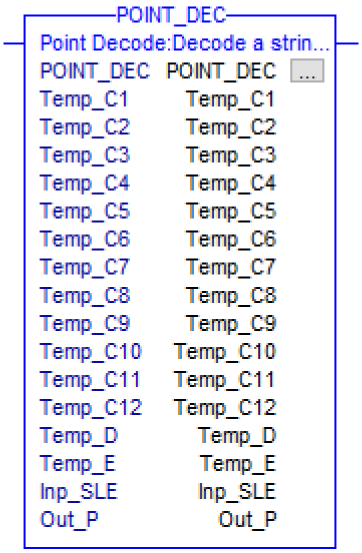
POINT_ENC
Point Encode: Encode point ...

POINT_ENC	POINT_ENC
Temp_C1	Temp_C1
Temp_C2	Temp_C2
Temp_C3	Temp_C3
Temp_C4	Temp_C4
Temp_D	Temp_D
Temp_E	Temp_E
Inp_P	Inp_P
Out_E	Out_E

Figure 24: POINT_ENC Add-On Instruction

5.5.6 POINT_DEC

The "POINT_DEC" AOI is an add-on instruction designed to take a STRING representing an encoded point (x,y) stored in the little-endian format and decode it to a point in extended homogenous coordinates (X, Y, Z, T). Typically these STRING values representing an encoded point come from another node as part of KEP when performing the ECDH key exchange to create a shared symmetric key, which is then used to encrypt/decrypt **Ks'**. The figure below shows the AOI interface:

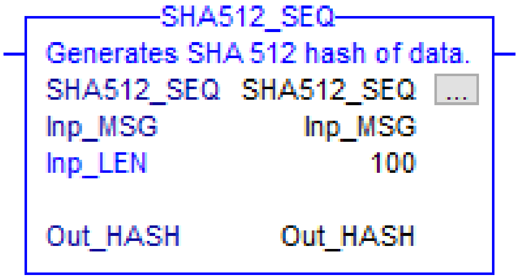


POINT_DEC
Point Decode:Decode a strin...
POINT_DEC POINT_DEC ...
Temp_C1 Temp_C1
Temp_C2 Temp_C2
Temp_C3 Temp_C3
Temp_C4 Temp_C4
Temp_C5 Temp_C5
Temp_C6 Temp_C6
Temp_C7 Temp_C7
Temp_C8 Temp_C8
Temp_C9 Temp_C9
Temp_C10 Temp_C10
Temp_C11 Temp_C11
Temp_C12 Temp_C12
Temp_D Temp_D
Temp_E Temp_E
Inp_SLE Inp_SLE
Out_P Out_P

Figure 25: POINT_DEC Add-On Instruction

5.5.7 SHA512_SEQ

The “SHA512_SEQ” AOI performs the functions of the SHA512 algorithm. The figure below shows the AOI interface:



SHA512_SEQ
Generates SHA 512 hash of data.
SHA512_SEQ SHA512_SEQ ...
Inp_MSG Inp_MSG
Inp_LEN 100
Out_HASH Out_HASH

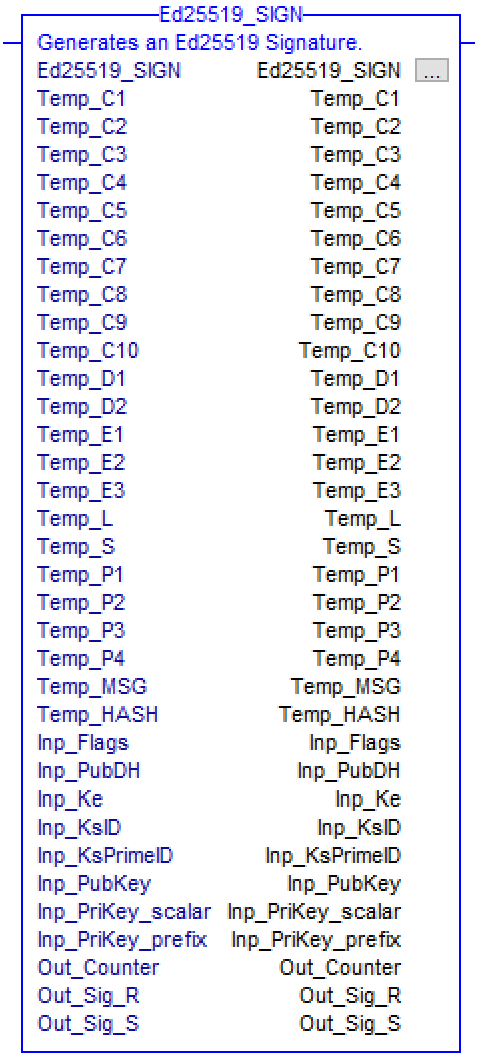
Figure 26: SHA512_SEQ Add-On Instruction

The regular SHA512 algorithm is capable of operating over a variable length of bits, which is not possible in a PLC. A PLC is also not capable of piping in the message as a bit or byte stream. Therefore the message is stored as an array of LINT, with a maximum capacity of up to 1600 LINTs. The Inp_LEN parameter then defines how many of the

LINTs are used out of the possible 1600 to contain the message. Since SHA512 is a Θ(n) algorithm the length of time for the sequencer to complete is dependent on the length of the message. In practice, SHA512_SEQ is only used in KEP for signature generation and verification which has a fixed length input as shown in Section 2.7. The output is stored in an array of eight LINTs, where LINT[0] thru LINT[7] represent parts *a* thru *h* of the hash respectively.

5.5.8 Ed25519_SIGN

The "Ed25519_SIGN" AOI creates a digital signature of a message using the Ed25519 curve via the process shown in Section 2.7.2. This AOI, along with its partner "Ed25519_VERIFY" are the most complicated AOIs mathematically and contain several of the other AOIs listed above. The implementation is capable of processing a digital signature on an RSLogix5000 PLC in approximately 3 minutes with a scan time impact of less than 10ms. The figure below shows the AOI interface:



Ed25519_SIGN

Generates an Ed25519 Signature.

Ed25519_SIGN	Ed25519_SIGN	...
Temp_C1	Temp_C1	
Temp_C2	Temp_C2	
Temp_C3	Temp_C3	
Temp_C4	Temp_C4	
Temp_C5	Temp_C5	
Temp_C6	Temp_C6	
Temp_C7	Temp_C7	
Temp_C8	Temp_C8	
Temp_C9	Temp_C9	
Temp_C10	Temp_C10	
Temp_D1	Temp_D1	
Temp_D2	Temp_D2	
Temp_E1	Temp_E1	
Temp_E2	Temp_E2	
Temp_E3	Temp_E3	
Temp_L	Temp_L	
Temp_S	Temp_S	
Temp_P1	Temp_P1	
Temp_P2	Temp_P2	
Temp_P3	Temp_P3	
Temp_P4	Temp_P4	
Temp_MSG	Temp_MSG	
Temp_HASH	Temp_HASH	
Inp_Flags	Inp_Flags	
Inp_PubDH	Inp_PubDH	
Inp_Ke	Inp_Ke	
Inp_KsID	Inp_KsID	
Inp_KsPrimeID	Inp_KsPrimeID	
Inp_PubKey	Inp_PubKey	
Inp_PriKey_scalar	Inp_PriKey_scalar	
Inp_PriKey_prefix	Inp_PriKey_prefix	
Out_Counter	Out_Counter	
Out_Sig_R	Out_Sig_R	
Out_Sig_S	Out_Sig_S	

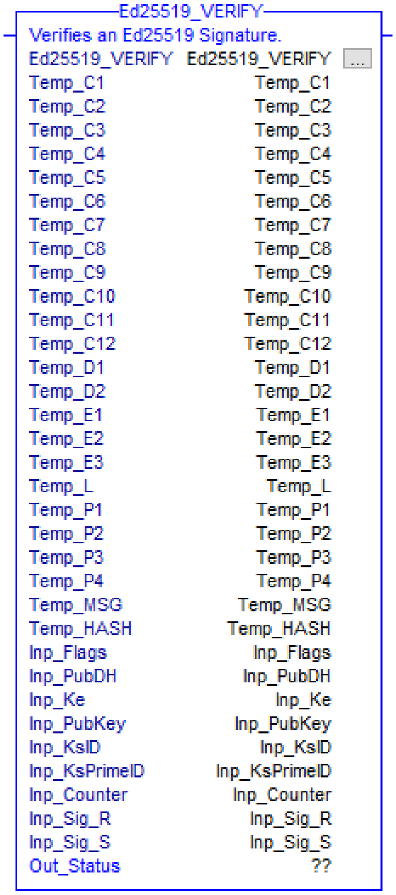
Figure 27: Ed25510_SIGN Add-On Instruction

This AOI was designed for signing messages specifically for KEP, and all the parameters listed as inputs are ultimately piped into an array of LINTs that then are piped into the SHA512_SEQ AOI. It is possible to modify the AOI to make it more generic, and to perform the message alignment outside of the AOI. However this interface improves the overall readability of the KEP code, and ultimately this AOI is only used in KEP. The discussion of the parameters used as part of the message to be signed is discussed further in Section 6.2, which describes the KEP message structure. However, three inputs here are the private and public keys which are not part of the message structure. Those values are:

- Inp_PubKey: a string representing an encoded point and generated offline via a python script or some other equivalent means (see Appendix E)
- Inp_PriKey_scalar: a string representing a scalar and generated offline in the same way as Inp_PubKey
- Inp_PriKey_prefix: a string representing representing part of the private key and generated offline in the same way as Inp_PubKey

5.5.9 Ed25519_VERIFY

The "Ed25519_VERIFY" AOI verifies a digital signature of a message using the Ed25519 curve via the process shown in Section 2.7.2. The implementation is capable of verifying a digital signature on an RSLogix5000 PLC in approximately 3 minutes with a scan time impact of less than 10ms. Like "Ed25519_SIGN" the AOI is designed to process messages and digital signatures specifically used by KEP, and outputs a status of 1 for a successfully verified message, 666 for a message that has failed verification, and 0 for a verification in progress. The figure below shows the AOI interface:



Ed25519_VERIFY

Verifies an Ed25519 Signature.

Ed25519_VERIFY	Ed25519_VERIFY
Temp_C1	Temp_C1
Temp_C2	Temp_C2
Temp_C3	Temp_C3
Temp_C4	Temp_C4
Temp_C5	Temp_C5
Temp_C6	Temp_C6
Temp_C7	Temp_C7
Temp_C8	Temp_C8
Temp_C9	Temp_C9
Temp_C10	Temp_C10
Temp_C11	Temp_C11
Temp_C12	Temp_C12
Temp_D1	Temp_D1
Temp_D2	Temp_D2
Temp_E1	Temp_E1
Temp_E2	Temp_E2
Temp_E3	Temp_E3
Temp_L	Temp_L
Temp_P1	Temp_P1
Temp_P2	Temp_P2
Temp_P3	Temp_P3
Temp_P4	Temp_P4
Temp_MSG	Temp_MSG
Temp_HASH	Temp_HASH
Inp_Flags	Inp_Flags
Inp_PubDH	Inp_PubDH
Inp_Ke	Inp_Ke
Inp_PubKey	Inp_PubKey
Inp_KsID	Inp_KsID
Inp_KsPrimeID	Inp_KsPrimeID
Inp_Counter	Inp_Counter
Inp_Sig_R	Inp_Sig_R
Inp_Sig_S	Inp_Sig_S
Out_Status	??

Figure 28: Ed25519_VERIFY Add-On Instruction

5.5.10 PRNG

The “PRNG” AOI creates a pseudo-random number and is based loosely on the Hash_DRBG algorithm described in NIST SP 800-90A [37]. Ideally the PRNG AOI would be replaced with a hardware-based true random number generator, but as discussed in Section 3.2 that is currently not an option. The figure below shows the AOI interface:

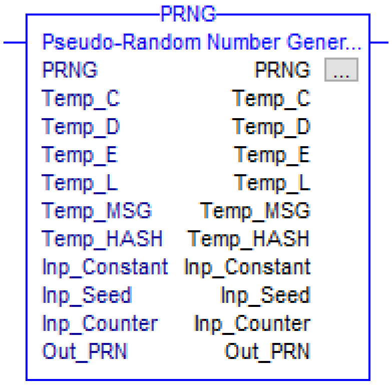


Figure 29: PRNG Add-On Instruction

Any secure PRNG can be used in KEP, and if a PRNG is proven to not be secure it can be easily replaced. For this work the PRNG algorithm works as shown below, however it should be noted that a detailed security proof for this has not been developed as part of this work. Note that both the constant and the seed should be initialized to random numbers using a true random number generator, and that this can be done offline.

<u>Step 1</u>: Build the base input as the following:

- LINT[0]: The current timestamp as a 64-bit value representing the number of nanoseconds since epoch time
- LINT[1]: A 64-bit counter that is incremented with each use of the PRNG
- LINT[2]: A 64-bit constant value that does not change
- LINT[6 thru 18]: The seed which is a 832-bit value stored as 13 LINTs that will be changed as described below with each execution of this algorithm

Note that by most conventions the "seed" would be considered a constant value and not something that would change. However this implementation follows the approach taken in NIST SP 800-90A which uses a constant value and then a seed which changes as part of the PRNG execution [37].

Step 2: Using the SHA512_SEQ AOI, compute the hash of the input and output the left part of the hash result as a 256-bit pseudo-random number.

Step 3: Update the seed for the next execution of the algorithm using the following algorithm:

- Add the *e* part of the hash to the first 64-bits of the seed (*Inp_Seed[0]*), and then do a circular right shift rotation by 1 bit.
- Add the *f* part of the hash to the next 64-bits of the seed (*Inp_Seed[1]*), and then do a circular right shift rotation by 2 bit.
- Add the *g* part of the hash to the next 64-bits of the seed (*Inp_Seed[2]*), and then do a circular right shift rotation by 3 bit.
- Add the *f* part of the hash to the next 64-bits of the seed (*Inp_Seed[3]*), and then do a circular right shift rotation by 4 bit.
- For n=4 to 11
    - Add Inp_Seed[n-4] to Inp_Seed[n]
    - Do a *n+1* bit circular right shift rotation on the result
- *Inp_Seed[0]* to *Inp_Seed[12]* and then do a 13-bit circular right shift rotation
- Swap Inp_Seed[0] and Inp_Seed[12]

CHAPTER 6: KEY EXCHANGE PROTOCOL

6.1 Introduction

A multicast group can be efficiently protected through use of a single symmetric key which can then be used to encrypt traffic between nodes or to provide message authentication and verification using algorithms such as VMAC. The challenge comes in generating the symmetric key and transmitting that key to the other nodes. As stated in Section 4.1 it is possible to use VMAC without KEP, however only in the case when the system designer has complete control of every node in the control system and does not have to interface with a 3rd party vendor. In practice this is unlikely, since control systems are often distributed, with a centralized "overarching" system that then interfaces to a number of controllers contained in panels that come with various machines produced by OEMs to provide local control. Therefore KEP provides a means by which a distributed control system with components by different OEMs can interface to each other and update the symmetric key used by VMAC. Each vendor would generate an Ed25519 public/private key pair using a process similar to the script provided in Appendix E, and then the vendors would exchange the public keys as part of the Interface Design Documentation. These keys can then be used to generate the symmetric key.

Section 3.5 described the main approaches to generating and handling the symmetric key in multicast groups, but these approaches have additional complications when applied to control systems. The limited processing power is the primary challenge which impacts each of the multicast solutions in a number of ways, primarily that they can not process the number of steps that would be required to update keys each time group membership changes while at the same time meeting performance requirements.

Additionally there are significant challenges that stem from how message structures and send rates work (Section 3.7), since most of these algorithms require that the system send out specialized “one-time” messages as part of the group join and leave processes. An implementation of these existing algorithms which must transmit the control messages used to exchange the symmetric key would likely suffer performance degradation.

Thankfully, the design of control systems does mean that the requirements for multicast messaging are more limited than the requirements for the generalized multicast group scenarios that are addressed in [30, 32, 33]. The rate of group membership changes is relatively low and is primarily the result of equipment or power failures where there is a desire to rejoin the node to the group as speedily as possible. This eliminates the need to change the group key solely based on group membership changes which is a primary driver for the approaches taken in the literature. However this does add a unique challenge in that a node must be capable of being restored to the group in near instantaneous time even during the middle of a group key change. Even an algorithm capable of updating a key with an *O(log n)* efficiency is not going to be fast enough given the inherent limitations of processing speed. This is commonly referred to in control systems engineering as a “bumpless transfer” and is not addressed in the literature simply because it is not a requirement for an information system.

To solve these challenges the Key Exchange Protocol (KEP) for control systems was developed. The primary purpose of KEP is to securely generate and transmit a **Ks’**, which serves as the primary shared symmetric key, along with the identity number of the **Ks’** and the **Ks** currently in use. These identity numbers are 64-bit numbers that represent the number of microseconds since epoch time at the time these were generated.

During a key exchange a new **Ks’** is created and distributed, however the old **Ks** is used in VMAC until a node confirms that all nodes it communicates with have received **Ks’** via the identity number. The nodes then begin to transmit messages using **Ks’** vice **Ks**. When a node receives valid VMAC messages from each node, which were generated using **Ks’**, the node sets **Ks** equal to **Ks’**.

KEP consists of two parts, the Listener and the Processor, that run in parallel with each other. The Listener is responsible for receiving messages from other nodes and verifying the messages of these nodes. The Processor is a state-machine that uses verified messages from the Listener along with internal data to perform the bulk of the KEP logic. KEP operates using a client-server model, however unlike the Iolus framework [30] which uses GSAs, each node in KEP can act as both a client and a server depending on the state of the node. This is done by each node having an internal “priority table” for all of the other nodes. A node finds the highest priority node that is advertising that it has a **Ks’** available for distribution and selects that node as its server to obtain the key. It then goes through an Ed25519 ECDH process to generate a one-time symmetric key with the server node, with that key known as **Kp**. **Kp** is then used to encrypt a copy of **Ks’** via an XOR operation, with the encrypted key known as **Ke**. **Ke** is then transmitted from the server node to the client node and subsequently decrypted using the client’s copy of **Kp** to obtain **Ks’**.

Since KEP, and control systems in general, do not require the symmetric key (i.e. **Ks’**) to change each time a node leaves and joins the group the efficiency of the key exchange is *Θ(1)***.** Also note that since the key does not change due to a node dropping out of a group, a power loss and subsequent power restore does not require a node to go through the key exchange process before it can begin communicating. Even if a key update has begun while a node is down, the key update will wait for the node to be restored and obtain a copy of the new **Ks’** before the current **Ks** is abandoned. This means that the node which experienced the failure and subsequent restoration will be able to use **Ks** to immediately authenticate and verify messages, meeting overall control system performance requirements.

Node priorities and trust relationships are established as part of the control system design. The trust relationships are established by each node having a copy of the Ed25519 public keys of the other nodes. The priorities are set by entering values into an internal table of the node, with “1” being the highest priority. Note that it is not necessary for all nodes to have the same priorities, i.e. if there are four nodes it is not required that each node consider the same node (say Node1) to be the highest priority node. This means that it is possible to use KEP in a tree configuration with a root node and have other nodes be branches and leaves of the tree. This improves the speed at which a key change propagates throughout the tree, with KEP having a key change efficiency of *O(log n).*

In general there is a root node for the KEP tree, however the root node does not necessarily have to be the node with the highest priority at all times. If the highest priority node “Node1” is down for some reason then the next highest priority node will take over as the root. This “Node2” can even perform a key update which will then propagate thru the rest of the tree. When “Node1” is restored it will perform the same function as any other leaf on the tree by examining the existing nodes to determine the highest priority node that is active and is advertising a **Ks’** available. “Node1” will then work to obtain a copy of the **Ks’** from “Node2”, and once that is obtained “Node1” will start to advertise itself as having the key available. Since it is the highest priority node it will automatically retake the position as the tree root.

It should be noted that when a node is first powered on it will enter a “powerup” state in the KEP Processor state machine that will clear **Ks’** from the node, with the assumption that **Ks’** might have changed while the node was down. However the node can continue to use **Ks** to authenticate packets because **Ks** does not get set equal to **Ks’** until a node has verified that all other nodes within its immediate vicinity in the tree are using the new **Ks’**. As a result KEP provides bumpless transfer. It is not suitable for scenarios where group membership in a multicast scheme is dynamic, but it is very efficient where the group membership is relatively static.

In general, we recommend that KEP be used to trigger a key update once per year in order to assure the validity of the VMAC key. This can be done on an automatic cycle by adding logic so that the root node triggers the update automatically when a timer expires, or can be done thru an message request sent by another node. Control engineers can further tailor KEP to only allow designated nodes, perhaps a particular administration console, to be allowed to instruct the root node to perform a key update. Note that the command to perform a key update is contained within a digitally signed message.

The sections below describe the KEP Listener and Processor in detail, beginning with a description of the message structure used in KEP which is digitally signed using the Ed25519 curve (refer to Section 5.5.8). It then continues by defining the flags that are used in KEP. The Listener is then described in detail, followed by the Processor and the various states that make up the processor. The state machine diagram is presented, and the chapter concludes by illustrating the paths of the state machine during a key generation and exchange. Appendix C contains source code from the proof of concept implementation.

