Control System Data Authentication and Verification Using Elliptic Curve Digital Signature Algorithm

By

Kenneth Alan Fischer

Control System Data Authentication and Verification Using Elliptic Curve Digital Signature Algorithm

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

MASTER OF SCIENCE

In

Computer Engineering

July, 2014

Villanova, Pennsylvania

This page intentionally left blank.

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 M.Sc studies at Villanova University. Firstly, I would like to thank my advisor, Dr. Richard Perry, for sharing his time, experience, and wisdom during my research. I also thank my supervisor, Michael Iacovelli (NSWCCD-SSES C955 Branch Manager) for his support in enabling me to pursue this work, as well as all of my colleagues within NSWCCD-SSES who took the time to listen to my ideas and share with me their own insight and practical experience in the field of Control System Engineering. Lastly and most importantly, I would like to thank my wife, Ana Fischer, for all the extra work she did in taking care of our three children while I pursued this work.

DEDICATION

I dedicate this dissertation to my daughters Liviya and Dalya, and to my son Isaac,

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.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

ABSTRACT

Recent endeavors such as the Smart Grid and attacks on control systems such as Stuxnet, have highlighted the need for improved security in control systems and control system communications. Control system components such as Programmable Logic Controllers (PLCs) and Human-Machine Interfaces (HMIs) can no longer rely on simple heartbeat logic algorithms in order to verify communications. Advanced cryptographic algorithms for data authentication and verification are needed in messaging protocols between PLCs, HMIs, and sensors.

Cryptographic algorithms such as RSA or the Digital Signature Algorithm (DSA) appear to provide a solution, however the key sizes required for implementing these solutions are not feasible for implementation in a control system. Elliptic Curve DSA (ECDSA) looks to be a promising solution due to the smaller key sizes and faster computations.

This work implements a prototype implementation of ECDSA that was developed utilizing ISaGRAF™ SoftPLCs. The implementation is primarily written in IEC 61131-3 ladder logic using specialized function blocks developed in C, utilizing the OpenSSL library. Four SoftPLCs were configured in a round robin architecture on two different test strings in order to determine the average time it takes to generate a message, sign it, transmit it, receive it, and verify the signature. A self-signing key distribution architecture was developed which relies on the commissioning process of PLCs to establish the initial trust relationship. Test string 1 gave an average scan time of 26-28ms ms and test string 2 gave an average time of 55-60ms with PLC scan times of 55-60 ms. These timing values confirm the potential for ECDSA to be used for control system data authentication and verification. C and ladder logic source code is included.

CHAPTER 1: INTRODUCTION

1.1 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 troubleshot 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.

The need for PLCs will significantly expand in the coming years, as countries with mature economies work tirelessly to develop new sophisticated power distribution networks required to support our growing economy. Our existing power grids were designed decades ago, with the main aim of delivering electricity from large power stations to households and businesses. The increasing efficiency and reliable requirements necessary to support our developing civilization in the face of increasing energy demands and the real threat of domestic terrorism and foreign aggression require significant modernization of these power distribution networks. 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.

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. To address these requirements, modern ship designs such as the USS ZUMWALT DESTROYER (DDG1000) class are using Integrated Power Systems (IPS) that provide electrical power to propulsion and electrical loads from a common set of sources. To provide direction for future IPS development, the Navy initiated the Next Generation Integrated Power Systems (NGIPS) effort to provide smaller, simpler, more affordable, and more capable systems for all Navy ships [8].

The NGIPS effort is remarkably similar to the Smart Grid effort in multiple respects, and in both there is an increasing consensus that the control communication infrastructure needs fundamental changes. In an automated electrical system, damage to a complex communication network, a hostile terrorist act, or even a failing component giving erroneous data can result in a control system taking improper actions that could result in large scale power failures on land and weapons, propulsion, or a complete electrical failure at sea or worse. Earlier this year, we at NSWCCD-SSES documented a case where erroneous data from a failing control system communications component in an Improved Navy Lighterage System (INLS) Warping Tug (WT) resulted in a complete loss of propulsion and steering control whenever a ship was placed into full speed, which would have resulted in the ship colliding into the shore if it were not for conveniently placed Emergency Stop pushbuttons. It has become clear to controls engineers that more sophisticated methods are needed for verifying the integrity of the data and commands being issued to and from control systems.

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 last year particularly has highlighted the deficiencies with this model, as viruses such as Stuxnet have become rapidly prevalent. There is also significantly more risk in a compromised control system than a compromised corporate system. For example, an attacker could compromise the control system of a nuclear power plant resulting in a failure of the reactor cooling system. Therefore control system designers are realizing that not only do we need improved algorithms to verify that control system data is accurate, we need algorithms to verify that the data and commands to the control systems are authenticated (i.e. coming from a valid, recognized source).

1.2 Current Practices

Controls engineers have long recognized the need to verify that components within a control system are communicating and that the failure of communications between control system components should result in critical high priority alarms with possible equipment shutdowns. Since control system communications operate in real time, 24 hours a day, 7 days a week, algorithms are needed to detect a failure in communications as soon as it occurs. Traditionally, "heartbeat" logic is implemented between each pair of communication devices. Algorithm 1.1.A below illustrates an example of commonly used "heartbeat" logic.

Algorithm 1.1.A – Traditional Control System "Heartbeat"

1. Initialize a bit to a known condition (typically 1 as will be used in this algorithm).
2. Transmit bit (call it B1) to communication partner. Start a 3 second timer (call it T1)
3. Communication partner receives the bit B1. Communication partner sets another bit (call it B2) to 1 to match the state of B1 and starts its own 3 second timer (call it T2).
4. Receive bit B2 from the partner. Verify that the state of B2 matches the state of B1 and that timer T1 has not timed out. If true, restart timer T1. Change state of B1 to be opposite that of B2. Transmit B1 back to partner.

5. Partner receives bit B1. Partner verifies that the state of B1 does not match the state of B2 and that timer T2 has not timed out. If true, partner restarts timer T2. Partner changes B2 to match state of B1, retransmits bit back, and go to step 4.
6. If T1 or T2 times out, generate alarm for communications failure.

As long as a communications failure alarm does not occur, then the data being transmitted between the two PLCs is considered to be both valid and sourced between the communicating pair. This kind of logic has proven to be very effective for general network health monitoring. Issues in communication, primarily in the physical or transport layer, can be easily detected using this method. For control system networks that are physically isolated from any other network, this is generally sufficient to implement an effective control scheme. Unfortunately, this method does not protect against any kind of more sophisticated failure or attack such as that documented for the INLS WT described earlier or a “man-in-the-middle” attack.

1.3 Literature Review on Smart Grid

A number of papers have been written to introduce the Smart Grid concepts and provide a general overview of the requirements and challenges involved in developing a Smart Grid.

Bouhafs, Mackay, and Merabti (2012) [1] identified a number of general requirements including communications and electrical generation needed in order to fully realize the Smart Grid vision. They noted that underlying communications protocols will need to be more flexible and enable horizontal (vice a master/slave top-down) data exchange between controllers and remote terminal units (RTUs). The current "heartbeat" logic concept would not be useful in an implementation where data could flow from a source through multiple sources to a target since it only verifies the link between pairs and not the data itself. They went on to note that in the event the Internet is used to connect equipment in the Smart Grid, strong encryption and authentication measures must be taken to ensure the security of the data in transit.

Yan, Qian, Sharif, and Tipper (2012) [2] noted that it is necessary to have guaranteed Quality of Service (QoS) for the communications and networking technology. In particular they highlighted the latency, bandwidth, interoperability, scalability, and security requirements. Of particular interest is the authors' analysis of bandwidth requirements which showed that there will be significant challenges in this area. Therefore, adding a significant number of bits in any communications protocol for control systems could have a profoundly negative impact on the operation of the Smart Grid as a whole. The authors also noted that the effort required to provision symmetric keys (i.e. keys between each pair of communicating devices) into thousands of devices would be too expensive or insecure. They noted that the development of key and trust management schemes for large network deployments would be required. While Navy systems are small enough that they would not suffer from the same kinds of limitations, it seems obvious that a solution must be developed for Navy systems that would be applicable to all future controls systems including the Smart Grid, particularly in support of modernized shore power connections for Navy systems.

Yan, Qian, Sharif, and Tipper (2012) [3] in a related paper noted that new functions in the Smart Grid such as demand response introduce significant new cyber attack vectors such as a malware that initiates a massive coordinated and instantaneous drop in demand. This attack could result in substantial damage to distribution, transmission, and generation facilities. Research ongoing at NSWCCD-SSES has also noted this risk as applicable to Navy systems, particularly in combat scenarios with the use of advanced weapon systems such as the railgun. The authors also noted that a major difference between Smart Grid controls communication and the Internet is that the controls data is significantly more concerned with message delay and timing constraints.

Liu, Ning, and Reiter (2009) [4] in their work presented a notable example of a new type of attack, called false data injection attacks, that highlights the very real risk of attacks targeting data integrity.

Baumeister (2011) [5] noted that most information systems use a Public Key Infrastruction (PKI) solution, but that the nature of power grid systems creates additional PKI requirements not present in traditional information systems. This same statement can be generalized to apply to all control systems. For example, Baumeister noted that control systems must make informed decisions regularly, and that it is unreasonable to expect a control system to go down or revert to a less efficient predecessor every time a certificate is unavailable. For example, what happens when a certificate from a sensor expires? In an information system, the impact of expired certificates is insignificant and they can be renewed when discovered. However, in a control system this could cause the process (such as electric flows) to be incorrectly altered.

1.3.1 SGiP Cyber Security Working Group NISTIR 7628

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) [6]. This document broke down the various kinds of communications that would be prevelant 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 NGIPS 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 Literature Review on NGIPS

Most of the literature on the NGIPS effort has focused on areas such as electrical generation, propulsion, power conversion and distribution, energy storage, and zonal survivability. The NGIPS architecture is broken up into seven modules types:

- Power Generation Modules (PGM)
- Power Distribution Modules (PDM)

- Power Conversion Modules (PCM)
- Energy Storage Modules (ESM)
- Power Loads
- Propulsion Motor Modules (PMM)
- Power Control Modules (PCON)

The PCON module is of particular interest to controls engineers, as it consists of the software and communications protocols necessary to operate the system. Doerry (2009) [8] noted that PCON should implement the following functions listed below. He also noted that the software should be developed for robustness in anticipation of future changes in the life of a ship, and for modifying for use across multiple ship classes.

- Remote monitoring and control of NGIPS modules and controllable loads
- Resource Planning
- System Configuration
- Mission Priority Load Shedding
- Quality of Service Load Shedding
- Fault Detection and Isolation
- Maintenance Support
- Training

These functions are remarkably similar to the control system functions required for the development of a Smart Grid, with the notable exceptions of the electrical distribution Quality of Service (QoS) and Mission Priority Load Shedding. As a result, the same need for data authentication and verification in the Smart Grid would be applicable to NGIPS, particularly in functions such as maintenance support where it becomes increasingly common for ships to transmit data to and from shore based services for software upgrades and maintenance / troubleshooting support.

Desired requirements for QoS also introduce the need to ensure that commands being transmitted across the ship for electrical service are genuine. As noted by Doerry, a typical cause of a QoS failure is the shifting of electrical power sources from ship to shore, and that communications will be required with the terrestrial power system command and control centers. Failure of the ship and shore to properly establish valid communications could result in power instabilities for both.

The increasing prevalence of computer viruses specifically targeting control systems will introduce new challenges to the mission readiness of a ship in times of war. By attacking PCON, an enemy may be able to cause a control system to incorrectly transfer loads which could result in a failure of propulsion or weapon systems (or both) at a critical moment. Modern weapon systems produce substantial electrical loads that may require realigning of the ship's electrical distribution prior to being operational (such as performing bus-tie operations and shedding non-critical loads).

The Navy has been putting significant effort into open architecture approaches in the development of control system software for fleet wide applications, encompassing a much larger scope then NGIPS. Doerry, Scherer, Cohen, and Guertin (2011) [9] pointed out that information assurance and security needs to be thought of at the outset of any new machinery control system design, stating that confidentiality, integrity, and availability of data must be assured. They also highlight that the software should perform error detection (and error correction if possible) along with filtering of the sensor data.

CHAPTER 2: CURRENT PRACTICES

2.1 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
- Data Integrity – ensuring that the data has not been modified by unauthorized means
- 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 an 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.

2.2 Limitations of Control Systems compared to Information / Corporate Systems

Information / Corporate Systems typically consist of x86-based architecture computers running either Windows or Linux operating systems and a host of other software programs provided by multiple vendors to provide an integrated solution. In contrast, at the heart of the Control System are Programmable Logic Controllers (PLCs), which use vendor specific developer environments to write software following IEC 61131-3 guidelines (ladder logic, function blocks, etc) to implement a solution that is both easy and cheap to design and is very effective for controls. The downside of these PLCs is that they tend to have significantly less processing power and storage capabilities as they are designed to run very specific software programs extremely efficiently, non-stop, for 20 years or more.

An alternative to PLCs are VERSAmodule Eurocards (VME) which tend to have greater processing power and contain the same input / output processing capabilities as PLCs but add significant complexity to the design of a control system. The pros and cons of PLCs and VMEs are described below. Another alternative to PLCs are SoftPLCs. SoftPLCs are essentially programmed in the same manner as regular PLCs, but contain additional underlying base code designed to interface with an operating system (typically Windows NT based operating systems) in order to run the IEC 61131-3 code on an x86-based architecture. Figures 2.2A and 2.2.B show running PLC and VME racks on control systems for Navy Ships

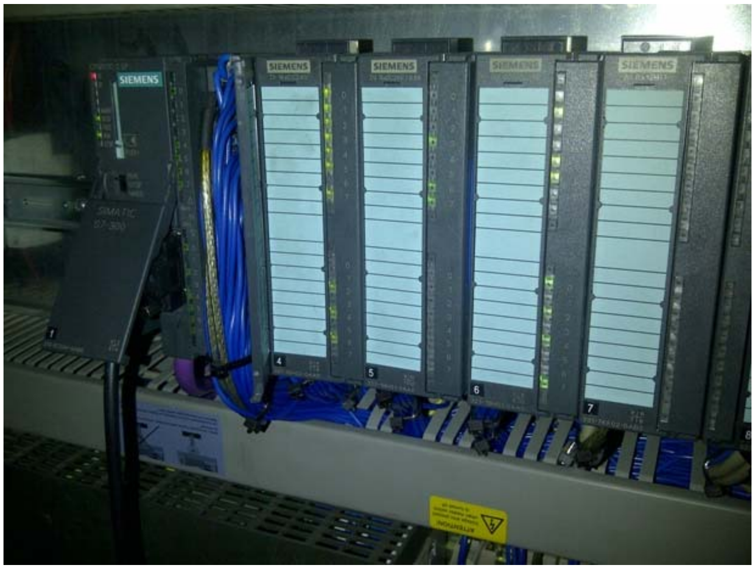


Figure 2.2.A PLC Rack

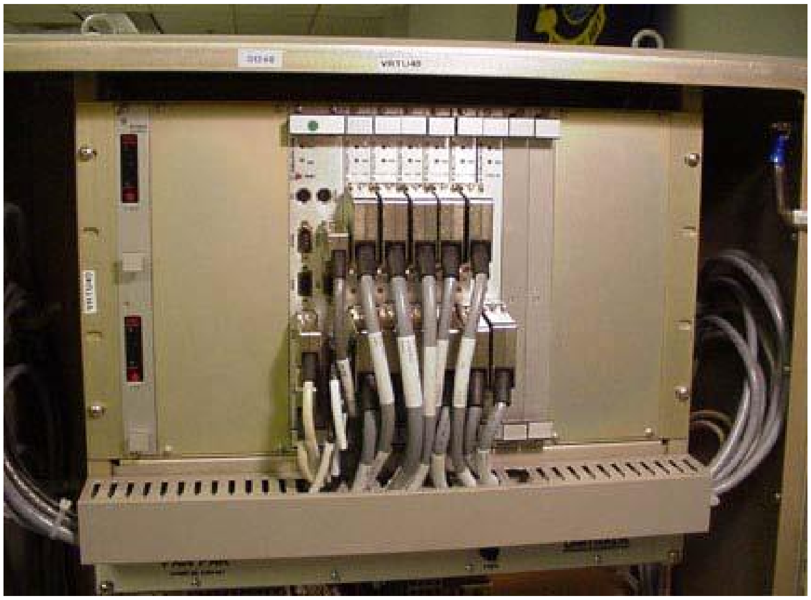


Figure 2.2.B VME Rack

Since VME cards can be obtained that use the x86 architecture, in recent years the Navy has been implementing control systems on ship classes that use SoftPLCs running on VMEs to obtain the best of both worlds. This can be a complicated and expensive solution that is still more in the research and development stage and will likely not be implemented in either the Smart Grid or regular industrial control systems. However it is possible from a research perspective to perform cryptographic testing on SoftPLCs using VMEs to do "proof of concept" testing in order to determine the validity of a solution before expending significant resources in developing an independent and complete PLC solution.

2.2.1 PLC versus VME

In order to give greater perspective on the usage of PLCs versus VMEs in control systems, the pros and cons of both technologies are listed below. These SoftPLCs may become more prevalent in industrial control systems with the advent of new projects such as OpenPLC which aims to develop an open source software and hardware platform for industrial control systems.

VME Pros

- Analog and digital I/O boards are available from a large number of vendors
- VME components are open architecture
- Standardized circuit card form factor and data bus

- Significantly greater flexibility in software for VME than compared to PLC, allowing for advanced processing not available with PLCs (such as required for the Navy's new advanced guided missile destroy program)
- Ability to implement secure communication protocols
- Operating Temperature range of -40^{o}C to +85^{o}C

VME Cons

- Development of software is complex and difficult, developer must design not only the control system application but also the low-level system interactions
- Widespread use of proprietary operating systems often creates a virtual sole-source situation
- Instability in VME Operating System market means it is unlikely developers will have experience with the operating system chosen for a new project, leading to longer ramp-up time and increased risk for software defects
- Obsolescence is a major problem
- Integration of new components into an existing system is NOT "plug-and-play"

PLC Pros

- Cost is less than for VME systems
- Programming time is reduced due to ease of programming language (ladder-logic)
- Risk is significantly reduced when using all products from the same vendor
- Integration of new components into an existing system is typically "plug-and-play"

- PLC vendors have a strong record of supporting their products for 20 years or longer
- Enhanced software troubleshooting features not available with VMEs

PLC Cons

- PLC vendor products generally not compatible with another vendor's products, requiring a single vendor to provide all processor, I/O, and network communication boards
- No standards for PLC form factor or electrical characteristics
- Secure communication protocols are not a common feature with many PLC vendors
- Increased risk in relying on a single vendor to support their products
- Operating Temperature range of only 0^{o}C to $+60^{o}$C

2.3 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.

2.3.1 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 [10]. 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 (for NGIPS) it will not be practical for large scale systems such as the Smart Grid.

2.3.2 Public-key Cryptography

Public-key cryptography began in 1975 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.

2.3.2.1 RSA

The most commonly used public-key cryptography scheme is RSA, named after its inventors Rivest, Shamir, and Adleman [11]. 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 2.3.2.1.A below shows how to generate an RSA key pair. RSA encryption and signature schemes use the fact that $m^{ed} = m \ (mod\ n)$. Algorithms 2.3.2.1.B and 2.3.2.1.C 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 2.3.2.1.D and 2.3.2.1.E. As in all signature schemes, the signer first generates a cryptographic hash function *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 of *e* are $e = 3$ and $e = 65537$ for encryption and signature verification [11]. 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 2.3.2.1.A [10] – 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 2.3.2.1.B [10] – 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 2.3.2.1.C [10] – 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 2.3.2.1.D [10] – 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 2.3.2.1.E [10] – 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 $h' = s^e \; mod \; n$
3. If $h = h'$ then accept the signature, else reject

2.3.2.2 Digital Signature Algorithm

In 1976 Diffie and Hellman proposed developing a key agreement protocol based on the discrete logarithm problem (DLP) [10], 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 2.3.2.2.A shows how these domain parameters are generated, and Algorithm 2.3.2.2.B 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) [10]. 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 [12]. Algorithms 2.3.2.2.C and 2.3.2.2.D shown below give the procedures respectively for DSA signature generation and verification.