6.2 KEP Message Structure

As described in Sections 5.5.8 and 5.5.9, KEP uses the Ed25519 curve to digitally sign and verify a specific message structure which contains the various control data. The message structure is as follows:

- Inp_FLAGS: a 32-bit number containing various control flags
- Inp_PubDH: a 64 character string containing hexadecimal values representing an encoded point
- Inp_Ke: a 64 character string containing hexadecimal values representing **Ks'** encrypted with a unique one-time symmetric key known as **Kp** that is generated via an ECDH key exchange
- Inp_KsID: a LINT representing the ID of the node's **Ks**
- Inp_KsPrimeID: a LINT representing the ID of the node's **Ks'**
- Out_Counter: a LINT value representing a timestamp of the message, and represents the number of microseconds since epoch time when the message was generated
- Out_Sig_R and Out_Sig_S: two 64 character strings containing hexadecimal values that combined represent the digital signature

Note that PLCs in general do not have a good way of easily transmitting large numbers. Therefore it was determined that sending a string of hexadecimal characters is one of the most efficient ways to send the value over the wire. It is possible to send a string of Base64 encoded characters, but this adds extra processing time and does not improve message size efficiently significantly. A string of ASCII characters is unusable because a PLC uses the apostrophe character to denote the beginning and end of a string. If a digital signature was encoded in ASCII characters the encoding might result in an apostrophe appearing in the middle of the string, resulting in the PLC cutting off a portion of the string leading to an invalid signature.

The table below defines the flags. Note that most of the 32-bits are spares for potential status indications to provide additional details or diagnostics information that may be desired in a future implementation, such as a KEP version number.

Bit	Flag Name	Usage Description
0	All OK	Indicates that node believes its overall health is OK, and that there are no internal errors that the node is detecting within itself.
7	Ks’ Request	Requests that a node transmit a copy of the Ks’
8	Create New Ks’ Request	Requests that a node create a new Ks’, will only be processed by a node that believes it is a server
9	Ks’ Available	Status that this node has a Ks’ that exists and that either matches the node’s server, in the case where the node believes another node is the server
12	Ks = Ks’ Status	Status that this node has Ks = Ks’
14	Key Server Yes	Status that this node believes it is a server
15	Key Server No	Status that this node believes it is a client
18	Ks ID Matches Server	Status that this node’s Ks ID matches the Ks ID of the node that it believes is the server
19	Ks’ ID Matches Server	Status that this node’s Ks’ ID matches the Ks’ ID of the node that it believes is the server
20	Ks ID Matches Me	Status that this node’s Ks ID matches the Ks ID of the node receiving the message (based on the last message this node received from that node)
21	Ks’ ID Matches Me	Status that this node’s Ks’ ID matches the Ks’ ID of the node receiving the message (based on the last message this node received from that node)

Figure 30: KEP Message Flags

6.3 Listener

The Listener runs in parallel with the Processor, and is responsible for verifying incoming messages from different nodes. As mentioned previously, PLCs and other embedded controllers continually send messages, so the detection of a new message to be verified is not based on the presence of a message but rather a change in the message’s content. Each of the messages being sent to the controller from the various nodes are examined for a change in the content (note that a counter change only counts if the new value is greater than the last value that was previously verified). If a change in the content is detected the message is added to a verification queue.

The Listener is continually examining messages for new content, while at the same time verifying the messages that have been added to the verification queue one at a time. If a new message comes in for a node while the message for that node is being verified the new message must wait until the current verification is complete before the new one can be added to the queue. However if a new message comes in for a node that is not currently undergoing verification than the new message is added to the queue and the old message along with its position in the queue is discarded. This protects the Listener against the case where one node is misbehaving, potentially due to a hardware or networking failure, resulting in the node continually sending invalid messages. The remaining nodes will be able to operate without any issue. Also note that if the failing node is sending the **same** invalid message over and over again the verification will only be performed once on the invalid message, which helps protect the Listener from consuming unnecessary processing resources.

An attacker could potentially attempt to perform a denial of service attack by flooding the Listener with false messages that appear to be from the different nodes. As a result the KEP as a whole would essentially idle because it would continually detect a series of unverifiable messages, which would then trigger an alarm to note that there appears to be misbehavior of the system for further investigation. Since a controller sends out messages continually, with a fixed message size and fixed message rate, detecting this denial of service attack on a network is a relatively trivial task by analyzing the bandwidth of messages being sent to the node. Essentially, if the bandwidth utilization is above normal and you are getting alarms then you are under a denial of service attack.

Since KEP works by only switching out **Ks** after it successfully detects that all nodes have **Ks'** the KEP can essentially idle indefinitely until such an attack is resolved. So long as the attack is resolved before a key expires the overall control system data authentication and verification component remains unaffected. This is particularly true for VMAC, where the analysis in Section 4.3.3 shows that there is significant time before a key expires.

The efficiency of the Listener is *O(n)*, as it is possible that at any given moment every node may be sending a new message to a node. The speed of the Listener is ultimately dictated by the time in which it takes the Ed25519_VERIFY AOI to verify a message (see Section 5.5.9).   The scan time impact of the Listener is dependent on the scan time impact of the Ed25519_VERIFY AOI, and is generally less than 1ms when no verification is being performed on a Rockwell 1756-L8x processor.

6.4 Processor

The Processor is a state machine consisting of a total of ten states numbered from 0 to 9. Each state performs a sequence of logical functions designed to perform a specific task. Once those functions are preformed the state machine than transitions to a new state, performing those functions in the new state. This process repeats continually, resulting in a state machine that is in constant motion. The design of the Processor was inspired by a typical CPU processor, in which instructions are read from memory, the instruction is processed, actions are taken, and then the process repeats for the next set of instructions.

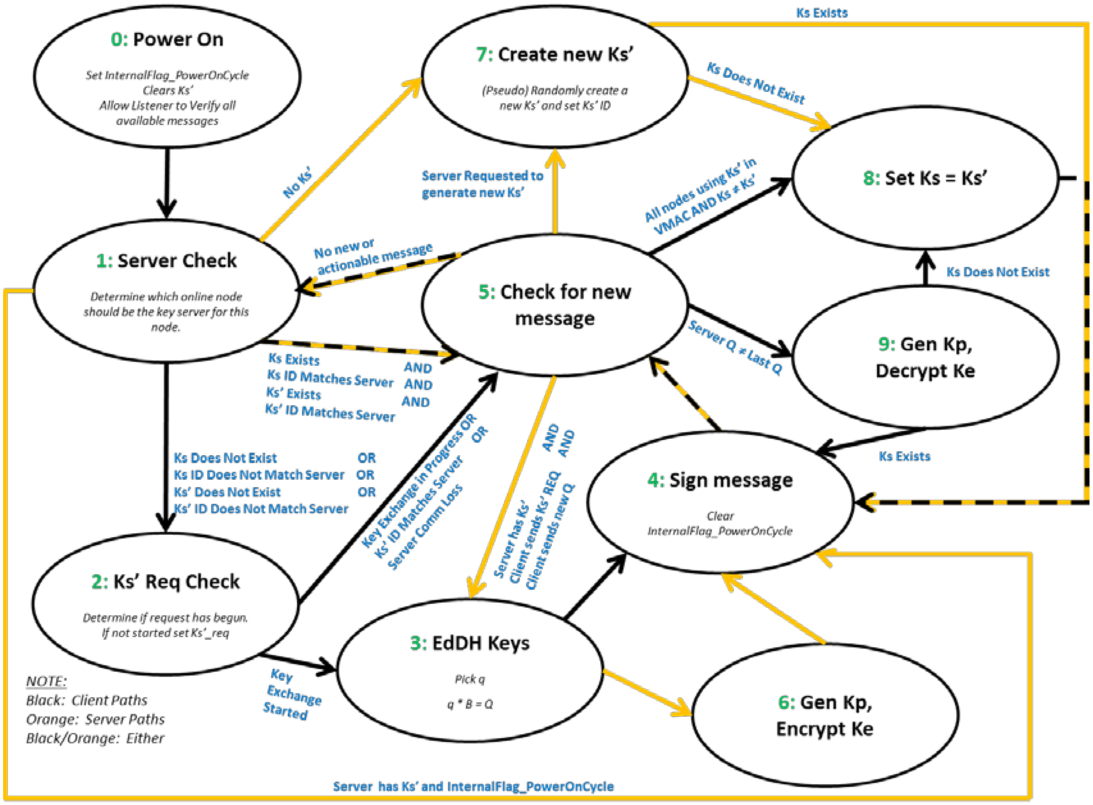


Figure 31: Processor State Machine Diagram

Figure 31 above shows the Processor State Machine Diagram. The green numbers indicate the state number, the black arrows indicate transitions only made by a node that has not determined it is a server (i.e. client side transitions), and the orange arrows indicate transitions only made by a node that has determined it is a server (i.e. server side transitions). Transitions indicated by black/orange arrows can be made by either clients or servers. The blue text indicates the conditions that dictate which state is next after the current state completes its functions. Arrows without blue text simply indicate that there is only one possible option for the next state. For example, state 4 is always followed by state 5, however there are multiple options for the next state after state 5. The sections below describe the functions that are performed in each state.

During KEP operation there is significant discussion on “server” nodes versus “client” nodes. It is important to remember that these are just roles that a node takes with respect to a different node. A node can be a server for one node and a client to a different node at the same time. Therefore a node is exclusively limited to just the “black” or just the “orange” transitions at any given time.

6.4.1 State 0: Power On

State 0 is known as the “Power On” state because this is the state that all nodes begin operation when power is freshly applied. The sequence chart for this state is shown in Figure 32. The state begins by clearing the identity of the key server node which will then be freshly determined in state 1. The state then goes into idle for a period of time (20 minutes was used in the proof of concept) to allow the Listener (which is running in parallel of the processor) to examine incoming messages from the various nodes. After the timer expires the state saves a copy of the current public Diffie-Hellman values being advertised by the various nodes as old messages, and sets an internal flag indicating that the node has just left the Power On state.

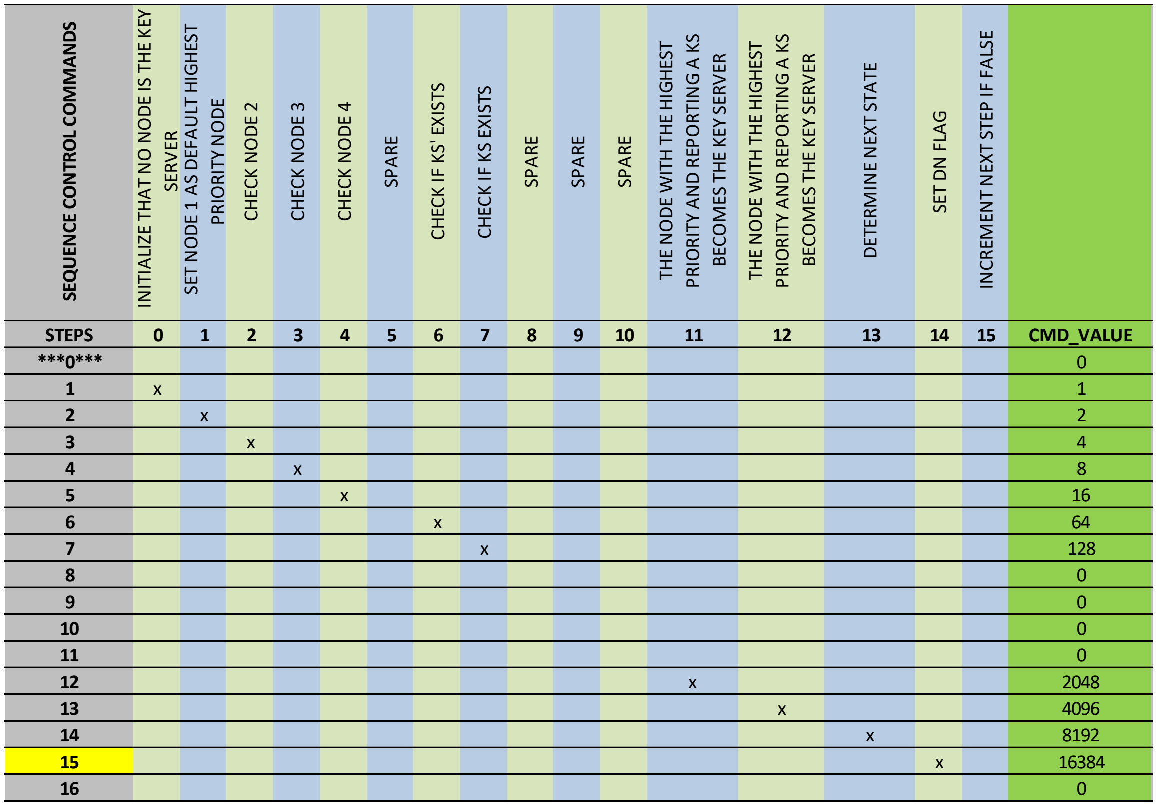
SEQUENCE CONTROL COMMANDS	CLEAR KEY SERVER NODE	SAVE CURRENT PubDH AS OLD	DELAY FOR 20 MINUTES	SPARE	SET InternalFlag_PowerOnCycle	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	DETERMINE NEXT STATE	SET DN FLAG	INCREMENT NEXT STEP IF FALSE	
STEPS	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**CMD_VALUE**
*****0*****																	0
1	x																1
2			x														4
3		x															2
4																	0
5					x												16
6																	0
7														x			8192
8																	0
9															x		16384
10																	0

Figure 32: Processor State 0 Sequence Chart

Note that while State 0 is in operation the Listener and VMAC algorithms are still operating in parallel. This means that if a node is being powered on for the first time ever it will not have a valid **Ks** or **Ks'** in order to being using to authenticate messages. This scenario will most commonly occur when commissioning part of the control system for the first time or when replacing a failed processor that did not have a "hot backup" available (NOTE: Many PLC vendors offer a "hot backup" solution where two controllers have a dedicated line between them and run in parallel with each other to act as only one node). However for a node that has merely been shut down temporarily, such as due to a power loss, the VMAC will still continue to work because it will have a copy of **Ks** and also possibly **Ks'** if a key exchange had not been initiated while the node was powered off.

6.4.2 State 1: Server Check

State 1 is known as the “Server Check” state because this is the state in which a node determines who is the key server for the node. This determination is made by scanning the verified messages from the nodes and finding the highest priority node that has set the **Ks’ Available** flag. If no node has the flag set then the state selects the online node that has the highest priority. Since priorities are based on the nodes internal settings a tree can be created. For example, call this node A and call the server for A node B. It is possible to configure B such that a third node C is the server for node B. As a result key propagation would go down from C to B to A. The sequence chart for State 1 is shown in Figure 33.



SEQUENCE CONTROL COMMANDS	INITIALIZE THAT NO NODE IS THE KEY SERVER	SET NODE 1 AS DEFAULT HIGHEST PRIORITY NODE	CHECK NODE 2	CHECK NODE 3	CHECK NODE 4	SPARE	CHECK IF KS' EXISTS	CHECK IF KS EXISTS	SPARE	SPARE	SPARE	THE NODE WITH THE HIGHEST PRIORITY AND REPORTING A KS BECOMES THE KEY SERVER	THE NODE WITH THE HIGHEST PRIORITY AND REPORTING A KS BECOMES THE KEY SERVER	DETERMINE NEXT STATE	SET DN FLAG	INCREMENT NEXT STEP IF FALSE	
STEPS	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**CMD_VALUE**
*****0*****																	0
1	x																1
2		x															2
3			x														4
4				x													8
5					x												16
6							x										64
7								x									128
8																	0
9																	0
10																	0
11																	0
12												x					2048
13													x				4096
14														x			8192
15															x		16384
16																	0

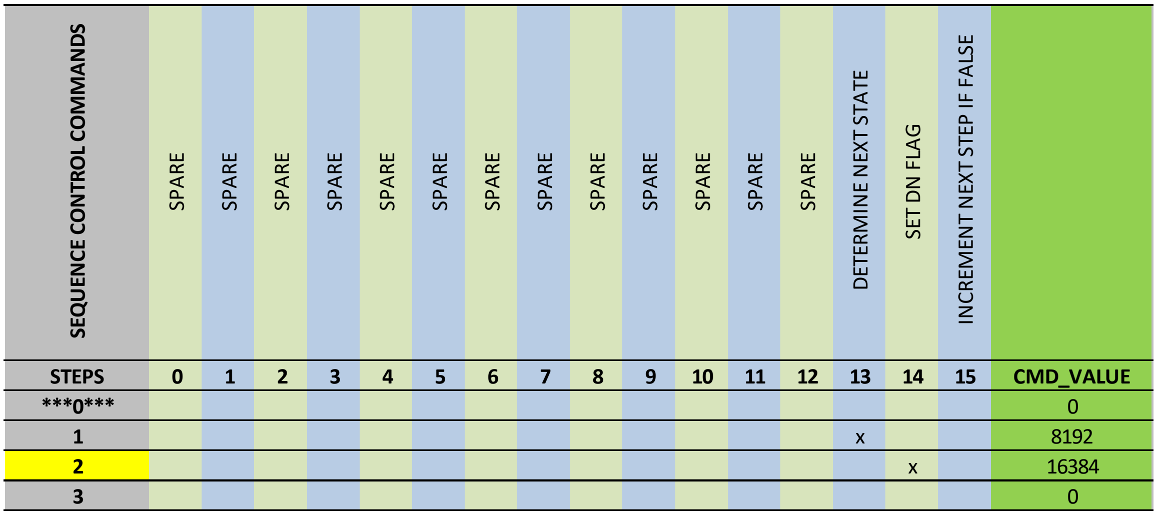
Figure 33: Processor State 1 Sequence Chart

After the server is selected the state must determine the next transition. Normally under a “steady state” operation (meaning that all keys have been exchanged, **Ks=Ks’**, and VMAC is authenticating data regularly) the state will transition into State 5 to check for new messages, and then back to State 1. This cycle repeats ad infimum until some change occurs. If a node determines that it is its own server, meaning that it is the root of the tree, and does not have a **Ks’** it will go about the process of creating a new **Ks’** by transitioning to State 7. Note that just because a node is normally the root does not necessary mean that it will always act as the root. If the root node is power cycled then it is possible that the second highest priority node will have a copy of the **Ks’** and will have set the **Ks’ Available** flag. In this case the root node will instead act a client to the second highest priority node, remaining a client until it has a copy of **Ks’** in which case it will then resume its normal role as a server. The status of each node will be clearly identifiable thru the use of the **Key Master Yes** and **Key Master No** flags, where the former will be set by Node 2 and the latter by Node 1, and then switched once Node 1 obtains a copy of **Ks’**.

Nodes that have determined that they are clients will transition to State 2 if they determine that they need to request a copy of the **Ks’**, either because they have an invalid or nonexistent **Ks’** or **Ks**. The transition to State 2 essentially means that a node realizes that something is wrong and is unsure of what to do. While in State 1 the node does not know if it is in the process of obtaining the keys or if it needs to initiate that process, so it moves into a state that is meant to handle this condition.

6.4.3 State 2: Ks’ Req Check

State 2 is known as the “Ks’ Req Check” state because this is the state in which a node determines if it is currently in the process of performing a key exchange by sending a “**Ks’ Request**” flag to the server node. If the node has determined that it has sent a request to the current server node, or that this node’s **Ks’ ID** matches the server’s **Ks’ ID** it transitions into State 5. Otherwise the node transitions into State 3 to start the key exchange process. The sequence chart for State 2 is shown in Figure 34.



SEQUENCE CONTROL COMMANDS	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	DETERMINE NEXT STATE	SET DN FLAG	INCREMENT NEXT STEP IF FALSE	
STEPS	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**CMD_VALUE**
*****0*****																	0
1														x			8192
2															x		16384
3																	0

Figure 34: Processor State 2 Sequence Chart

State 2 is one of the simplest states in the Processor and consideration was made to combine it with State 1. It was determined, however, that combining the two states overcomplicates the design of State 1 because it takes two fundamentally different questions/actions and combines them. However during a key exchange process these states are strongly interrelated because it is possible that a node’s current server node could change during the key exchange. For example, assume you are node A and the server node is node B. It is possible that during the key exchange node B will go offline for some reason while node A was waiting for a response to the last KEP message (which would normally contain a public Diffie-Hellman value and encrypted Ks’). State 1 will determine that a new node, call it node C, is now the server. It will transition to State 2, which will then realize that a key exchange process has not been started with Node C and therefore initiate a fresh request. Requests to nodes that are offline are automatically cleared by the processor, so the previous key exchange with node B will be abandoned.

6.4.4 State 3: EdDH Keys

State 3 is known as the “EdDH Keys” state because this is the state in which a node generates the public and private Diffie-Hellman keys using the Ed25519 curve. This is done by pseudo randomly picking a private value called **q** and then performing a point multiplication against the base point **B** to get the public value **Q** which represents a point on the Ed25519 curve. The point is then encoded as a 64-character hexadecimal string. The sequence chart for State 3 is shown in Figure 35.