Algorithm 2.3.2.2.A [10] – 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 $h \in [1, p-1]$ and compute $g = h^{(p-1)/q} \bmod p$
           b. If $g = 1$ then repeat 2.a.
    3. Return $(p, q, g)$
```

Algorithm 2.3.2.2.B [10] – Discrete Logarithm Key Pair Generation

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

Algorithm 2.3.2.2.C [10] – 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 2.3.2.2.D [10] – 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

2.3.2.3 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 [13] 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 1024-bit keys are likely to become insecure sometime between 2006 and 2010 and 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. 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.

CHAPTER 3: ELLIPTIC CURVE CRYPTOGRAPHY

3.1 Background

Elliptic curve public key cryptosystems were first independently proposed by V.S. Miller (1985) [14] and by N. Koblitz (1987) [15]. 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) [16, 17], IEEE (Institute of Electrical and Electronics Engineers) [18], ISO (International Standards Organization [19, 20], and NIST (National Institute of Standards and Technology [21].

The most prominent group for the standardization and propagation of ECC technology is SECG (Standards for Efficient Cryptography Group) [22]. They have published numerous and detailed works on the subject, including documents on how to implement ECC and on recommended elliptic curve domain parameters [23, 24]. 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 [25].

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 [6]. 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." [26]

3.2 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 [15], Miller [14], Hankerson et al [10], and the SECG [23] 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.

3.2.1 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 [10]. 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.

3.2.1.1 Prime Finite Fields [23]

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

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

$$[log_2p] \epsilon \{192, 224, 256, 384, 521]$$

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.

3.2.1.2 Binary Finite Fields [23]

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:

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

and an irreducible polynomial $f(x)$.

Operations on binary finite fields are defined as follows:

- Addition: If $a = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \ldots + a_0$, $b = b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \ldots + b_0 \epsilon F_{2^m}$, then $a + b = r$ in 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 $a = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \ldots + a_0$, $b = b_{m-1}x^{m-1} + b_{m-2}x^{m-2} + \ldots + b_0 \epsilon F_{2^m}$, then $ab = s$ in 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 $a \epsilon F_{2^m}$, then the additive inverse ($-a$) of a in F_{2^m} is the unique solution to the equation $a + x \equiv 0$ in F_{2^m}.

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

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

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

These fields were chosen in order to construct a suitable Koblitz curve whose order is 2 or 4 times a prime over 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 Figure 3.2.1.2.A 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)
$F_{2^{163}}$	$f(x) = x^{163}+x^7+x^6+x^3+1$
$F_{2^{233}}$	$f(x) = x^{233}+x^{74}+1$
$F_{2^{239}}$	$f(x) = x^{239}+x^{36}+1$ or $x^{239}+x^{158}+1$
$F_{2^{283}}$	$f(x) = x^{283}+x^{12}+x^7+x^5+1$
$F_{2^{409}}$	$f(x) = x^{409}+x^{87}+1$
$F_{2^{571}}$	$f(x) = x^{571}+\mathrm{x}^{10}+\mathrm{x}^5+\mathrm{x}^2+1$

Figure 3.2.1.2.A Binary Finite Field Reduction Polynomials

3.2.2 Elliptic Curves

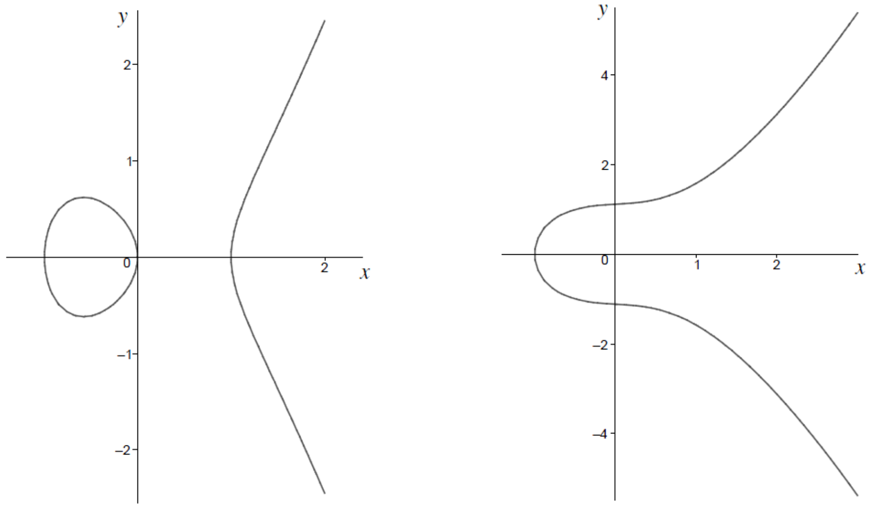
Elliptic curves are most commonly shown in the form of the simplified Weierstrass equation in the form of:

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

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 Figure 3.2.2.A 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 3.2.2.A Sample Elliptic Curves [10]

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 [10]: given an elliptic curve E defined over a finite field $\boldsymbol{F}_q$, a point $P \epsilon E(\boldsymbol{F}_q)$ of order n, and a point

Q ϵ <P>, find the integer *l ϵ [0, n-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. Figure 3.2.2.B below shows a geometric representation of both of these rules.

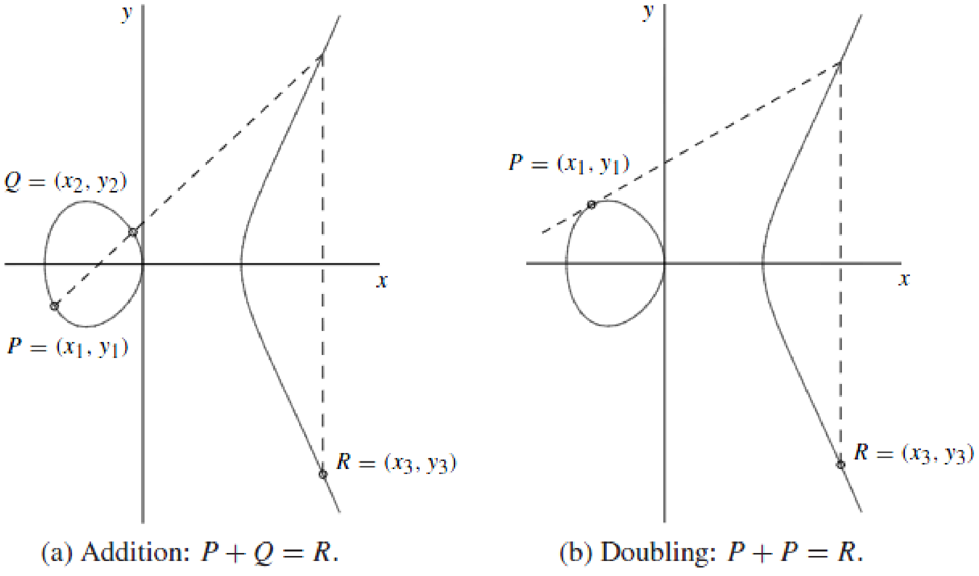
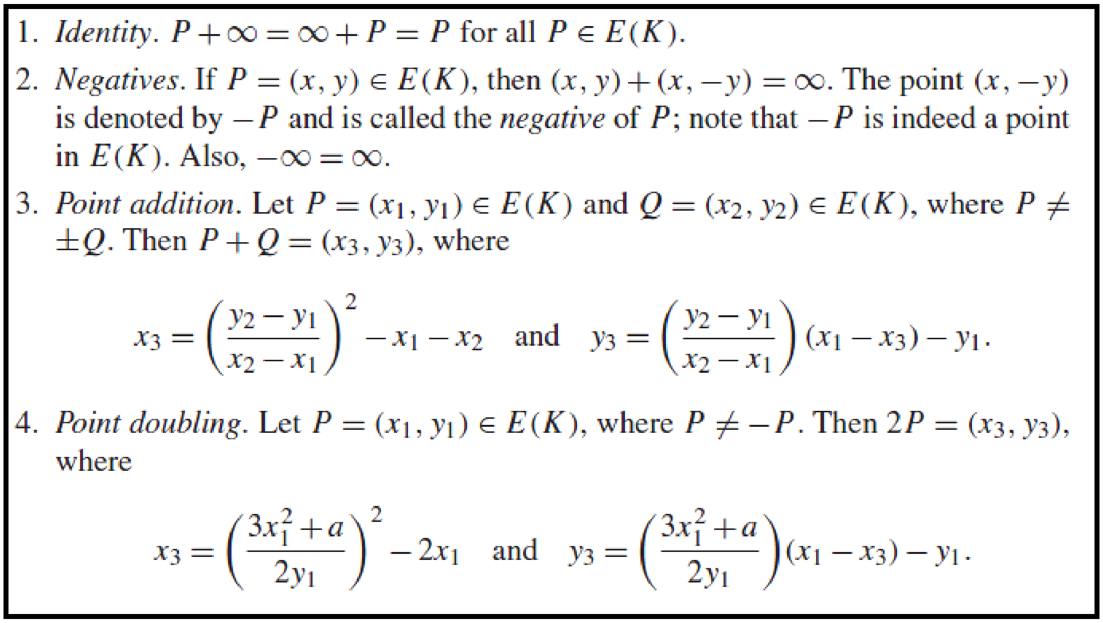


Figure 3.2.2.B Geometric Representation of Point Addition and Point Doubling [10]

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 [10]. We consider these cases:

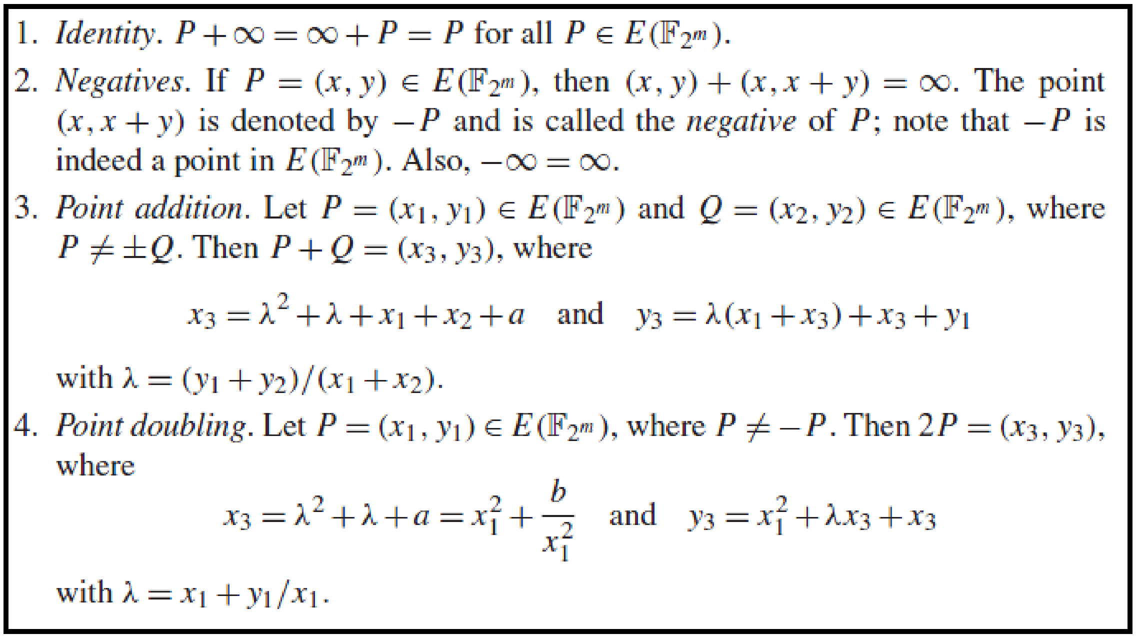
- The characteristic of the underlying field K is not 2 or 3 (e.g. $K = \boldsymbol{F}_p$ where $p > 3$ is a prime)
- The curve E is non-supersingular of the form over $K = \boldsymbol{F}_{2^m}$
- The curve E is supersingular of the form over $K = \boldsymbol{F}_{2^m}$

The easiest group law to understand is for that of the simplified Weierstrass form for *char(K)≠2,3*, shown in Figure 3.2.2.C. Group laws for the simplified Weierstrass form for *char(K)=2* are shown in Figures 3.2.2.D and 3.2.2.E 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 3.2.2.C Group Law for E($\boldsymbol{F}_p$): $y^2=x^3+ax+b$, char(K) ≠2,3 [10]



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

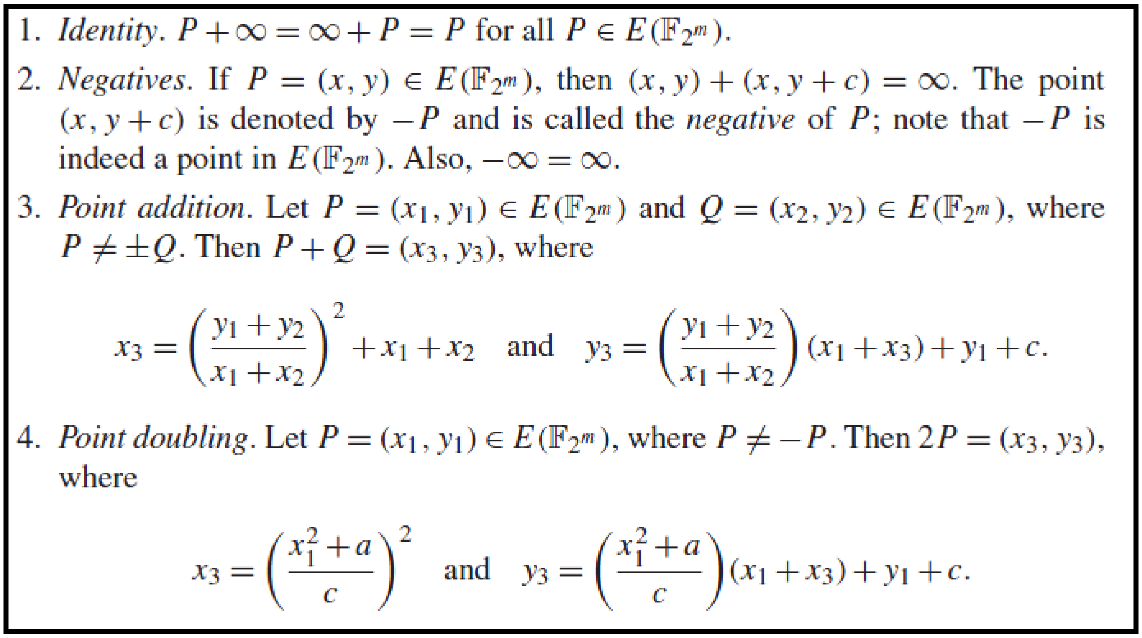
   $$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \quad \text{and} \quad y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$

   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 3.2.2.D Group Law for non-supersingular E(F_{2^m}): $y^2+xy=x^3+ax^2+b$ [10]



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 3.2.2.E Group Law for supersingular E(F_{2^m}): $y^2+cy=x^3+ax+b$ [10]

3.2.3 Projective Coordinates

The group laws shown in section 3.2.2 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 [10]), then the use of a technique known as projective coordinates may be advantageous to use.

Projective coordinates essentially works 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:

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

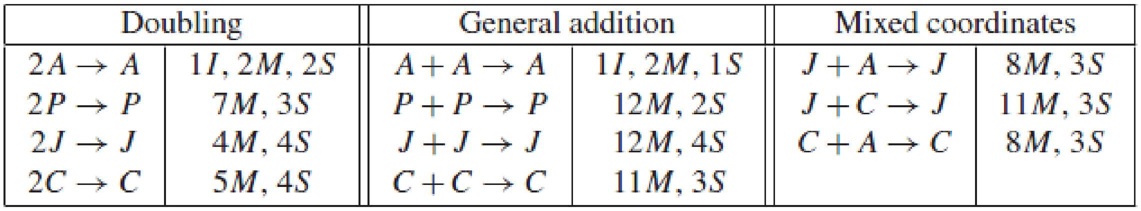
to the projective form:

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

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 [10]. 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. Figure 3.2.3.A 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	
$2A \rightarrow A$	$1I, 2M, 2S$	$A + A \rightarrow A$	$1I, 2M, 1S$	$J + A \rightarrow J$	$8M, 3S$
$2P \rightarrow P$	$7M, 3S$	$P + P \rightarrow P$	$12M, 2S$	$J + C \rightarrow J$	$11M, 3S$
$2J \rightarrow J$	$4M, 4S$	$J + J \rightarrow J$	$12M, 4S$	$C + A \rightarrow C$	$8M, 3S$
$2C \rightarrow C$	$5M, 4S$	$C + C \rightarrow C$	$11M, 3S$		

Figure 3.2.3.A Operation Counts on $y^2 = x^3 - 3x+b$ [10]

3.2.4 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 [10]. Shamir's Trick is yet another method used specifically to speed up the calculation of $kP + lQ$ by performing simultaneous multiple point multiplication [10].

3.3 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.

3.3.1 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 [24]. If the elliptic curve domain parameters are verifiably random than they should be accompanied by the seed value *S* from which they are derived [24].

3.3.2 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 [24]. If the elliptic curve domain parameters are verifiably random than they should be accompanied by the seed value *S* from which they are derived [24].

3.3.3 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 [24] 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 [10].

The conventional wisdom of ECC has been, as described by Koblitz [27]:

- 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 [27].

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 NGIPS following the NIST recommendation seems to be the most prudent.

3.4 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 [24] will provide protection against all known attacks (i.e. render these attacks computationally infeasible).

3.4.1 Naïve Method

The most naïve method for solving the ECDLP is to perform an exhaustive search where one computes the sequence of points $1P, 2P, 3P, \ldots lP$ 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. $n = 2^{80}$) [10]. 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 [10]. 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 [10].

3.4.2 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.

3.4.3 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 [10]. 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.

3.4.4 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 [10].

3.4.5 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 [10].

3.5 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 in section 2.3.2.2. However, as discussed in section 2.3.2.3 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). Since ECDSA is approved by NIST and included in their NSA Suite B it is therefore the most suitable candidate for use in control systems. The subsections below detail the algorithm, beginning with generating private and public keys for use in ECDSA.

3.5.1 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 3.5.1.A 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 3.5.1.A [10] – 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 3.5.1.B illustrates how to perform the required validation.

Algorithm 3.5.1.B [10] – 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 of Algorithm 3.5.1.B 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 [24]) then successful completion of the checks in steps 1 through 3 imply that $nQ = \infty$ [10].

3.5.2 Elliptic Curve Digital Signature Algorithm (ECDSA)

Algorithms 3.5.2A and 3.5.2.B 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. More information on hash functions can be found in section 3.5.3.

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

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

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:

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

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

Algorithm 3.5.2.A [10] – 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 $k \in_R [1, n\text{-}1]$
2. Compute $kP = (x_1, y_1)$ and convert x_1 to an integer $\overline{x_1}$
3. Compute $r = \overline{x_1} \ mod\ n$ and if $r = 0$ go to step 1
4. Compute $e = H(m)$
5. Compute $s = k^{-1}(e + dr)\ mod\ n$ and if $s = 0$ go to step 1
6. Return (r, s)

Algorithm 3.5.2.B [10] – 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 $w = s^{-1}\ mod\ n$
4. Compute $u_1 = ew\ mod\ n$ and $u_2 = rw\ mod\ n$
5. Compute $X = u_1P + u_2Q$
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

3.5.3 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 [24] supported hash functions for ECC are:

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

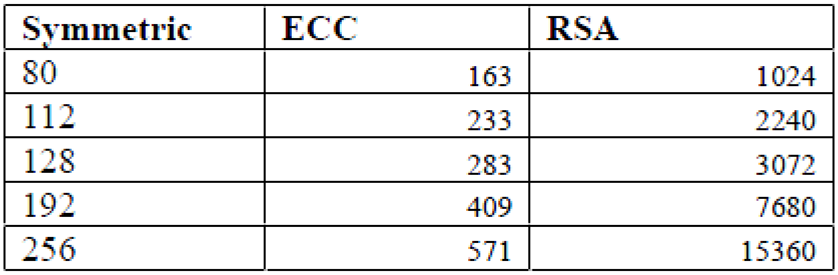
On October 2, 2012 NIST concluded a competition for a new SHA-3 algorithm, selecting Keccak as the winner. Future versions of SECG standards are likely to allow use of the new SHA-3 [23].

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. Recent results have shown that SHA-1 provides less than 80 bits of collision resistance [23] and therefore should be used with ECDSA only when providing backwards compatibility.

3.6 Comparing RSA Signatures to ECDSA

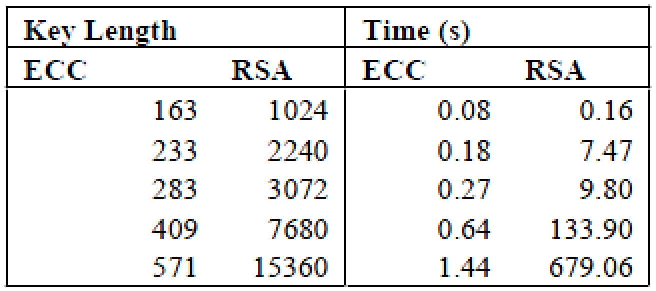
It has already been stated that ECDSA offers security equivalent to RSA using much smaller key sizes which can lead to increased efficiency. Figure 3.6.A below shows a chart of comparable key sizes for equivalent levels of security. Figures 3.6.B through 3.6.D below show published literature execution 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 [29]. 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. Further research is required in this area to determine what the actual timings would be on control system hardware.



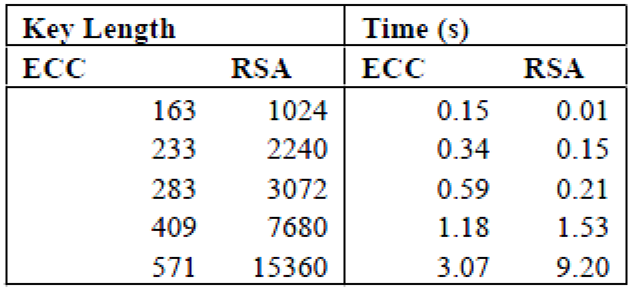
Symmetric	ECC	RSA
80	163	1024
112	233	2240
128	283	3072
192	409	7680
256	571	15360

Figure 3.6.A ECC vs RSA Comparable Key Sizes (in bits) [29]



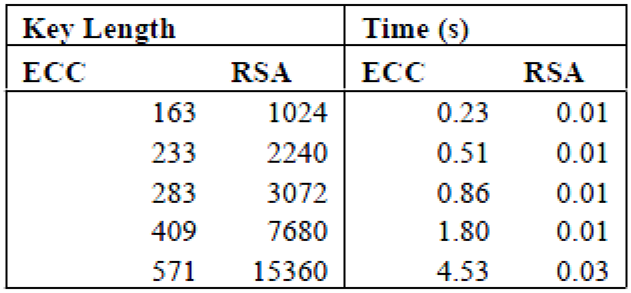
Key Length		Time (s)	
ECC	RSA	ECC	RSA
163	1024	0.08	0.16
233	2240	0.18	7.47
283	3072	0.27	9.80
409	7680	0.64	133.90
571	15360	1.44	679.06

Figure 3.6.B ECC vs RSA Key Generation [29]



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 3.6.C ECC vs RSA Signature Generation [29]



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 3.6.D ECC vs RSA Signature Verification [29]

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) may hold promise if ECDSA does not prove to be efficient for use in control systems. In EC-KCDSA the signer's private key is an integer $d \in_R [1, n-1]$ 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 should ECDSA prove impractical. 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) [32], 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.

3.7 OpenSSL ECC Implementation

As much as has been discussed up to this point on the underlying mathematics and implementation theory of ECC and ECDSA, in particular, most engineers never develop their own implementations. They instead rely on existing implementations which they incorporate into their own products. OpenSSL provides a suite of cryptographic toolkits including toolkits for ECC written in C that can be readily incorporated into new products.

The ECC implementations present in OpenSSL were contributed by Sun (now Oracle) and offered freely with "patent peace provision" language (meaning they will not sue anyone for using their implementation and ask, but not require, that you do not sue them if they use a product you develop with their technology). This implementation was theoretically written in a way that avoids any patented method by basing the implementation on the current IETF [26] draft [30]. However the issue of patents appears to be far from settled, and some versions of Linux such as Red Hat do not include the ECC toolkits in their versions of OpenSSL. There also exist JAVA and .NET implementations.

While it is true that in control systems the OpenSSL toolkit cannot be used by PLCs (since they cannot run C binaries), VME technologies including SoftPLC may be able to leverage the OpenSSL implementation. Currently there are no known implementations of ECC written specifically for control systems that are compliant with IEC 61131-3 or IEC 61499.

3.8 ECC Certificates

As discussed in section 3.5.1, certificates play a key role in cryptographic systems. In ECC, they are used in order to associate a public key with a set of domain parameters. The problem with ECC is that current there are no Certificate Authorities supported by major web browsers for ECC, causing some to not consider ECC a true public-key cryptography scheme. SECG is working hard on changing this, establishing itself as an ECC certificate authority and publishing standards to indicate ECC keys and their usage within X.509 certificates [31]. However there is still significant work to do in this area in order to truly make ECC a viable solution for complex control systems such as that in the Smart Grid. For smaller control systems such as those planned for usage in NGIPS (which are still vastly complex by industry standards with tens of thousands of I/O points) the lack of a strong ECC certificate authority is not as much of a roadblock.

CHAPTER 4: PROTOTYPE OVERVIEW

4.1 Prototype Objectives

ECDSA shows promise for use in control systems, however there are a number of questions that arise from the perspective of a controls engineer such as:

- How difficult will this be to implement?
- What impacts will this have on the performance of my controls algorithms?
- What kind of software maintenance is needed to support ECDSA in control systems?
- What are the costs of implementing these algorithms?

In order to begin to answer these questions, we developed a prototype control system that matches architectures used in real applications to run actual control algorithms. The primary objective of the prototype was be to determine the viability of using ECDSA in control system data authentication and verification.

Given the sheer complexity of developing a brand new implementation of ECDSA in IEC 61131-3 code a "proof of concept" study was needed to more accurately assess the validity of using ECC technology in control systems before significant time and money are invested. SoftPLCs provide a unique opportunity to perform this analysis by developing a prototype system that is predominately written in IEC 61131-3 code but allows the use of specialized custom function blocks written in other high level languages. This allows the development of a prototype implementation of a control system ECDSA algorithm that is able to reuse existing software libraries in order to avoid the expense of developing ladder logic cryptographic functions. The prototype implementation, being predominately written in ladder logic, can then be reused for a full

IEC61131-3 implementation by simply replacing the custom function blocks with IEC61131-3 versions.

Please note that currently there are a large number of SoftPLC based control systems in use in both Naval and industrial control system applications which would be better suited for the prototype implementation. Therefore, an auxiliary goal of the project beyond testing the capability of ECDSA is to create a software template that can be easily reused by control system engineers in other applications at minimal cost. The prototype project also included additional features beyond messaging such as enhanced alarming functions that not only indicates communications status but failures in signature verifications, indicating a potential hardware failure or adversary attack.

The remainder of this dissertation presents the system architecture, source code, timing results, and stability results for a prototype ISaGRAF™ SoftPLC implementation of ECDSA for control systems. ISaGRAF™ SoftPLCs are currently in use on a wide range of industrial control applications around the world, and are currently employed by the Navy in two major programs: the Littoral Combat Ship (LCS) Class and the Mobile Landing Platform (MLP) Class for ship wide machinery control. The challenges and solutions uncovered when developing the implementation are discussed, and guidelines for converting the prototype into a full IEC 61131-3 compliant implementation is presented.

4.2 System Architecture Overview

The primary goal of the prototype is to determine the overhead the ECDSA implementation would have on control system operation. In order to have effective control, the scan times for the logic must be sufficiently low enough to approximate real time operation (less than 100 ms and ideally less than 50 ms). Additionally, for PLC to PLC communications effective control requires transmitting and processing at least one set of actions every 300 ms or less (ideally every 100 ms or less). There are a number of different things that can impact both scan times and transmission times, particularly for SoftPLCs running on a Windows OS, such as network latency and individual hardware I/O access rates.

In order to obtain averages for scan times and transmission times a simple "round robin" architecture was chosen as shown in Figure 5.A below. Four SoftPLCs, labelled 1 through 4, were configured so that each SoftPLC processes a piece of data and then subsequently transmits it down the line, repeating the process in an indefinite loop. The data transmitted included a number of simulated signals, including BOOL, INT, REAL, DATE, and STRING values that were part of a small logic simulation routine.

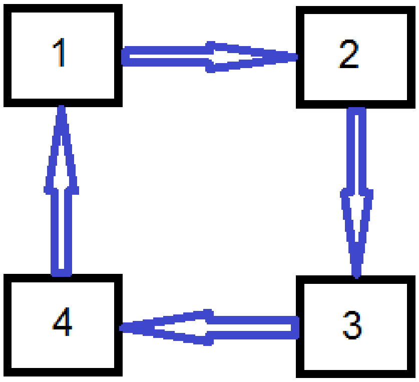


Figure 4.2.A. SoftPLC Round Robin Architecture Concept

The primary data point of interest for this system is a DINT value called COUNT. COUNT is a number that is initialized to 0 at the very start of the system. Each SoftPLC receives COUNT as an input from its predecessor, verifies the digital signature that came with the COUNT value, and if verification is successful it increments the COUNT value by 1, generates a new message, and transmits the new message to the next PLC in the loop. Figure 5.B below illustrates the count increment lifecycle within the PLC's execution processes. As can be seen, each value of count represents a complete set of signature generation, data transmission, and signature verification actions. Transmission time is then synonymous with the timing of the COUNT lifecycle. In order to determine an average time, the system was run until it had reached a certain value of COUNT. The runtime was then captured and the runtime divided by the COUNT gives us the average transmission time.

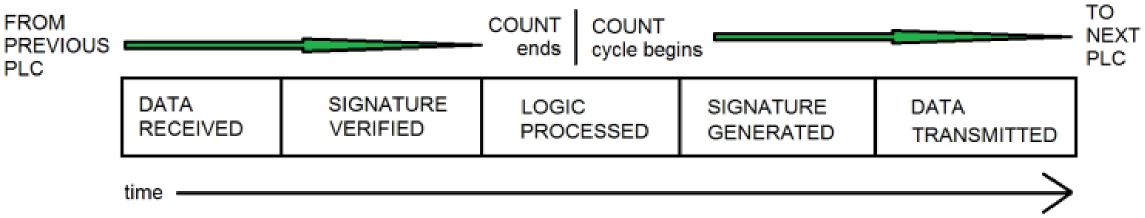


Figure 4.2.B. Count Increment Lifecycle Per SoftPLC Scan

Please note that some SoftPLC products offer an additional "WAIT" option as a step after the DATA TRANSMISSION step in their PLC. This feature normally works by predefining a PLC execution cycle time, like 100 ms, and essentially takes the 100 ms minus the time it took to complete all of the above steps and then waits for that time until it begins executing the next cycle. This is normally used to add a bit more determinism to the SoftPLC and provides a window of opportunity for the operating system to execute tasks. Since the goal of this prototype to determine loads by monitoring scan and transmission times the WAIT feature was disabled.

The PLC scan times are monitored by using built-in tools that come with all major SoftPLC and regular PLC packages. Scan time is not synonymous with transmission time since the PLCs are not synchronized (as is typical with industrial operations). Additionally, it is possible that a PLC might receive an invalid signature in a data transmission due to events like network transmission errors that have damaged the data. Any COUNT with an invalid signature is disregarded, making COUNT representative for only successful transmissions with correct signature generation and verification.

4.3 Hardware Overview

Hardware Two test strings were used in the development and testing of the prototype, Test String 1 and Test String 2.

4.3.1 Test String 1

Test String 1 used four Windows 7 virtual machines running on VMWare Player on top of a Windows 7 Pro 64-bit machine. This system was primarily used for initial development and testing. The ISaGRAF software was run on the main machine in order to view the PLC code. The four virtual machines were bridged to the host NIC card which was then connected to a router. DHCP was used to assign IP addresses to both the host and virtual machines.

The main machine had the following relevant hardware specifications:

- I5-2500K processor, 4C, overclocked to 4.3 GHz, with 4x256KB of L2 Cache and 6MB of L3 Cache
- 16 GB RAM

- 1 GB Hardwired NIC
- 1.5 TB SATA 3 HDD

4.3.2 Test String 2

Test String 2 used four General Micro System VS275 Single Board Computer VME boards in the same VME chassis. No communications across the VME backplane were used. Each board was running Windows XP Professional, 32-bit, SP2. Figure X below shows the front panel of the VS275.

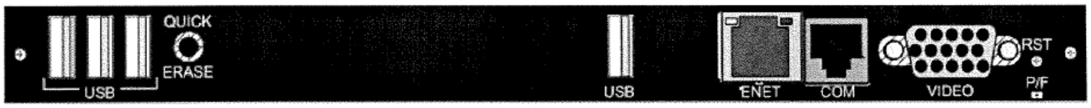


Figure 4.3.2.A. VS275 Board

Each VS275 board had the following specifications:

- 2.16 GHz Core 2 Duo Processor, 4-MB L2 cache
- 3 GB of 667-MHz DDR-2 SDRAM
- 1 GB Hardwired NIC
- 64 GB SATA2 SSD

Hardwired Ethernet communications were used via the ENET port on the front of the board. All four boards were connected to an 8 port flat 10/100 MB hub that was in turn connected to a 10/100/1000MB router. A development station running ISaGRAF was connected to the router directly in order to download the software to the boards and to go online to the SoftPLCs. DHCP was used to assign IP addresses to each of the boards and development station.

CHAPTER 5: SOFTWARE ARCHITECTURE

5.1 SoftPLC Package

There are two popular SoftPLC packages currently in use by the Navy: Rockwell Canada's ISaGRAF package and Siemens WinAC. Both packages can be used to develop IEC 61131-3 ladder logic and include custom C function blocks. The primary difference between the packages is that Siemens WinAC includes additional functionality installed on the target hardware that adds real-time determinism and disables certain functions in Windows that could result in a "blue screen of death" error. ISaGRAF, on the other hand, is more comparable to Java in that it includes a simple executable known as the ISaVM that in turn executes the logic. Like Java, ISaGRAF has multiple ISaVM implementations that allow the software to be used with multiple platforms including both Linux and Windows. Ultimately, ISaGRAF was chosen for this implementation because it used on more US Navy Ships than Siemens WinAC and because Rockwell Canada was willing to provide free licenses to support development of this prototype (special thanks to Stephen Mizera at Rockwell Canada for his support).

The ISaVM executable lives with a collection of related executables in a folder that is collectively referred to as the "target". The target is installed onto the hardware via a simple copy operation which can be performed anywhere within the file structure of the SoftPLC's operating system. The target includes a main program, "ISaGRAF.exe" that is manually started by the user. This program then starts up the subprograms including both ISaVM.exe and the default Ethernet communications program "ETCP.exe". The target also includes a text file known as the "target definition" file that describes details about the target such as what custom C function blocks are available.

In order to develop custom C function blocks for inclusion in the ladder logic a tool called TDBuild (Target Definition Builder) is used to define the input and output variables of the function blocks. The tool is then used to auto-generate C code which essentially provides the API between the ladder logic and the custom C code that was developed as part of this prototype. TDBuild is also used to update the target definition files, which are then read into the main ISaGRAF program in order to allow the software to include the specialized function blocks in the ladder logic.

For Windows-based targets the custom C code is eventually compiled as a dynamic link library (DLL) which is then copied into the target folder. This DLL, combined with the target definition files discussed previously, provide full runtime access to the custom C code for program execution. Note that you do not have to register the DLL with Windows which greatly simplifies the install.

5.2 OpenSSL Usage

As stated previously, developing a custom IEC 61131-3 implementation of ECDSA is a complicated endeavour and a prototype system that utilizes existing implementations is a necessary first step. For this prototype the OpenSSL ECC implementation discussed in Section 3.7 was been chosen for both its efficiency and its free and open source availability to all controls engineers. There are three methods by which OpenSSL can be included in a project:

1. By installing OpenSSL onto the target system and using its command line interface.

2. By performing a fresh compile of OpenSSL as a dynamic library which can then be copied onto the target system and used after first properly registering the DLL.

3. By performing a fresh compile of OpenSSL as a static library which can then be included in your application without having to first install OpenSSL onto the target system or register any DLLs including the DLL used by the ISaGRAF software itself.

For this prototype OpenSSL was compiled as a static library in order to simplify the installation process onto the target machines (i.e. to keep the ISaGRAF target install down to a simple copy operation). The downside to this option is that it does complicate the structure of the actual C source code and increases the development time of the prototype. It was decided that the increase in development time is justified by the increased ability to use the prototype on existing Navy ship classes that are using SoftPLCs and do not require a 100% IEC 61131-3 implementation, such as Littoral Combat Ships (LCS) and Mobile Landing Platform (MLP) ships.

The following steps were performed to compile OpenSSL as a static library on a Windows 7 machine:

1. Install Visual Studio 2010
2. Install Active Pearl 32-bit
3. Download the latest version of OpenSSL
4. Open the Visual Studio Command Prompt
5. Unzip OpenSSL to a directory such as C:\openssl-src-32
6. CD to the directory

7. Run the command: perl Configure VC-WIN32 –prefix=C:\Build-OpenSSL-VC-32

8. Run the command: ms\do_ms

9. Run the command: nmake –f ms\nt.mak

10. Run the command: nmake –f ms\nt.mak install

The compiled library will end up in the C:\Build-OpenSSL-VC-32 directory.

5.3 Visual Studio 2010 Express

Visual Studio 2010 Express (a free product) was used to develop and compile the custom C function blocks into a DLL. Importing the C code and header files generated by TDBuild and to the Visual Studio project is a simple matter, but there are additional steps that must be taken in order to configure the compiler to properly utilize the dependent libraries in order to generate the DLL. The two critical steps required are:

1. Under Linker – General – Additional Library Dependencies add the OpenSSL static library.
2. Under Linker – Input – Additional Dependencies add the libeay32.lib and the ssleay32.lib.

Note that Visual Studio 2012 was also used successfully in the later stages of the project to recompile the DLL and other C source code developed for the prototype as needed.

5.4 Cryptographic Algorithms Used

As discussed previously, creating a digital signature requires the use of two different cryptographic algorithms: a hash function and a public-key cryptography function. The hash function is performed first in order to process the arbitrarily long amount of data into a fixed length tag. The cryptographic function then encrypts the data using the private key, which can then be decrypted using the public key by an agent desiring to verify the public signature.

For this work the SHA-512 algorithm was used in combination with the P-521 ECDSA algorithm. The goal was to use the strongest measure of security possible at the time of development, and it is worth noting that the weaker P-384 ECDSA algorithm is considered valid for even TOP SECRET data. OpenSSL includes implementations of both algorithms in the static library that was compiled.

Using the SHA-512 algorithm was relatively trivial, but the P-521 algorithm proved to be a bit trickier than originally anticipated. The P-521 algorithm consists of 65 bytes of data plus 1 bit. The extra bit, when true, results in an output of 66 bytes. When the extra bit is false, the result is an output of 65 bytes. As will be illustrated in the walkthrough of the C code (and as is shown in the source code comments) special considerations were needed to monitor the output of the algorithm to ensure that the correct amount of data was read.

5.5 Base64 Encoding / Decoding

Transmitting cryptographic keys and digital signatures between PLCs is not a trivial task. PLCs only include a predefined number of data types such as BOOL, INT, DINT, and STRING which can be transmitted between devices. All of the data types except for STRING are therefore unsuitable for transmitting the keys and signatures due to the insufficient bit length.

In most PLC products, the STRING variable can consist of any ASCII character array up to a length of 255 characters according to the literature of the various vendors. This would appear to give a data value of 2040 bits using ASCII encoding, sufficiently long enough to transmit ECDSA keys and signatures. Note that this length is still too short for the use of RSA keys and digital signatures which is one of the reasons why ECC technologies were chosen.

The problem is that the 255 character set is misleading, since in ISaGRAF and in other PLC products 3 of the 255 characters are reserved to process the STRING. One of the three character slots must be used for the NULL character (which should indicate to the experienced C programmer that the STRING data type is really a C-String and not the more advanced string data type found in C++ or other high level languages). Additionally, the STRING must begin with an apostrophe and end with an apostrophe. The "apostrophe" problem becomes our greatest concern. When representing an ECDSA signature or key in ASCII it is possible that the resulting ASCII string will output an apostrophe in the middle of the key or signature. The PLC will interpret the second apostrophe (and note that you must always start with an apostrophe) as the end of the string and subsequently cut off the remaining data.

In order to work around this problem, a different character encoding must be used that maximizes the data compression of the string representation. Hexadecimal representation of the data was considered, but at only 4 bits per character it was considered insufficient. A custom variant of ASCII was considered that didn't use the apostrophe, but in the interests of conforming to widely accepted standards this idea was dropped. Base64 encoding, using 6 bits per character, became the most logical choice.

Base64 encoding allows the cryptographic keys to be transmitted as a single string variable for each key. The private key is represented by a 88 character string (plus 3 for the null and the leading/lagging apostrophes). The public key, which represents a point on an elliptic curve, is represented by a 178 character string (plus 3). The digital signature is represented by two values, SIG_R and SIG_S, each with a length equivalent to the private key. These values were transmitted as two separate strings to help facilitate the need for possible future expansion.

Note that the STRING variables themselves which are storing the data in the PLC ladder logic are required to be set to a fixed length. This is a fairly standard requirement by most PLC manufacturers. In order to facilitate the need for future strong cryptography it was decided to set the length for these variables in the ladder logic to the largest possible value. As a consequence, the PLC STRING variables include padded data in addition to the cryptographic keys or digital signatures.

5.6 Self-Signing Keys

One of the largest areas of discussion with using a public-key cryptography system for control system security has been a concern about how to setup a public-key infrastructure (PKI). In order for two systems to be able to communicate securely, there needs to be some initial trust relationship established so that a receiving system which obtains a public key from a sending system knows that the public key really belongs to that system. Put another way, how does the receiver really know that the sender is who it claims to be?

In traditional information systems a Certificate Authority (CA) is used to issue certificates to users. This certificate contains the identity of the key pair owner, the owners public key, and a digital signature of the Certificate Authority. When users communicate securely, they receive a copy of each other's signed certificate (minus the private key) and they establish trust in each other's identity based on the successful validation of the CA's signature. Put another way, they trust each other because someone else told them it was okay. Ultimately, even trusting the digital signature of a CA is based on the idea that some human ultimately made the decision that the CA who issued the certificates is of good repute.

This kind of complicated infrastructure is necessary in IT systems, since two communicating systems may send a wide range of different kinds of data with a wide disparity in both timing and content. Furthermore, an IT system may communicate only once with another system it never heard of before and then never communicate to that system again. This constant flux requires a PKI in order to establish trust relationships with CAs and individual users. Unfortunately, running a PKI system requires a significant amount of processing power and adds significant complexity to a control system. Research is ongoing to develop a modified version of a PKI that can be executed on operational technology platforms, however most of these are simply slightly modified versions of the same technologies used in IT systems.

We believe that implementing a PKI for control systems is unnecessary. The two central problems that a PKI system solves are 1) establishing a trust relationship between two systems and 2) handling the large amount of flux in data transmitted between systems. In Operational Technology systems, neither of these problems really exists. OT systems are designed to communicate in a very consistent and precise manner, transmitting the same basic message structure at a consistent interval to the exact same targets. The relationship between these targets is established when the system is first commissioned for operation (i.e. on the day of birth) and no new communication partnerships are ever established without significant software changes and a recommissioning of the system.

Therefore, in designing this prototype, a different system for establishing communications and updating keys was created. During commissioning, each system is preconfigured with a public / private key pair, and the public keys of each part of the system are given to each of the other parts. When an agent in the system decides to change its public / private key pair (this decision is made at regular intervals with the interval length configured at commissioning) it first generates a new key pair and then signs the new public key with the old private key. This information is then transmitted to each of the other agents in the system who then verify the key change message with the old public key they currently have on file. If the digital signature is valid, the new public key is accepted and the old public key is disregarded.

In order to ensure that an adversary will not be able to attack the system using the original preconfigured public/private key pair each agent immediately changes its key to a new random key pair at startup, before any other logic is processed. Additionally, since there is no certificate authority and communications are highly deterministic there is little risk in changing the key pairs at a much more frequent interval than what is typically found in Information Technology systems (where certificates and keys can be valid for years). In the prototype, each of the four SoftPLC agents were configured to change their key pairs at 53 minutes, 59 minutes, 61 minutes, and 67 minutes respectively.

As a result, this system effectively generates keys 175,000 to 265,000 times more often the PKI CA systems used in IT applications and greatly weakens an adversaries capabilities to brute force crack the system. Ultimately, this allows for smaller key sizes to be used by OT systems and thus reduces the required processing power and memory required to effectively implement an ECDSA algorithm.

The major downside with this implementation is that an agent in the control system which is powered down must retain their current public/private key pair in memory and resume use of that key pair upon startup (though it can immediately change it). A agent which does not have this capability may default back to the original pre-commissioned key pair and will therefore be considered to be a bad actor by the other agents in the system, requiring a complete re-initialization of the entire system. Fortunately, this problem is easily solved by adding the required memory capabilities and is mitigated by the fact that these kinds of systems are highly redundant and designed to run uninterrupted for years without failures.

One alternative variation of this system under consideration is to commission each PLC with a master public-private key pair, and then subsequently generate session public-private key pairs which are used for the actual data verification signatures. An advantage of this method is that the system is much more robust and capable of dealing with circumstances such as power outages that may result in significant downtime. The disadvantage is that more stringent controls will have to be put in place on the PLC source code and design artifacts which may contain a particular PLC's master public-private key pair.

CHAPTER 6: SOURCE CODE WALKTHROUGH

Copies of the source code developed for this prototype with comments are available as appendices to this dissertation, and key details and decisions made in designing the software have been presented in the previous sections. This subsection presents a high level description of each of the modules that together make up the prototype, in an effort to provide context for the source code modules.

Note that for the C code, the portions of the ISaGRAF code generated by the TDBuild tool are not shown as that code is considered proprietary. Fortunately, the code not shown simply serves as the API to the PLC ladder logic and is specific to the ISaGRAF platform. Current users of ISaGRAF will be able to use the same tools with the information provide in this document to quickly regenerate this code. Users of other SoftPLC products should be able to develop their own variations using the information provided here.

6.1 C Code Walkthrough

The subsections below describe the model and C code for each of the custom function blocks that are used in the IEC61131-3 ladder logic. A fully compliant IEC61131-3 implementation will replace the C code inside each of these function blocks with a ladder logic version.

In developing the prototype, debug logic was added into each of the function blocks. Two different mechanisms for debug logic were included. The first is that each block outputs an integer status variable, with a 1 indicating successful operation and a negative number indicating a failure. The code of the blocks is organized internally in steps as shown in the source code and comments. A failed step will output a negative number equal to the step number (i.e. a fail on step 3 will output a -3 value for the status).

The second debug logic is normally turned off for full operation and was only added for development. This logic generates text files during operation that contain key status variables useful for debugging the code. This adds overhead to the program execution and creates a security risk, but is extremely valuable for debugging. In order to turn the logic on or off, the variable "ISaDEBUG" must be set true for on or false for off. This variable is located in the "debug.h" file, and a change in the status of the variable requires a recompile of the DLL. All the timing results presented are with the ISaDEBUG variable set to false.

The custom C source code developed for the ISaGRAF implementation can be found in Appendix A. Please note that only the parts of the C code written by the author are included in this Appendix. Auto-generated code from the ISaGRAF TD Build tool is not included. Presence of auto-generated code in the source files below is indicated by the tag "---autocode---".

6.1.1 MsgGen

Figures 8.1.1.A and 8.1.1.B below define the details of the MsgGen block. In digital signature applications, the first step of an algorithm is to hash the data to be signed into one single value of fixed length, that will later be encrypted by the private key. The MsgGen block generates that hash value using the SHA-512 algorithm. The data to be signed is inputted into the B#, I#, R#, and STRING fields. The DATE and TIME fields should always be used for the current date and time stamps of the message or the entire algorithm is potentially subject to a replay attack. The block outputs the hash as a hexadecimal string.

Creating one generic message generation block that can be used in a wide range of applications is tricky, since one application could only need a few Booleans and another application could require 40 different real values. Therefore the block was structured generically in order to allow the widest range of possible inputs, and then a HASH_I value was added that allows the user to chain multiple blocks together. Readers familiar with the SHA-512 algorithm will recognize that there is no loss in processing the data in this manner, since the algorithm generates the hash in a "chain" fashion naturally. This implementation simply extends the chain, and there is no limit to the number of MsgGen blocks that can be chained together.

The key for implementing this function is that the data to be signed must have a hash generated on both the source PLC and the destination PLC. Therefore, during system design, it is critical that the layout of the MsgGen blocks is the same on each endpoint or the signature will never verify successfully.

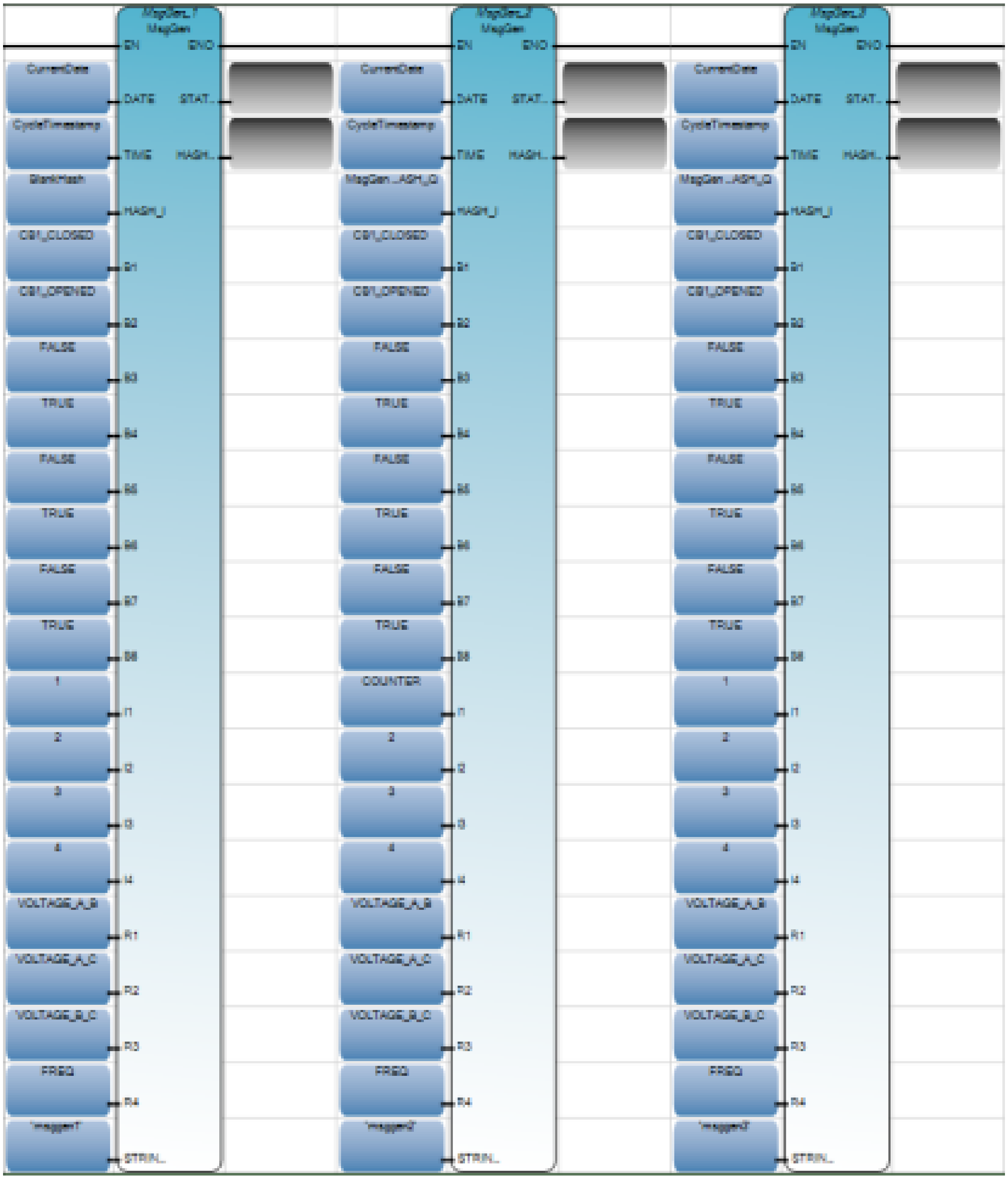


Figure 6.1.1.A Three MsgGen Blocks Connected Together

Variable Name	**I/O**	**Type**	**Description**
DATE	IN	DATE	Used to date the message in order to prevent replay attacks. Has resolution up to the second.
TIME	IN	TIME	Used to timestamp the message in order to prevent replay attacks occurring within the last second. Has resolution up to the millisecond.
HASH_I	IN	STRING(252)	Hash from previous MsgGen. Used to string multiple blocks together in order to generate messages containing large amounts of data.
B1 Thru B8	IN	BOOL	Boolean data inputs.
I1 Thru I4	IN	DINT	Integer data inputs.
R1 Thru R4	IN	REAL	Real data inputs.
STRING	IN	STRING(252)	String data inputs.
STATUS	OUT	INT	Status of the computation.
HASH_Q	OUT	STRING(252)	Hash of the message.

Figure 6.1.1.B MsgGen Input / Output Structure

6.1.2 KeyGen

The KeyGen block generates an ECC-521 prime public/private key pair. Ladder logic should be placed in the application to only enable the block at specific intervals when generation of a new key is desired. The new keys are outputted as base64 encoded strings. Figures 6.1.2.A and 6.1.2.B below define the details of the KeyGen block.

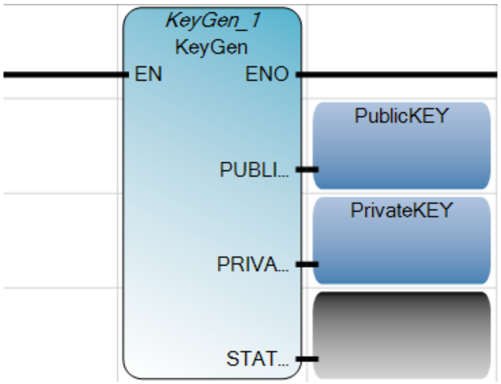


Figure 6.1.2.A KeyGen Block

Variable Name	I/O	Type	Description
PUBLIC_KEY	OUT	STRING(252)	ECC 521 public key
PRIVATE_KEY	OUT	STRING(252)	ECC 521 private key
STATUS	OUT	INT	Status of the computation

Figure 6.1.2.B KeyGen Input / Output Structure

6.1.3 KeyVerify

The KeyVerify block is used as an added check to verify the integrity of the public/private key pair by using the private key to first sign an internal dummy hash and then using the public key to verify the signature. This block is solely used in the prototype to verify that the system is functioning correctly. As the maturity of the design improves this block may be removed, thus reducing system overhead and improving communications results. Figures 6.1.3.A and 6.1.3.B below define the details of the KeyVerify block.

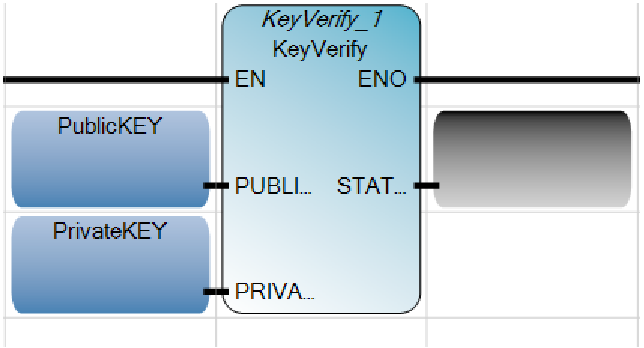


Figure 6.1.3.A KeyVerify Block

Variable Name	I/O	Type	Description
PUBLIC_KEY	IN	STRING(252)	ECC 521 public key
PRIVATE_KEY	IN	STRING(252)	ECC 521 private key
STATUS	OUT	INT	Status of the computation

Figure 6.1.3.B KeyVerify Input / Output Structure

6.1.4 SigGen

The SigGen block is the heart of the ECDSA implementation, and takes the hash from the MsgGen block and the PrivateKey from the KeyGen block to output an ECDSA P-521 signature. The signature itself is composed of two components, SIG_R and SIG_S, which are each a base64 encoded string. Figures 8.1.4.A and 8.1.4.B below define the details of the SigGen block.

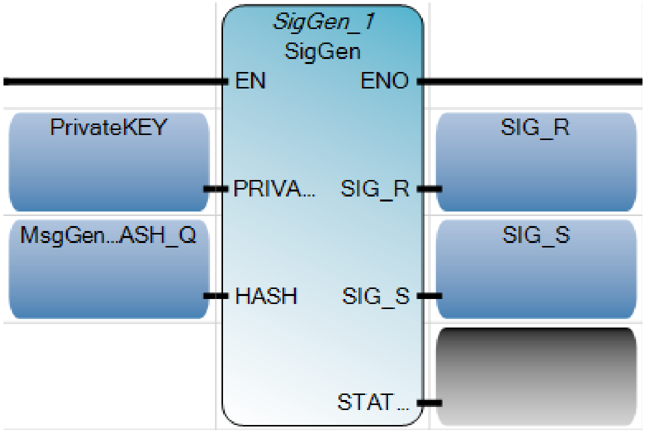


Figure 6.1.4.A SigGen Block

Variable Name	I/O	Type	Description
PRIVATE_KEY	IN	STRING(252)	ECC 521 private key
HASH	IN	STRING(252)	Data hash to be signed from the MsgGen block.
SIG_R	OUT	STRING(252)	ECDSA P-521 signature, R component.
SIG_S	OUT	STRING(252)	ECDSA P-521 signature, S component.
STATUS	OUT	INT	Status of the computation

Figure 6.1.4.B SigGen Input / Output Structure

6.1.5 SigVerify

The SigVerify block is used to verify the validity of a digital signature by the receiving agent. The receiving agent first uses the MsgGen block to generate the same hash that the original sender generated, and then uses the public key of the sender along with the signature to determine if the signature and the hash match. Figures 6.1.5.A and 6.1.5.B below define the details of the SigVerify block.

The SigVerify block is also used in the prototype system on the sender side to ensure that a signature is properly generated before transmission. As the design matures, the usage of the block on the sender side can be eliminated, reducing system overhead and improving communications results.

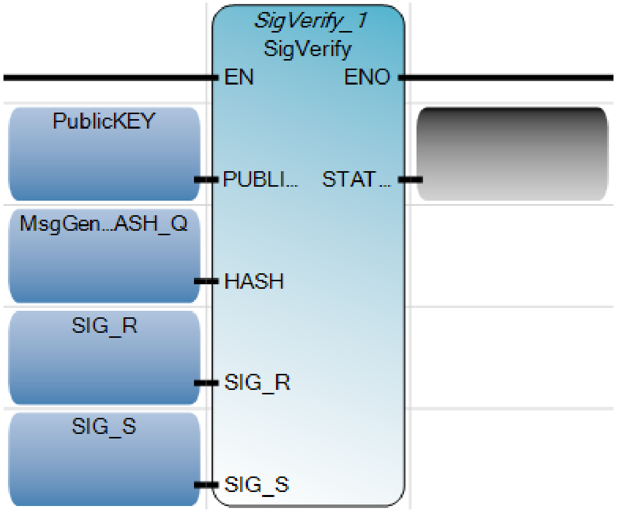


Figure 6.1.5.A SigVerify Block

Variable Name	I/O	Type	Description
PUBLIC_KEY	IN	STRING(252)	ECC 521 public key
HASH	IN	STRING(252)	Data hash to be verified from the MsgGen block.
SIG_R	IN	STRING(252)	ECDSA P-521 signature, R component.
SIG_S	IN	STRING(252)	ECDSA P-521 signature, S component.
STATUS	OUT	INT	Status of the computation

Figure 6.1.5.B SigVerify Input / Output Structure

6.2 Ladder Logic Walkthrough

For the prototype system, the ladder logic for each of the four SoftPLC agents is structured identically, and execution of the logic proceeds in the following order:

1. ECDSA_D#_I – process digital signatures for incoming data transmissions, including the validation of new public keys sent to the SoftPLC
2. MAIN – general logic such as determining current date, setting ALWAYS_ON and ALWAYS_OFF bits, and incrementing the counter value received during ECDSA_D#_I
3. SIM_BOOL and SIM_REAL – generic simulation routines that fill the place for where actual PLC logic would normally occur
4. ECDSA_KeyCntrl – controls the generation and validation of the SoftPLC's own private / public key pair
5. ECDSA_Q – generates digital signatures for the outgoing data transmissions

Full details for each of the ladder logic routines can be found in Appendix B below. This appendix includes a screenshot of the solution explorer detailing the layout of all four SoftPLC agents (indicated as Device 1 through Device 4). There is also a screenshot of the Device 1 to Device 2 data communication bindings which is representative of how each SoftPLC is binded to the next in the round-robin configuration. Additional information can be found in the comments section for each rung, indicated in the green highlighted fields above each ladder logic rung.

CHAPTER 7: RESULTS, FUTURE WORK, AND CONCLUSION

There were two main goals of this project:

1. Determine the validity of ECDSA for control system authentication and data verification by determining if an implementation can be developed that could be executed within the confines of a typical PLC scan time, and to decide if the development of a fully compliant IEC 61131-3 implementation is justified.
2. Determine an appropriate alternative method for key generation and distribution to solve the complexities of the public-key infrastructure problem discussed in the literature.

In order to satisfy goal 1, the prototype implementation was run on both test strings for periods ranging from several days to several weeks. COUNT values, reflecting the number of completed message transmissions (including both the signature generation and signature verification components) ranged from 10,000 to 4 million. In all scenarios the timing results were independent of the count length for values over 10,000. Test String 1 gave an average time of 26-28ms with PLC scan times of 22-24 ms. Test string 2 gave an average time of 55-60ms with PLC scan times of 55-60 ms. As expected, timing of the completed transmission is linearly related to PLC scan time.

Values under 10,000 counts gave slightly higher results (1-5 ms increase). During the first scan of the PLCs when the system is initializing there is an additional load on the system to establish TCP communications resulting in an increased scan time and transmission time for that cycle. For low COUNT values this initial time has a stronger effect on the average. As a result, for approximately the first 10 minutes of system runtime the average transmission time gradually decreased until the system settled at the values stated above.

It is believed that the timing results are the worst case scenario given that no real-time modifications were added to the operating system (which would be untypical for a SoftPLC control system). The entire timing cycle is in the order of tens of milliseconds (as opposed to seconds), falling within the confines of a typical PLC scan time and satisfying the first goal of the project. Based on these timings it has been concluded that future work on the development of a fully IEC 61131-3 compliant implementation is justified.

In order to satisfy goal 2, the self-signing key mechanism has been proposed which relies on the deterministic properties of a control system and the commissioning process to establish the initial trust relationship. Additional focus needs to be placed on the self-signing keys mechanism and is reserved for future work. While the prototype used an implementation that completely changed the public-private key pair the author does feel that the use of permanent master keys followed by individual session key pairs may increase system stability.

The difficulty of protecting the master keys from access by unauthorized individuals who obtain an offline copy of the code specifically needs further investigation. Most PLC vendors include a feature with their software that allows for developers to protect the source code with a password to prevent unauthorized viewing of the code both online and offline. The problem with this mechanism is that typically third party code reviews are necessary which means that the developer would need to supply this password to multiple parties, which would increase the exposure of the master key pair. A potential solution would be the inclusion of two different passwords by PLC vendors, one that is used to protect the ladder logic and another that is specifically to protect both the master and the session key pairs. This would allow third party audits and maintenance users to have access to the ladder logic code without exposing the key pairs.

REFERENCES

1. Bouhafs, F., Mackay, M., Merabti, M. (2012). “Links to the Future.” IEEE Power and Energy Magazine 1540-7977/12, pp. 24-32
2. Yan, Y., Qian, Y., Sharif, H., Tipper, D. (2012) “A Survey on Smart Grid Communication Infrastructures: Motivations, Requirements, and Challenges.” IEEE Communications Surveys & Tutorials, Vol. 15 #1, pp. 5-20
3. Yan, Y., Qian, Y., Sharif, H., Tipper, D. (2012) “A Survey on Cyber Security for Smart Grid Communications.” IEEE Communications Surveys & Tutorials, Accepted for Publication 1553-877X/12
4. Liu, Y., Ning, P., Reiter, M. (2009) “False data injection attacks against state estimation in electric power grids.” In Proc. ACM Conference on Computer and Communications Security (CCS 09)
5. Baumeister, T. (2011) “Adapting PKI for the Smart Grid.” IEEE SmartGridComm, 978-1-4577-1702-4/11
6. NISTIR 7628 Volume 1 (2010) “Guidelines for Smart Grid Cyber Security: Vol. 1, Smart Grid Cyber Security Strategy, Architecture, and High-Level Requirements.”
7. Naval Sea Systems Command (2007) “Next Generation Integrated Power System Technology Development Roadmap.” Ser 05D/349 of 30 Nov 2007
8. Doerry, N., CAPT USN, "Next Generation Integrated Power Systems for the Future Fleet," Presented at the Corbin A. McNeill Symposium, United States Naval Academy, Annapolis, MD, March 30, 2009

9. Doerry, N., Scherer, T., Cohen, J., Guertin, N., "Open Architecture Machinery Control System ," Presented at ASNE Intelligent Ships Symposium 2011, May 25-26, 2011, Philadelphia, PA.

   Also Published in ASNE Naval Engineers Journal, Mar 2012, Vol 124 No. 1, pp. 101-114.

10. Hankerson, D., Menezes, A., Vanstone, S. (2004) Guide to Elliptic Curve Cryptography, ©2004, Springer-Verlag New York, Inc.

11. Wikipedia: RSA Algorithm (http://en.wikipedia.org/wiki/RSA_%28algorithm%29) Accessed: 18 June, 2014

12. Wikipedia: Digital Signature Algorithm (http://en.wikipedia.org/wiki/Digital_Signature_Algorithm) Accessed: 18 June, 2014

13. Wikipedia: Key Size (http://en.wikipedia.org/wiki/Key_size) Accessed: 18 June 2014

14. Miller, V.S. (1985). "Use of elliptic curves in cryptography." Advances in Cryptology Proc. Crypto '85, LNCS 218, H.C. Williams, Ed., Springer-Verlag, pp. 417-426

15. Koblitz, N. (1987). "Elliptic curve cryptosystems." Mathematics of Computation, Vol. 48, No. 177, p. 279-287

16. ANSI X9.62 (1999). "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)

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

18. IEEE 1363-2000 (2000) "Standard Specifications for Public-Key Cryptography."

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

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

21. NIST FIPS Pub 186-4 (2013). "Digital Signature Standard." (http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf) Accessed: 28th July 2014

22. Standards For Efficient Cryptography Group (SECG) (http://www.secg.org) Accessed: 9th July 2012

23. SECG SEC 1 Version 2.0 (2009). "SEC 1: Elliptic Curve Cryptography." (http://www.secg.org) Accessed: 9th July 2012

24. SECG SEC 2 Version 2.0 (2010). "SEC 2: Recommended Elliptic Curve Domain Parameters." (http://www.secg.org) Accessed: 9th July 2012

25. Certicom (http://www.certicom.com) Accessed: 9th July 2012

26. McGrew, D. (IETF) (2009-Working Draft). "Fundamental Elliptic Curve Cryptography Algorithms." (http://tools.ietf.org/html/draft-mcgrew-fundamental-ecc-01) Accessed: 15th May 2012

27. Koblitz, N. (2010) "My Last 24 Years in Crypto: A Few Good Judgments and Many Bad Ones" (http://2010.eccworkshop.org/slides/Koblitz.pdf) Accessed: 18 June, 2014

28. Wikipedia: NIST hash function competition (http://en.wikipedia.org/wiki/NIST_hash_function_competition) Accessed: 18 June 2014

29. Jansma, N., Arrendondo, B. (2004). "Performance Comparison of Elliptic Curve and RSA Digital Signatures." (http://nicj.net/files/performance_comparison_of_elliptic_curve_and_rsa_digital_signatures.pdf) Accessed 12th July 2012

30. Oracle Labs: FAQ (related to ECC) (https://labs.oracle.com/projects/crypto/FrequenlyAskedQuestions.html) Accessed: 12th July 2012

31. SECG X.509 WG Working Group Draft Version 0.2 (1999). "ECC in X.509."

32. Antipa, A., Brown, D., Gallant, R., Lambert, R., Struik, R., Vanstone, S. (2005). "Accelerated Verification of ECDSA Signatures." (http://www.mathnet.or.kr/mathnet/preprint_file/cacr/2005/cacr2005-28.pdf) Accessed: 14th July 2012

APPENDIX A. C SOURCE CODE

A.1 MsgGen