SEQUENCE CONTROL COMMANDS	GET NEW RANDOM NUMBER (qa)	SAVE q AS Internal_PriDH	CONVERT RANDOM NUMBER INTO E	SPARE	BUILD BASE POINT	SPARE	CALCULATE Q = q * B	SPARE	ENCODE Q	SPARE	SAVE Q AS Internal_PubDH	SET PubDH Avail Flag	SPARE	DETERMINE NEXT STATE	SET DN FLAG	INCREMENT NEXT STEP IF FALSE	
STEPS	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**CMD_VALUE**
*****0*****																	0
1	x																1
2		x															2
3			x														4
4					x												16
5							x										64
6									x								256
7											x						1024
8												x					2048
9														x			8192
10															x		16384
11																	0

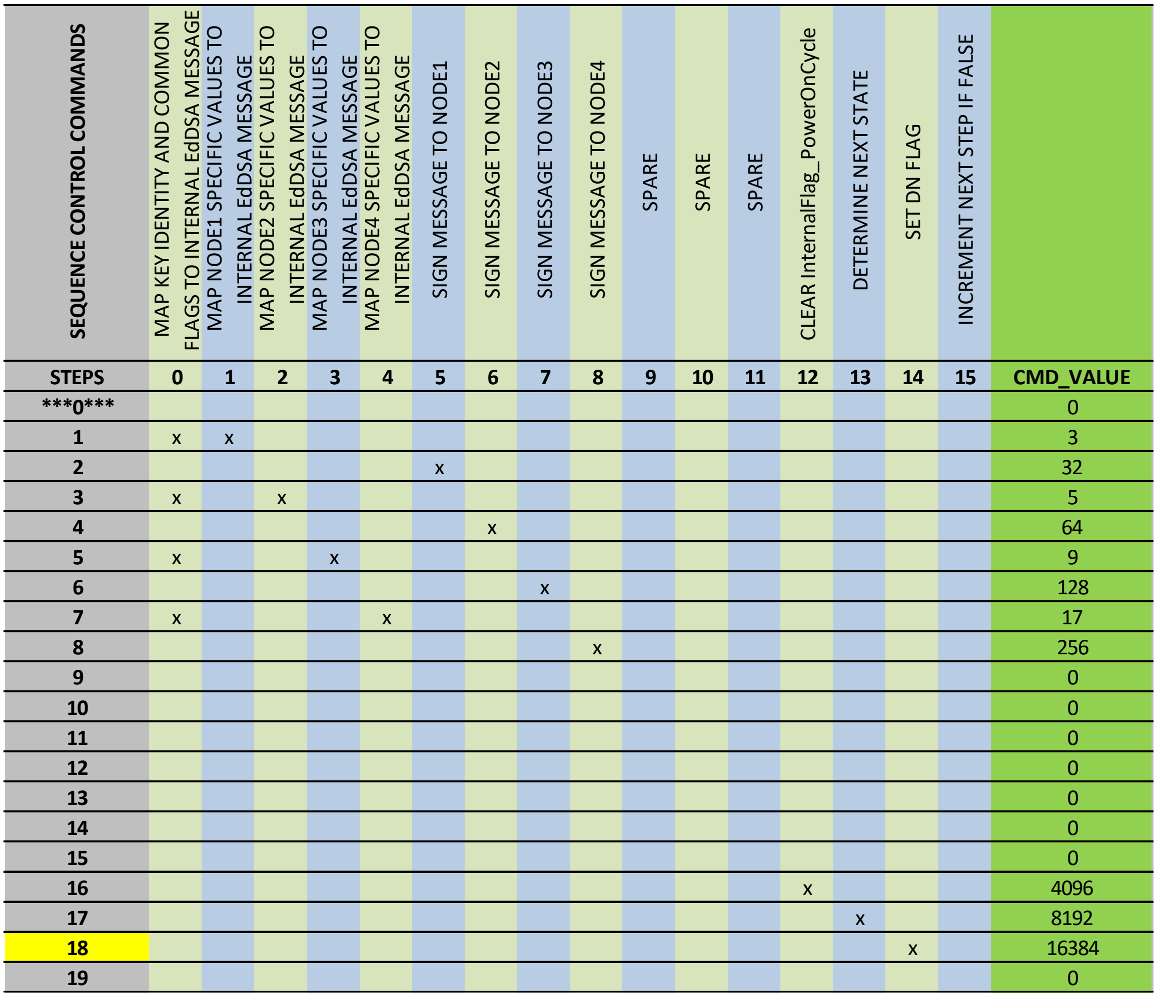
Figure 35: Processor State 3 Sequence Chart

For every node that a given node communicates with in KEP a table is kept in memory that keeps track of this node's current **q / Q** pair that is being used for that node. It also keeps track of the last public Diffie-Hellman value that provided by that node, since a change in that value indicates that the other node has created its own key pair which it used to encrypt **Ks'**. Each row in the table is reserved for a specific node and is only used for that node, which means that the table grows linearly with as the number of nodes increase. However by using a tree structure a node's table size only needs to be as large as the node's number of leaves plus 2 for the node itself and the node's root.

Note that State 3 is used to generate **q / Q** pairs for both servers and clients. The state essentially functions the exact same way regardless of the reason why it is creating the key pair and stores the results in the same table mentioned in the previous paragraph. The difference is whether the node transitions next into State 6 if this is a server side action or transitions into State 4 for a client side action.

6.4.5 State 4: Sign message

State 4 is where the digital signatures are created. Depending on how entry is made into State 4 it is possible that the state will sign only a message to a given client, only a message to the server node, or a message to all nodes. All nodes, in this context, includes only the nodes in this node's immediate proximity within the tree. For example, when transitioning from State 3 to State 4 the message to be signed is a client message that only needs to go to a server. Similarly when transitioning from State 6 to State 4 the message is a server message that only needs to go to a specific client (in this case the one who initiated the process by sending a **Ks' Request**). When transitioning from State 8 to State 4 a message is sent to all nodes advertising the new key IDs, so that all nodes can use this information to determine if that node can act as a viable server node. The sequence chart for State 4 is shown in Figure 36.



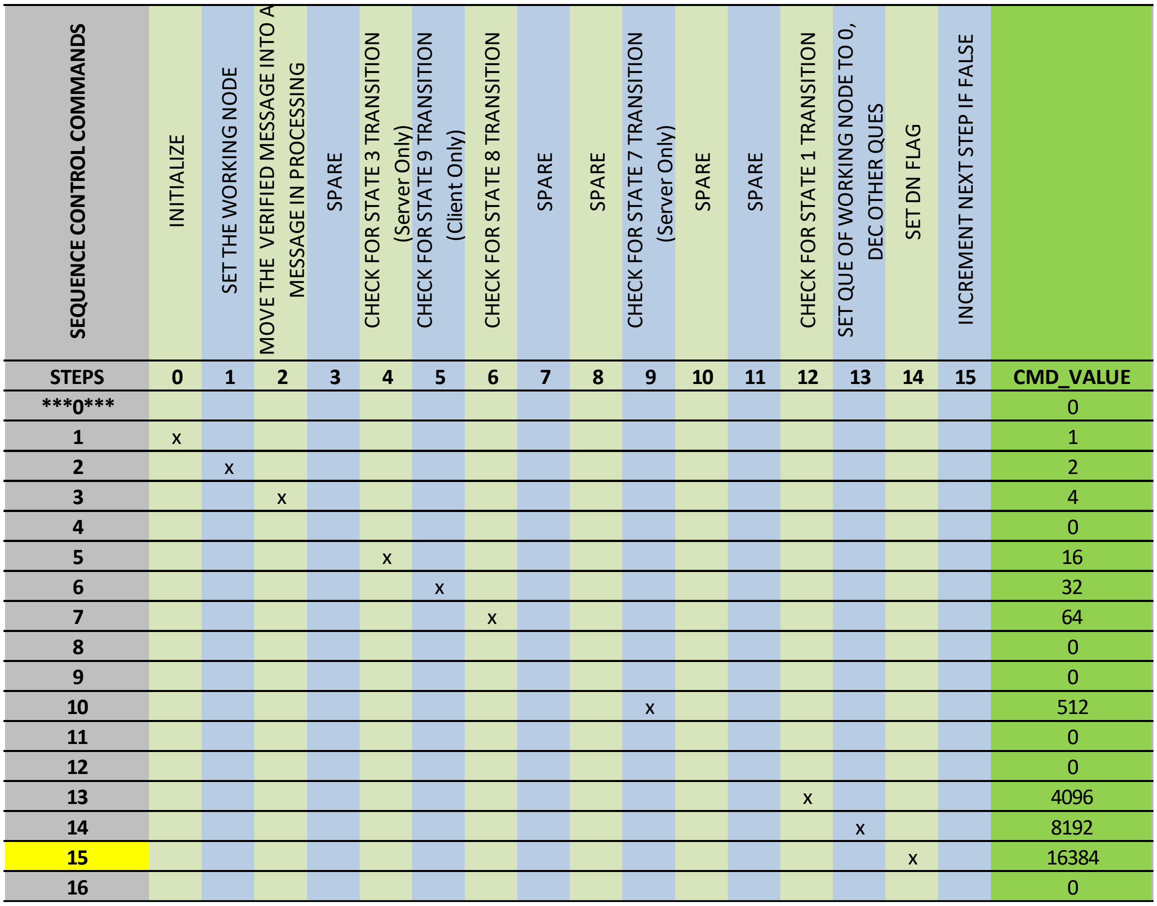
SEQUENCE CONTROL COMMANDS	MAP KEY IDENTITY AND COMMON FLAGS TO INTERNAL EdDSA MESSAGE	MAP NODE1 SPECIFIC VALUES TO INTERNAL EdDSA MESSAGE	MAP NODE2 SPECIFIC VALUES TO INTERNAL EdDSA MESSAGE	MAP NODE3 SPECIFIC VALUES TO INTERNAL EdDSA MESSAGE	MAP NODE4 SPECIFIC VALUES TO INTERNAL EdDSA MESSAGE	SIGN MESSAGE TO NODE1	SIGN MESSAGE TO NODE2	SIGN MESSAGE TO NODE3	SIGN MESSAGE TO NODE4	SPARE	SPARE	SPARE	CLEAR InternalFlag_PowerOnCycle	DETERMINE NEXT STATE	SET DN FLAG	INCREMENT NEXT STEP IF FALSE	
STEPS	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**CMD_VALUE**
*****0*****																	0
1	x	x															3
2						x											32
3	x		x														5
4							x										64
5	x			x													9
6								x									128
7	x				x												17
8									x								256
9																	0
10																	0
11																	0
12																	0
13																	0
14																	0
15																	0
16													x				4096
17														x			8192
18															x		16384
19																	0

Figure 36: Processor State 4 Sequence Chart

This sequence chart from the proof of concept does have room for future improvement, particularly to streamline the chart and make the entire state more arbitrary for a *n-nodes*. However it illustrates the critical concept that you first need to map the values that you desire to sign into a working memory area, and then leave that memory area untouched while the digital signature is being processed or else you will have an invalid signature generated. The sequence chart also indicates that this is the state in which the internal power on flag set in State 0 is finally cleared. This means that the flag essentially acts as just a one shot to go straight from State 1 to State 4 in the circumstances where the node that just powered on determined it was its own server (i.e. is the root node) and already had the current **Ks'**. This allows the root node to resume its place in the tree as quickly as possible.

6.4.6 State 5: Check for new message

The purpose of this state is to check the incoming message of a node and compare the information in that message with the information this node already has to determine if some action needs to be taken. The various conditions that induce the transitions are shown in Figure 31, and the sequence chart is shown in Figure 37. There is also an internal "processing required" flag set by the Listener whenever a new message has come in that is successfully verified, but this flag was just used for diagnostic purposes. The state of the internal flag does not have an operational impact.



SEQUENCE CONTROL COMMANDS	INITIALIZE	SET THE WORKING NODE	MOVE THE VERIFIED MESSAGE INTO A MESSAGE IN PROCESSING	SPARE	CHECK FOR STATE 3 TRANSITION (Server Only)	CHECK FOR STATE 9 TRANSITION (Client Only)	CHECK FOR STATE 8 TRANSITION	SPARE	SPARE	CHECK FOR STATE 7 TRANSITION (Server Only)	SPARE	SPARE	CHECK FOR STATE 1 TRANSITION	SET QUE OF WORKING NODE TO 0, DEC OTHER QUES	SET DN FLAG	INCREMENT NEXT STEP IF FALSE	
STEPS	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**CMD_VALUE**
*****0*****																	0
1	x																1
2		x															2
3			x														4
4																	0
5					x												16
6						x											32
7							x										64
8																	0
9																	0
10										x							512
11																	0
12																	0
13													x				4096
14														x			8192
15															x		16384
16																	0

Figure 37: Processor State 5 Sequence Chart

This state works by examining the messages from one node at a time with each transition into State 5, vice examining all messages each time. For example, when first entering the state it checks for a new message from Node A. If no new or actionable messages are found we transition into State 1. If nothing has changed with respect to the server node we transition back into State 5, this time checking for a new message from Node B. The process repeats cyclically, skipping nodes that are offline. This prevents a possible denial of service attack where one node is constantly providing new messages without allowing the next node to get a chance to check its functions.

6.4.7 State 6: Gen Kp, Encrypt Ke

State 6 is a “server only” state that is executed as a result of the server receiving a new message from a client in State 5 which initiates a Ed25519 Diffie-Hellman key exchange. State 5 transitions into State 3 to generate the server’s key pair, and then transitions into State 6 where a point multiplication between the server’s private value (acting a scalar) and the client’s public value (acting as a point on the Ed25519 curve) generates a new point on the curve. This new point is then hashed using SHA512 to provide extra security, and the results of the hash (parts *a* thru *d*) are used as a one-time symmetric key called **Kp** with a length of 256-bits. **Kp** is then used to encrypt **Ks’** via an XOR operation. The sequence chart for State 6 is shown in Figure 38.

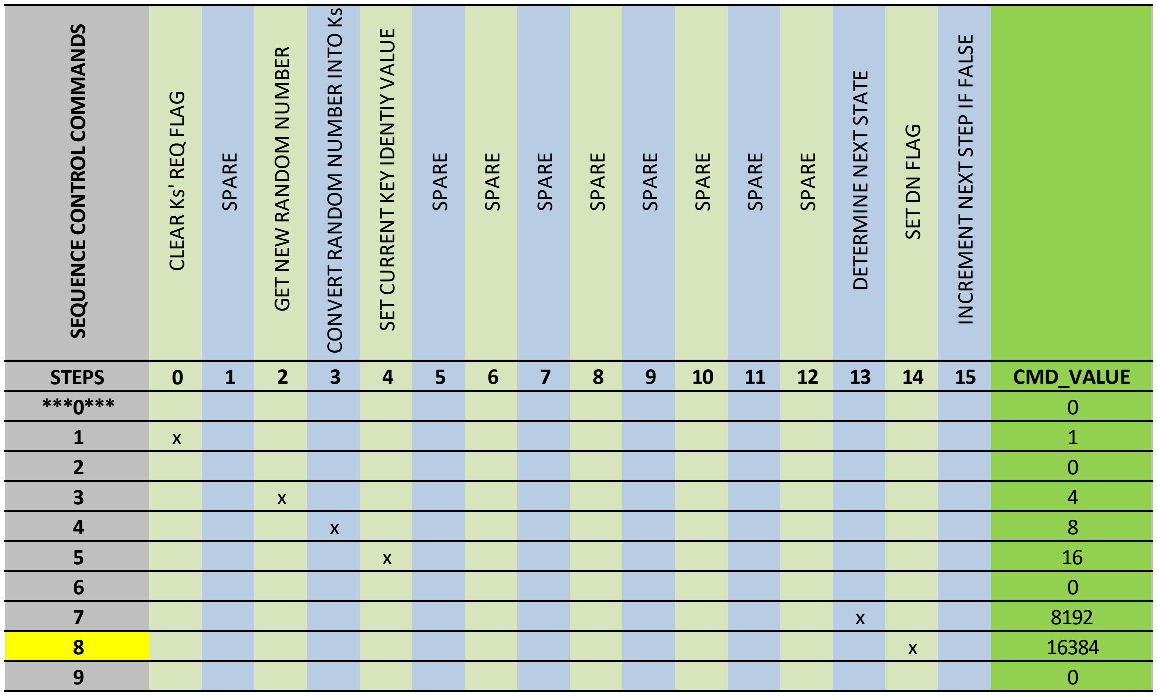
Note that XOR can be used to securely encrypt data when the key is the same length as the data and is only used once. Therefore when a public Diffie-Hellman value is provided to the server or client the value is saved as the last value used. A new value must be provided in order to get into State 6, which is controlled on the server side as one of the criteria to initiate the State 5 to State 3 transition.

SEQUENCE CONTROL COMMANDS	INTERPRET PriDH STRING AS LITTLE ENDIAN INTEGER	DECODE PubDH POINT	PERFORM POINT MULTIPLICATION	ENCODE THE RESULT	CONVERT RESULT TO LINTs FOR HASHING	HASH	STORE FIRST 256 BITS AS Kp	ENCRYPT Ks' AS Ke USING Kp	CONVERT Ke INTO STRING TO BE SIGNED	SET FLAGS	SPARE	SPARE	SPARE	DETERMINE NEXT STATE	SET DN FLAG	INCREMENT NEXT STEP IF FALSE	
STEPS	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**CMD_VALUE**
*****0*****																	0
1	x																1
2		x															2
3			x														4
4				x													8
5					x												16
6						x											32
7							x										64
8								x									128
9									x								256
10										x							512
11														x			8192
12															x		16384
13																	0

Figure 38: Processor State 6 Sequence Chart

6.4.8 State 7: Create new Ks'

This state is a "server only" state that is used to create a new **Ks'**, and is typically initiated as a result of a scheduled automatic update. The sequence diagram for the state is shown in Figure 39. Essentially the state clears the request and then pseudo randomly generates a new 256-bit value which becomes the new **Ks'**. The ID of **Ks'** is then set as a 64-bit number representing the number of microseconds since epoch time. If **Ks** does not exist then the state transitions to State 8, otherwise it transitions to State 4 to send a new digitally signed message to all nodes indicating that a new **Ks'** is available. The other nodes will receive this message, verify it, and then initiate the key exchange process with the server.

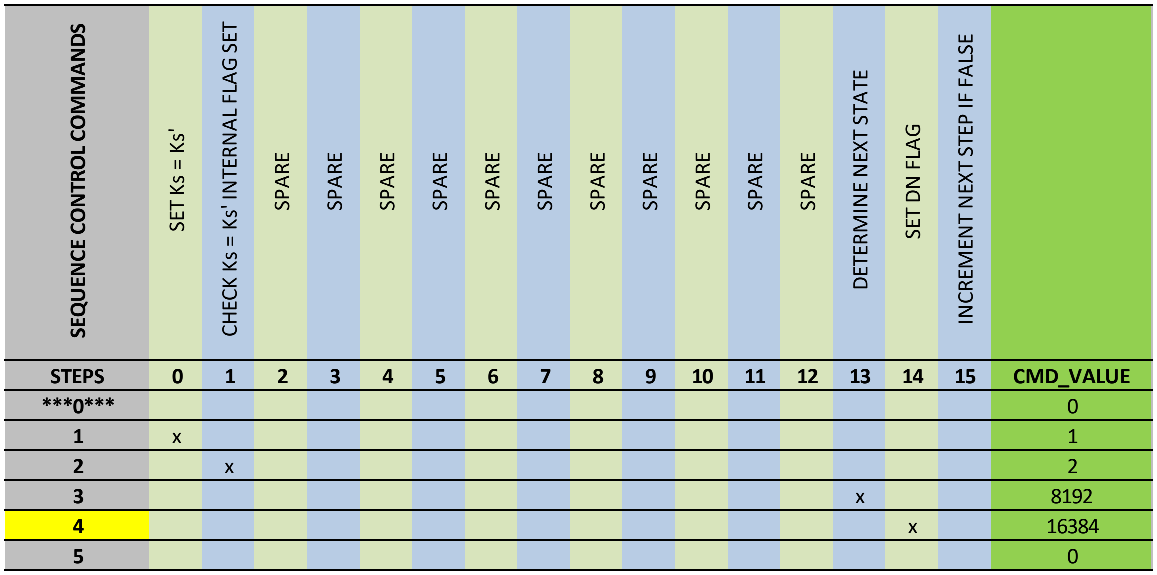


SEQUENCE CONTROL COMMANDS	CLEAR Ks' REQ FLAG	SPARE	GET NEW RANDOM NUMBER	CONVERT RANDOM NUMBER INTO Ks	SET CURRENT KEY IDENTIY VALUE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	DETERMINE NEXT STATE	SET DN FLAG	INCREMENT NEXT STEP IF FALSE	
STEPS	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**CMD_VALUE**
*****0*****																	0
1	x																1
2																	0
3			x														4
4				x													8
5					x												16
6																	0
7														x			8192
8															x		16384
9																	0

Figure 39: Processor State 7 Sequence Chart

6.4.9 State 8: Set Ks = Ks'

This state is used to disregard the old **Ks** value and set it equal to the new **Ks'** value, as well as update the ID. Transition into this state is based on a node obtaining a new **Ks'** and not having a **Ks**, or more commonly on a node detecting that all nodes have sent valid VMAC messages using **Ks'** as described previously. The sequence diagram for the state is shown in Figure 40. The state completes when it has verified that **Ks** is equal to **Ks'** and the IDs of the keys match. The state then transitions to State 4 and sends a message out to all nodes indicating the updated status of the node.