```
/***************************************************************************
File:                    msggen.c
Author:                  kfischer
Creation date:           05/11/2013 - 19:52
POU name:                MsgGen
***************************************************************************/

---autocode---

//START OF CUSTOM CODE HEADERS
#include <debug.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/sha.h>              //for SHA512
//END OF CUSTOM CODE HEADERS

---autocode---


   //START OF CUSTOM CODE

   //--Declarations
    /*debug variables*/
       #if ISaDEBUG
              FILE *fp;
       #endif

       /*string and hash variables*/
    #define       P_PTR_HASH_I_DATA       (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA-
>hash_i))+2)
       #define     P_PTR_STRING_I_DATA     (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA-
>string))+2)
       unsigned char digest[SHA512_DIGEST_LENGTH];
       char mdString[SHA512_DIGEST_LENGTH*2+1];
       char *hash_Q;
       const char *string;
       char *tstring;
       char str[STRLEN];
       char tstr[STRLEN/4];
       int i;

       /*base64 variables*/

   //--Logic

       /*STEP 0 - Initialization*/
       P_STATUS = 0;

       memset(digest,0,SHA512_DIGEST_LENGTH);
       memset(mdString,0,SHA512_DIGEST_LENGTH*2+1);
       memset(str,0,STRLEN);
       memset(tstr,0,STRLEN/4);
```

```
hash_Q = mdString;
string = str;
tstring = tstr;

/*STEP 1 - build string for hashing*/
strcpy(str, "DATE=");
sprintf(tstring,"%u,",P_DATE);
strcat(str,tstring);

strcat(str, "TIME=");
sprintf(tstring,"%u,",P_TIME);
strcat(str,tstring);

strcat(str, "HASH_I=");
sprintf(tstring,"%s,",P_PTR_HASH_I_DATA);
strcat(str,tstring);

strcat(str,"P_B1=");
sprintf(tstring,"%s,", (P_B1)?"TRUE":"FALSE");
strcat(str,tstring);

strcat(str,"P_B2=");
sprintf(tstring,"%s,", (P_B2)?"TRUE":"FALSE");
strcat(str,tstring);

strcat(str,"P_B3=");
sprintf(tstring,"%s,", (P_B3)?"TRUE":"FALSE");
strcat(str,tstring);

strcat(str,"P_B4=");
sprintf(tstring,"%s,", (P_B4)?"TRUE":"FALSE");
strcat(str,tstring);

strcat(str,"P_B5=");
sprintf(tstring,"%s,", (P_B5)?"TRUE":"FALSE");
strcat(str,tstring);

strcat(str,"P_B6=");
sprintf(tstring,"%s,", (P_B6)?"TRUE":"FALSE");
strcat(str,tstring);

strcat(str,"P_B7=");
sprintf(tstring,"%s,", (P_B7)?"TRUE":"FALSE");
strcat(str,tstring);

strcat(str,"P_B8=");
sprintf(tstring,"%s,", (P_B8)?"TRUE":"FALSE");
strcat(str,tstring);

strcat(str,"P_I1=");
sprintf(tstring,"%d,",P_I1);
strcat(str,tstring);

strcat(str,"P_I2=");
sprintf(tstring,"%d,",P_I2);
strcat(str,tstring);

strcat(str,"P_I3=");
```

```
sprintf(tstring,"%d,",P_I3);
strcat(str,tstring);

strcat(str,"P_I4=");
sprintf(tstring,"%d,",P_I4);
strcat(str,tstring);

strcat(str,"P_R1=");
sprintf(tstring,"%f,",P_R1);
strcat(str,tstring);

strcat(str,"P_R2=");
sprintf(tstring,"%f,",P_R2);
strcat(str,tstring);

strcat(str,"P_R3=");
sprintf(tstring,"%f,",P_R3);
strcat(str,tstring);

strcat(str,"P_R4=");
sprintf(tstring,"%f,",P_R4);
strcat(str,tstring);

strcat(str, "P_STRING = ");
sprintf(tstring,"%s,",P_PTR_STRING_I_DATA);
strcat(str,tstring);

if (NULL == str)
{
       P_STATUS = -1; /*failed to create string*/
}
else
{
       /*STEP 2 - hash string*/
       SHA512_CTX ctx;
       SHA512_Init(&ctx);
       SHA512_Update(&ctx, string, strlen(string));
       SHA512_Final(digest, &ctx);

       for(i = 0; i < SHA512_DIGEST_LENGTH; i++)
       {
              sprintf(&mdString[i*2], "%02x", (unsigned int)digest[i]);
       }

       if (NULL == mdString)
       {
              P_STATUS = -2; /*failed to create hash*/
       }
       else
       {
              /*STEP 3 - output result*/
              strcpy(P_HASH_Q,hash_Q);
              HASH_Q_MAXLEN = 128;
              HASH_Q_CURLEN = 128;
              P_STATUS = 1;
       }
}
```

```
       /*debug logic*/
       #if ISaDEBUG
          fp = fopen("debug-message.txt", "w");
          if (fp == NULL) {
                 P_STATUS = 666;
                 exit(0);
          }

          fprintf(fp, "string = %s\n", string);
          fprintf(fp, "digest = %s\n", digest);
          fprintf(fp, "mdString = %s\n", mdString);


          fprintf(fp, "\n\nDATE = %u\n", P_DATE);
          fprintf(fp, "HASH_I = %s\n", P_PTR_HASH_I_DATA);
          fprintf(fp, "BOOL1 = %s\n", (P_B1)?"TRUE":"FALSE");
          fprintf(fp, "BOOL2 = %s\n", (P_B2)?"TRUE":"FALSE");
          fprintf(fp, "BOOL3 = %s\n", (P_B3)?"TRUE":"FALSE");
          fprintf(fp, "BOOL4 = %s\n", (P_B4)?"TRUE":"FALSE");
          fprintf(fp, "BOOL5 = %s\n", (P_B5)?"TRUE":"FALSE");
          fprintf(fp, "BOOL6 = %s\n", (P_B6)?"TRUE":"FALSE");
          fprintf(fp, "BOOL7 = %s\n", (P_B7)?"TRUE":"FALSE");
          fprintf(fp, "BOOL8 = %s\n", (P_B8)?"TRUE":"FALSE");
          fprintf(fp, "INT1  = %i\n", P_I1);
          fprintf(fp, "INT2  = %i\n", P_I2);
          fprintf(fp, "INT3  = %i\n", P_I3);
          fprintf(fp, "INT4  = %i\n", P_I4);
          fprintf(fp, "REAL1 = %f\n", P_R1);
          fprintf(fp, "REAL2 = %f\n", P_R2);
          fprintf(fp, "REAL3 = %f\n", P_R3);
          fprintf(fp, "REAL4 = %f\n", P_R4);

          fclose(fp);
       #endif


   //END OF CUSTOM CODE


}

/* eof *********************************************************************/
```

A.2 KeyGen

```
/**************************************************************************
File:               keygen.c
Author:             kfischer
Creation date:      15/06/2013 - 20:02
POU name:           KeyGen
***************************************************************************/

---autocode---

//START OF CUSTOM CODE HEADERS
#include <debug.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ec.h>         //  for  EC_GROUP_new_by_curve_name,  EC_GROUP_free,
EC_KEY_new, EC_KEY_set_group, EC_KEY_generate_key, EC_KEY_free
#include <openssl/obj_mac.h> // for NID_secp384r1
#include <openssl/bn.h>             // for BIGNUM conversions
#include <openssl/bio.h>
#include <openssl/evp.h>     // for base64 conversions
#include <openssl/buffer.h>
#include <math.h>
#include <base64.h>                 // for base64 conversions

//END OF CUSTOM CODE HEADERS
---autocode---

   //START OF CUSTOM CODE

   //--Declarations


    /*debug variables*/
       #if ISaDEBUG
              FILE *fp;
              FILE *fpa;
              FILE *fpb;
              int i,j;
       #endif

   /*status variables*/
   const int set_group_success = 1;
   const int gen_success = 1;
   int set_group_status;
   int gen_status;

   /*key and base 64 variables*/

   const BIGNUM *PrivateKeyBN;
   BIGNUM *PublicKeyBN;
   const EC_POINT *PublicKeyPoint;
   EC_KEY *eckey;
   EC_GROUP *ecgroup;
   unsigned char *PublicKeyBinaryUnsigned, PublicKeyBinaryUnsignedArray[250];
   unsigned char *PrivateKeyBinaryUnsigned, PrivateKeyBinaryUnsignedArray[250];

   int encodedSize;
   char *buffer, BufferArray[250];
   unsigned char *PrivateKeyBase64, PrivateKeyBase64Array[250];

   int encodedSize_b;
   char *buffer_b, BufferArray_b[250];
   unsigned char *PublicKeyBase64, PublicKeyBase64Array[250];


   //--Logic

   /*STEP 0 - Initialization*/
       P_STATUS = 0;

       memset(PrivateKeyBinaryUnsignedArray,0,250);
       PrivateKeyBinaryUnsigned = PrivateKeyBinaryUnsignedArray;

       memset(PublicKeyBinaryUnsignedArray,0,250);
       PublicKeyBinaryUnsigned = PublicKeyBinaryUnsignedArray;

       memset(PrivateKeyBase64Array,0,250);
       PrivateKeyBase64 = PrivateKeyBase64Array;

       memset(PublicKeyBase64Array,0,250);
       PublicKeyBase64 = PublicKeyBase64Array;

       memset(BufferArray,0,250);
       buffer = BufferArray;
       memset(BufferArray_b,0,250);
       buffer_b = BufferArray_b;

       /*STEP 1 - create key object*/
    eckey=EC_KEY_new();
    if (NULL == eckey)
    {
        P_STATUS = -1;  /*failed to create key object*/
    }
    else
    {
              /*STEP 2 - create EC_GROUP object*/
        //ecgroup = EC_GROUP_new_by_curve_name(NID_secp384r1);     --worked for 384
ECDSA, will want to come back and add an option to turn this on and off
              ecgroup = EC_GROUP_new_by_curve_name(NID_secp521r1);
        if (NULL == ecgroup)
        {
            P_STATUS = -2; /*failed to create new EC Group*/
        }
        else
        {
                     /*STEP 3 - associate key with group*/
            set_group_status = EC_KEY_set_group(eckey,ecgroup);
            if (set_group_success != set_group_status)
            {
                            P_STATUS = -3; /*failed to associate key with group*/
            }
            else
                     {
                            /*STEP 4 - create PRIVATE and PUBLIC keys*/
                gen_status = EC_KEY_generate_key(eckey);
                if (gen_success != gen_status)
                {
                    P_STATUS = -4; /*failed to generate EC Key*/
                }
                else
                            {
                                   /*STEP 5 - extract PRIVATE keys*/
                                   PrivateKeyBN = EC_KEY_get0_private_key(eckey);
                                   if (PrivateKeyBN == NULL)
                                   {
                                          P_STATUS   =   -5;   /*failed   to   extract
PRIVATE key*/
                                   }
                                   else
                                   {
                                          /*STEP 6 - extract PUBLIC KEY*/
                                          PublicKeyPoint                         =
EC_KEY_get0_public_key(eckey);
                                          if (PublicKeyPoint == NULL)
                                          {
                                                 P_STATUS = -6; /*failed to extract
PUBLIC key*/
                                          }
                                          else
                                          {
                                                 /*STEP  7a  -  Private  Key  convert
from BN to binary and then encode as base64*/


       BN_bn2bin(PrivateKeyBN,PrivateKeyBinaryUnsigned);
                                                 encodedSize                      =
EVP_EncodeBlock(PrivateKeyBase64,PrivateKeyBinaryUnsigned,89);

                                                 /*STEP  7b  -  Public  Key  convert
from Point to BN to binary and then encode as base64*/

                                                 PublicKeyBN = BN_new();

       EC_POINT_point2bn(ecgroup,PublicKeyPoint,POINT_CONVERSION_UNCOMPRESSED,Publ
icKeyBN,NULL);

       BN_bn2bin(PublicKeyBN,PublicKeyBinaryUnsigned);
                                                 encodedSize_b                    =
EVP_EncodeBlock(PublicKeyBase64,PublicKeyBinaryUnsigned,179);



                                                 /*STEP 7c - set outputs*/

                                                 buffer = (char*) PrivateKeyBase64;
                                                 buffer_b         =         (char*)
PublicKeyBase64;
```

```
                                                 strcpy(P_PRIVATE_KEY_BUFADD,
buffer);
                                                 PRIVATE_KEY_MAXLEN = 250;
                                                 PRIVATE_KEY_CURLEN = encodedSize;

                                                 strcpy(P_PUBLIC_KEY_BUFADD,
buffer_b);
                                                 PUBLIC_KEY_MAXLEN = 250;
                                                 PUBLIC_KEY_CURLEN = encodedSize_b;

                                                 P_STATUS = 1; /*success*/
/*debug logic*/
#if ISaDEBUG
       fpb = fopen("debug-keygen-BIGNUM.txt", "w");
       fprintf(fpb, "PublicKeyBigNumber = ");
       BN_print_fp(fpb, PublicKeyBN);
       fprintf(fpb, " \n");
       fprintf(fpb, " \n");
       fprintf(fpb, "PrivateKeyBigNumber = ");
       BN_print_fp(fpb, PrivateKeyBN);
       fclose(fpb);

       fp = fopen("debug-keygen-Private.txt", "w");
       fprintf(fp, "hello world\n");
       fprintf(fp,"encodedSize = %i\n", encodedSize);
       fprintf(fp,"encodedSize_b = %i\n", encodedSize_b);
       fprintf(fp,"array dump = ");
       for (i = 0; i<250; i++)
       {
              fprintf(fp, "%c", PrivateKeyBinaryUnsignedArray[i]);
       }
       fprintf(fp, " \n");
       fprintf(fp,"PrivateKeyBinaryUnsigned = %s\n", PrivateKeyBinaryUnsigned);
       fprintf(fp,"PrivateKeyBase64 = %s\n", PrivateKeyBase64);
       fprintf(fp,"buffer = %s\n", buffer);
       fclose(fp);
#endif
                                          }
                                   }
                            }
                     }
              }
       }
   //END OF CUSTOM CODE

}

/* eof ********************************************************************/
```

A.3 KeyVerify

```
/***************************************************************************
File:               keyverify.c
Author:             kfischer
Creation date:      15/06/2013 - 13:53
POU name:           KeyVerify
***************************************************************************/

---autocode---

//START OF CUSTOM CODE HEADERS
#include <debug.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ec.h>         //  for  EC_GROUP_new_by_curve_name,  EC_GROUP_free,
EC_KEY_new, EC_KEY_set_group, EC_KEY_generate_key, EC_KEY_free
#include <openssl/ecdsa.h>   // for ECDSA_do_sign, ECDSA_do_verify
#include <openssl/obj_mac.h> // for NID_secp384r1
#include <openssl/bn.h>            // for BIGNUM conversions
#include <openssl/bio.h>
#include <openssl/evp.h>
#include <openssl/buffer.h>
#include <math.h>
#include <base64.h>                // for base64 conversions
//END OF CUSTOM CODE HEADERS

---autocode---

   //START OF CUSTOM CODE


   //--Declarations

   /*debug variables*/
       #if ISaDEBUG
       FILE *fp;
       FILE *fpa;
       FILE *fpb;
       int i, j;
       #endif

   /*status variables*/
       #define   P_PTR_PUBLIC_KEY_DATA    (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA-
>public_key))+2)
       #define   P_PTR_PRIVATE_KEY_DATA   (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA-
>private_key))+2)

       const int set_group_success = 1;
       const int gen_success = 1;
       const int set_public_success = 1;
       const int set_private_success = 1;
       const int verify_success = 1;

       int set_group_status;
       int set_public_status;
       int set_private_status;
       int verify_status;

       unsigned char hash[] = "c7fbca202a95a570285e3d700eb04ca2";

   /*base64 variables*/
       EC_KEY *eckey;
       EC_GROUP *ecgroup;
       EC_POINT *PublicKeyPoint;
       BIGNUM *PrivateKeyBN;
       BIGNUM *PublicKeyBN;
       ECDSA_SIG *signature;

       unsigned char *buffer;
       unsigned char *buffer_b;

       unsigned char *PublicKeyBinaryUnsigned, PublicKeyBinaryUnsignedArray[250];
       unsigned                   char                       *PrivateKeyBinaryUnsigned,
PrivateKeyBinaryUnsignedArray[250];

       const unsigned char *PublicKeyBinaryConst;
       const unsigned char *PrivateKeyBinaryConst;



   //--Logic
       P_STATUS = 0;

       memset(PrivateKeyBinaryUnsignedArray,0,250);
       PrivateKeyBinaryUnsigned = PrivateKeyBinaryUnsignedArray;

       memset(PublicKeyBinaryUnsignedArray,0,250);
       PublicKeyBinaryUnsigned = PublicKeyBinaryUnsignedArray;


       /*STEP 1 - create key object*/
       eckey=EC_KEY_new();
    if (NULL == eckey)
    {
        P_STATUS = -1;  /*failed to create key object*/
    }
    else
    {
              /*STEP 2 - create EC_GROUP object*/
              ecgroup = EC_GROUP_new_by_curve_name(NID_secp521r1);
        if (NULL == ecgroup)
        {
            P_STATUS = -2; /*failed to create new EC Group*/
        }
        else
        {
                     /*STEP 3 - associate key with group*/
            set_group_status = EC_KEY_set_group(eckey,ecgroup);
            if (set_group_success != set_group_status)
            {
                            P_STATUS = -3; /*failed to associate key with group*/
            }
            else
                     {
                            /*STEP 4 - set public key*/
                            PublicKeyPoint = EC_POINT_new(ecgroup);
                            PublicKeyBN = BN_new();

                            buffer_b = (unsigned char*) P_PTR_PUBLIC_KEY_DATA;

                            EVP_DecodeBlock(PublicKeyBinaryUnsigned,buffer_b,240);

                            PublicKeyBinaryConst    =    (const    unsigned    char*)
PublicKeyBinaryUnsigned;

                            BN_bin2bn(PublicKeyBinaryConst,133,PublicKeyBN);


       EC_POINT_bn2point(ecgroup,PublicKeyBN,PublicKeyPoint,NULL);


                            set_public_status                                 =
EC_KEY_set_public_key(eckey,PublicKeyPoint);
                            if (set_public_success != set_public_status)
                            {
                                   P_STATUS = -4; /*failed to set public key*/
                            }
                            else
                            {
                                   /*STEP 5 - set private key*/
                                   PrivateKeyBN = BN_new();
                                   buffer         =         (unsigned         char*)
P_PTR_PRIVATE_KEY_DATA;

       EVP_DecodeBlock(PrivateKeyBinaryUnsigned,buffer,120);
                                   PrivateKeyBinaryConst  =  (const  unsigned  char*)
PrivateKeyBinaryUnsigned;

/*521 ECDSA is unique due to the extra bit resulting in a length of 65.125 bytes,
which  means  that  when  converting  from  bin  to  bn  we  will  alternatively  be
converting 66 or 65 bytes depending on if the extra bit is a 1 or not.      The
presence  of  the  extra  bit  is  encoded  in  the  in  the  first  base64  characters.  The
byte   0000 0001 will be broken in base64 to a pair of 0000000 and 1XXXXXX, which
will  result  in  the  key  having  a  leading  A  character  representing  the  0000000.
However, the byte 0000 0000 will not be encoded at all, which will result in the
key  not  having  a  leading  A.   Detection  of  the  leading  A  is  therefore  critical  to
properly convert the bin to BN.*/

                                   if (P_PTR_PRIVATE_KEY_DATA[0] == 'A')
                                   {

       BN_bin2bn(PrivateKeyBinaryConst,66,PrivateKeyBN);
                                   }
                                   else
                                   {

       BN_bin2bn(PrivateKeyBinaryConst,65,PrivateKeyBN);
                                   }
```

```
                    set_private_status =
EC_KEY_set_private_key(eckey,PrivateKeyBN);
                    if (set_private_success != set_private_status)
                    {
                        P_STATUS = -5; /*failed to set private
key*/
                    }
                    else
                    {
                        /*STEP 6 - create signature on dummy
hash*/
                        signature = ECDSA_do_sign(hash,32,eckey);
                        if (NULL == signature)
                        {
                            P_STATUS = -6; /*failed to
generate signature*/
                        }
                        else
                        {
                            /*STEP 7 - verify signature*/
                            verify_status =
ECDSA_do_verify(hash,32,signature,eckey);
                            if (verify_success !=
verify_status)
                            {
                                P_STATUS = -7;
/*verification failed*/
                            }
                            else
                            {
                                P_STATUS = 1; /*keys
verified*/
                            }
                        }
                    }
                }
            }
        }
    }

    /*CLEANUP - Release memory structures to prevent memory leaks*/
    EC_KEY_free(eckey);                     /*frees EC_KEY memory allocation*/
    EC_GROUP_clear_free(ecgroup);   /*destroys any sensitive data in the group
and then frees the memory*/
    EC_POINT_free(PublicKeyPoint);  /*frees EC_POINT memory allocation*/
    BN_clear_free(PrivateKeyBN);    /*overwrites the BN before returning memory
to the system*/
    BN_clear_free(PublicKeyBN);
    ECDSA_SIG_free(signature);        /*frees ECDSA_SIG memory allocation*/


/*debug logic*/
#if ISaDEBUG
    fpa = fopen("debug-keyverify-public.txt", "w");
    fprintf(fpa, "hello world \n");
    fprintf(fpa, "P_PTR_PUBLIC_KEY_DATA = %s\n", P_PTR_PUBLIC_KEY_DATA);
    fprintf(fpa, " \n");
    fprintf(fpa, " \n");
```

```
       fprintf(fpa, "Public Key Array dump = ");
       for (j = 0; j<250; j++)
       {
              fprintf(fpa, "%c", PublicKeyBinaryUnsignedArray[j]);
       }
       fprintf(fpa, " \n");
       fprintf(fpa, " \n");
       fclose(fpa);

       fp = fopen("debug-keyverify-private.txt", "w");
       fprintf(fp, "hello world\n");
       fprintf(fp, "P_PTR_PRIVATE_KEY_DATA = %s\n", P_PTR_PRIVATE_KEY_DATA);
       fprintf(fp, "P_PTR_PRIVATE_KEY_DATA[0] = %c\n", P_PTR_PRIVATE_KEY_DATA[0]);
       fprintf(fp, "Private Key Array dump = ");
       for (i = 0; i<250; i++)
       {
              fprintf(fp, "%c", PrivateKeyBinaryUnsignedArray[i]);
       }
       fprintf(fp, " \n");
       fclose(fp);


       fpb = fopen("debug-keyverify-BIGNUM.txt", "w");
       fprintf(fpb, "PublicKeyBigNumber = ");
       BN_print_fp(fpb, PublicKeyBN);
       fprintf(fpb, " \n");
       fprintf(fpb, " \n");
       fprintf(fpb, "PrivateKeyBigNumber = ");
       BN_print_fp(fpb, PrivateKeyBN);
       fprintf(fpb, " \n");
       fprintf(fpb, " \n");
       fprintf(fpb, "bnlen = %i\n", bnlen);
       fclose(fpb);
#endif




   //END OF CUSTOM CODE


}

/* eof **********************************************************************/
```

A.4 SigGen

```
/**************************************************************************
File:               siggen.c
Author:             kfischer
Creation date:      10/07/2013 - 22:27
POU name:           SigGen
***************************************************************************/

---autocode---

//START OF CUSTOM CODE HEADERS
#include <debug.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ec.h>         // for EC_GROUP_new_by_curve_name, EC_GROUP_free,
EC_KEY_new, EC_KEY_set_group, EC_KEY_generate_key, EC_KEY_free
#include <openssl/ecdsa.h>   // for ECDSA_do_sign, ECDSA_do_verify
#include <openssl/obj_mac.h> // for NID_secp384r1
#include <openssl/bn.h>            // for BIGNUM conversions
#include <openssl/bio.h>
#include <openssl/evp.h>
#include <openssl/buffer.h>
#include <math.h>
#include <base64.h>                // for base64 conversions
//END OF CUSTOM CODE HEADERS

---autocode---

   //START OF CUSTOM CODE


   //--Declarations

   /*debug variables*/
       #if ISaDEBUG
       FILE *fp;
       int i;
       #endif

   /*status variables*/

       #define   P_PTR_PRIVATE_KEY_DATA   (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA-
>private_key))+2)
       #define P_PTR_HASH_DATA (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA->hash))+2)

       const int set_group_success = 1;
       const int gen_success = 1;
       const int set_public_success = 1;
       const int set_private_success = 1;
       const int verify_success = 1;

       int set_group_status;
       int set_private_status;
```

```
    unsigned char hash[128];


/*EC Variables*/
    EC_KEY *eckey;
    EC_GROUP *ecgroup;
    ECDSA_SIG *signature;

/*base64 variables for decomposing private key*/
    BIGNUM *PrivateKeyBN;
    unsigned char *buffer;
 unsigned char *PrivateKeyBinaryUnsigned, PrivateKeyBinaryUnsignedArray[250];
    const unsigned char *PrivateKeyBinaryConst;

/*base64 variables for encoding SIG_R and SIG_S*/

    int SIG_R_EncodedSize;
    unsigned char *SIG_R_BinaryUnsigned, SIG_R_BinaryUnsignedArray[250];
 unsigned char *SIG_R_Base64, SIG_R_Base64Array[250];
    int SIG_S_EncodedSize;
    unsigned char *SIG_S_BinaryUnsigned, SIG_S_BinaryUnsignedArray[250];
 unsigned char *SIG_S_Base64, SIG_S_Base64Array[250];

    char *SIG_R_Buffer, SIG_R_BufferArray[250];
    char *SIG_S_Buffer, SIG_S_BufferArray[250];

//--Logic

    /*STEP 0 - Initialization*/
    P_STATUS = 0;

    memcpy(hash,P_PTR_HASH_DATA,128);

    memset(PrivateKeyBinaryUnsignedArray,0,250);
    PrivateKeyBinaryUnsigned = PrivateKeyBinaryUnsignedArray;

    memset(SIG_R_BinaryUnsignedArray,0,250);
    SIG_R_BinaryUnsigned = SIG_R_BinaryUnsignedArray;
    memset(SIG_R_Base64Array,0,250);
    SIG_R_Base64 = SIG_R_Base64Array;
    memset(SIG_R_BufferArray,0,250);
    SIG_R_Buffer = SIG_R_BufferArray;

    memset(SIG_S_BinaryUnsignedArray,0,250);
    SIG_S_BinaryUnsigned = SIG_S_BinaryUnsignedArray;
    memset(SIG_S_Base64Array,0,250);
    SIG_S_Base64 = SIG_S_Base64Array;
    memset(SIG_S_BufferArray,0,250);
    SIG_S_Buffer = SIG_S_BufferArray;

    /*STEP 1 - create key object*/
    eckey=EC_KEY_new();
 if (NULL == eckey)
 {
     P_STATUS = -1;  /*failed to create key object*/
 }
 else
```

```
    {
              /*STEP 2 - create EC_GROUP object*/
              ecgroup = EC_GROUP_new_by_curve_name(NID_secp521r1);
        if (NULL == ecgroup)
        {
            P_STATUS = -2; /*failed to create new EC Group*/
        }
        else
        {
                     /*STEP 3 - associate key with group*/
            set_group_status = EC_KEY_set_group(eckey,ecgroup);
            if (set_group_success != set_group_status)
            {
                            P_STATUS = -3; /*failed to associate key with group*/
            }
            else
                     {
                            /*STEP 4 - set private key*/
                            PrivateKeyBN = BN_new();

                            buffer = (unsigned char*) P_PTR_PRIVATE_KEY_DATA;
                            EVP_DecodeBlock(PrivateKeyBinaryUnsigned,buffer,120);

                            PrivateKeyBinaryConst = (const unsigned char*)
PrivateKeyBinaryUnsigned;

/*521 ECDSA is unique due to the extra bit resulting in a length of 65.125 bytes,
which means that when converting from bin to bn we will alternatively be
converting 66 or 65 bytes depending on if the extra bit is a 1 or not. The
presence of the extra bit is encoded in the in the first base64 characters.
The byte 0000 0001 will be broken in base64 to a pair of 0000000 and 1XXXXXX,
which will result in the key having a leading A character representing the
0000000. However, the byte 0000 0000 will not be encoded at all, which will
result in the key not having a leading A. Detection of the leading A is therefore
critical to properly convert the bin to BN.*/

                            if (P_PTR_PRIVATE_KEY_DATA[0] == 'A')
                            {

       BN_bin2bn(PrivateKeyBinaryConst,66,PrivateKeyBN);
                            }
                            else
                            {

       BN_bin2bn(PrivateKeyBinaryConst,65,PrivateKeyBN);
                            }

                            set_private_status =
EC_KEY_set_private_key(eckey,PrivateKeyBN);
                            if (set_private_success != set_private_status)
                            {
                                   P_STATUS = -4; /*failed to set private key*/
                            }
                            else
                            {
                                   /*STEP 5 - create signature on dummy hash*/
                                   signature = ECDSA_do_sign(hash,128,eckey);
                                   if (NULL == signature)
                                   {
                                          P_STATUS   =   -5;   /*failed   to   generate
signature*/
                                   }
                                   else
                                   {
                                          /*STEP 6 - base64 encode SIG_R and SIG_S
and output result*/

                                          BN_bn2bin(signature-
>r,SIG_R_BinaryUnsigned);
                                          SIG_R_EncodedSize                          =
EVP_EncodeBlock(SIG_R_Base64,SIG_R_BinaryUnsigned,89);

                                          BN_bn2bin(signature-
>s,SIG_S_BinaryUnsigned);
                                          SIG_S_EncodedSize                          =
EVP_EncodeBlock(SIG_S_Base64,SIG_S_BinaryUnsigned,89);

                                          SIG_R_Buffer = (char *) SIG_R_Base64;
                                          SIG_S_Buffer = (char *) SIG_S_Base64;

                                          strcpy(P_SIG_R_BUFADD,SIG_R_Buffer);
                                          SIG_R_MAXLEN = 250;
                                          SIG_R_CURLEN = SIG_R_EncodedSize;

                                          strcpy(P_SIG_S_BUFADD,SIG_S_Buffer);
                                          SIG_S_MAXLEN = 250;
                                          SIG_S_CURLEN = SIG_S_EncodedSize;

                                          P_STATUS = 1; /*success*/
                                   }
                            }
                     }
              }
       }

       ///*CLEANUP - Release memory structures to prevent memory leaks*/
       EC_KEY_free(eckey);                      /*frees EC_KEY memory allocation*/
       EC_GROUP_clear_free(ecgroup);    /*destroys any sensitive data in the group
and then frees the memory*/
       BN_clear_free(PrivateKeyBN);     /*overwrites the BN before returning memory
to the system*/
       ECDSA_SIG_free(signature);         /*frees ECDSA_SIG memory allocation*/


/*debug logic*/
#if ISaDEBUG
       fp = fopen("debug-siggen.txt", "w");
       fprintf(fp, "hello world\n");
       fprintf(fp, " \n");
       fprintf(fp, "PrivateKeyBigNumber = ");
       BN_print_fp(fp, PrivateKeyBN);
       fprintf(fp, " \n");
       fprintf(fp, "signature->r = ");
       BN_print_fp(fp, signature->r);
       fprintf(fp, " \n");
       fprintf(fp, "signature->s = ");
       BN_print_fp(fp, signature->s);
       fprintf(fp, " \n");
       fclose(fp);

#endif

   //END OF CUSTOM CODE

}

/* eof ********************************************************************/
```

A.5 SigVerify

```
/**************************************************************************
File:               sigverify.c
Author:             kfischer
Creation date:      12/07/2013 - 18:52
POU name:           SigVerify
***************************************************************************/

---autocode---

//START OF CUSTOM CODE HEADERS
#include <debug.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ec.h>         // for EC_GROUP_new_by_curve_name, EC_GROUP_free,
EC_KEY_new, EC_KEY_set_group, EC_KEY_generate_key, EC_KEY_free
#include <openssl/ecdsa.h>   // for ECDSA_do_sign, ECDSA_do_verify
#include <openssl/obj_mac.h> // for NID_secp384r1
#include <openssl/bn.h>            // for BIGNUM conversions
#include <openssl/bio.h>
#include <openssl/evp.h>
#include <openssl/buffer.h>
#include <math.h>
#include <base64.h>                // for base64 conversions
//END OF CUSTOM CODE HEADERS

---autocode---

   //START OF CUSTOM CODE


   //--Declarations

   /*debug variables*/
       #if ISaDEBUG
       FILE *fp;
       int i;
       #endif

       /*status variables*/
       #define   P_PTR_PUBLIC_KEY_DATA    (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA-
>public_key))+2)
       #define P_PTR_HASH_DATA (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA->hash))+2)
       #define     P_PTR_SIG_R_DATA      (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA-
>sig_r))+2)
       #define     P_PTR_SIG_S_DATA      (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA-
>sig_s))+2)

       const int set_group_success = 1;
       const int gen_success = 1;
       const int set_public_success = 1;
       const int set_private_success = 1;
       const int verify_success = 1;
```

```
    int set_group_status;
    int set_public_status;
    int verify_status;

    unsigned char hash[128];

 /*EC Variables*/
    EC_KEY *eckey;
    EC_GROUP *ecgroup;
    EC_POINT *PublicKeyPoint;
    ECDSA_SIG *signature;


 /*base64 variables for decoding public key*/
    BIGNUM *PublicKeyBN;
    unsigned char *buffer;
 unsigned char *PublicKeyBinaryUnsigned, PublicKeyBinaryUnsignedArray[250];
    const unsigned char *PublicKeyBinaryConst;

    /*base64 variables for decoding SIG_R key*/
    BIGNUM *SIG_R_BN;
    unsigned char *SIG_R_buffer;
 unsigned char *SIG_R_BinaryUnsigned, SIG_R_BinaryUnsignedArray[250];
    const unsigned char *SIG_R_BinaryConst;

    /*base64 variables for decoding SIG_S key*/
    BIGNUM *SIG_S_BN;
    unsigned char *SIG_S_buffer;
 unsigned char *SIG_S_BinaryUnsigned, SIG_S_BinaryUnsignedArray[250];
    const unsigned char *SIG_S_BinaryConst;

//--Logic
    P_STATUS = 0;

    memcpy(hash,P_PTR_HASH_DATA,128);

    memset(PublicKeyBinaryUnsignedArray,0,250);
    PublicKeyBinaryUnsigned = PublicKeyBinaryUnsignedArray;

    memset(SIG_R_BinaryUnsignedArray,0,250);
    SIG_R_BinaryUnsigned = SIG_R_BinaryUnsignedArray;

    memset(SIG_S_BinaryUnsignedArray,0,250);
    SIG_S_BinaryUnsigned = SIG_S_BinaryUnsignedArray;

    /*STEP 1 - create key object*/
    eckey=EC_KEY_new();
 if (NULL == eckey)
 {
     P_STATUS = -1;  /*failed to create key object*/
 }
 else
 {
           /*STEP 2 - create EC_GROUP object*/
           ecgroup = EC_GROUP_new_by_curve_name(NID_secp521r1);
     if (NULL == ecgroup)
     {
         P_STATUS = -2; /*failed to create new EC Group*/
```

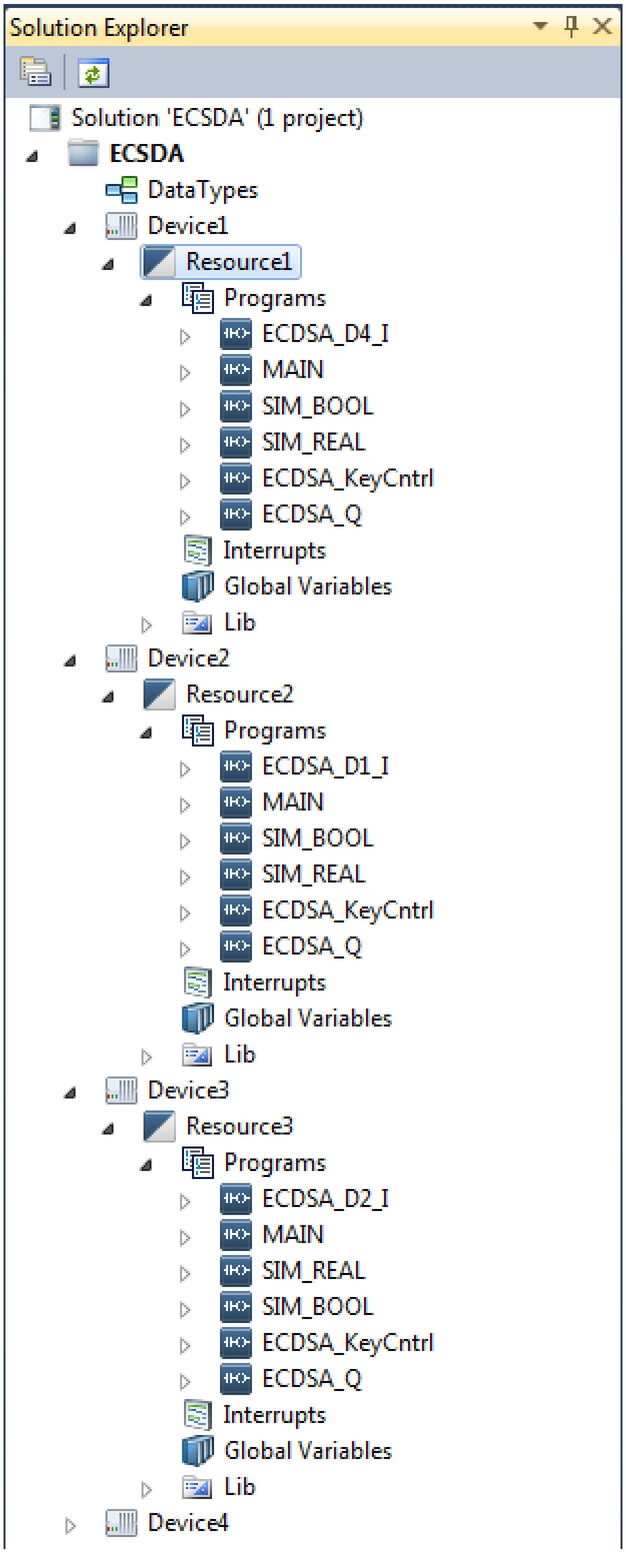
```
        }
        else
        {
                     /*STEP 3 - associate key with group*/
            set_group_status = EC_KEY_set_group(eckey,ecgroup);
            if (set_group_success != set_group_status)
            {
                            P_STATUS = -3; /*failed to associate key with group*/
            }
            else
                     {
                            /*STEP 4 - set public key*/
                            PublicKeyPoint = EC_POINT_new(ecgroup);
                            PublicKeyBN = BN_new();
                            buffer = (unsigned char*) P_PTR_PUBLIC_KEY_DATA;
                            EVP_DecodeBlock(PublicKeyBinaryUnsigned,buffer,240);
                            PublicKeyBinaryConst    =    (const    unsigned    char*)
PublicKeyBinaryUnsigned;
                            BN_bin2bn(PublicKeyBinaryConst,133,PublicKeyBN);