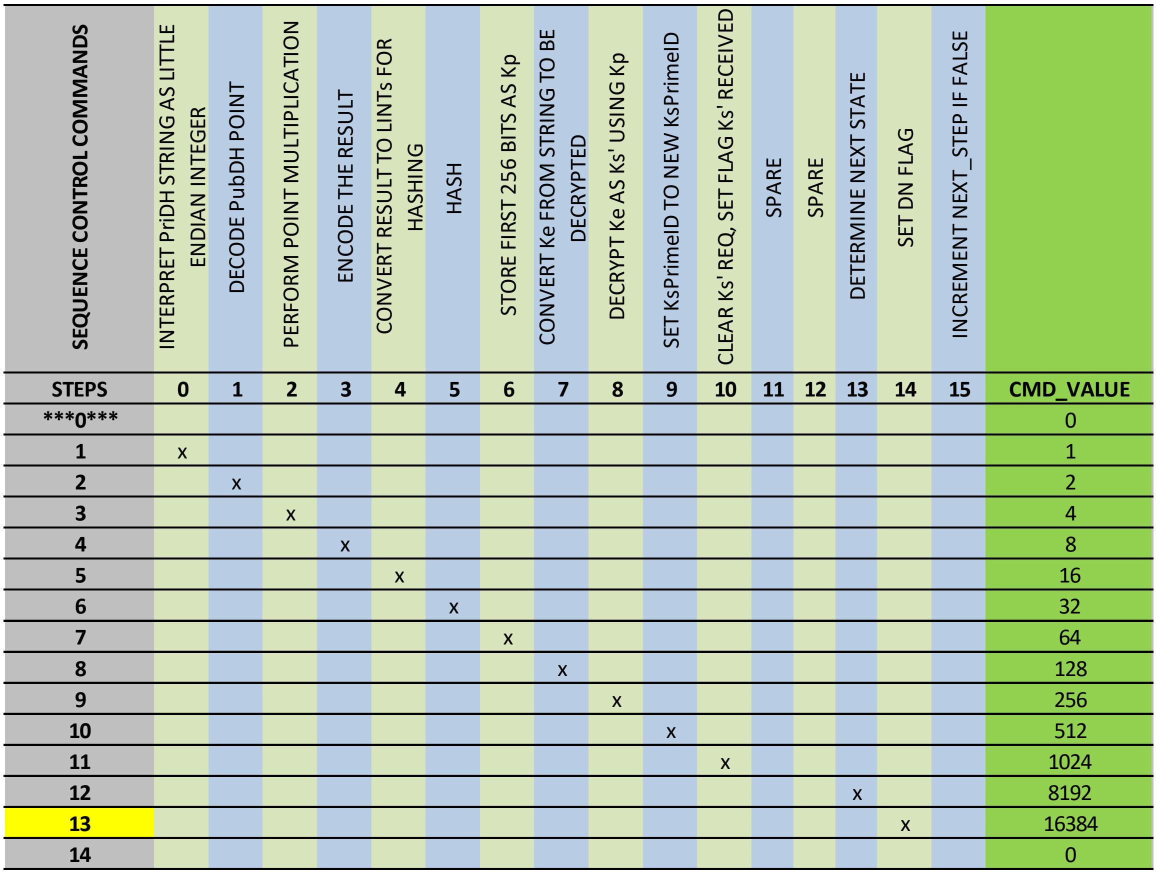


SEQUENCE CONTROL COMMANDS	SET Ks = Ks'	CHECK Ks = Ks' INTERNAL FLAG SET	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	DETERMINE NEXT STATE	SET DN FLAG	INCREMENT NEXT STEP IF FALSE	
STEPS	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**CMD_VALUE**
*****0*****																	0
1	x																1
2		x															2
3														x			8192
4															x		16384
5																	0

Figure 40: Processor State 8 Sequence Chart

6.4.10 State 9: Gen Kp, Decrypt Ke

State 9 is the client side equivalent of State 6 described previously. It essentially does the same process as State 6 to generate **Kp**, and this time uses XOR on **Ke** to recover **Ks'**. The sequence chart is shown in Figure 41.



SEQUENCE CONTROL COMMANDS	INTERPRET PriDH STRING AS LITTLE ENDIAN INTEGER	DECODE PubDH POINT	PERFORM POINT MULTIPLICATION	ENCODE THE RESULT	CONVERT RESULT TO LINTs FOR HASHING	HASH	STORE FIRST 256 BITS AS Kp	CONVERT Ke FROM STRING TO BE DECRYPTED	DECRYPT Ke AS Ks' USING Kp	SET KsPrimeID TO NEW KsPrimeID	CLEAR Ks' REQ, SET FLAG Ks' RECEIVED	SPARE	SPARE	DETERMINE NEXT STATE	SET DN FLAG	INCREMENT NEXT_STEP IF FALSE	
STEPS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	CMD_VALUE
0																	0
1	x																1
2		x															2
3			x														4
4				x													8
5					x												16
6						x											32
7							x										64
8								x									128
9									x								256
10										x							512
11											x						1024
12														x			8192
13															x		16384
14																	0

Figure 41: Processor State 9 Sequence Chart

6.5 Typical Operation

This section describes the typical sequences of events that would occur during operation of the KEP when powering on for the first time when the system is freshly commissioned. In this example assume that there are four nodes, labeled A thru D, where A has the highest priority and D has the lowest priority. This section is intended as an example to demonstrate the general behavior of KEP in this particular scenario and configuration. In general, both server nodes and client nodes will begin operation in state

0, which is the starting point for when power is applied. The node does not know whether it is a server or a client until after it completes state 1.

When the four nodes begin powering up, they will each move into State 1 to do a server check. In this example each 10of the nodes will determine that A is the server, and that no one is advertising **Ks’ Available**. In fact at this point since each node is powering on for the first time the KEP message between them will be completely empty. Nodes B thru D will each go through the process of preparing to get a new key from the server by executing the sequence of operations shown in Figure 42 on the right. Node A will recognize that it is the server and go thru the sequence of operations shown on the left in order to generate a new **Ks’**. Since this is a first time power up it will set **Ks = Ks’**. All nodes will conclude by generating digitally signed messages, with B thru D each sending a single message to node A. This message will contain the **Ks’ Request** along with the public Diffie-Hellman value necessary to do a key exchange. Node A will be sending a message to all nodes advertising **Ks’ Available** along with the **Ks’ ID** and **Ks ID**.

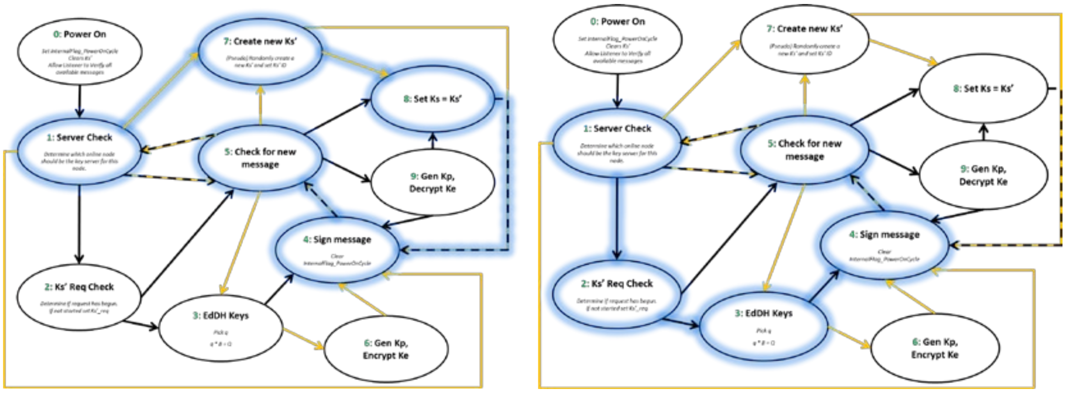

0: Power On
Set InternalFlag_PowerOnCycle
Clears Ks'
Allow Listener to Verify all available messages
7: Create new Ks'
(Pseudo) Randomly create a new Ks' and set Ks' ID
8: Set Ks = Ks'
1: Server Check
Determine which online node should be the key server for this node.
5: Check for new message
9: Gen Kp, Decrypt Ke
4: Sign message
Clear InternalFlag_PowerOnCycle
2: Ks' Req Check
Determine if request has begun, if not started set Ks'_req
3: EdDH Keys
Pick q
q * B = Q
6: Gen Kp, Encrypt Ke
0: Power On
Set InternalFlag_PowerOnCycle
Clears Ks'
Allow Listener to Verify all available messages
7: Create new Ks'
(Pseudo) Randomly create a new Ks' and set Ks' ID
8: Set Ks = Ks'
1: Server Check
Determine which online node should be the key server for this node.
5: Check for new message
9: Gen Kp, Decrypt Ke
4: Sign message
Clear InternalFlag_PowerOnCycle
2: Ks' Req Check
Determine if request has begun, if not started set Ks'_req
3: EdDH Keys
Pick q
q * B = Q
6: Gen Kp, Encrypt Ke

Figure 42: LEFT: A generating Ks’ and Ks, RIGHT: B thru D requesting Ks’

All of the nodes will then enter into State 5 to check for a new message. Depending on processing speeds and network latency it could take some time for the nodes to receive the various messages and verify them. Therefore the nodes will enter into an idle mode where they will cycle between different states looking for some kind of change in status. This change could be a change in the server identity, or in the receipt of a verified and actionable message. Figure 43 shows the sequence of operations that the different nodes will take. Note that the client nodes will cycle differently than the server node at this point because the server has the keys.

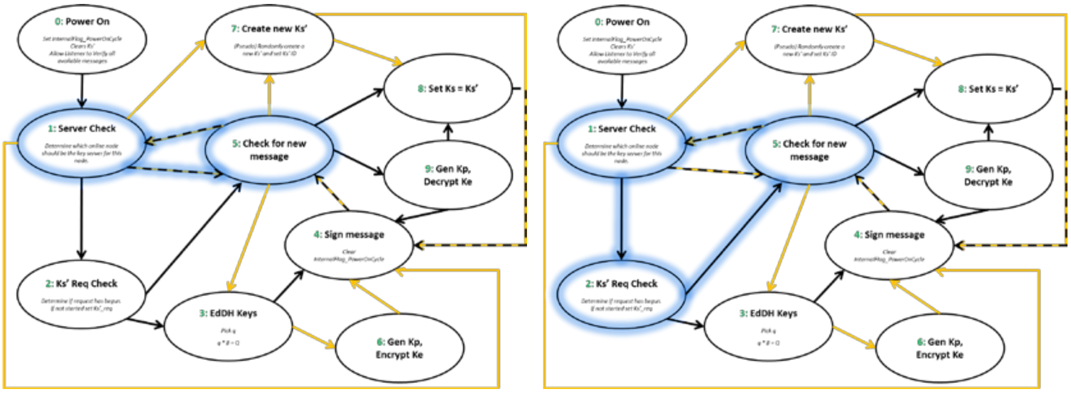

0: Power On
Set internalFlag_PowerOnCycle Clears Ks' Allow Listener to Verify all available messages
7: Create new Ks'
(Pseudo) Randomly create a new Ks' and set Ks' ID
8: Set Ks = Ks'
1: Server Check
Determine which online node should be the key server for this node.
5: Check for new message
9: Gen Kp, Decrypt Ke
4: Sign message
Clear internalFlag_PowerOnCycle
2: Ks' Req Check
Determine if request has begun. If not started set Ks'_req
3: EdDH Keys
Pick q
q * B = O
6: Gen Kp, Encrypt Ke
0: Power On
Set internalFlag_PowerOnCycle Clears Ks' Allow Listener to Verify all available messages
7: Create new Ks'
(Pseudo) Randomly create a new Ks' and set Ks' ID
8: Set Ks = Ks'
1: Server Check
Determine which online node should be the key server for this node.
5: Check for new message
9: Gen Kp, Decrypt Ke
4: Sign message
Clear internalFlag_PowerOnCycle
2: Ks' Req Check
Determine if request has begun. If not started set Ks'_req
3: EdDH Keys
Pick q
q * B = O
6: Gen Kp, Encrypt Ke

Figure 43: LEFT: A in Idle, RIGHT: B thru D in Idle but Exchange in Progress

At some point the server node and client nodes will receive verified messages. A verified message from the server will be not be actionable at this point, since the message will only confirm that A is the server. If the configuration was different and another node was a higher priority than node A (call it E), which was previously offline for some reason but now has come online with a valid **Ks' ID**, then the clients would abandon the request to A and generate a new request to E. In such a scenario A should also be configured to see that E is higher priority and switch its role over to a client, since it would detect that the **Ks' ID** that it generated did not match the server's values.

In this configuration, however, the client nodes will continue to idle while the server node processes the message for each node. The server node will receive the requests for **Ks'** and go thru the process of generating the Diffie-Hellman keys, generating **Kp**, encrypting **Ks'**, generating a message, and signing the message. This process will be repeated for each of the nodes as shown in the figure below on the left. Meanwhile, as each node receives the message it will leave the idle cycle shown in the previous figure and instead begin to process the message and decrypt the key as shown in Figure 44 on the right.

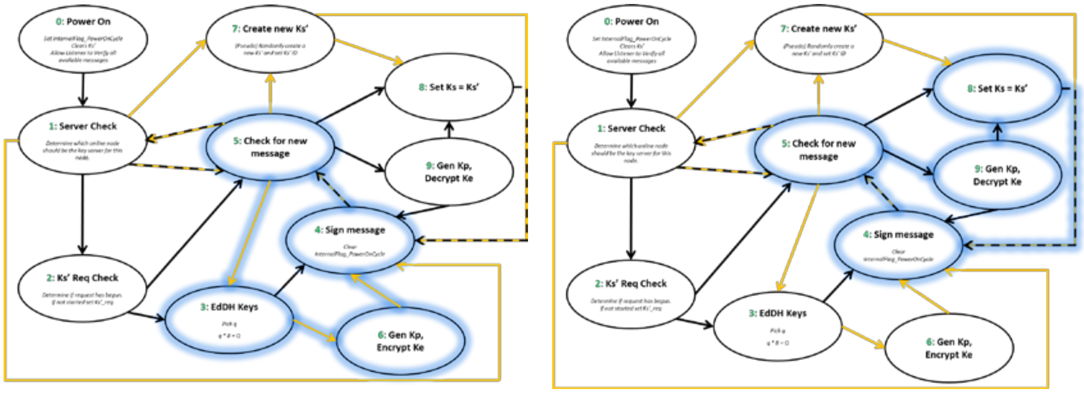

0: Power On
Set InternalFlag_PowerOnCycle
Clears Ks'
Allow Listener to Verify all available messages
7: Create new Ks'
(Pseudo) Randomly create a new Ks' and set Ks' ID
8: Set Ks = Ks'
1: Server Check
Determine which online node should be the key server for this node.
5: Check for new message
9: Gen Kp, Decrypt Ke
4: Sign message
Clear InternalFlag_PowerOnCycle
2: Ks' Req Check
Determine if request has begun. If not started set Ks'_req
3: EdDH Keys
Pick q
q * B = Q
6: Gen Kp, Encrypt Ke
0: Power On
Set InternalFlag_PowerOnCycle
Clears Ks'
Allow Listener to Verify all available messages
7: Create new Ks'
(Pseudo) Randomly create a new Ks' and set Ks' ID
8: Set Ks = Ks'
1: Server Check
Determine which online node should be the key server for this node.
5: Check for new message
9: Gen Kp, Decrypt Ke
4: Sign message
Clear InternalFlag_PowerOnCycle
2: Ks' Req Check
Determine if request has begun. If not started set Ks'_req
3: EdDH Keys
Pick q
q * B = Q
6: Gen Kp, Encrypt Ke

Figure 44: LEFT: A encrypting Ks', RIGHT: B thru D decrypting Ks'

In this example, each of the clients did not have an older **Ks**, so they immediately set **Ks = Ks'**. However during subsequent key exchanges the client nodes will only enter State 8 after confirming that all nodes have successfully received and verified VMAC messages. Figure 45 and Figure 46 show the process that client node would go through to update **Ks’** if it already has a **Ks**, as well as the process the client node will go thru once it has received and verified VMAC messages from all nodes using **Ks’**.

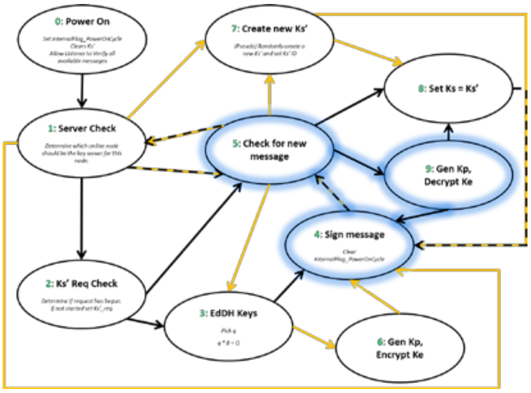

0: Power On
Set InternalFlag_PowerOnCycle
Clears Ks'
Allow Listener to Verify all available messages
7: Create new Ks’
(Pseudo) Randomly create a new Ks' and set Ks' ID
8: Set Ks = Ks’
1: Server Check
Determine which online node should be the key server for this node.
5: Check for new message
9: Gen Kp, Decrypt Ke
4: Sign message
Clear InternalFlag_PowerOnCycle
2: Ks’ Req Check
Determine if request has begun. If not started set Ks'_req
3: EdDH Keys
Pick q
6: Gen Kp, Encrypt Ke

Figure 45: Client Has Ks and Decrypting new Ks’

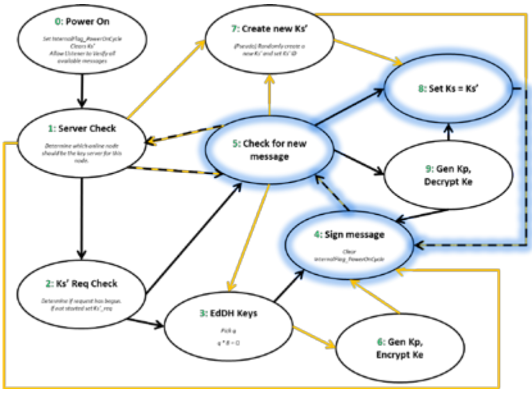

0: Power On
Set InternalFlag_PowerOnCycle
Clears Ks'
Allow Listener to Verify all available messages
7: Create new Ks’
(Pseudo) Randomly create a new Ks' and set Ks' ID
8: Set Ks = Ks’
1: Server Check
Determine which online node should be the key server for this node.
5: Check for new message
9: Gen Kp, Decrypt Ke
4: Sign message
Clear InternalFlag_PowerOnCycle
2: Ks’ Req Check
Determine if request has begun. If not started set Ks'_req
3: EdDH Keys
Pick q
6: Gen Kp, Encrypt Ke

Figure 46: RIGHT: Client updating Ks = Ks’

Now that the key exchange is complete, all servers and clients will enter into the same idle cycle shown in Figure 47. The nodes will remain in this cycle until there is either a change in server status, a key update, or the node experience some kind of failure that causes it to go down, such as a power loss.

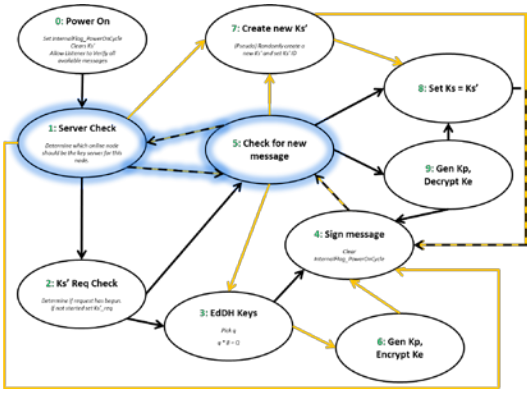

0: Power On
Set InternalFlag_PowerOnCycle
Clears Ks'
Allow Listener to Verify all available messages
7: Create new Ks'
(Pseudo) Randomly create a new Ks' and set Ks' ID
8: Set Ks = Ks'
1: Server Check
Determine which online node should be the key server for this node.
5: Check for new message
9: Gen Kp, Decrypt Ke
4: Sign message
Clear InternalFlag_PowerOnCycle
2: Ks' Req Check
Determine if request has begun. If not started set Ks'_req
3: EdDH Keys
Pick q
q * B = 0
6: Gen Kp, Encrypt Ke

Figure 47: Client or Server in Idle Cycle with Ks = Ks'

CHAPTER 7: RESULTS AND FUTURE WORK

The proof of concept was tested using four RSLogix 5000 1756-L83 PLCs that were placed in the same rack. Messages were transmitted using produce/consume tags across the backplane, however the work presented here is protocol independent. Any transportation mechanism including Ethernet and Fieldbus could be used, which is a feature inherent for both this work and produce/consume tags in general. The controllers were configured initially with Node 1 having the highest priority and Node 4 the lowest, all in a flat configuration. As testing continued variations on the tree configuration were tested, such as nodes farther down on the tree from the root node. VMAC was used to provide data authentication during the entire process.

In general for KEP it was found that the timing of KEP ranged from almost a nonexistent impact on scan time to at most a 20 ms impact on scan time, depending on what KEP was doing at the moment in time. The results on scan time are not dependent on the number of nodes involved in the KEP, however the time to complete an entire key change is dependent on the tree configuration and the number of nodes. The scan time impact is ultimately driven by the processing of Ed25519_SIGN and Ed25519_VERIFY, each of which has a maximum 10 ms scan time. Therefore during KEP it is possible to have both of them running in the worst case scenario at a point in time resulting in the 20ms scan time impact. In general though the results showed that during an active key exchange KEP spent about 30% of the time with a less than 1 ms impact, approximately 50% of the time with a 5-10 ms scan time impact, and remaining 20% of time between 10-20ms scan time impact.

Ultimately the biggest scan time impacts come from VMAC itself. Figure 17 provided the timing results for VMAC using various rounds and Section 4.3.3 provides the analysis for key reversal attacks. For the proof-of-concept 20 rounds were used with a message length of 100 DINTs, or 400 bytes. Produce/consume tags have a maximum length of 500 bytes so the VMAC data consumed 400 of the 500 bytes. The remaining 100 bytes was reserved for some overhead such as the VMAC itself (32 bytes), flags indicating the number of rounds and the length of the message (8 bytes total), and information on the connection status which is a generic part of using produce/consume tags in a Rockwell PLC. With each node generating one VMAC message and processing 3 incoming messages the total scan time of the program (including all the other I/O handling for none-VMAC data and all the implementation details described in Section 4.4) was between 51.7 ms and 52.1 ms.

This result is interesting because the results shown in Figure 17 indicate that running four VMACs containing 400 bytes each should take a total of approximately 51.6 ms, indicating that the overwhelming majority of the run time is spent running VMAC itself. Therefore the implementation details provided in Section 4.4 have a negligible impact on overall system performance but are critical for providing overall security. The result also suggests that if additional alarms or monitoring capability were to be added that capability would have a negligible impact on performance.

The one area of potential improvement for VMAC is the determination of the status of a loss of communications to a node. In the proof of concept this was done using built in system values and running a Get System Values (GSV) instruction that would provide the status of communications from the PLC. In most PLC applications if the status was "good" then "COMMSOK" would be triggered immediately, and the status would have to go "bad" for three seconds before we would state the communications was lost. In this proof of concept status this was reversed, meaning that we would have to be "good" for three seconds before setting "COMMSOK" and a "bad" status would immediately indicate communications had been lost. The main driver for this change was to ensure that stable communications had been established before cryptographic functions would be initialized to improve the overall operation of KEP and VMAC. It is the author's opinion that this change would not negatively impact overall system performance, but that might not hold true depending on the specific control system application.