       EC_POINT_bn2point(ecgroup,PublicKeyBN,PublicKeyPoint,NULL);
                            set_public_status                                     =
EC_KEY_set_public_key(eckey,PublicKeyPoint);
                            if (set_public_success != set_public_status)
                            {
                                   P_STATUS = -4; /*failed to set public key*/
                            }
                            else
                            {
                                   /*STEP  5  -  convert  Base64  SIG_R  and  SIG_S  into
BN and set signature values*/

                                   signature = ECDSA_SIG_new();
                                   SIG_R_BN = signature->r;
                                   SIG_S_BN = signature->s;

                                   SIG_R_buffer       =       (unsigned       char*)
P_PTR_SIG_R_DATA;

       EVP_DecodeBlock(SIG_R_BinaryUnsigned,SIG_R_buffer,120);
                                   SIG_R_BinaryConst   =   (const   unsigned   char*)
SIG_R_BinaryUnsigned;
                                   if (P_PTR_SIG_R_DATA[0] == 'A')
                                   {
                                          BN_bin2bn(SIG_R_BinaryConst,66,SIG_R_BN);
                                   }
                                   else
                                   {
                                          BN_bin2bn(SIG_R_BinaryConst,65,SIG_R_BN);
                                   }

                                   SIG_S_buffer       =       (unsigned       char*)
P_PTR_SIG_S_DATA;

       EVP_DecodeBlock(SIG_S_BinaryUnsigned,SIG_S_buffer,120);
                                   SIG_S_BinaryConst   =   (const   unsigned   char*)
SIG_S_BinaryUnsigned;
```

```
                    if (P_PTR_SIG_S_DATA[0] == 'A')
                    {
                        BN_bin2bn(SIG_S_BinaryConst,66,SIG_S_BN);
                    }
                    else
                    {
                        BN_bin2bn(SIG_S_BinaryConst,65,SIG_S_BN);
                    }

                    if (NULL==signature)
                    {
                        P_STATUS = -5;
                    }
                    else
                    {
                        /*STEP 6 - Verify Signature*/
                        verify_status                          =
ECDSA_do_verify(hash,128,signature,eckey);
                        if (verify_success != verify_status)
                        {
                            P_STATUS   =   -6;   /*verification
failed*/
                        }
                        else
                        {
                            P_STATUS    =    1;    /*signature
verified*/
                        }
                    }
                }
            }
        }
    }

    /*CLEANUP - Release memory structures to prevent memory leaks*/

        EC_KEY_free(eckey);                     /*frees    EC_KEY    memory
allocation*/
        EC_GROUP_clear_free(ecgroup);   /*destroys any sensitive data in the
group and then frees the memory*/
        EC_POINT_free(PublicKeyPoint);  /*frees EC_POINT memory allocation*/
        BN_clear_free(PublicKeyBN);     /*overwrites the BN before returning
memory to the system*/
        ECDSA_SIG_free(signature);        /*frees      ECDSA_SIG      memory
allocation*/



/*debug logic*/
#if ISaDEBUG
    fp = fopen("debug-sigverify.txt", "w");
    fprintf(fp, "hello world\n");
    fprintf(fp, " \n");
    fprintf(fp, "PublicKeyBigNumber = ");
    BN_print_fp(fp, PublicKeyBN);
    fprintf(fp, " \n");
    fprintf(fp, "signature->r = ");
    BN_print_fp(fp, signature->r);
```

```
        fprintf(fp, " \n");
        fprintf(fp, "signature->s = ");
        BN_print_fp(fp, signature->s);
        fprintf(fp, " \n");
        fclose(fp);