A long term solution to improve efficiency in the algorithm would be to create a hardware based solution, especially one that could perform 64-bit math natively in the PLC. If a PLC was capable of performing 64-bit math natively the performance of VMAC would be at least doubled because we could use an SHA512 based solution that would double the size of VMAC data blocks, although it is likely we would have to modify the message scheduling portion of SHA512. Additionally a hardware based solution could include a built-in true-random number generator that could be integrated into the control system platform (such as a card that would fit in the chassis). It is suggested, however, that if PLC vendors do create such a solution they offer at least three different options based on different technologies in case a flaw is discovered with one option that is not solvable with a firmware update.

Finally, this work has been focused on providing a mechanism for protecting and verifying data integrity between nodes, and allowing a mechanism for the operator to detect if there is a problem. However this work has not defined what an operator should do if problem is detected. In IT systems the usual answer is to disconnect the systems, however in a control system it is possible that disconnecting the system could result in even higher risk. The decision tree is therefore based on a wide range of factors that require further study with the goal of producing an automated system that can respond intelligently to detected cyber threats. Further work is required to map out these threat profiles and corresponding decision trees, and ultimately to develop the response solution.

REFERENCES

1. NISTIR 7628 Volume 1 (2010) "Guidelines for Smart Grid Cyber Security: Vol. 1, Smart Grid Cyber Security Strategy, Architecture, and High-Level Requirements."
2. D. Hankerson, A. Menezes, S. Vanstone, (2004) Guide to Elliptic Curve Cryptography, ©2004, Springer-Verlag New York, Inc.
3. Wikipedia: RSA Algorithm (http://en.wikipedia.org/wiki/RSA_%28algorithm%29) Accessed: 28 September 2017
4. Wikipedia: Digital Signature Algorithm (http://en.wikipedia.org/wiki/Digital_Signature_Algorithm) Accessed: 28 September 2017
5. NIST Special Publication 800-57 Part 1 Revision 4 (2016) "Recommendation for Key Management." (http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf) Accessed: 28 September 2017
6. V.S. Miller, (1985). "Use of elliptic curves in cryptography." Advances in Cryptology Proc. Crypto '85, LNCS 218, H.C. Williams, Ed., Springer-Verlag, pp. 417-426
7. N. Koblitz, (1987). "Elliptic curve cryptosystems." Mathematics of Computation, Vol. 48, No. 177, p. 279-287
8. ANSI X9.62 (1999). "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)

9. ANSI X9.63 (2000- Working Draft). “Public Key Cryptography for the Financial Services Industry: Elliptic Curve Key Agreement and Key Transport Protocols.”

10. IEEE 1363-2000 (2000) “Standard Specifications for Public-Key Cryptography.”

11. ISO/IEC 14888-3 (1998). “Information Technology – Security Techniques – Digital Signatures with Appendix – Part 3: Certificate Based Mechanisms.”

12. ISO/IEC 15946 (1999 – Committee Draft). “Information Technology – Security Techniques – Cryptographic Techniques Based on Elliptic Curves.”

13. NIST FIPS Pub 186-4 (2013). “Digital Signature Standard.” (http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf) Accessed: 28 September 2017

14. Standards For Efficient Cryptography Group (SECG) (http://www.secg.org) Accessed: 28 September 2017

15. SECG SEC 1 Version 2.0 (2009). “SEC 1: Elliptic Curve Cryptography.” (http://www.secg.org) Accessed: 28 September 2017

16. SECG SEC 2 Version 2.0 (2010). “SEC 2: Recommended Elliptic Curve Domain Parameters.” (http://www.secg.org) Accessed: 28 September 2017

17. Certicom (http://www.certicom.com) Accessed: 28 September 2017

18. D. McGrew, K. Igoe, M. Salter. (IETF) (RFC 6090). “Fundamental Elliptic Curve Cryptography Algorithms.” (http://tools.ietf.org/html/draft-mcgrew-fundamental-ecc-01) Accessed: 28 September 2017

19. N. Koblitz, (2010) “My Last 24 Years in Crypto: A Few Good Judgments and Many Bad Ones” (http://2010.eccworkshop.org/slides/Koblitz.pdf) Accessed: 28 September 2017

20. N. Jansma, B. Arrendondo, (2004). "Performance Comparison of Elliptic Curve and RSA Digital Signatures." (http://nicj.net/files/performance_comparison_of_elliptic_curve_and_rsa_digital_signatures.pdf) 28 September 2017

21. A. Antipa, D. Brown, R. Gallant, R. Lambert, R. Struik, S. Vanstone, (2005). "Accelerated Verification of ECDSA Signatures." (http://www.mathnet.or.kr/mathnet/preprint_file/cacr/2005/cacr2005-28.pdf) Accessed: 28 September 2017

22. H. Krawczyk, M. Bellare, R. Canetti, "Request for Comments: 2104, HMAC: Keyed-Hashing for Message Authentication." http://www.ietf.org/rfc/rfc2104.txt Last accessed: 28 September 2017

23. D. Bernstein, "The cr.yp.to blog: How to Design an Elliptic-curve Signature System" (https://blog.cr.yp.to/20140323-ecdsa.html) Accessed: 12 February 2017

24. H. Edwards, (2007). "A normal form for elliptic curves." Bulletin of the American Mathematical Society 44, pp. 393-422

25. D. Bernstein, T. Lange, (2007). "Faster addition and doubling on elliptic curves" (http://cr.yp.to/newelliptic/newelliptic-20070906.pdf) Accessed: 17 September 2017

26. D. Bernstein, N. Duif, T. Lange, P. Schwabc, B. Yang, (2011). "High-speed high-security signatures" (https://ed25519.cr.yp.to/ed25519-20110926.pdf) Accessed: 17 September 2017

27. S. Josefsson., I. Liusvaara, (IETF) . "Edwards-curve Digital Signature Algorithm (EdDSA)." (https://tools.ietf.org/html/rfc8032) Accessed: 17 September 2017

28. K. Fischer, “L5K Files Link” (http://fog.misty.com/perry/ccs/ec/KF-PhD/) Accessed: 26 November 2017

29. R. Perry, “EdDSA Reference Implementation.” (http://fog.misty.com/perry/ccs/EdDSA/impl/impl.html) Accessed: 17 September 2017

30. S. Mittra, “Iolus: A Framework for Scalable Secure Multicasting” (http://conferences.sigcomm.org/sigcomm/1997/papers/p113.pdf) Accessed: 17 September 2017

31. L. Dondeti, S. Mukherjee, A. Samal, “DISEC: A Distributed Framework for Scalable Secure Many-to-many Communication” (http://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1030&context=cseconfwork) Accessed: 17 September 2017

32. O. Rodeh, K. Birman, D. Dolev, “Optimized Group Rekey for Group Communication System” (https://pdfs.semanticscholar.org/e000/fae10daa731a1faa6eeb82102912d0acf8fd.pdf) Accessed: 17 September 2017

33. M. Baugher, R. Canetti, F. Lindholm, “RFC 4046: Multicast Security (MSEC) Group Key Management Architecture” (https://www.ietf.org/rfc/rfc4046.txt) Accessed: 17 September 2017

34. J. Guo, T. Peyrin, A. Poschmann, “The PHOTON Family of Lightweight Hash Functions” (https://eprint.iacr.org/2011/609.pdf) Accessed: 2 October 2018

35. W. Diffie, M. Hellman (1976) “New Directions in Cryptography”, IEEE Transactions on Information Theory, VOL IT-22, No. 6, pp 644-654

36. NIST FIPS Pub 202 (2015). “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.” (http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf) Accessed: 2 October 2017

37. NIST SP 800-90A (2015). “Recommendation for Random Number Generation Using Deterministic Random Bit Generators.” (http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf) Accessed: 2 October 2017

38. T. Krovetz, W. Dai, “VMAC: Message Authentication Code using Universal Hashing <draft-krovetz-vmac-01.txt>” (http://fastcrypto.org/vmac/draft-krovetz-vmac-01.txt) Accessed: 17 November 2017

APPENDIX A. VMAC ANALYSIS

A.1 Testing SHA-256

```
/************************************************************************
File:           sha.py
************************************************************************/
```

```
"""
SHA-256 constants, functions, processing one block, and testing

Also includes KF VMAC function for one block
"""

import random, zlib

# offset to add to negative values to make them unsigned
#
_offset = (1 << 32)

# mask to just keep 32 bits
#
_mask = _offset - 1

def ROTR(x,n):
  """Circular Rotate Right by n bits"""
  return (x >> n) | ((x << 32-n) & _mask)

def ROTL(x,n):
  """Circular Rotate Left by n bits, not used in SHA-256"""
  return ((x << n) & _mask) | (x >> 32-n)

def SHR(x,n):
  """Shift Right by n bits"""
  return (x >> n)

# ~x produces negative value

def Ch(x,y,z):
  """SHA-256 Ch Function"""
  return (x & y) ^ (((~x)+_offset) & z)

def Maj(x,y,z):
  """SHA-256 Maj Function"""
  return (x & y) ^ (x & z) ^ (y & z)

def Sigma0(x):
  """SHA-256 Sigma0 Function"""
  return ROTR(x,2) ^ ROTR(x,13) ^ ROTR(x,22)

def Sigma1(x):
  """SHA-256 Sigma1 Function"""
  return ROTR(x,6) ^ ROTR(x,11) ^ ROTR(x,25)

def sigma0(x):
  """SHA-256 sigma0 Function"""
  return ROTR(x,7) ^ ROTR(x,18) ^ SHR(x,3)

def sigma1(x):
  """SHA-256 sigma1 Function"""
  return ROTR(x,17) ^ ROTR(x,19) ^ SHR(x,10)

# initial hash value
_H0 = [ 0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
        0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19 ]
```

```
# constants
_K = [ 0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
       0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
       0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
       0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
       0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
       0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
       0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
       0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
       0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
       0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
       0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
       0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
       0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
       0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
       0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
       0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2 ]

def SHA256(M, N=64):
  """
  SHA256 on data with size <= 447 bits so it fits in one 512-bit block M
  e.g. M = 32-bit-data1, ..., data13, 0x80000000, 0x0, data_bit_length

  N >= 16 is required.  N < 64 represents a reduced-round SHA-256.
  """
  W = [0 for t in range(N)]
  # unpack M into array of 16 32-bit values
  for t in range(15,-1,-1):
    W[t] = M & _mask
    M >>= 32
  for t in range(16,N):
    W[t] = (sigma1(W[t-2]) + W[t-7] + sigma0(W[t-15]) + W[t-16]) & _mask
  a = _H0[0]; b = _H0[1]; c = _H0[2]; d = _H0[3]
  e = _H0[4]; f = _H0[5]; g = _H0[6]; h = _H0[7]
  for t in range(N):
    T1 = (h + Sigma1(e) + Ch(e,f,g) + _K[t] + W[t]) & _mask
    T2 = (Sigma0(a) + Maj(a,b,c)) & _mask
    h = g; g = f; f = e; e = (d + T1) & _mask
    d = c; c = b; b = a; a = (T1 + T2) & _mask
  H =             (a + _H0[0]) & _mask
  H <<= 32;  H |= (b + _H0[1]) & _mask
  H <<= 32;  H |= (c + _H0[2]) & _mask
  H <<= 32;  H |= (d + _H0[3]) & _mask
  H <<= 32;  H |= (e + _H0[4]) & _mask
  H <<= 32;  H |= (f + _H0[5]) & _mask
  H <<= 32;  H |= (g + _H0[6]) & _mask
  H <<= 32;  H |= (h + _H0[7]) & _mask
  return H

def VMAC(M, N=20, KEY=0):
  """
  KF VMAC on one block
  """
  # debug:
  # global a, b, c, d, e, f, g, h, T0, T1, T2
  # global Key
  global hprev, W
  #
  # unpack KEY into array of 8 32-bit values
  #
  Key = [0 for t in range(8)]
  for t in range(7,-1,-1):
    Key[t] = KEY & _mask
    KEY >>= 32
  #
  # unpack M into array of 16 32-bit values XOR'd with Key
  #
  W = [0 for t in range(N)]
  KeyIndex = [ 0, 1, 2, 3, 4, 5, 6, 7, 3, 4, 5, 6, 7, 1, 2, 0]
  for t in range(15,-1,-1):
    ki = KeyIndex[t]
    W[t] = (M & _mask) ^ ROTR( Key[ki], 3*ki+(t>>3))
    M >>= 32
  for t in range(16,N):
    ki = (63-t)%8
    W[t] = ((sigma1(W[t-2]) + W[t-7] + sigma0(W[t-15]) + W[t-16]) & _mask) \
           ^ ROTR( Key[ki], 3*ki+(t>>3))
  #
  a = _H0[0]; b = _H0[1]; c = _H0[2]; d = _H0[3]
  e = _H0[4]; f = _H0[5]; g = _H0[6]; h = _H0[7]
  #
  for t in range(N):
    T0 = (h + Sigma1(e) + Ch(e,f,g) + _K[t]) & _mask
    T1 = (T0 + W[t]) & _mask
    T2 = (Sigma0(a) + Maj(a,b,c)) & _mask
    hprev = h;
    h = g; g = f; f = e; e = (d + T1) & _mask
    d = c; c = b; b = a; a = (T1 + T2) & _mask
  #
  H = (a + _H0[0]) & _mask
  H <<= 32;  H |= (b + _H0[1]) & _mask
  H <<= 32;  H |= (c + _H0[2]) & _mask
  H <<= 32;  H |= (d + _H0[3]) & _mask
  H <<= 32;  H |= (e + _H0[4]) & _mask
  H <<= 32;  H |= (f + _H0[5]) & _mask
  H <<= 32;  H |= (g + _H0[6]) & _mask
  H <<= 32;  H |= (h + _H0[7]) & _mask
  return H

def bitstats(MSG,N):
  """
  count how many bits change vs. 1-bit change on input
  using SHA-256 main loop range 1:N (instead of 1:64)

  measure randomness by size of compressed hash (Z)

  returns (C,Cmin,Cmax,Cavg,Zmin,Zmax,Zavg)
  """
  C = [0 for i in range(257)]
  Cmin = Zmin = 256; Cmax = Cavg = Zmax = Zavg = 0
  mask = 1 << 511
  MD = SHA256( MSG, N) # original hash
  while mask > 0:
    # MSG ^ mask flips one bit in MSG
    # h ^ MD has non-zero bits where result != MD
    h = SHA256( MSG ^ mask, N)
    d = h ^ MD
    n = bin(d).count('1')
    C[n] += 1
    if n < Cmin: Cmin = n
    if n > Cmax: Cmax = n
    # compressed length will be > 32 bytes if h is very random
    Z = len(zlib.compress( h.to_bytes((h.bit_length()+7)//8,'big') ))
    Zavg += Z
    if Z < Zmin: Zmin = Z
    if Z > Zmax: Zmax = Z
    mask >>= 1
  Zavg /= 512
  for i in range(257): Cavg += i*C[i]
  Cavg /= 512 # divide by number of trials
  return (C,Cmin,Cmax,Cavg,Zmin,Zmax,Zavg)

def block(MSG):
  """put 16 32-bit values into one 512-bit block"""
  M = MSG[0]
  for i in range(1,16):
    M = (M << 32) | MSG[i]
  return M

# some test inputs and outputs from SHA256ShortMsg.rsp
# http://csrc.nist.gov/groups/STM/cavp/documents/shs/shabytetestvectors.zip
#
```

```
_MSG_0 = [ 0x80000000, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
           0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 ]
MSG_0 = block(_MSG_0)
MD_0 = 0xe3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855

_MSG_8 = [ 0xd3800000, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
          0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x8 ]
MSG_8 = block(_MSG_8)
MD_8 = 0x28969cdfa74a12c82f3bad960b0b000aca2ac329deea5c2328ebc6f2ba9802c1

MSG_440 =
0x3ebfb06db8c38d5ba037f1363e118550aad94606e26835a01af05078533cc25f2f39573c04b632f62f68c29
4ab31f2a3e2a1a0d8c2be51800000000000000001b8
MD_440 = 0x6595a2ef537a69ba8583dfbf7f5bec0ab1f93ce4c8ee1916eff44a93af5749c4

del _MSG_0, _MSG_8
```

/**

File: test.py

**/

```
from sha import *

tests = [ "MSG_0, 64", "MSG_8, 64", "MSG_440, 64",
           "MSG_0, 32", "MSG_8, 32", "MSG_440, 32",
           "MSG_0, 20", "MSG_8, 20", "MSG_440, 20",
           "MSG_0, 16", "MSG_8, 16", "MSG_440, 16" ]

for what in tests:
  test = "bitstats(" + what + ")"
  print(test)
  (C,Cmin,Cmax,Cavg,Zmin,Zmax,Zavg) = eval(test)
  print(Cmin,Cmax,Cavg,Zmin,Zmax,Zavg)
```