#endif



}

/* eof ********************************************************************/
```

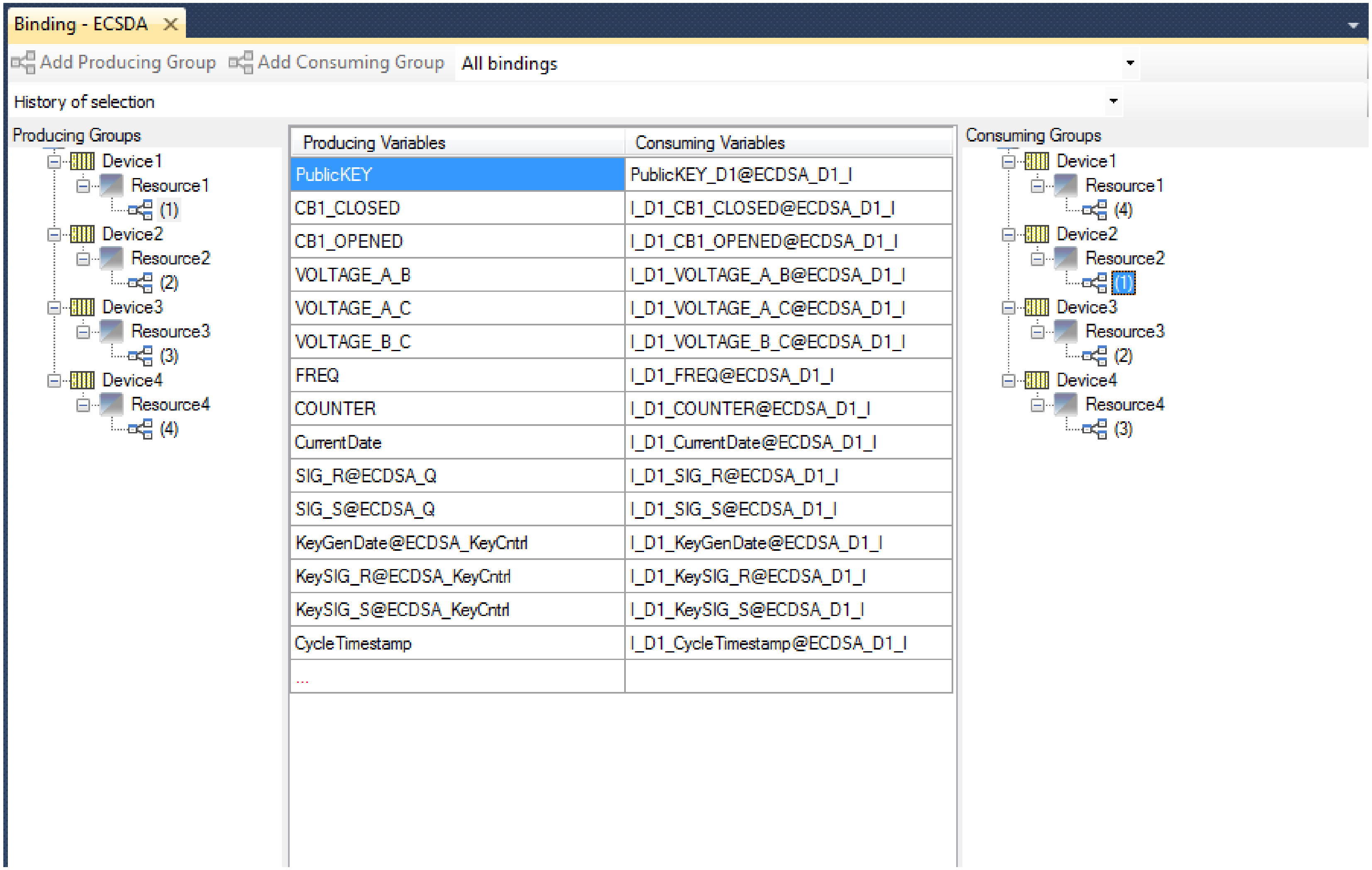
APPENDIX B. LADDER LOGIC CODE

B.1 Solution Explorer



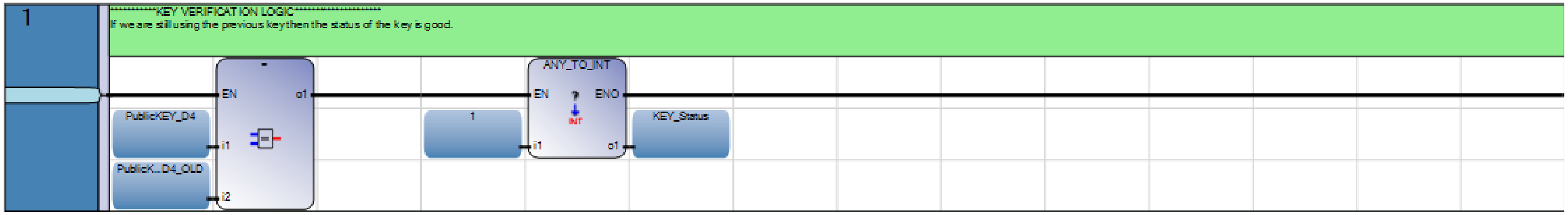
```
Solution Explorer
Solution 'ECSDA' (1 project)
◢ ECSDA
      DataTypes
   ◢ Device1
      ◢ Resource1
         ◢ Programs
            ▷ ECDSA_D4_I
            ▷ MAIN
            ▷ SIM_BOOL
            ▷ SIM_REAL
            ▷ ECDSA_KeyCntrl
            ▷ ECDSA_Q
            Interrupts
            Global Variables
         ▷ Lib
   ◢ Device2
      ◢ Resource2
         ◢ Programs
            ▷ ECDSA_D1_I
            ▷ MAIN
            ▷ SIM_BOOL
            ▷ SIM_REAL
            ▷ ECDSA_KeyCntrl
            ▷ ECDSA_Q
            Interrupts
            Global Variables
         ▷ Lib
   ◢ Device3
      ◢ Resource3
         ◢ Programs
            ▷ ECDSA_D2_I
            ▷ MAIN
            ▷ SIM_REAL
            ▷ SIM_BOOL
            ▷ ECDSA_KeyCntrl
            ▷ ECDSA_Q
            Interrupts
            Global Variables
         ▷ Lib
   ▷ Device4
```

B.2 Bindings Sample

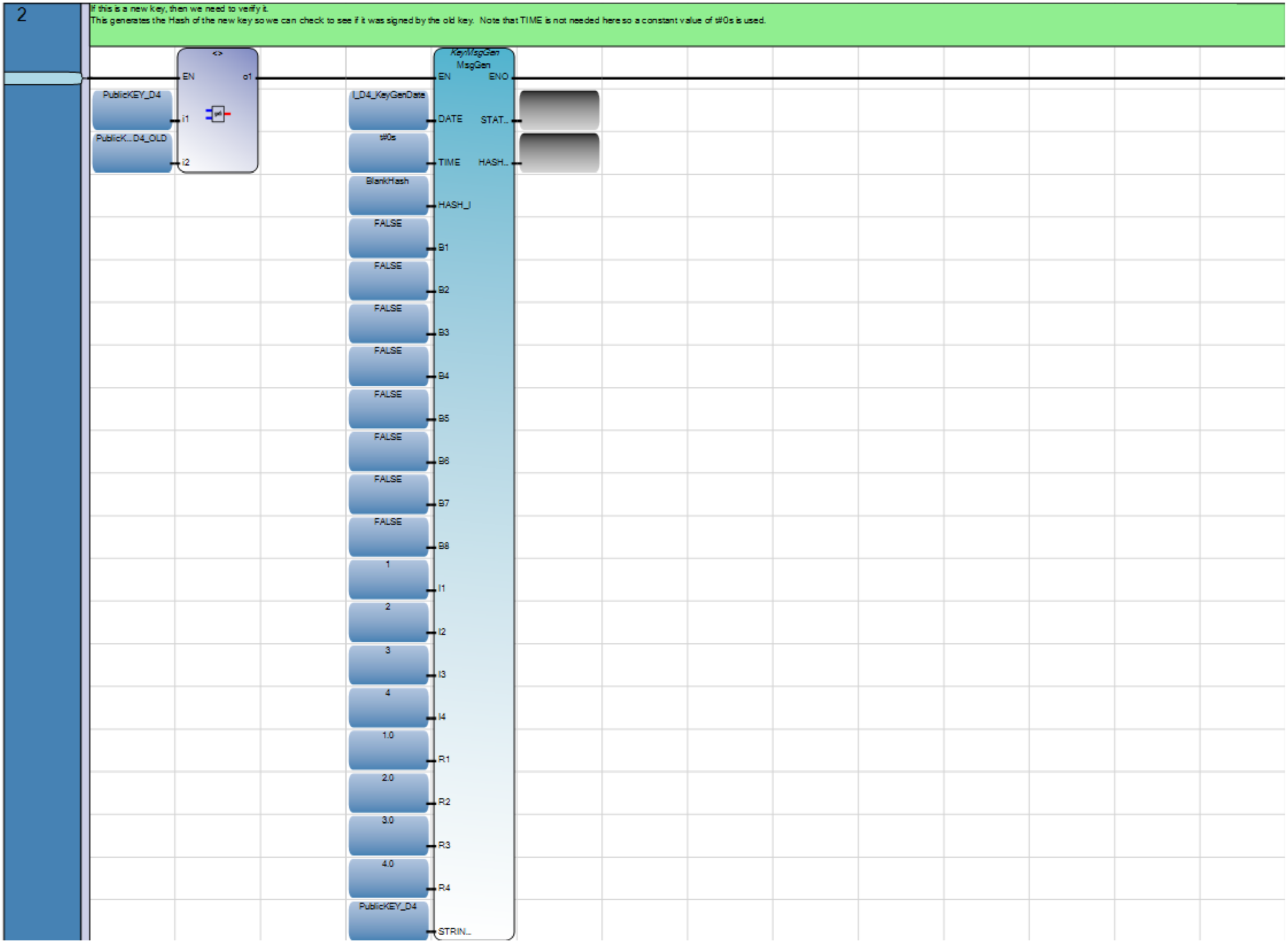


Producing Variables	Consuming Variables
PublicKEY	PublicKEY_D1@ECDSA_D1_I
CB1_CLOSED	I_D1_CB1_CLOSED@ECDSA_D1_I
CB1_OPENED	I_D1_CB1_OPENED@ECDSA_D1_I
VOLTAGE_A_B	I_D1_VOLTAGE_A_B@ECDSA_D1_I
VOLTAGE_A_C	I_D1_VOLTAGE_A_C@ECDSA_D1_I
VOLTAGE_B_C	I_D1_VOLTAGE_B_C@ECDSA_D1_I
FREQ	I_D1_FREQ@ECDSA_D1_I
COUNTER	I_D1_COUNTER@ECDSA_D1_I
CurrentDate	I_D1_CurrentDate@ECDSA_D1_I
SIG_R@ECDSA_Q	I_D1_SIG_R@ECDSA_D1_I
SIG_S@ECDSA_Q	I_D1_SIG_S@ECDSA_D1_I
KeyGenDate@ECDSA_KeyCntrl	I_D1_KeyGenDate@ECDSA_D1_I
KeySIG_R@ECDSA_KeyCntrl	I_D1_KeySIG_R@ECDSA_D1_I
KeySIG_S@ECDSA_KeyCntrl	I_D1_KeySIG_S@ECDSA_D1_I
CycleTimestamp	I_D1_CycleTimestamp@ECDSA_D1_I
...	

B.3 ECDSA_D4_I

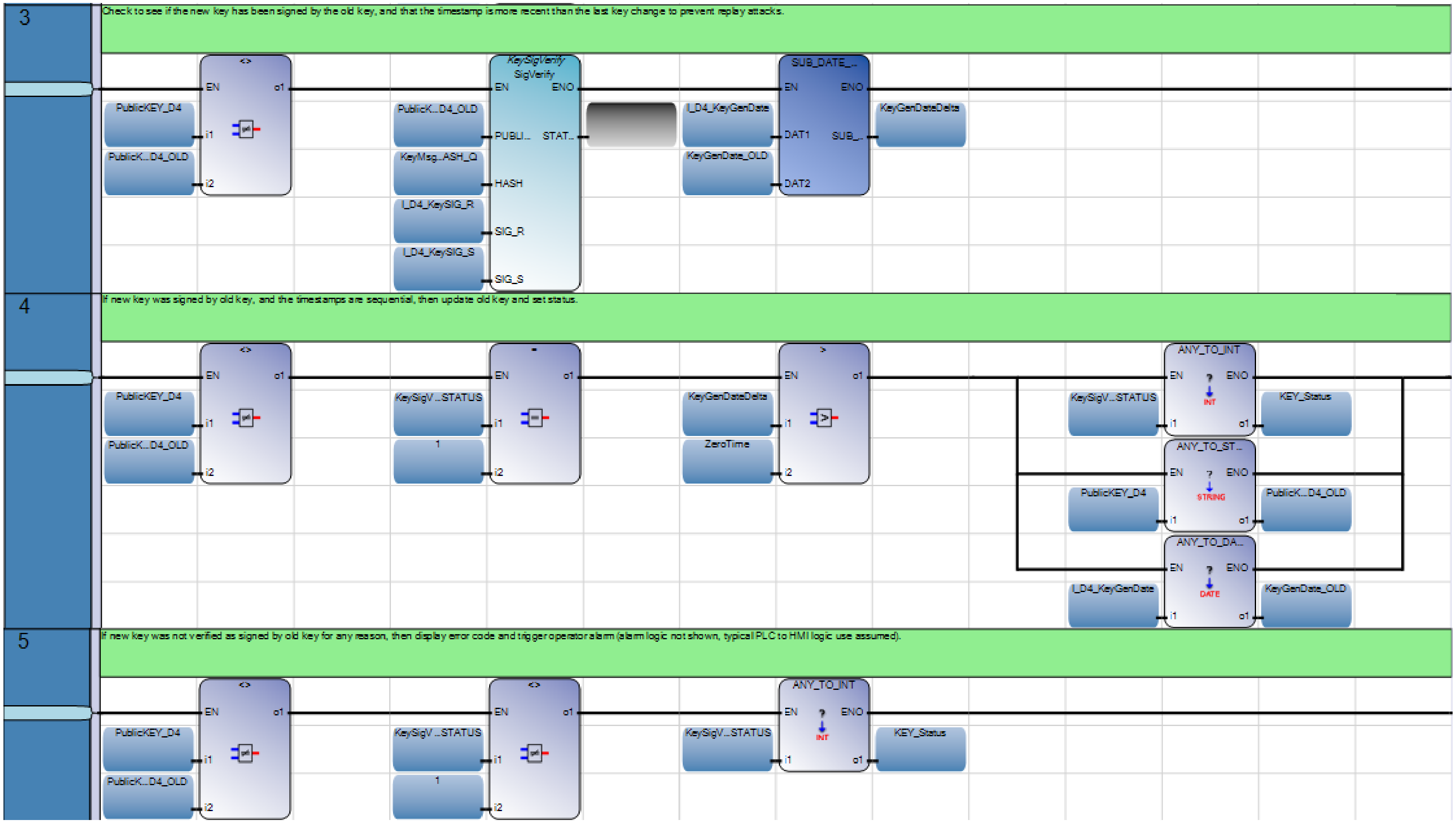


************KEY VERIFICATION LOGIC**********************
If we are still using the previous key then the status of the key is good.



2

If this is a new key, then we need to verify it.
This generates the Hash of the new key so we can check to see if it was signed by the old key. Note that TIME is not needed here so a constant value of t#0s is used.



3

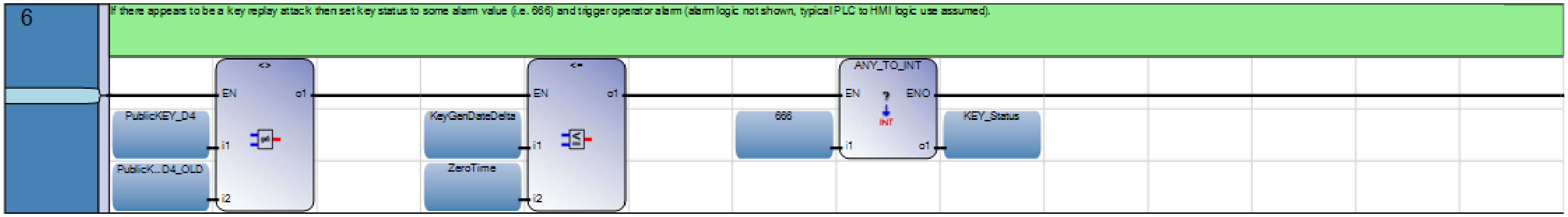
Check to see if the new key has been signed by the old key, and that the timestamp is more recent than the last key change to prevent replay attacks.

4

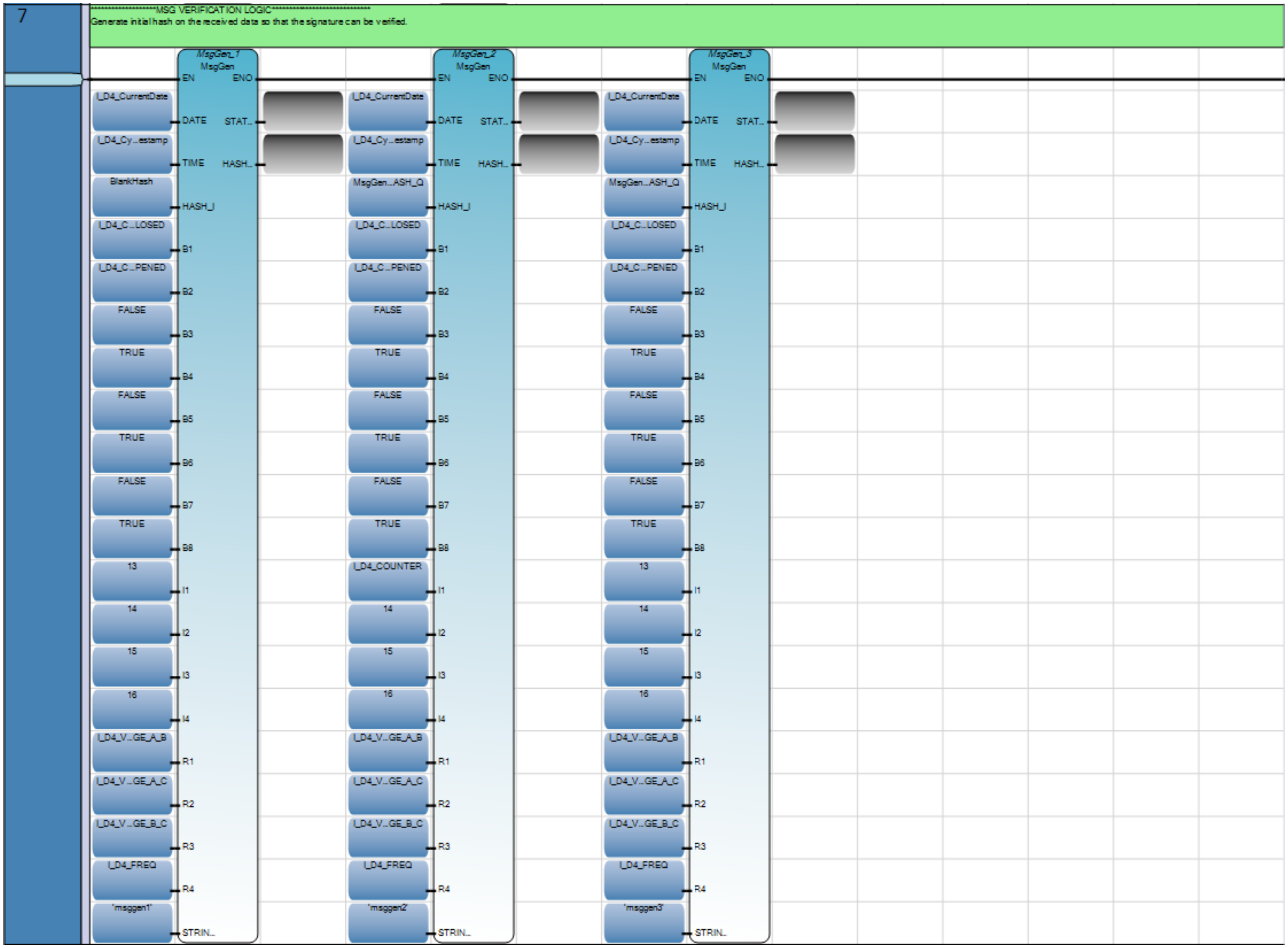
If new key was signed by old key, and the timestamps are sequential, then update old key and set status.

5

If new key was not verified as signed by old key for any reason, then display error code and trigger operator alarm (alarm logic not shown, typical PLC to HMI logic use assumed).



If there appears to be a key replay attack then set key status to some alarm value (i.e. 666) and trigger operator alarm (alarm logic not shown, typical PLC to HMI logic use assumed).

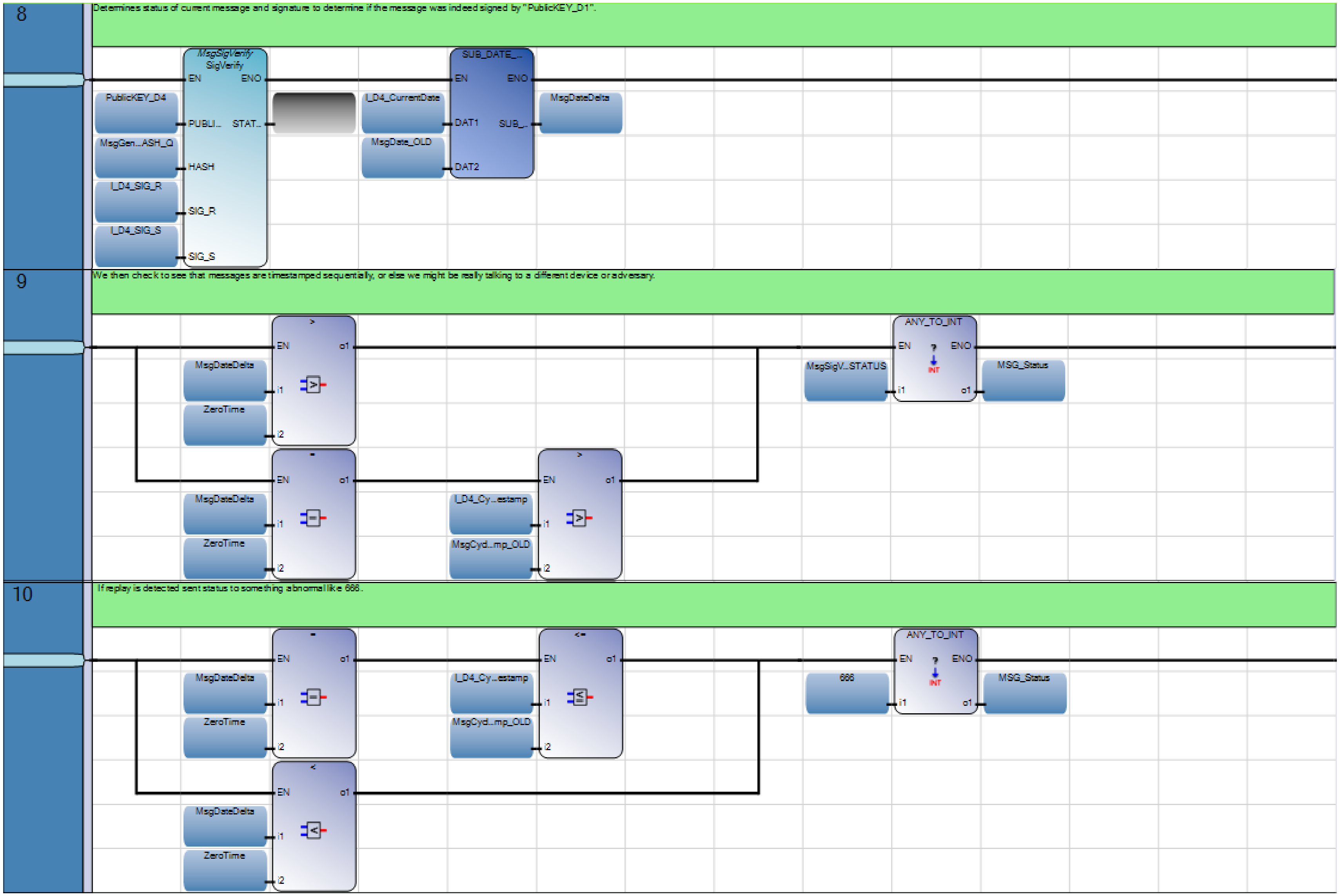


7

********************MSG VERIFICATION LOGIC**********************************

Generate initial hash on the received data so that the signature can be verified.

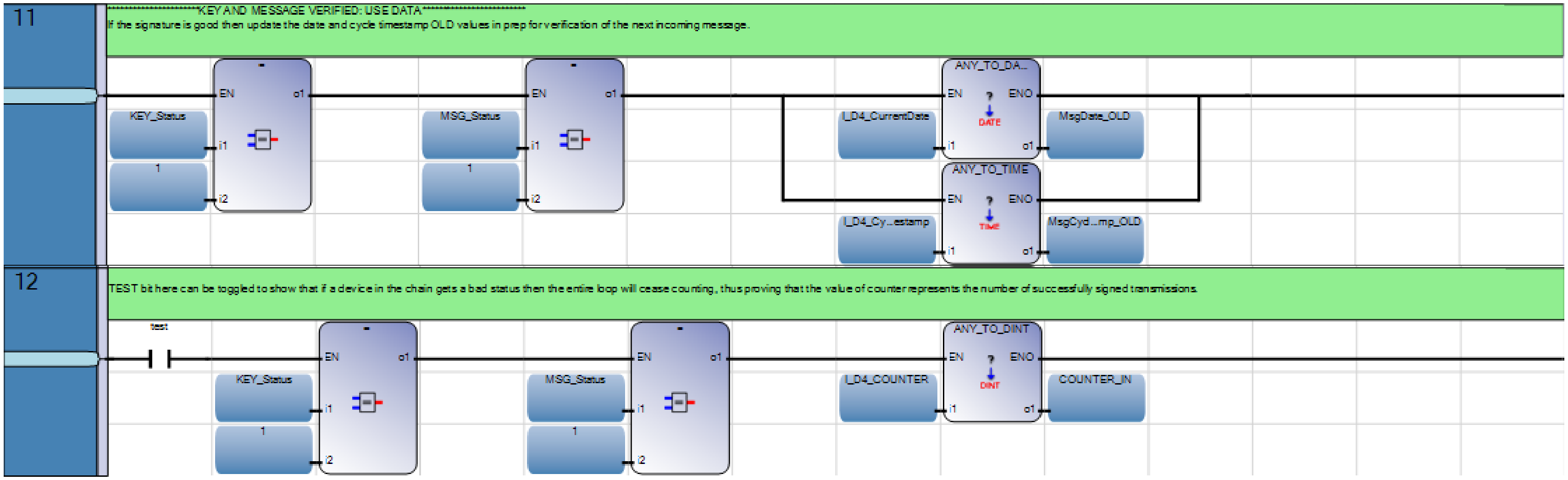
	MsgGen_1 MsgGen	*MsgGen_2* MsgGen	*MsgGen_3* MsgGen
EN / ENO			
DATE	I_D4_CurrentDate	I_D4_CurrentDate	I_D4_CurrentDate
TIME	I_D4_Cy...estamp	I_D4_Cy...estamp	I_D4_Cy...estamp
HASH_I	BlankHash	MsgGen...ASH_Q	MsgGen...ASH_Q
B1	I_D4_C...LOSED	I_D4_C...LOSED	I_D4_C...LOSED
B2	I_D4_C...PENED	I_D4_C...PENED	I_D4_C...PENED
B3	FALSE	FALSE	FALSE
B4	TRUE	TRUE	TRUE
B5	FALSE	FALSE	FALSE
B6	TRUE	TRUE	TRUE
B7	FALSE	FALSE	FALSE
B8	TRUE	TRUE	TRUE
I1	13	I_D4_COUNTER	13
I2	14	14	14
I3	15	15	15
I4	16	16	16
R1	I_D4_V...GE_A_B	I_D4_V...GE_A_B	I_D4_V...GE_A_B
R2	I_D4_V...GE_A_C	I_D4_V...GE_A_C	I_D4_V...GE_A_C
R3	I_D4_V...GE_B_C	I_D4_V...GE_B_C	I_D4_V...GE_B_C
R4	I_D4_FREQ	I_D4_FREQ	I_D4_FREQ
STRIN..	'msggen1'	'msggen2'	'msggen3'
STAT..			
HASH..			



8 Determines status of current message and signature to determine if the message was indeed signed by "PublicKEY_D1".

9 We then check to see that messages are timestamped sequentially, or else we might be really talking to a different device or adversary.

10 If replay is detected sent status to something abnormal like 666.



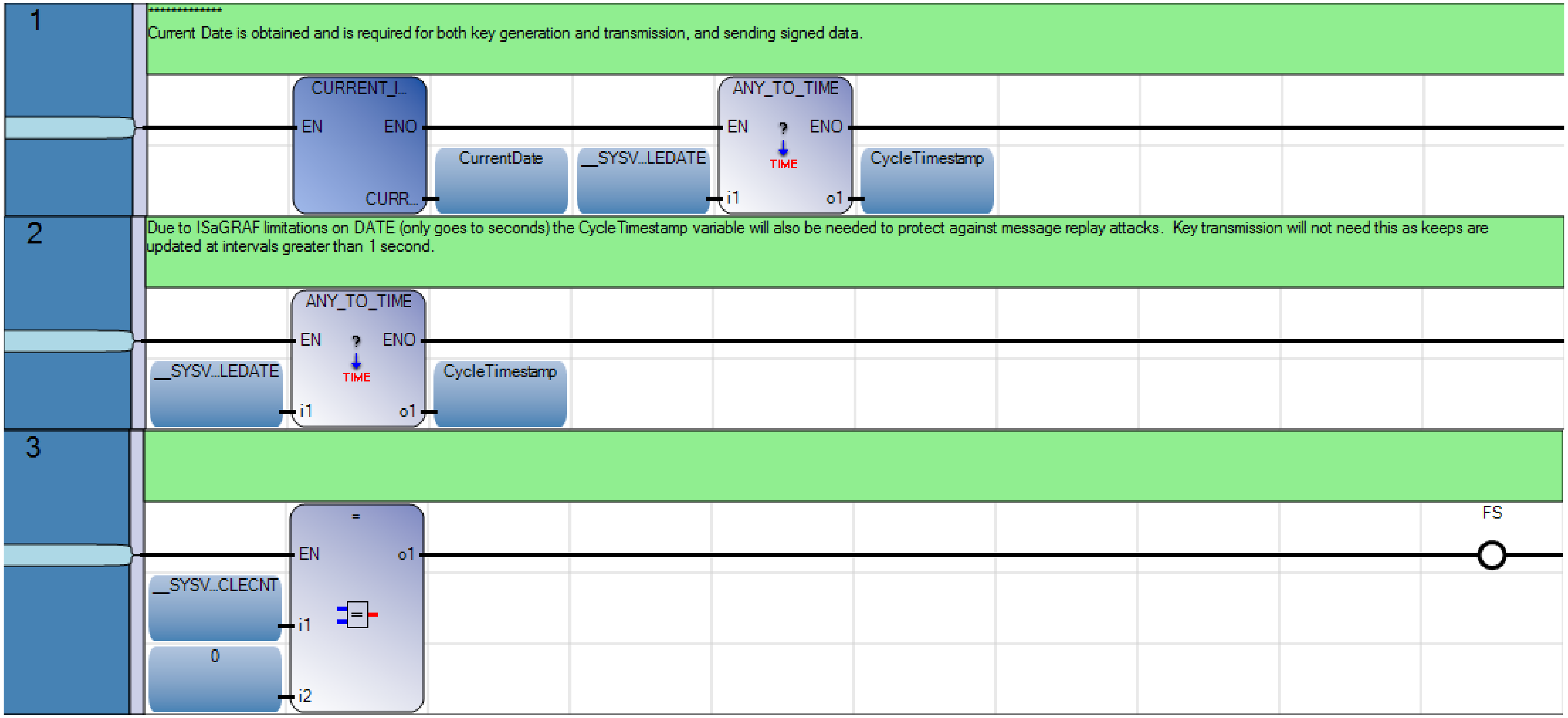