A.2 Cracking 16-Rounds

/**

File: crack16.py

**/

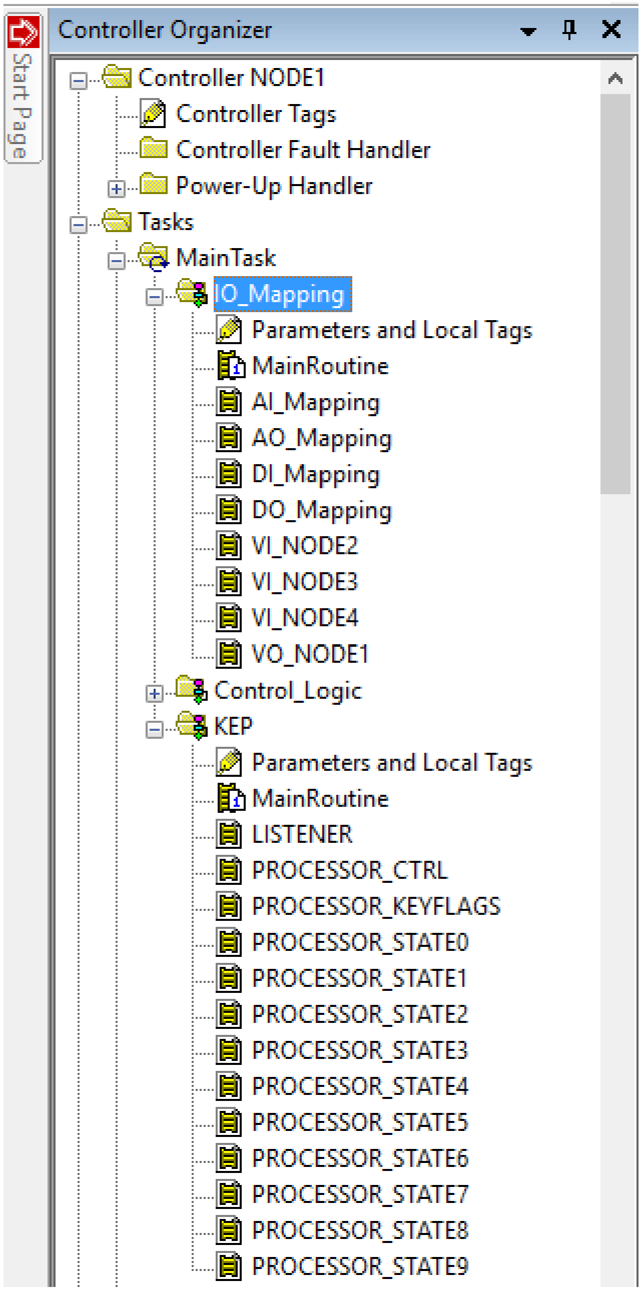
```
# Crack one-block VMAC with N=16 rounds using arbitrary input data
#
# The input data does not conform to the SHA-256 padding requirements.
#
# This finds bits 0 to 30 of the rotated Key[0] used in round 16.
#
# Bit 31 can not be found using this method because the carry bit related to
# that is lost in the 32-bit additions.  To continue, try both ways: with
# that bit low and with that bit high.  For each way, unwind H0 and H1 one
# more round, and find bits 0 to 30 of rotated Key[1].  Again bit 31 will
# be unknown, so another branch into two ways is required.  After 8 unwind
# levels 8 bits of KEY will remain unknown, and those 256 possibilities can
# just be tested to determine those bits.
#
# Only one unwind is done here, demonstrating the concept.
```

```
import sha, random

N = 16

# extract internal SHA-256 variables
#
def extract(H):
  h = (H - sha._H0[7]) & sha._mask; H >>= 32
  g = (H - sha._H0[6]) & sha._mask; H >>= 32
  f = (H - sha._H0[5]) & sha._mask; H >>= 32
  e = (H - sha._H0[4]) & sha._mask; H >>= 32
  d = (H - sha._H0[3]) & sha._mask; H >>= 32
  c = (H - sha._H0[2]) & sha._mask; H >>= 32
  b = (H - sha._H0[1]) & sha._mask; H >>= 32
  a = (H - sha._H0[0]) & sha._mask;
  #
  # check
  #
  # H = (a + sha._H0[0]) & sha._mask
  # H <<= 32;  H |= (b + sha._H0[1]) & sha._mask
  # H <<= 32;  H |= (c + sha._H0[2]) & sha._mask
  # H <<= 32;  H |= (d + sha._H0[3]) & sha._mask
  # H <<= 32;  H |= (e + sha._H0[4]) & sha._mask
  # H <<= 32;  H |= (f + sha._H0[5]) & sha._mask
  # H <<= 32;  H |= (g + sha._H0[6]) & sha._mask
  # H <<= 32;  H |= (h + sha._H0[7]) & sha._mask
  #
  return (a,b,c,d,e,f,g,h)

# extract V = T1 - offset = h + W[N-1]
#
# The previous round values a,b,c,d,e,f,g,T1,T2 can be determined directly,
# but not h (see notes in file REVERSE here).
#
def reverse( D, N, K):
  H = sha.VMAC( D, N, K)
  W = sha.W[N-1]
  hprev = sha.hprev
  (an,b,c,d,e,f,g,h) = extract(H)
  #
  a = b; b = c; c = d; e = f; f = g; g = h
  T2 = (sha.Sigma0(a) + sha.Maj(a,b,c)) & sha._mask
  T1 = (an - T2) & sha._mask
  V = (T1 - sha.Sigma1(e) - sha.Ch(e,f,g) - sha._K[N-1]) & sha._mask
  #
  if V != (hprev + W) & sha._mask: print( "error in reverse, V is wrong")
  return V

# random KEY
#
KEY = random.getrandbits(256)

Key0 = sha.ROTR( KEY>>224, 3*0+(15>>3)) # Key[0] XOR with W[15] in VMAC()

print( " Key0 =", format(Key0,'032b'))

# K0 will hold the cracked key bits
#
K0 = 0

# R will hold the h bits
#
R = 0

# arbitrary random input Data
#
Data = random.getrandbits(512)

# masks to set bit 0 of Data high or low
#
```

```
ones  = (1 << 512) - 1 # 11...11
mask1 = 1              # 00...01
mask0 = ones ^ mask1   # 11...10

# find bit n of Key0
#
for n in range(31):
  #
  # W[15][n] = 0 or 1
  #
  V0 = reverse(Data & mask0,16,KEY)
  # W0 = sha.W[N-1]

  if n == 0: hprev0 = sha.hprev
  elif hprev0 != sha.hprev: print( "error, hprev0 changed")

  V1 = reverse(Data | mask1,16,KEY)
  # W1 = sha.W[N-1]

  if hprev0 != sha.hprev: print( "error, hprev1 changed")

  # for next iteration
  #
  mask1 <<= 1
  mask0 = ones ^ mask1

  # T1 ~= h + (Data XOR Key0), with T1 and Data known, h and Key0 unknown.
  #
  # A single bit flip in Data will cause a corresponding bit flip in T1,
  # and based on the change in the carry bit related to that flip, we can
  # determine a bit of h and Key0.
  #
  # V = T1 - offset = h + W[N-1]
  #
  # where offset includes R, the previously calculated low-order bits of h,
  # to eliminate carry propagation from those bits in the addition

  V0 = (V0 - R) & sha._mask

  V1 = (V1 - R) & sha._mask

  # print( "V0 =", format(V0,'032b'))
  # print( "V1 =", format(V1,'032b'))

  # print( "W0 =", format(W0,'032b'))
  # print( "W1 =", format(W1,'032b'), "\n")

  # get r = bit n of h, and k = bit n of Key0
  #
  # bits: i = input data, k = key, (v1,v0) = (carry, V bit n)
  #
  #   i k r (v1,v0) = (i XOR k) + r
  #   -----  -----
  # * 0 0 0   0  0
  #   1 0 0   0  1 no change in carry
  #
  # * 0 1 0   0  1
  #   1 1 0   0  0 no change
  #
  # * 0 1 1   1  0
  #   1 1 1   0  1 change
  #
  # * 0 0 1   0  1
  #   1 0 1   1  0 change
  #
  # As the input bit i changes 0->1, v0 changes 0->1 or 1->0,
  # and if the v1 carry bit changes then r = 1 else r = 0
  #
  # Take the four rows marked with (*) from the table above
  # (with i = 0), and rearrange the columns:
  #
  #  v0 r  k
  #  ----  -
  #   0 0  0
  #   1 0  1
  #   0 1  1
  #   1 1  0
  #
  # So k = v0 XOR r
  #
  # Alternate derivation:  v0 = i XOR k XOR r  ==>  k = v0 XOR r XOR i
  #
  v0 = (V0 >> n) & 1; w0 = (V1 >> n) & 1 # these bits must differ

  if v0 == w0: print( "error, v0 = w0") # consistency check

  r = ((V0 >> (n+1)) & 1) ^ ((V1 >> (n+1)) & 1) # change in carry

  k = v0 ^ r # key bit

  # print( "k =", k)

  if k != ((Key0 >> n) & 1): print( "error, k bit", n, "is wrong")

  if r != ((hprev0 >> n) & 1): print( "error, r bit", n, "is wrong")

  # insert k into K0 and r into R
  #
  K0 |= k << n
  R  |= r << n

print( "   K0 =", format(K0,'032b'))
print( "hprev =", format(hprev0,'032b'))
print( "    R =", format(R,'032b'))

# sample runs:
#
#  Key0 = 10010101111101010001000100010011
#    K0 = 00010101111101010001000100010011
# hprev = 10010001110011101110001100001011
#     R = 00010001110011101110001100001011
#
#  Key0 = 11001100001100000111011111011010
#    K0 = 01001100001100000111011111011010
# hprev = 00001011000100000001101010010000
#     R = 00001011000100000001101010010000
#
#  Key0 = 00110100000010110000100010010010
#    K0 = 00110100000010110000100010010010
# hprev = 11100011110111001011001101110100
#     R = 01100011110111001011001101110100
# --------^
# this bit is not determined here
```

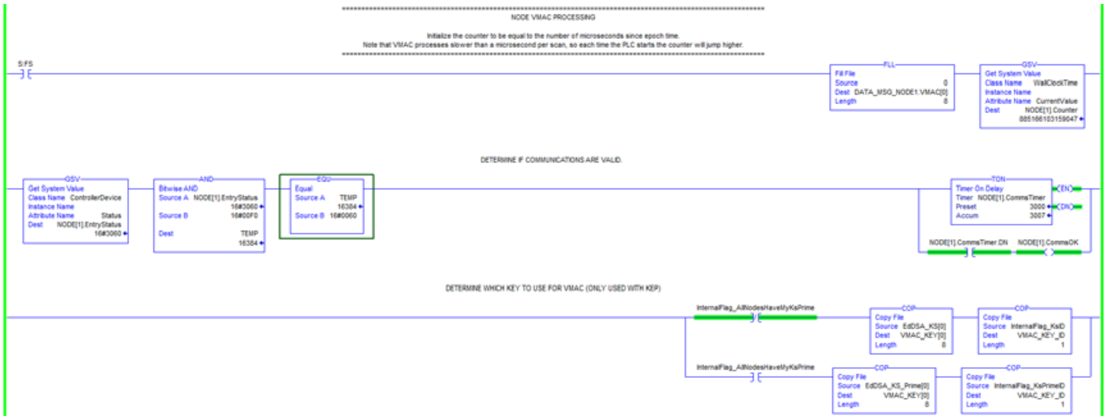
APPENDIX B. GRAPHICAL VMAC IMPLEMENTATION

B.1 Controller Organizer

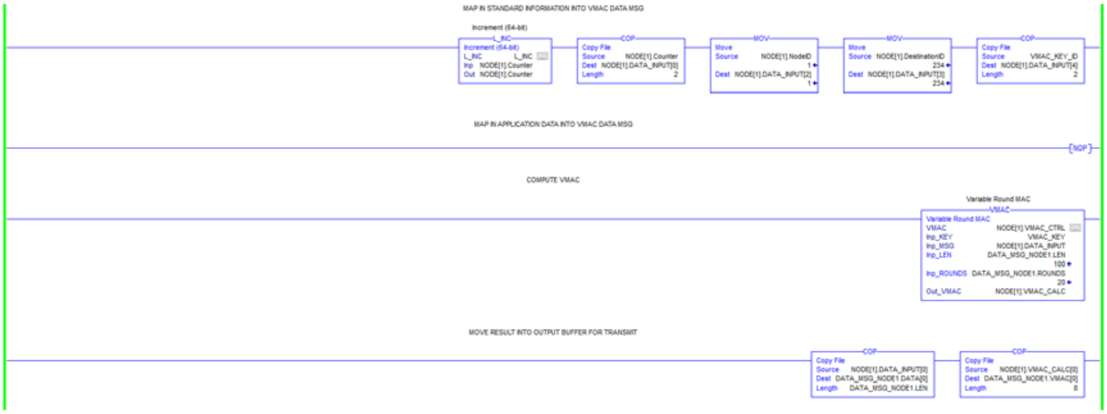


```
Controller NODE1
    Controller Tags
    Controller Fault Handler
    Power-Up Handler
Tasks
    MainTask
        IO_Mapping
            Parameters and Local Tags
            MainRoutine
            AI_Mapping
            AO_Mapping
            DI_Mapping
            DO_Mapping
            VI_NODE2
            VI_NODE3
            VI_NODE4
            VO_NODE1
        Control_Logic
        KEP
            Parameters and Local Tags
            MainRoutine
            LISTENER
            PROCESSOR_CTRL
            PROCESSOR_KEYFLAGS
            PROCESSOR_STATE0
            PROCESSOR_STATE1
            PROCESSOR_STATE2
            PROCESSOR_STATE3
            PROCESSOR_STATE4
            PROCESSOR_STATE5
            PROCESSOR_STATE6
            PROCESSOR_STATE7
            PROCESSOR_STATE8
            PROCESSOR_STATE9
```

B.2 VO_NODE1

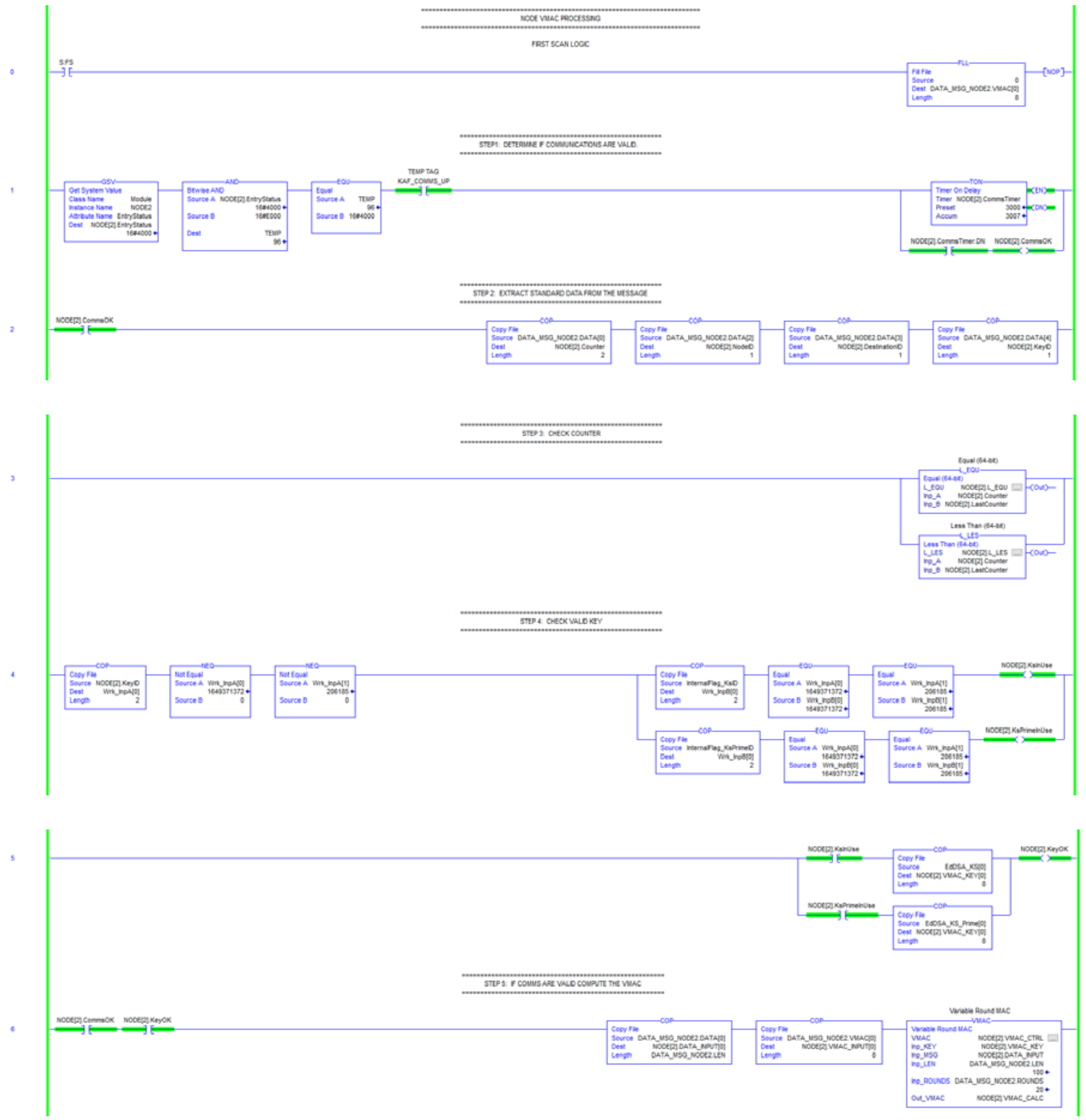


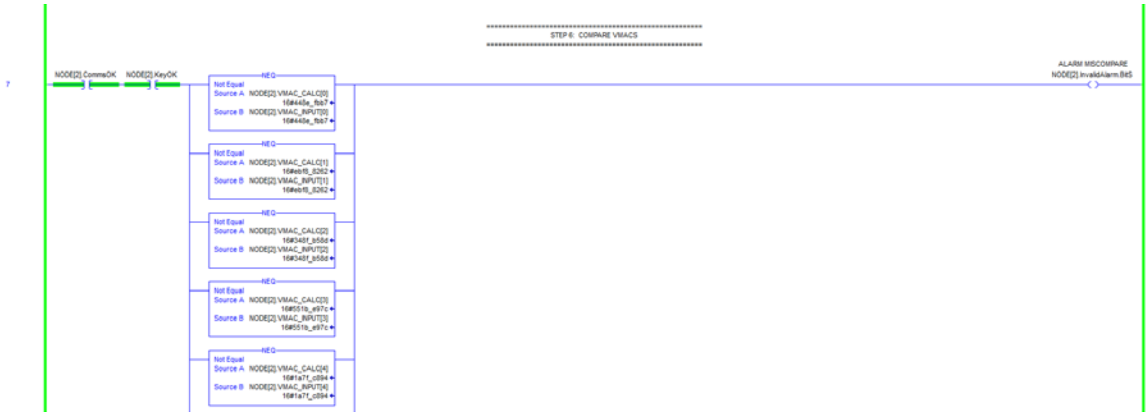
```
************************************************************************************************
NODE VMAC PROCESSING

Initialize the counter to be equal to the number of microseconds since epoch time.
Note that VMAC processes slower than a microsecond per scan, so each time the PLC starts the counter will jump higher.
************************************************************************************************
0   S:FS
    ] [                                                     -FLL-                          -GSV-
                                                            Fill File                      Get System Value
                                                            Source                   0     Class Name      WallClockTime
                                                            Dest  DATA_MSG_NODE1.VMAC[0]   Instance Name
                                                            Length                   8     Attribute Name  CurrentValue
                                                                                           Dest            NODE[1].Counter
                                                                                                   885166103159047

DETERMINE IF COMMUNICATIONS ARE VALID.
1   -GSV-                            -AND-                         -EQU-                          -TON-
    Get System Value                 Bitwise AND                   Equal                          Timer On Delay                    (EN)
    Class Name  ControllerDevice     Source A  NODE[1].EntryStatus Source A   TEMP                Timer  NODE[1].CommsTimer
    Instance Name                                      16#3060                16384               Preset              3000      (DN)
    Attribute Name        Status     Source B          16#00F0     Source B  16#0060              Accum               3007
    Dest     NODE[1].EntryStatus
                          16#3060    Dest                 TEMP                                    NODE[1].CommsTimer.DN   NODE[1].CommsOK
                                                         16384                                    ] [                     ( )

DETERMINE WHICH KEY TO USE FOR VMAC (ONLY USED WITH KEP)
2   InternalFlag_AllNodesHaveMyKsPrime     -COP-                      -COP-
    ]/[                                    Copy File                  Copy File
                                           Source  EdDSA_KS[0]        Source  InternalFlag_KsID
                                           Dest    VMAC_KEY[0]        Dest    VMAC_KEY_ID
                                           Length            8        Length            1
    InternalFlag_AllNodesHaveMyKsPrime     -COP-                          -COP-
    ] [                                    Copy File                      Copy File
                                           Source  EdDSA_KS_Prime[0]      Source  InternalFlag_KsPrimeID
                                           Dest          VMAC_KEY[0]      Dest           VMAC_KEY_ID
                                           Length                  8      Length                   1
```



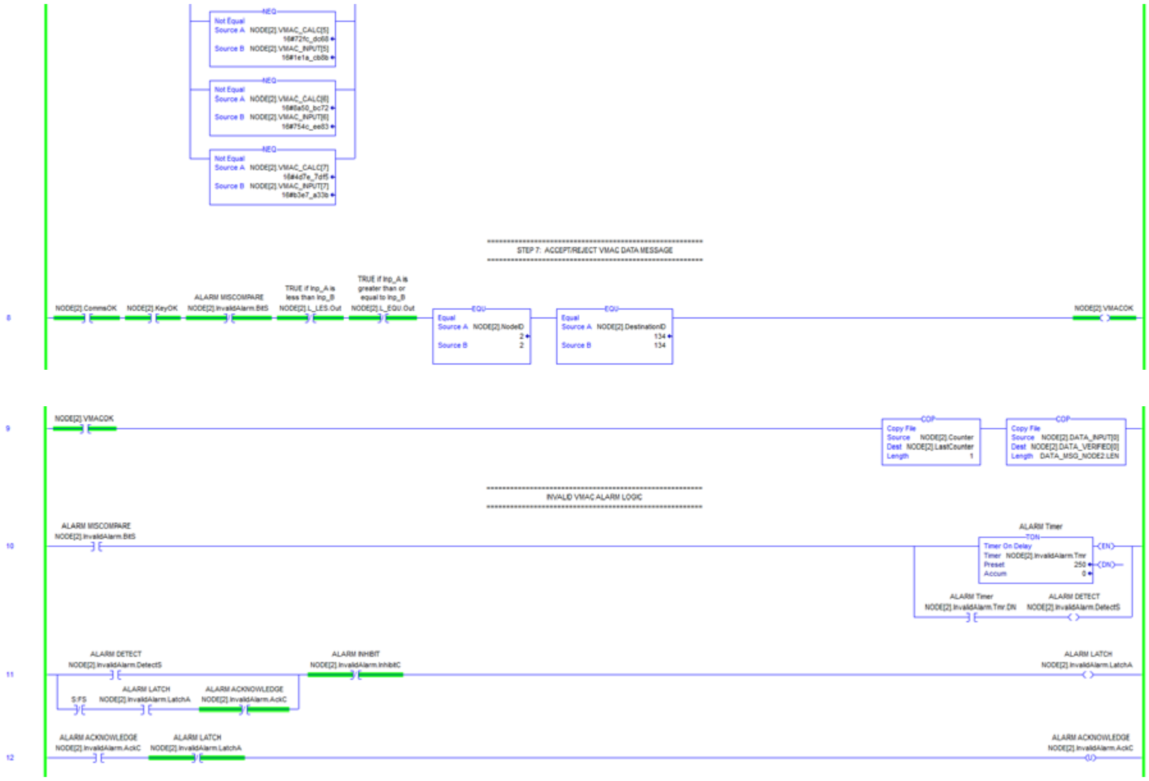
```
MAP IN STANDARD INFORMATION INTO VMAC DATA MSG
    Increment (64-bit)
3   -L_INC-                      -COP-                           -MOV-                            -MOV-                                -COP-
    Increment (64-bit)           Copy File                       Move                             Move                                 Copy File
    L_INC          L_INC  ...    Source      NODE[1].Counter     Source         NODE[1].NodeID    Source  NODE[1].DestinationID        Source        VMAC_KEY_ID
    Inp  NODE[1].Counter         Dest  NODE[1].DATA_INPUT[0]                                 1                                  234    Dest  NODE[1].DATA_INPUT[4]
    Out  NODE[1].Counter         Length                    2     Dest  NODE[1].DATA_INPUT[2]      Dest  NODE[1].DATA_INPUT[3]          Length                    2
                                                                                             1                                  234

MAP IN APPLICATION DATA INTO VMAC DATA MSG
4                                                                                                                                  {NOP}

COMPUTE VMAC
                                                           Variable Round MAC
5                                                          -VMAC-
                                                           Variable Round MAC
                                                           VMAC                 NODE[1].VMAC_CTRL  ...
                                                           Inp_KEY                       VMAC_KEY
                                                           Inp_MSG             NODE[1].DATA_INPUT
                                                           Inp_LEN            DATA_MSG_NODE1.LEN
                                                                                              100
                                                           Inp_ROUNDS  DATA_MSG_NODE1.ROUNDS
                                                                                               20
                                                           Out_VMAC             NODE[1].VMAC_CALC

MOVE RESULT INTO OUTPUT BUFFER FOR TRANSMIT
6                                     -COP-                                 -COP-
                                      Copy File                             Copy File
                                      Source     NODE[1].DATA_INPUT[0]      Source    NODE[1].VMAC_CALC[0]
                                      Dest  DATA_MSG_NODE1.DATA[0]          Dest  DATA_MSG_NODE1.VMAC[0]
                                      Length     DATA_MSG_NODE1.LEN         Length                       8
```