11

*********************KEY AND MESSAGE VERIFIED: USE DATA***********************
If the signature is good then update the date and cycle timestamp OLD values in prep for verification of the next incoming message.

12

TEST bit here can be toggled to show that if a device in the chain gets a bad status then the entire loop will cease counting, thus proving that the value of counter represents the number of successfully signed transmissions.

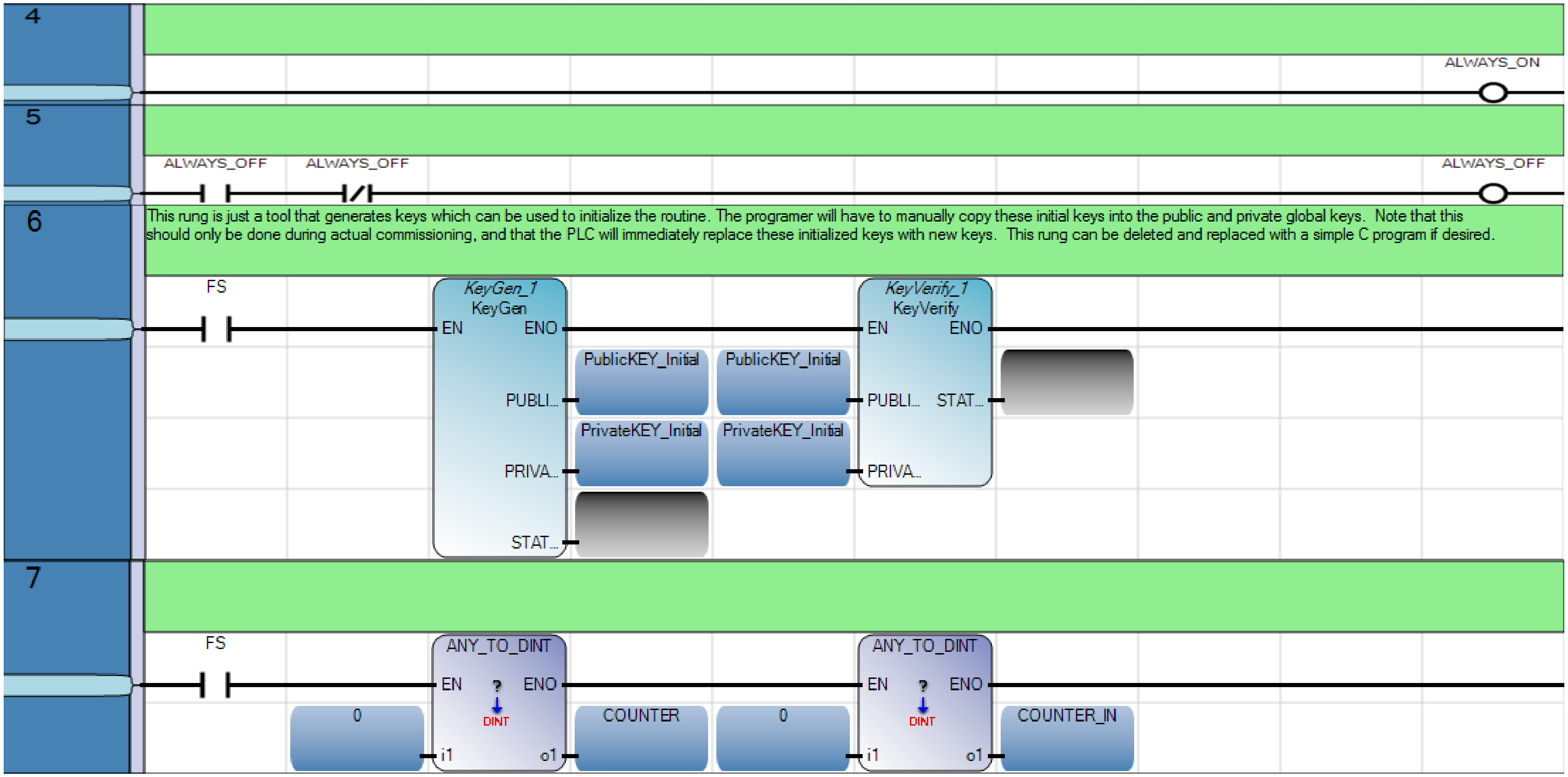
B.4 MAIN



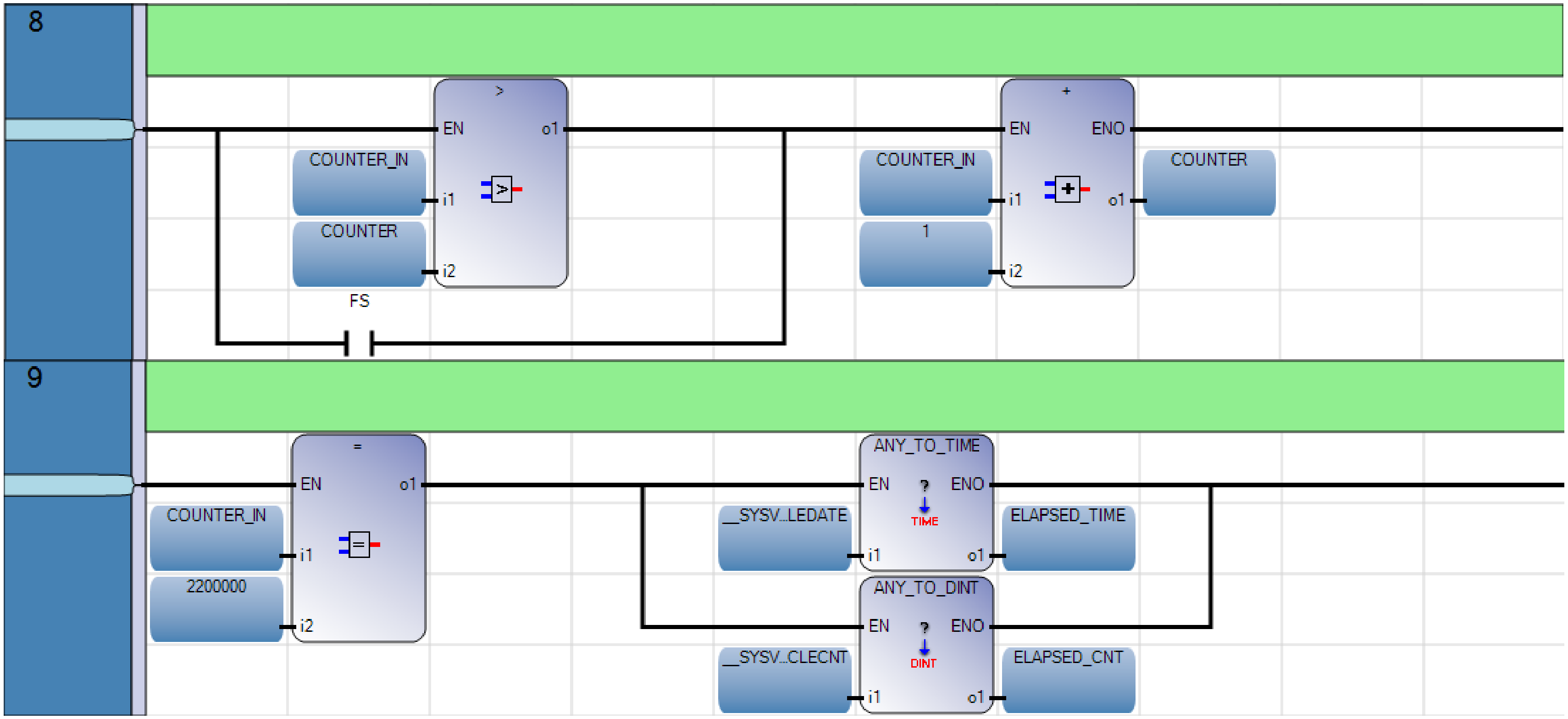
****

Current Date is obtained and is required for both key generation and transmission, and sending signed data.

Due to ISaGRAF limitations on DATE (only goes to seconds) the CycleTimestamp variable will also be needed to protect against message replay attacks. Key transmission will not need this as keeps are updated at intervals greater than 1 second.

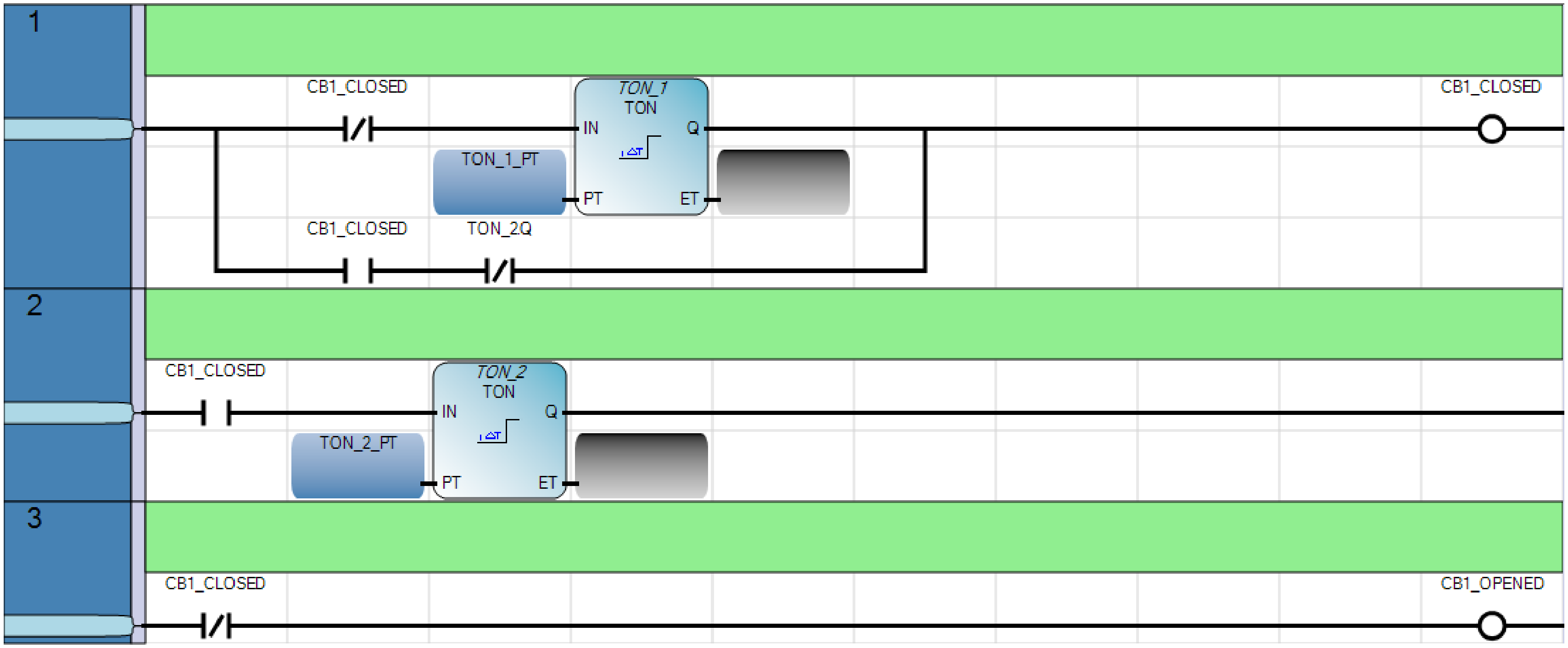


This rung is just a tool that generates keys which can be used to initialize the routine. The programer will have to manually copy these initial keys into the public and private global keys. Note that this should only be done during actual commissioning, and that the PLC will immediately replace these initialized keys with new keys. This rung can be deleted and replaced with a simple C program if desired.

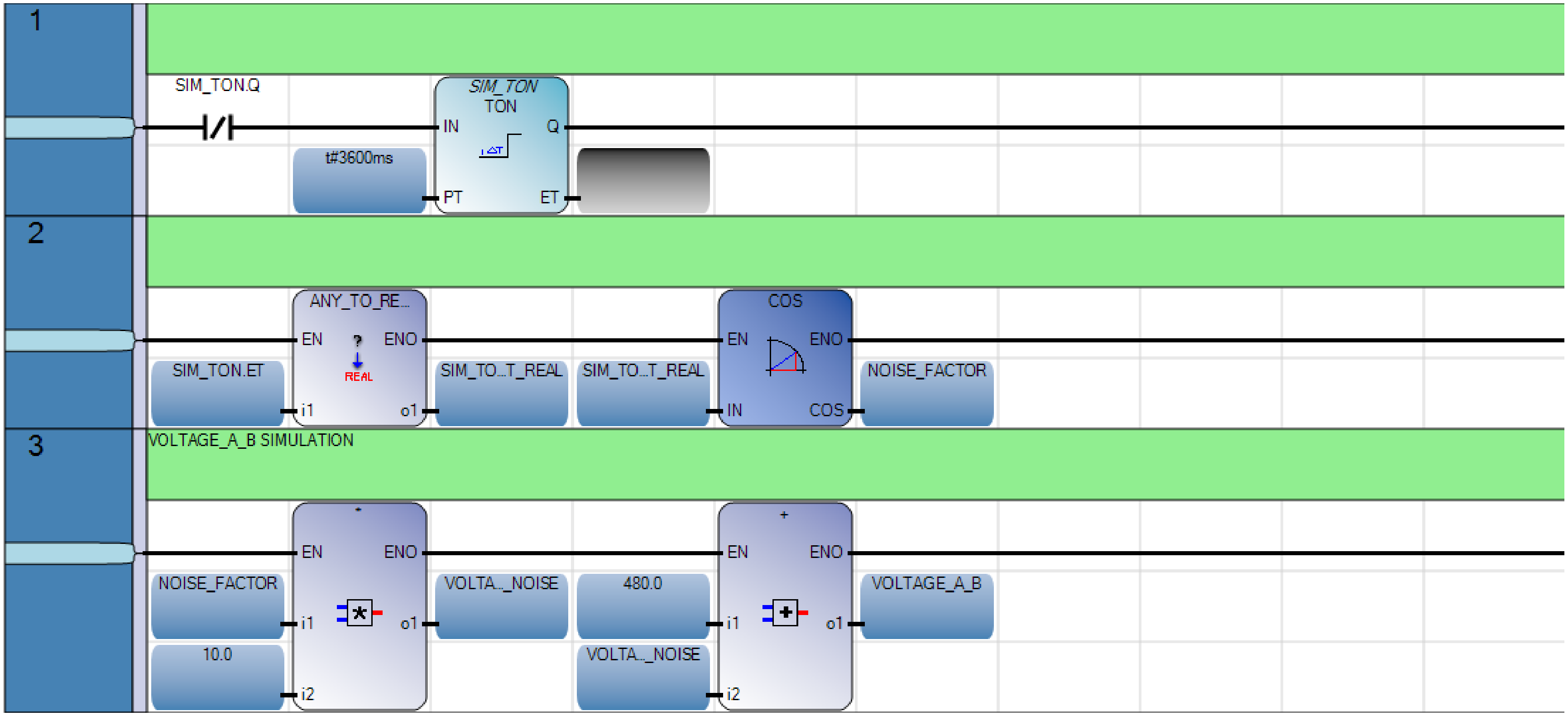


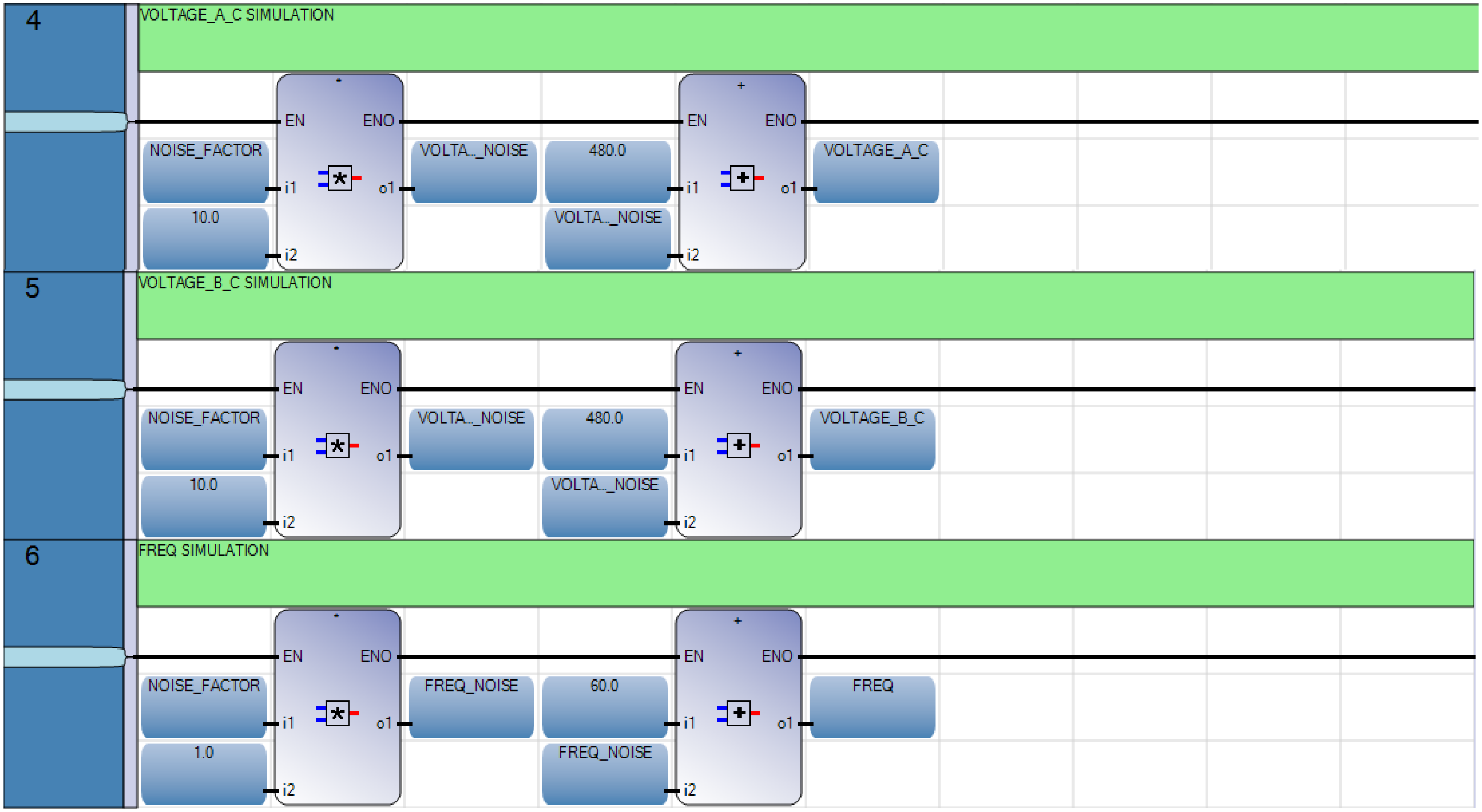
Note: Rungs 8 and 9 are only used for the main routine of Device 1 to initialize the device and to trap the number elapsed time and cycle counts. The other devices have a rung similar to rung 8 but without the FS XIC latch.

B.5 SIM_BOOL

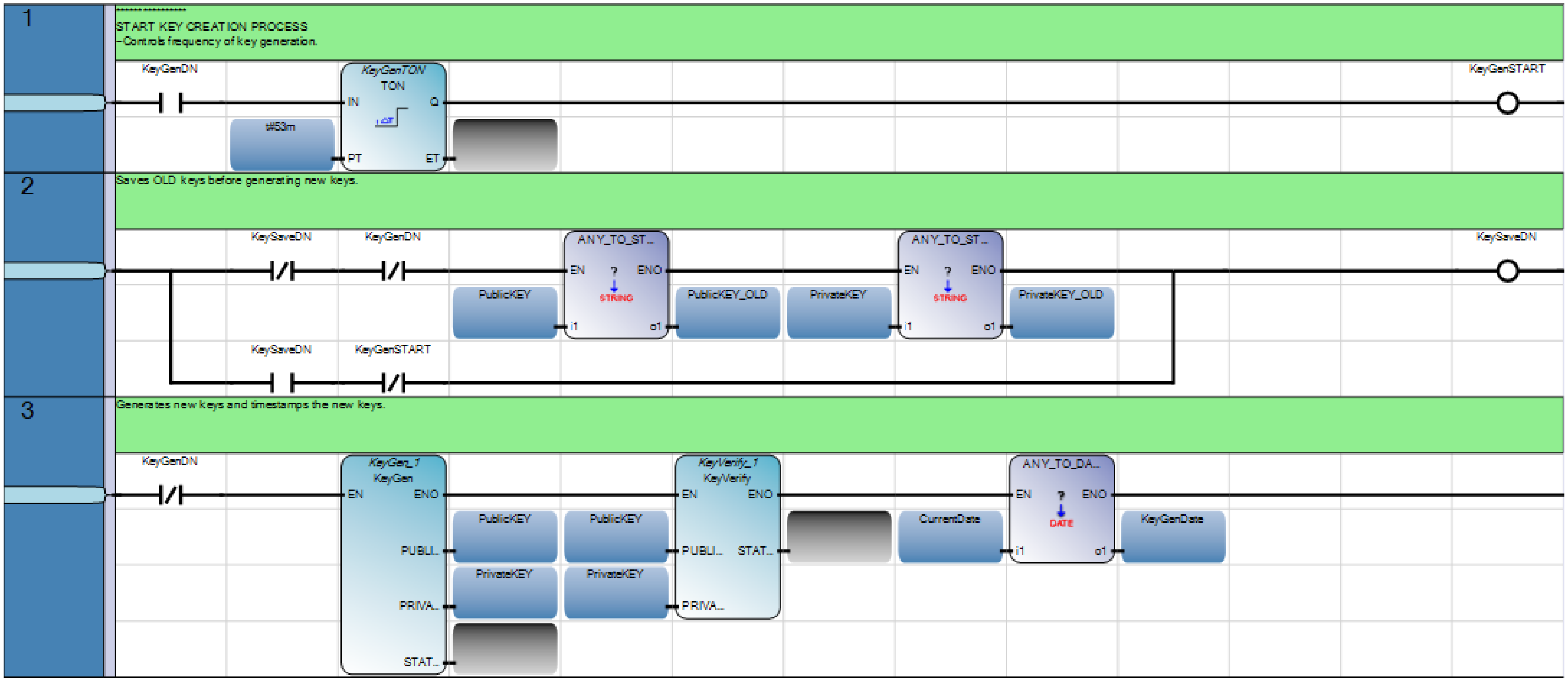

1
CB1_CLOSED
TON_1
TON
IN
Q
TON_1_PT
PT
ET
CB1_CLOSED
CB1_CLOSED
TON_2Q
2
CB1_CLOSED
TON_2
TON
IN
Q
TON_2_PT
PT
ET
3
CB1_CLOSED
CB1_OPENED

B.6 SIM_REAL


1
SIM_TON.Q
SIM_TON
TON
IN
Q
t#3600ms
PT
ET
2
ANY_TO_RE...
EN
ENO
REAL
SIM_TON.ET
i1
o1
SIM_TO...T_REAL
SIM_TO...T_REAL
COS
EN
ENO
IN
COS
NOISE_FACTOR
3
VOLTAGE_A_B SIMULATION
*
EN
ENO
NOISE_FACTOR
i1
o1
10.0
i2
VOLTA..._NOISE
480.0
+
EN
ENO
i1
o1
VOLTA..._NOISE
i2
VOLTAGE_A_B


4
VOLTAGE_A_C SIMULATION
*
EN
ENO
NOISE_FACTOR
i1
o1
10.0
i2
VOLTA..._NOISE
480.0
+
EN
ENO
i1
o1
VOLTA..._NOISE
i2
VOLTAGE_A_C
5
VOLTAGE_B_C SIMULATION
*
EN
ENO
NOISE_FACTOR
i1
o1
10.0
i2
VOLTA..._NOISE
480.0
+
EN
ENO
i1
o1
VOLTA..._NOISE
i2
VOLTAGE_B_C
6
FREQ SIMULATION
*
EN
ENO
NOISE_FACTOR
i1
o1
1.0
i2
FREQ_NOISE
60.0
+
EN
ENO
i1
o1
FREQ_NOISE
i2
FREQ

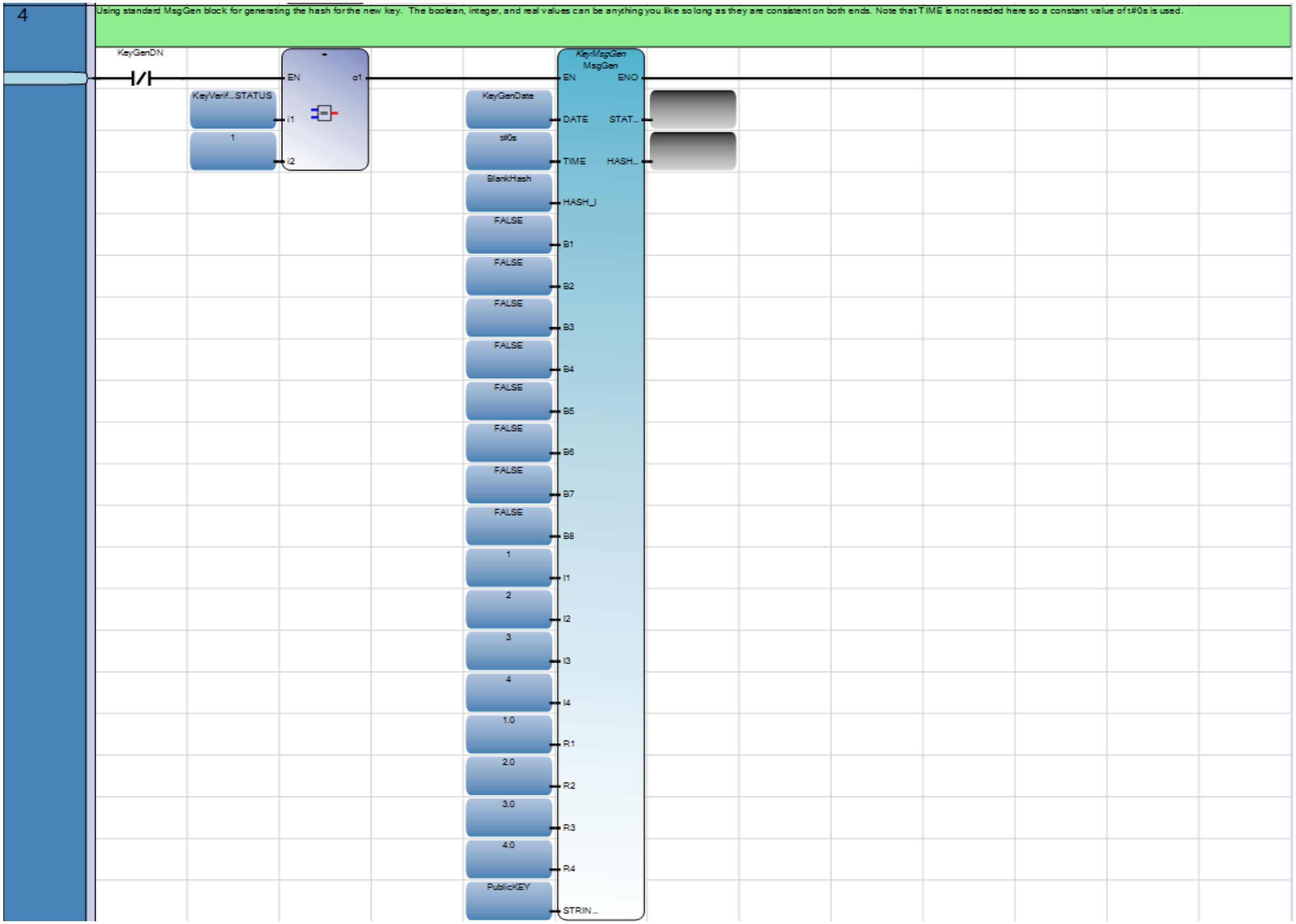
B.7 ECDSA_KeyCntrl



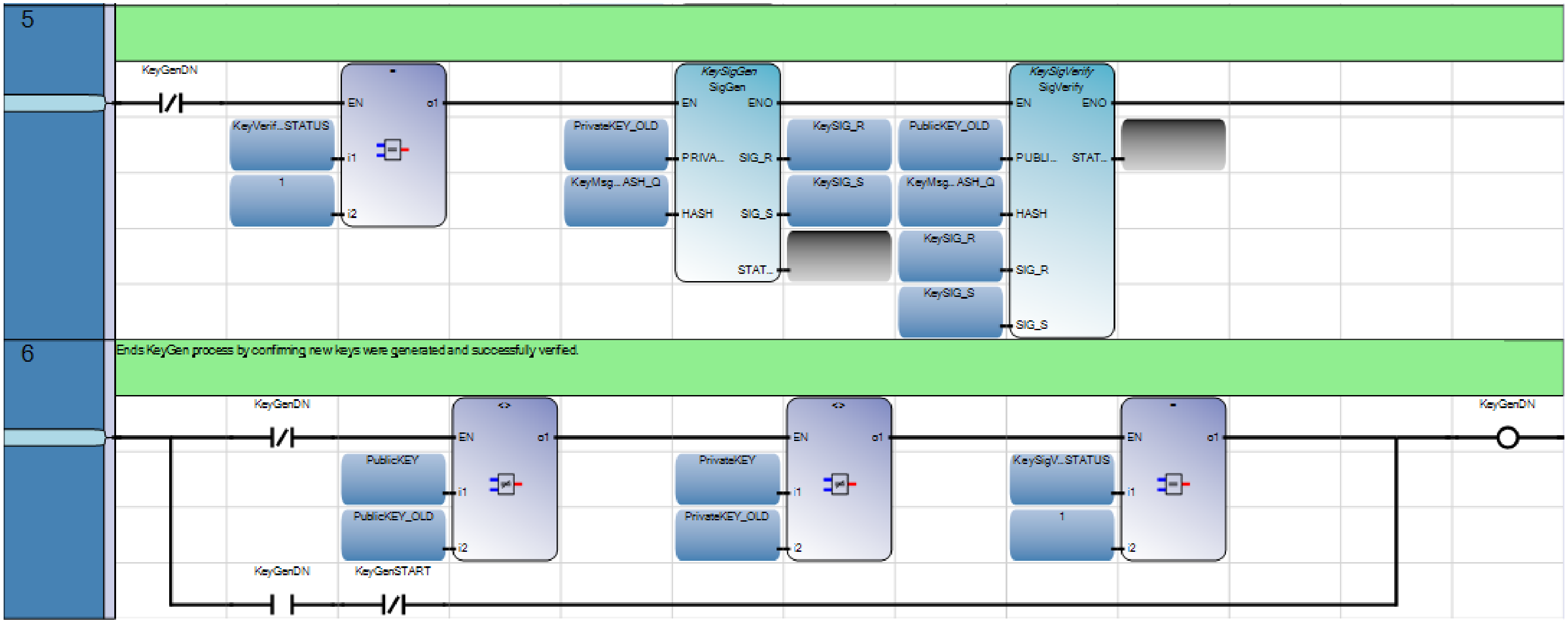
******* **********
START KEY CREATION PROCESS
--Controls frequency of key generation.

Saves OLD keys before generating new keys.

Generates new keys and timestamps the new keys.

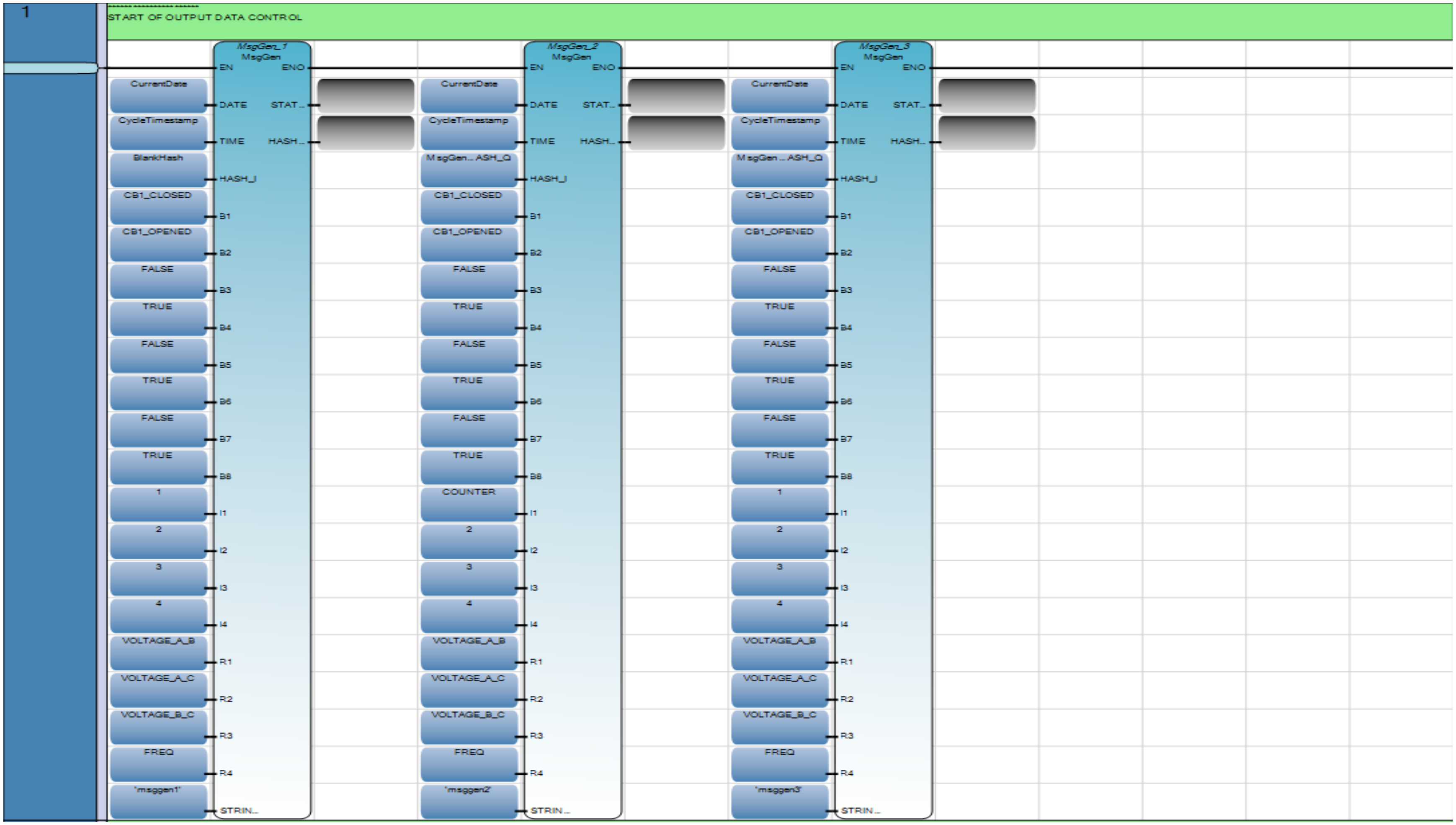


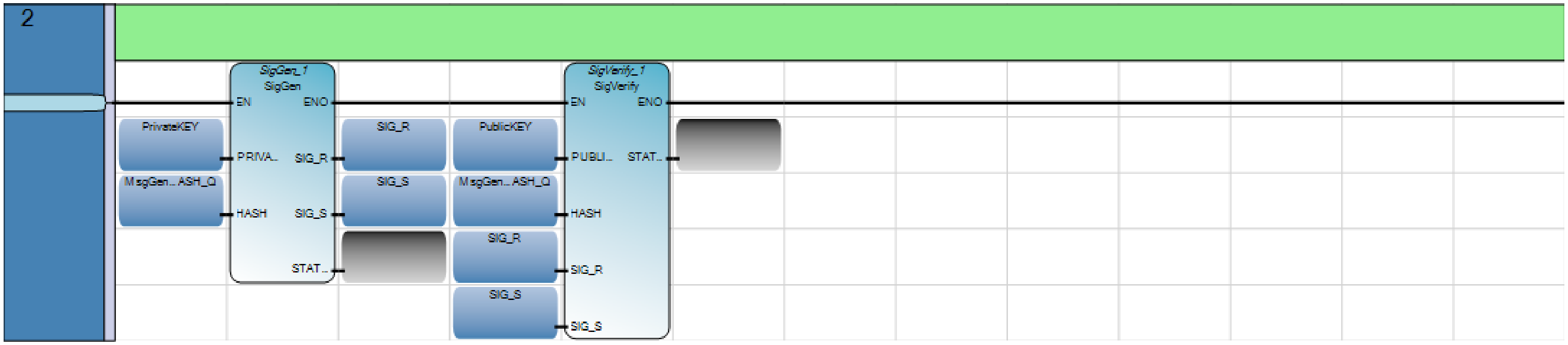
Using standard MsgGen block for generating the hash for the new key. The boolean, integer, and real values can be anything you like so long as they are consistent on both ends. Note that TIME is not needed here so a constant value of t#0s is used.



Ends KeyGen process by confirming new keys were generated and successfully verified.

B.8 ECDSA_Q


1
START OF OUTPUT DATA CONTROL
MsgGen_1
MsgGen
EN
ENO
CurrentDate
DATE
STAT...
CycleTimestamp
TIME
HASH...
BlankHash
HASH_I
CB1_CLOSED
B1
CB1_OPENED
B2
FALSE
B3
TRUE
B4
FALSE
B5
TRUE
B6
FALSE
B7
TRUE
B8
1
I1
2
I2
3
I3
4
I4
VOLTAGE_A_B
R1
VOLTAGE_A_C
R2
VOLTAGE_B_C
R3
FREQ
R4
'msggen1'
STRIN...
MsgGen_2
MsgGen
EN
ENO
CurrentDate
DATE
STAT...
CycleTimestamp
TIME
HASH...
MsgGen... ASH_Q
HASH_I
CB1_CLOSED
B1
CB1_OPENED
B2
FALSE
B3
TRUE
B4
FALSE
B5
TRUE
B6
FALSE
B7
TRUE
B8
COUNTER
I1
2
I2
3
I3
4
I4
VOLTAGE_A_B
R1
VOLTAGE_A_C
R2
VOLTAGE_B_C
R3
FREQ
R4
'msggen2'
STRIN...
MsgGen_3
MsgGen
EN
ENO
CurrentDate
DATE
STAT...
CycleTimestamp
TIME
HASH...
MsgGen... ASH_Q
HASH_I
CB1_CLOSED
B1
CB1_OPENED
B2
FALSE
B3
TRUE
B4
FALSE
B5
TRUE
B6
FALSE
B7
TRUE
B8
1
I1
2
I2
3
I3
4
I4
VOLTAGE_A_B
R1
VOLTAGE_A_C
R2
VOLTAGE_B_C
R3
FREQ
R4
'msggen3'
STRIN...


2
SigGen_1
SigGen
EN
ENO
PrivateKEY
PRIVA...
SIG_R
MsgGen...ASH_Q
HASH
SIG_S
STAT...
SIG_R
SIG_S
SigVerify_1
SigVerify
EN
ENO
PublicKEY
PUBLI...
STAT...
MsgGen...ASH_Q
HASH
SIG_R
SIG_R
SIG_S
SIG_S