B.2 VI_NODE2

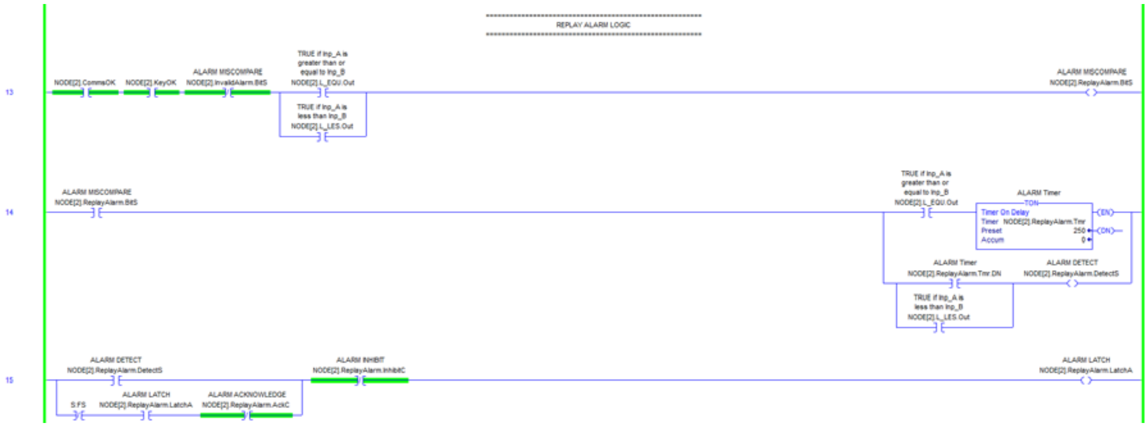

NODE VMAC PROCESSING
FIRST SCAN LOGIC
0
S:FS
FLL
Fill File
Source 0
Dest DATA_MSG_NODE2.VMAC[0]
Length 8
NOP
STEP1: DETERMINE IF COMMUNICATIONS ARE VALID.
1
GSV
Get System Value
Class Name Module
Instance Name NODE2
Attribute Name EntryStatus
Dest NODE[2].EntryStatus
16#4000
AND
Bitwise AND
Source A NODE[2].EntryStatus
16#4000
Source B 16#E000
Dest TEMP
96
EQU
Equal
Source A TEMP
96
Source B 16#4000
TEMP TAG
KAF_COMMS_UP
TON
Timer On Delay
Timer NODE[2].CommsTimer
Preset 3000
Accum 3007
EN
DN
NODE[2].CommsTimer.DN
NODE[2].CommsOK
STEP 2: EXTRACT STANDARD DATA FROM THE MESSAGE
2
NODE[2].CommsOK
COP
Copy File
Source DATA_MSG_NODE2.DATA[0]
Dest NODE[2].Counter
Length 2
COP
Copy File
Source DATA_MSG_NODE2.DATA[2]
Dest NODE[2].NodeID
Length 1
COP
Copy File
Source DATA_MSG_NODE2.DATA[3]
Dest NODE[2].DestinationID
Length 1
COP
Copy File
Source DATA_MSG_NODE2.DATA[4]
Dest NODE[2].KeyID
Length 1
STEP 3: CHECK COUNTER
3
Equal (64-bit)
L_EQU
Equal (64-bit)
L_EQU NODE[2].L_EQU
Inp_A NODE[2].Counter
Inp_B NODE[2].LastCounter
Out
Less Than (64-bit)
L_LES
Less Than (64-bit)
L_LES NODE[2].L_LES
Inp_A NODE[2].Counter
Inp_B NODE[2].LastCounter
Out
STEP 4: CHECK VALID KEY
4
COP
Copy File
Source NODE[2].KeyID
Dest Wrk_InpA[0]
Length 2
NEQ
Not Equal
Source A Wrk_InpA[0]
1649371372
Source B 0
NEQ
Not Equal
Source A Wrk_InpA[1]
206185
Source B 0
COP
Copy File
Source InternalFlag_KsID
Dest Wrk_InpB[0]
Length 2
EQU
Equal
Source A Wrk_InpA[0]
1649371372
Source B Wrk_InpB[0]
1649371372
EQU
Equal
Source A Wrk_InpA[1]
206185
Source B Wrk_InpB[1]
206185
NODE[2].KsInUse
COP
Copy File
Source InternalFlag_KsPrimeID
Dest Wrk_InpB[0]
Length 2
EQU
Equal
Source A Wrk_InpA[0]
1649371372
Source B Wrk_InpB[0]
1649371372
EQU
Equal
Source A Wrk_InpA[1]
206185
Source B Wrk_InpB[1]
206185
NODE[2].KsPrimeInUse
5
NODE[2].KsInUse
COP
Copy File
Source EdDSA_KS[0]
Dest NODE[2].VMAC_KEY[0]
Length 8
NODE[2].KeyOK
NODE[2].KsPrimeInUse
COP
Copy File
Source EdDSA_KS_Prime[0]
Dest NODE[2].VMAC_KEY[0]
Length 8
STEP 5: IF COMMS ARE VALID COMPUTE THE VMAC
6
NODE[2].CommsOK
NODE[2].KeyOK
COP
Copy File
Source DATA_MSG_NODE2.DATA[0]
Dest NODE[2].DATA_INPUT[0]
Length DATA_MSG_NODE2.LEN
COP
Copy File
Source DATA_MSG_NODE2.VMAC[0]
Dest NODE[2].VMAC_INPUT[0]
Length 8
Variable Round MAC
VMAC
Variable Round MAC
VMAC NODE[2].VMAC_CTRL
Inp_KEY NODE[2].VMAC_KEY
Inp_MSG NODE[2].DATA_INPUT
Inp_LEN DATA_MSG_NODE2.LEN
100
Inp_ROUNDS DATA_MSG_NODE2.ROUNDS
20
Out_VMAC NODE[2].VMAC_CALC



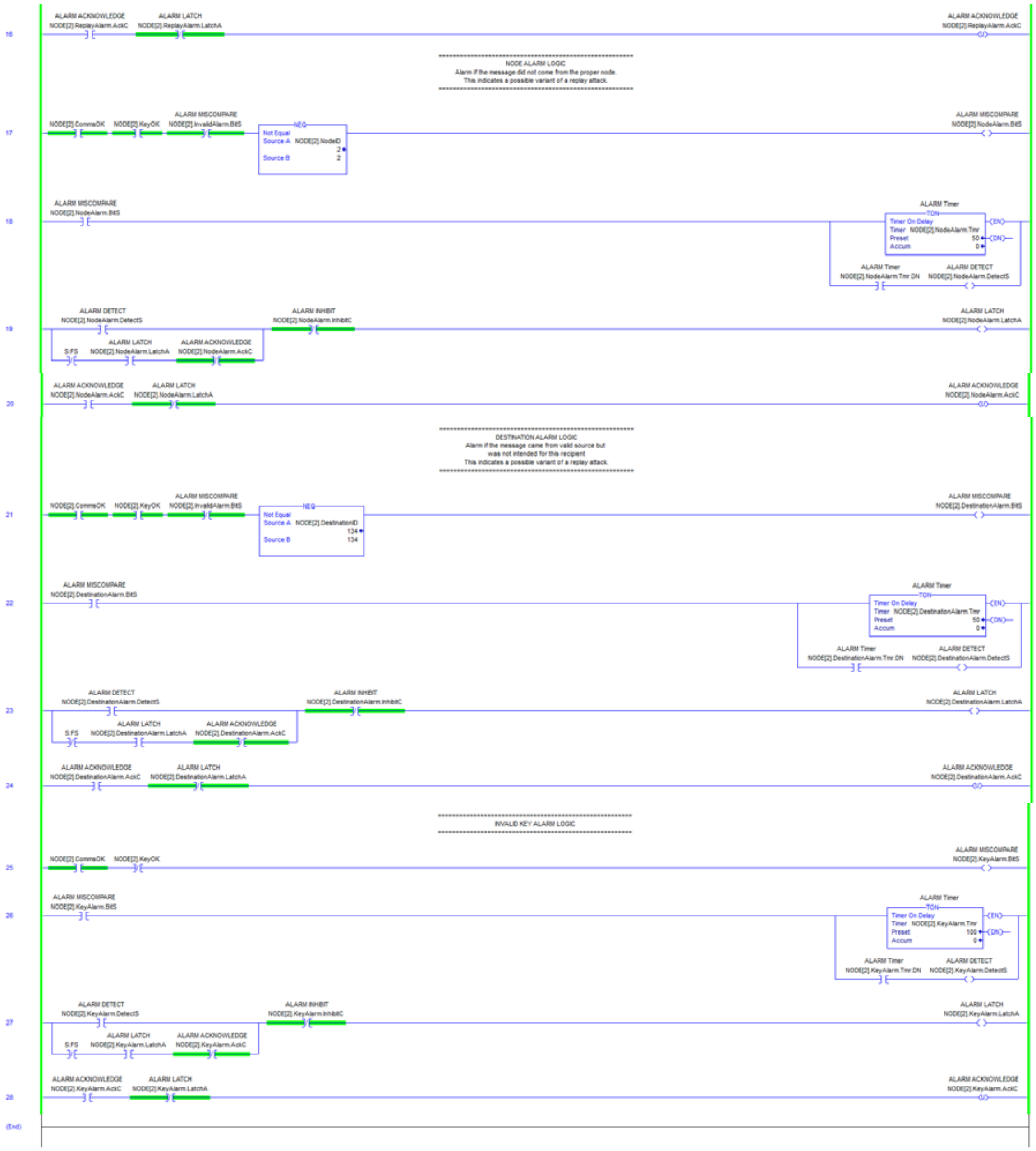
STEP 6: COMPARE VMACS
7
NODE[2].CommsOK
NODE[2].KeyOK
NEQ
Not Equal
Source A NODE[2].VMAC_CALC[0]
16#448e_fbb7
Source B NODE[2].VMAC_INPUT[0]
16#448e_fbb7
NEQ
Not Equal
Source A NODE[2].VMAC_CALC[1]
16#ebf8_8262
Source B NODE[2].VMAC_INPUT[1]
16#ebf8_8262
NEQ
Not Equal
Source A NODE[2].VMAC_CALC[2]
16#348f_b58d
Source B NODE[2].VMAC_INPUT[2]
16#348f_b58d
NEQ
Not Equal
Source A NODE[2].VMAC_CALC[3]
16#551b_e97c
Source B NODE[2].VMAC_INPUT[3]
16#551b_e97c
NEQ
Not Equal
Source A NODE[2].VMAC_CALC[4]
16#1a7f_c894
Source B NODE[2].VMAC_INPUT[4]
16#1a7f_c894
ALARM MISCOMPARE
NODE[2].InvalidAlarm.BitS



NEQ
Not Equal
Source A NODE[2].VMAC_CALC[5]
16#72fc_dc68
Source B NODE[2].VMAC_INPUT[5]
16#1e1a_cb8b
NEQ
Not Equal
Source A NODE[2].VMAC_CALC[6]
16#8a50_bc72
Source B NODE[2].VMAC_INPUT[6]
16#754c_ee83
NEQ
Not Equal
Source A NODE[2].VMAC_CALC[7]
16#4d7e_7df5
Source B NODE[2].VMAC_INPUT[7]
16#b3e7_a33b
STEP 7: ACCEPT/REJECT VMAC DATA MESSAGE
8
NODE[2].CommsOK
NODE[2].KeyOK
ALARM MISCOMPARE
NODE[2].InvalidAlarm.BitS
TRUE if Inp_A is
less than Inp_B
NODE[2].L_LES.Out
TRUE if Inp_A is
greater than or
equal to Inp_B
NODE[2].L_EQU.Out
EQU
Equal
Source A NODE[2].NodeID
2
Source B
2
EQU
Equal
Source A NODE[2].DestinationID
134
Source B
134
NODE[2].VMACOK
9
NODE[2].VMACOK
COP
Copy File
Source NODE[2].Counter
Dest NODE[2].LastCounter
Length 1
COP
Copy File
Source NODE[2].DATA_INPUT[0]
Dest NODE[2].DATA_VERIFIED[0]
Length DATA_MSG_NODE2.LEN
INVALID VMAC ALARM LOGIC
10
ALARM MISCOMPARE
NODE[2].InvalidAlarm.BitS
ALARM Timer
TON
Timer On Delay
Timer NODE[2].InvalidAlarm.Tmr
Preset 250
Accum 0
EN
DN
ALARM Timer
NODE[2].InvalidAlarm.Tmr.DN
ALARM DETECT
NODE[2].InvalidAlarm.DetectS
11
ALARM DETECT
NODE[2].InvalidAlarm.DetectS
S:FS
ALARM LATCH
NODE[2].InvalidAlarm.LatchA
ALARM ACKNOWLEDGE
NODE[2].InvalidAlarm.AckC
ALARM INHIBIT
NODE[2].InvalidAlarm.InhibitC
ALARM LATCH
NODE[2].InvalidAlarm.LatchA
12
ALARM ACKNOWLEDGE
NODE[2].InvalidAlarm.AckC
ALARM LATCH
NODE[2].InvalidAlarm.LatchA
ALARM ACKNOWLEDGE
NODE[2].InvalidAlarm.AckC
U



REPLAY ALARM LOGIC
13
NODE[2].CommsOK
NODE[2].KeyOK
ALARM MISCOMPARE
NODE[2].InvalidAlarm.BitS
TRUE if Inp_A is
greater than or
equal to Inp_B
NODE[2].L_EQU.Out
TRUE if Inp_A is
less than Inp_B
NODE[2].L_LES.Out
ALARM MISCOMPARE
NODE[2].ReplayAlarm.BitS
14
ALARM MISCOMPARE
NODE[2].ReplayAlarm.BitS
TRUE if Inp_A is
greater than or
equal to Inp_B
NODE[2].L_EQU.Out
ALARM Timer
TON
Timer On Delay
Timer NODE[2].ReplayAlarm.Tmr
Preset 250
Accum 0
EN
DN
ALARM Timer
NODE[2].ReplayAlarm.Tmr.DN
ALARM DETECT
NODE[2].ReplayAlarm.DetectS
TRUE if Inp_A is
less than Inp_B
NODE[2].L_LES.Out
15
ALARM DETECT
NODE[2].ReplayAlarm.DetectS
S:FS
ALARM LATCH
NODE[2].ReplayAlarm.LatchA
ALARM ACKNOWLEDGE
NODE[2].ReplayAlarm.AckC
ALARM INHIBIT
NODE[2].ReplayAlarm.InhibitC
ALARM LATCH
NODE[2].ReplayAlarm.LatchA



NODE ALARM LOGIC
Alarm if the message did not come from the proper node.
This indicates a possible variant of a replay attack.

DESTINATION ALARM LOGIC
Alarm if the message came from valid source but
was not intended for this recipient
This indicates a possible variant of a replay attack.

21
NODE[2].CommsOK
NODE[2].KeyOK
ALARM MISCOMPARE
NODE[2].InvalidAlarm.BitS
NEQ
Not Equal
Source A NODE[2].DestinationID
134
Source B 134
ALARM MISCOMPARE
NODE[2].DestinationAlarm.BitS
22
ALARM MISCOMPARE
NODE[2].DestinationAlarm.BitS
ALARM Timer
TON
Timer On Delay
Timer NODE[2].DestinationAlarm.Tmr
Preset 50
Accum 0
EN
DN
ALARM Timer
NODE[2].DestinationAlarm.Tmr.DN
ALARM DETECT
NODE[2].DestinationAlarm.DetectS
23
ALARM DETECT
NODE[2].DestinationAlarm.DetectS
S:FS
ALARM LATCH
NODE[2].DestinationAlarm.LatchA
ALARM ACKNOWLEDGE
NODE[2].DestinationAlarm.AckC
ALARM INHIBIT
NODE[2].DestinationAlarm.InhibitC
ALARM LATCH
NODE[2].DestinationAlarm.LatchA
24
ALARM ACKNOWLEDGE
NODE[2].DestinationAlarm.AckC
ALARM LATCH
NODE[2].DestinationAlarm.LatchA
ALARM ACKNOWLEDGE
NODE[2].DestinationAlarm.AckC
U

INVALID KEY ALARM LOGIC

APPENDIX C. SOURCE CODE

NOTE: Double click on the icon below in order to open the file. For printed copies of this thesis refer to reference [28] to download the source files.

C.1 Custom Data Types

DATA TYPES FINAL.zip

C.2 Add-On Instructions

AOI MASTERS FINAL.zip

C.3 VMAC and IO_Mapping

VMAC AND IO MAPPING FINAL.zip

C.4 KEP Program

KEP FINAL.zip

APPENDIX D. SEQUENCER CHARTS

D.1 Ed25519_SIGN

SEQUENCE CONTROL COMMANDS	BUILD BASE POINT AND INCREMENT COUNTER	COMBINE RH AND DH INTO MSG1	ADD ENCRYPTED VMAC KEY, CONTROL FLAGS, KEYIDs, AND COUNTER INTO MSG1	h1 = SHA512(MSG,LEN=16)	TEMP_D1 = B512_HtoD (h1)	E TEMP_E1 (re) = D TEMP_D1 mod L	SPARE	P q = POINT_MUL(E r * B)	E SIG_R = POINT_ENC (P q)	OUTPUT RESULT: S SIG_R = B256_StoSLE(B256_CtoS(B256_EtoC	COMBINE POINT R MSG2	ADD PUB KEY A AND DH INTO MSG2	ADD ENCRYPTED VMAC KEY, CONTROL FLAGS, KEYIDs, AND COUNTER INTO MSG2	h2 = SHA512(MSG,LEN=20)	E TEMP_E2 (ke) = D TEMP_D1 mod L	SPARE	SIGN: C TEMP_C1 (rc) = B256_EtoC(TEMP_E1(re))	SIGN: C TEMP_C2 (kc) = B256_EtoC(TEMP_E2 (ke))	SIGN: COP w[0..21]=r[0..21], FLL w[22..43]=0	SPARE	SIGN: Initialize i=-1 and Inp_PriKey_scalar into Temp_C10	SIGN: Increment I and Reset j=0	SIGN: Increment j	SIGN: w[i+j] += k[i] * a[j]	SIGN: B256_NNRED(w)	SPARE	SPARE	OUTPUT RESULT: SIG_S = B256_CtoS(Sc)	SPARE	Set Next_Step = 19 if i<21	Increment Next_Step if FALSE or (STEP==62 and i >= 21)	SET DN FLAG				
STEPS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	CMD_VALUE	i	j	COMMENTS
0																																	0			***NOT USED TO ALLOW FOR FIRST SCAN INPUT***
1	x																																1	?	?	Initialize Variables
2		x																															2			
3			x																														4			
4				x																													8			
5					x																												16			
6						x																											32			
7								x																									128			
8									x																								256			
9										x																							512			
10											x	x																					3072			
11													x																				4096			Controller stores its own public key A (point) as an array of LINTs
12														x																			8192			Other side stores it as a string value representing encoded point
13					x																												16			
14															x																		16384			
15																	x																65536			
16																		x															131072			
17																			x														262144			
18																					x												1048576	-1		
19																						x											2097152	0	0	i loop
20																								x									8388608	0	0	
21																							x										4194304	0	1	
22																								x									8388608	0	1	
23																							x										4194304	0	2	
24																								x									8388608	0	2	
25																							x										4194304	0	3	
26																								x									8388608	0	3	
27																							x										4194304	0	4	
28																								x									8388608	0	4	
29																							x										4194304	0	5	
30																								x									8388608	0	5	
31																							x										4194304	0	6	
32																								x									8388608	0	6	
33																							x										4194304	0	7	
34																								x									8388608	0	7	
35																							x										4194304	0	8	
36																								x									8388608	0	8	
37																							x										4194304	0	9	
38																								x									8388608	0	9	
39																							x										4194304	0	10	
40																								x									8388608	0	10	
41																							x										4194304	0	11	
42																								x									8388608	0	11	
43																							x										4194304	0	12	
44																								x									8388608	0	12	
45																							x										4194304	0	13	
46																								x									8388608	0	13	
47																							x										4194304	0	14	
48																								x									8388608	0	14	
49																							x										4194304	0	15	
50																								x									8388608	0	15	
51																							x										4194304	0	16	
52																								x									8388608	0	16	
53																							x										4194304	0	17	
54																								x									8388608	0	17	
55																							x										4194304	0	18	
56																								x									8388608	0	18	
57																							x										4194304	0	19	
58																								x									8388608	0	19	
59																							x										4194304	0	20	
60																								x									8388608	0	20	
61																							x										4194304	0	21	
62																								x						x	x		1619001344	0	21	
63																									x								16777216			
64															x																		16384			
65																		x															131072			
66																												x					134217728			
67																																	0			
68																																x	-2147483648			
69																																	0			
70																																	0			

D.2 Ed25519_VERIFY

SEQUENCE CONTROL COMMANDS	BUILD BASE POINT AND INCREMENT COUNTER	COMBINE RH AND DH INTO MSG1	ADD ENCRYPTED VMAC KEY, CONTROL FLAGS, KEYIDs, AND COUNTER INTO MSG1	h1 = SHA512(MSG,LEN=16)	TEMP_D1 = B512_HtoD (h1)	E TEMP_E1 (re) = D TEMP_D1 mod L	SPARE	P q = POINT_MUL(E r * B)	E SIG_R = POINT_ENC (P q)	OUTPUT RESULT: S SIG_R = B256_StoSLE(B256_CtoS(B256_EtoC	COMBINE POINT R MSG2	ADD PUB KEY A AND DH INTO MSG2	ADD ENCRYPTED VMAC KEY, CONTROL FLAGS, KEYIDs, AND COUNTER INTO MSG2	h2 = SHA512(MSG,LEN=20)	E TEMP_E2 (ke) = D TEMP_D1 mod L	SPARE	SIGN: C TEMP_C1 (rc) = B256_EtoC(TEMP_E1(re))	SIGN: C TEMP_C2 (kc) = B256_EtoC(TEMP_E2 (ke))	SIGN: COP w[0..21]=r[0..21], FLL w[22..43]=0	SPARE	SIGN: Initialize i=-1 and Inp_PriKey_scalar into Temp_C10	SIGN: Increment I and Reset j=0	SIGN: Increment j	SIGN: w[i+j] += k[i] * a[j]	SIGN: B256_NNRED(w)	SPARE	SPARE	OUTPUT RESULT: SIG_S = B256_CtoS(Sc)	SPARE	Set Next_Step = 19 if i<21	Increment Next_Step if FALSE or (STEP==62 and i >= 21)	SET DN FLAG				
STEPS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	CMD_VALUE	i	j	COMMENTS
0																																	0			***NOT USED TO ALLOW FOR FIRST SCAN INPUT***
1	x																																1	?	?	Initialize Variables
2		x																															2			
3			x																														4			
4				x																													8			
5					x																												16			
6						x																											32			
7								x																									128			
8									x																								256			
9										x																							512			
10											x	x																					3072			
11													x																				4096			Controller stores its own public key A (point) as an array of LINTs
12														x																			8192			Other side stores it as a string value representing encoded point
13					x																												16			
14															x																		16384			
15																	x																65536			
16																		x															131072			
17																			x														262144			
18																					x												1048576	-1		
19																						x											2097152	0	0	i loop
20																								x									8388608	0	0	
21																							x										4194304	0	1	
22																								x									8388608	0	1	
23																							x										4194304	0	2	
24																								x									8388608	0	2	
25																							x										4194304	0	3	
26																								x									8388608	0	3	
27																							x										4194304	0	4	
28																								x									8388608	0	4	
29																							x										4194304	0	5	
30																								x									8388608	0	5	
31																							x										4194304	0	6	
32																								x									8388608	0	6	
33																							x										4194304	0	7	
34																								x									8388608	0	7	
35																							x										4194304	0	8	
36																								x									8388608	0	8	
37																							x										4194304	0	9	
38																								x									8388608	0	9	
39																							x										4194304	0	10	
40																								x									8388608	0	10	
41																							x										4194304	0	11	
42																								x									8388608	0	11	
43																							x										4194304	0	12	
44																								x									8388608	0	12	
45																							x										4194304	0	13	
46																								x									8388608	0	13	
47																							x										4194304	0	14	
48																								x									8388608	0	14	
49																							x										4194304	0	15	
50																								x									8388608	0	15	
51																							x										4194304	0	16	
52																								x									8388608	0	16	
53																							x										4194304	0	17	
54																								x									8388608	0	17	
55																							x										4194304	0	18	
56																								x									8388608	0	18	
57																							x										4194304	0	19	
58																								x									8388608	0	19	
59																							x										4194304	0	20	
60																								x									8388608	0	20	
61																							x										4194304	0	21	
62																								x						x	x		1619001344	0	21	
63																									x								16777216			
64															x																		16384			
65																		x															131072			
66																												x					134217728			
67																																	0			
68																																x	-2147483648			
69																																	0			
70																																	0			

D.3 B256_MODINV

SEQUENCE CONTROL COMMANDS	Initialize u = 1, i=-1	Initialize t = v	Increment i and Reset j=-1	Extract the byte from exponent	Increment j to get next bit in exponent byte	Look at Exponent Bit, if true u=u*t else u=u*1	t=t*t	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	Set DN Flag	Set Next_Step = 3 if i<31	Increment Next_Step if FALSE or i>=31				
STEPS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	CMD_VALUE	i	j	COMMENTS
0																	0			***NOT USED TO ALLOW FOR FIRST SCAN INPUT***
1	x																1	-1	?	Initialize Variables
2		x															2			
3			x														4	0	0	
4				x													8	0	0	
5																	0	0	0	
6						x											32	0	0	
7							x										64	0	0	
8					x												16	0	1	
9						x											32	0	1	This subsection is representative of
10							x										64	0	1	what it takes to complete a single j-loop
11					x												16	0	2	
12						x											32	0	2	
13							x										64	0	2	
14					x												16	0	3	
15						x											32	0	3	
16							x										64	0	3	
17					x												16	0	4	
18						x											32	0	4	
19							x										64	0	4	
20					x												16	0	5	
21						x											32	0	5	
22							x										64	0	5	
23					x												16	0	6	
24						x											32	0	6	
25							x										64	0	6	
26					x												16	0	7	
27						x											32	0	7	
28							x								x	x	-16320	0	7	
29														x			8192			TRIGGER DONE FLAG FOR EXTERNAL STATUS REPORT
30																	0			DONE WITH MODINV, GO BACK TO STEP 0

D.4 B256_MODL

SEQUENCE CONTROL COMMANDS	Initialize i=43, p=0, q=0	decrement i and Reset j=-1, s=Inp_D[i], Inp_D[i]=0:	increment j	k= i+j-N+1	math operations to calculate w[k]	SPARE	adjust negative values	increment p	math operations to calculate w[q]	increment q	Copy lower half of Inp_D to c	CtoE	Set DN Flag	Set Next_Step = 87 If w[21] > 0, else Increment Next_Step	Set Next_Step = 2 if i>21	Increment Next_Step if FALSE or (i<=21 & current_step = 4						
STEPS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	CMD_VALUE	i	j	p	q	COMMENTS
0																	0					***NOT USED TO ALLOW FOR FIRST SCAN INPUT***
1	x																1	43	?	0	0	Initialize Variables
2		x															2	42	-1	0	0	
3			x														4	42	0	0	0	
4				x													8	42	0	0	0	
5					x												16	42	0	0	0	
6			x														4	42	1	0	0	
7				x													8	42	1	0	0	
8					x												16	42	1	0	0	
9			x														4	42	2	0	0	
10				x													8	42	2	0	0	
11					x												16	42	2	0	0	
12			x														4	42	3	0	0	
13				x													8	42	3	0	0	
14					x												16	42	3	0	0	
15			x														4	42	4	0	0	
16				x													8	42	4	0	0	
17					x												16	42	4	0	0	
18			x														4	42	5	0	0	
19				x													8	42	5	0	0	
20					x												16	42	5	0	0	
21			x														4	42	6	0	0	
22				x													8	42	6	0	0	
23					x												16	42	6	0	0	
24			x														4	42	7	0	0	
25				x													8	42	7	0	0	
26					x												16	42	7	0	0	
27			x														4	42	8	0	0	
28				x													8	42	8	0	0	
29					x												16	42	8	0	0	
30			x														4	42	9	0	0	
31				x													8	42	9	0	0	
32					x												16	42	9	0	0	
33			x														4	42	10	0	0	
34				x													8	42	10	0	0	
35					x												16	42	10	0	0	
36			x														4	42	11	0	0	
37				x													8	42	11	0	0	
38					x												16	42	11	0	0	
39			x														4	42	12	0	0	
40				x													8	42	12	0	0	
41					x										x	x	-16368	42	12	0	0	
42							x	x									192	21	12	0	0	
43							x	x									192			1	0	
44							x	x									192			2	0	
45							x	x									192			3	0	
46							x	x									192			4	0	
47							x	x									192			5	0	
48							x	x									192			6	0	
49							x	x									192			7	0	
50							x	x									192			8	0	
51							x	x									192			9	0	
52							x	x									192			10	0	
53							x	x									192			11	0	
54							x	x									192			12	0	
55							x	x									192			13	0	
56							x	x									192			14	0	
57							x	x									192			15	0	
58							x	x									192			16	0	
59							x	x									192			17	0	
60							x	x									192			18	0	
61							x	x									192			19	0	
62							x	x									192			20	0	
63							x	x									192			21	0	
64														x		x	-24576			22	0	
65									x	x							768				0	this section only done to adjust negative values
66									x	x							768				1	
67									x	x							768				2	
68									x	x							768				3	
69									x	x							768				4	
70									x	x							768				5	
71									x	x							768				6	
72									x	x							768				7	
73									x	x							768				8	
74									x	x							768				9	
75									x	x							768				10	
76									x	x							768				11	
77									x	x							768				12	
78									x	x							768				13	
79									x	x							768				14	
80									x	x							768				15	
81									x	x							768				16	
82									x	x							768				17	
83									x	x							768				18	
84									x	x							768				19	
85									x	x							768				20	
86									x	x							768				21	
87											x						1024				22	
88												x					2048					
89													x				4096					TRIGGER DONE FLAG FOR EXTERNAL STATUS REPORT
90																	0					DONE WITH MOD_L, GO BACK TO STEP 0

D.5 B256_MODPOW

SEQUENCE CONTROL COMMANDS	Initialize u = 1, i=-1	Initialze t = v	Increment i and Reset j=-1	Extract the byte from exponent	Increment j to get next bit in exponent byte	Look at Exponent Bit, if true u=u*t else u=u*1	t=t*t	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	Set DN Flag	Set Next_Step = 3 if i<31	Increment Next_Step if FALSE or i>=31				
STEPS	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**CMD_VALUE**	**i**	**j**	**COMMENTS**
*****0*****																	0			***NOT USED TO ALLOW FOR FIRST SCAN INPUT***
1	x																1	-1	?	Initialize Variables
2		x															2			
3			x														4	0	0	
4				x													8	0	0	
5																	0	0	0	
6						x											32	0	0	
7							x										64	0	0	
8					x												16	0	1	
9						x											32	0	1	This subsection is representative of
10							x										64	0	1	what it takes to complete a single j-loop
11					x												16	0	2	
12						x											32	0	2	
13							x										64	0	2	
14					x												16	0	3	
15						x											32	0	3	
16							x										64	0	3	
17					x												16	0	4	
18						x											32	0	4	
19							x										64	0	4	
20					x												16	0	5	
21						x											32	0	5	
22							x										64	0	5	
23					x												16	0	6	
24						x											32	0	6	
25							x										64	0	6	
26					x												16	0	7	
27						x											32	0	7	
28							x								x	x	-16320	0	7	
29														x			8192	31	7	TRIGGER DONE FLAG FOR EXTERNAL STATUS REPORT
30																	0			DONE WITH MODPOW, GO BACK TO STEP 0

D.6 POINT_DEC

SEQUENCE CONTROL COMMANDS	Clear the point and convert the String to a C value	Create a C value representing d of Ed25519	Create a C value representing m1=m-1 and an E value representing m58=(m-5)/8	Create a C value representing sqrtm1 = 2^((m-1)/4)	Find least significant bit of x-coordinate and Clear the bit to make the y-coordinate	Calculate v = d*y^2+1, v is not reduced	Calculate u = y^2 - 1, Note if y^2 == 0, u = m-1	Note if y^2 == 0, u = m-1	Calculate t1 = u*v^3	Calculate t2=u*v^7	Calculate t3=(u*v^7)^((m-5)/8)	Calculate x=(u/v)^((p+3)/8)	Calculate t2=x^2, t1=v*x^2, t2=-u	Determine if v x^2 is equal to u or -u If equal to -u then P.x = P.x * 2^((p-1)/4) = P.x * sqrt of -1	Determine if decoding failed by comparing to x_0, if P.x = 0 and x_0 = 1 decoding fails	If x_0 != x mod 2 set x = p - x	Determine if y is >= p, if it is then decoding fails	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	EXTENDED COORDINATES: Set Z = 1, X=x, Y=y, T=x*y	SPARE	SPARE	SPARE	SPARE	SET DN FLAG	Increment Next_Step if FALSE		
STEPS	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**CMD_VALUE**	**COMMENTS**
*****0*****																																	0	***NOT USED TO ALLOW FOR FIRST SCAN INPUT***
1	x																																1	Initialize Variables
2		x																															2	
3			x																														4	
4				x																													8	
5					x																												16	
6						x																											32	
7							x																										64	
8								x																									128	
9									x																								256	
10										x																							512	
11											x																						1024	
12												x																					2048	
13													x																				4096	
14														x																			8192	
15															x																		16384	
16																x																	32768	
17																	x																65536	
18																										x							33554432	
19																															x		1073741824	
20																																	0	

D.7 POINT_ENC

SEQUENCE CONTROL COMMANDS	Initialize u = 1, i=-1	Initialize t = v	modInv(z,p->z): Increment i and Reset j=-1	modInv(z,p->z): Extract the byte from exponent	modInv(z,p->z): Increment j to get next bit in exponent byt	modInv(z,p->z): Look at Exponent Bit, if true u=u*t else u=	modInv(z,p->z): t=t*t	mul (x, p->x, z) //z= Temp_C4	adjust(x) // x=Temp_C2	mul (y, p->y, z) //z= Temp_C4	adjust(y) // y=Temp_C3	CtoE(e,y) //y=Temp_C3	e[M-1] |= (x[0] & 1) << 7; //lsb of x goes to msb of e	Set DN Flag	Set Next_Step = 3 if i<31	Increment Next_Step if FALSE or i>=31				
STEPS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	CMD_VALUE	i	j	COMMENTS
0																	0			***NOT USED TO ALLOW FOR FIRST SCAN INPUT***
1	x																1	-1	?	Initialize Variables
2		x															2			
3			x														4	0	0	
4				x													8	0	0	
5																	0	0	0	
6						x											32	0	0	
7							x										64	0	0	
8					x												16	0	1	
9						x											32	0	1	This subsection is representative of
10							x										64	0	1	what it takes to complete a single j-loop
11					x												16	0	2	
12						x											32	0	2	
13							x										64	0	2	
14					x												16	0	3	
15						x											32	0	3	
16							x										64	0	3	
17					x												16	0	4	
18						x											32	0	4	
19							x										64	0	4	
20					x												16	0	5	
21						x											32	0	5	
22							x										64	0	5	
23					x												16	0	6	
24						x											32	0	6	
25							x										64	0	6	
26					x												16	0	7	
27						x											32	0	7	
28							x								x	x	-16320	0	7	
29								x									128	31	7	
30									x								256			
31										x							512			
32											x						1024			
33												x					2048			
34													x				4096			
35														x			8192			TRIGGER DONE FLAG FOR EXTERNAL STATUS REPORT
36																	0			DONE WITH POINT_ENC, GO BACK TO STEP 0

D.8 POINT_MUL

SEQUENCE CONTROL COMMANDS	INITIALIZE TEMP VARIABLES	COPY INPUT AND INITIALIZE TEMP POINT	INITIALIZE Q_p3 TO THE NEUTRAL ELEMENT POINT	EXTRACT THE BYTE FROM EXPONENT	INCREMENT i AND RESET j	INCREMENT j TO GET NEXT BIT IN EXPONENT BYTE	j Loop POINT_ADD (T_Pd, Q_p3, I_p1)	Look at Exponent Bit, Q_p3=d exle Q_p3=Q_p3	j Loop Point_Dbl (Q_p3, Q_p3)	ADJUST (Q_p3.X), ADJUST (Q_p3.Y)	SPARE	SPARE	SPARE	SET DN FLAG	Set Next_Step = 34 if i<31	Increment Next_Step if FALSE or i>=31				
STEPS	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	CMD_VALUE	i	j	COMMENTS
0																	0			***NOT USED TO ALLOW FOR FIRST SCAN INPUT***
1	x	x	x														7	0	0	Initialize Variables, Step 1 unique
2				x													8	0	0	Starting J loops
3							x										64			
4								x									128			
5									x								256			
6						x											32	0	1	
7							x										64			
8								x									128			
9									x								256			
10						x											32	0	2	
11							x										64			
12								x									128			
13									x								256			
14						x											32	0	3	
15							x										64			
16								x									128			
17									x								256			
18						x											32	0	4	
19							x										64			
20								x									128			
21									x								256			
22						x											32	0	5	
23							x										64			
24								x									128			
25									x								256			
26						x											32	0	6	
27							x										64			
28								x									128			
29									x								256			
30						x											32	0	7	First complete j-Loop Done
31							x										64			Loop back to i and begin next j Loop
32								x									128			
33									x								256			
34					x												16	1	0	This block is a representative sample
35				x													8			of what it takes to complete a single i-loop
36							x										64			Ultimately this results in lots of steps but
37								x									128			the objective is to complete the sequence
38									x								256			with minimum impact on overall scan time
39						x											32	1	1	
40							x										64			
41								x									128			
42									x								256			
43						x											32	1	2	This subsection is representative of
44							x										64			what it takes to complete a single j-loop
45								x									128			
46									x								256			
47						x											32	1	3	
48							x										64			
49								x									128			
50									x								256			
51						x											32	1	4	
52							x										64			
53								x									128			
54									x								256			
55						x											32	1	5	
56							x										64			
57								x									128			
58									x								256			
59						x											32	1	6	
60							x										64			
61								x									128			
62									x								256			
63						x											32	1	7	
64							x										64			
65								x									128			
66									x						x	x	-16128			
67										x							512	31	7	
68														x			8192	31	7	TRIGGER DONE FLAG FOR EXTERNAL STATUS REPORT
69																	0	31	7	DONE WITH POINT MULTIPLY, GO BACK TO STEP 0

D.9 PRNG

SEQUENCE CONTROL COMMANDS	GRAB CLOCK TIME FOR ADDITIONAL INPUT AND ADD INTO MSG	INCREMENT COUNTER AND ADD INTO MSG	ADD CONSTANT INTO MSG	ADD SEED INTO MSG	COMPUTE THE HASH	TAKE LEFT HAND PORTION OF THE HASH AS THE PRN	ROTR EACH PORTION OF THE SEED< ADDING IN UNUSED PORTIONS OF THE HASH	ROTR EACH PORTION OF THE SEED, ADDING IN PORTIONS OF THE SEED	ROTR EACH PORTION OF THE SEED, ADDING IN PORTIONS OF THE SEED	ADD FIRST PORTION OF SEED TO LAST, ROTATE, AND THEN SWAP	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	SPARE	INCREMENT NEXT STEP if FALSE	SET DN FLAG		
STEPS	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**CMD_VALUE**	**COMMENTS**
0																																	0	***NOT USED TO ALLOW FOR FIRST SCAN INPUT***
1	x																																1	
2		x																															2	
3			x																														4	
4				x																													8	
5					x																												16	
6						x																											32	
7							x																										64	
8								x																									128	
9									x																								256	
10										x																							512	
11																																x	-2147483648	
12																																	0	

D.10 SHA512_SEQ

SEQUENCE CONTROL COMMANDS	PREPARE DATA FOR HASH	PAD MSG	APPEND LENGTH	INITIALIZE HASH BUFFER	SPARE	SPARE	SCHEDULE MESSAGE: SAVE HASH BUFFER INPUT	SCHEDULE MESSAGE: SET W[0] THRU W[15]	INITIALIZE TEMP BUFFER	SCHEDULE MESSAGE: INITIALIZE i=16	SCHEDULE MESSAGE: CALCULATE SIGMA0	SCHEDULE MESSAGE: CALCULATE SIGMA1	SCHEDULE MESSAGE: CALCULATE W	SCHEDULE MESSAGE: INCREMENT i	SET Next_Step = 10 if i<80	PROCESS MESSAGE: INITIALIZE t=0	PROCESS MESSAGE: CALCULATE COND FUNC	PROCESS MESSAGE: CALCULATE MAJ FUNC	PROCESS MESSAGE: CALCULATE SIGMA A	PROCESS MESSAGE: CALCULATE SIGMA E	PROCESS MESSAGE: CALCULATE T1 AND T2	PROCESS MESSAGE: UPDATE HASH BUFFER	PROCESS MESSAGE: INCREMENT t	SET Next_Step = 16 if t<80		COMPUTE INTERMEDIATE HASH VALUE	INCREMENT M BY 16	SET Next_Step = 7 if M=<TOTAL_MSG_LENGTH_MINUS_16	SEND OUTPUT		Increment Next_Step if FALSE or (i>=80 & current_step = 14) or (t>=80 & current_step=23) or (m>TOTAL_MSG_LENGTH_MINUS_ & current step=26)	Set DN Flag	
STEPS	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**CMD_VALUE**
0																																	0
1	x																																1
2		x																															2
3			x																														4
4				x																													8
5					x																												16
6						x																											32
7							x																										64
8								x																									128
9									x	x																							768
10											x																						1024
11												x																					2048
12													x																				4096
13														x																			8192
14															x																x		1073758208
15									x							x																	33024
16																	x																65536
17																		x															131072
18																			x														262144
19																				x													524288
20																					x												1048576
21																						x											2097152
22																							x										4194304
23																								x							x		1082130432
24																										x							33554432
25																											x						67108864
26																												x			x		1207959552
27																													x				268435456
28																																x	-2147483648
29																																	0

APPENDIX E. KEY SCRIPTS

E.1 keygen.py

```
######################################################################
# Name:                   keygen.py
# Desc:      generates public and private keys for use in ControlLogix
# Usage:     python keygen.py < keygen.tests.input
# Input:     A series of standard test messages to ensure keys properly function
# Output:   Private_Key_a, Private_Key_UH, Public_Key, all as hex strings
#
# NOTE:    Uses the secrets module and the ed25519
# Last Modified:         5/20/2017 4:26PM
# Last Author:                              Ken Fischer, NSWCPD, C516
######################################################################

import sys
import secrets
import hashlib
import binascii
import ed25519

b = 256

def H(m):
return hashlib.sha512(m).digest()

random_number = secrets.token_hex(32)
sk = binascii.unhexlify(random_number)

h = H(sk)
Private_Key_scalar = 2**(b-2) + sum(2**i * ed25519.bit(h,i) for i in range(3,b-2))
h_string = binascii.hexlify(h)
Private_Key_prefix = h_string[64:]
A = ed25519.encodepoint(ed25519.scalarmult(ed25519.B,Private_Key_scalar))
Public_Key = binascii.hexlify(A)

#OUTPUT THE KEY VALUES
print ""
print "Private_Key_scalar: "
print format(Private_Key_scalar, 'x')
print ""
print "Private_Key_prefix: "
print Private_Key_prefix
print ""
print "Public_Key: "
print Public_Key

#HERE WE TEST THE KEYS TO ENSURE THEY WORK ON VARIOUS MESSAGES
print ""
print "Testing Keys... this can take up to a minute per test case..."

pk = binascii.unhexlify(Public_Key)
i = 0
while 1:
  line = sys.stdin.readline()
  if not line: break
  x = line.split(':')
 m = binascii.unhexlify(x[0])
  s = ed25519.signature(m,sk,pk)
  ed25519.checkvalid(s,m,pk)
  i += 1
  print "      Test {} complete.".format(i)
 sys.stdout.write("\033[F")

print ""
print "Testing Complete.  If no errors then use keys."
```