Advancements in Control System Data Authentication and Verification

Kenneth A. Fischer

Code 516, NSWC Philadelphia Division
5101 South 18th Street, Philadelphia PA 19112
Kenneth.a.fischer@navy.mil

Abstract—*The challenge of securing industrial control systems is significant, and the need to provide authentication and verification services for control system commands and data has been well established. However there is currently a lack of options for the control system engineer, and the options that do exist mainly involve purchasing additional hardware. However these solutions disrupt the use of some real time communications products such as the Profinet/iMap combination used on Siemens PLCs. Previous work demonstrated the use of the Elliptic Curve Digital Signature Algorithm (ECDSA) on SoftPLCs, but solutions implemented for traditional PLCs such as the Rockwell ControlLogix line were deferred.*

There are a number of challenges to implementing control system data authentication and verification in a PLC, including performing mathematical operations on BigIntegers; generating and distributing keys; generating cryptographically secure hash values; implementing random number generation; and ensuring that the operations can be performed without impacting normal operation / scan times. In this paper we present a generic solution to these problems for implementation on any IEC61131-3 compliant PLC, as well as specific implementations for these solutions on the ControlLogix platform. Advances in cryptography are discussed including new algorithms that make use of twisted Edwards-curves that allow control system applications such as PLCs to perform advanced cryptographic functions without impacting overall performance at the 128-bit security level.

Index Terms—*PLC, HMI, Ed25519, EdDSA, ICS*

1. INTRODUCTION

The challenge of securing Industrial Control Systems (ICS) is significant and the need to provide authentication and verification services for control system commands and data has been well established [1,2]. 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.

Unfortunately, the utilization of cryptographic algorithms in a control system is a complicated endeavour. The primary processing component of a control system is a PLC, with software written in a language known as “ladder logic” that is based on old electrical relay diagrams. Ladder logic offers a number of benefits for control systems, primarily the benefit of being easy to understand and implement. Additionally it offers the ability to watch the code execute “live” to debug without the need of more advanced features such breakpoints common when performing debugging of a higher level language. In many ways, implementing a solution in ladder logic is much more similar to implementing a solution in hardware then in software, without the cost of expensive hardware components and the difficulty of changing hardware to meet new requirements.

The result is that control systems are completely unable to use existing implementations of cryptographic algorithms such as those found in the OpenSSL library. Even basic functions such as declaring an unsigned integer and a standard “for loop” are dramatically more difficult than in a language like C/C++. Memory is extremely limited in most PLCs and they do not possess the ability to natively do 64-bit integer calculations which are required for most cryptographic algorithms such as SHA-512. Generation of random numbers, a must for cryptographic algorithms, is all but impossible by a PLC and concepts such as the storing of digital certificates are completely foreign to a control system.

All of these challenges require a new approach to solving control system security that takes the best of our cryptographic technologies and essentially “translates” them into an implementation suitable for control systems. A solution was prototyped [2] using ISaGRAF™ SoftPLCs that was primarily written in IEC 61131-3 ladder logic with the addition of five specialized function blocks developed in C [1]. The specialized function blocks utilized the OpenSSL library to implement ECDSA and SHA-512 hashing functionality, key generation and verification functionality, and signature generation and verification functionality. The blocks could then be used in the same manner as a regular ladder logic instruction in the prototype’s implementation.

The intent of the prototype was to develop a solution written primarily in ladder logic that could be used in existing control systems if the specialized function blocks in C could be converted to ladder. 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 relied on the commissioning process of PLCs to establish the initial trust relationship.

Results of this work indicated that development of a fully IEC 61131-3 compliant implementation was justified. Such an implementation would need to go further and address a number of design issues of the prototype and present a solution that could be implemented across a wide range of platforms without negatively impacting performance. This paper presents such a solution, providing a cryptographically secure message authentication protocol without sacrificing performance.

Section 2 below describes the challenges in implementing cryptography on an ICS. Section 3 presents the background and concept of the solution. Section 4 provides details of the solution with respect to the major challenges described in Section 2. Section 5 provides recommendations for PLC vendors to include in future iterations of their products. Finally Sections 6 and 7 describe future work opportunities and provide the conclusion to this work.

2. ICS Cryptography Challenges

There are a number of problems with implementing cryptography in ICS, such as the inability to natively perform 64 bit operations. This section presents the three most significant challenges the author faced in developing an ICS cryptography solution.

2.1. Random Numbers

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

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

At this time for most applications the only true reliable method of obtaining random numbers is to install an external hardware-based random number generator. There are a number of products on the market today that make the claim to be TRNG (evaluation of such claims is beyond the scope of this paper). However most of those devices were designed with a more standard computer in mind and are incompatible for use with a PLC. An example is the TRNG9815device commercially available at www.trng98.com and shown in Figure 1 below. This device is based upon a Zeener diode noise source which is then amplified to be read by a PC.

Figure 1. TRNG 9815 Device

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

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

Ultimately a solution is required that will significantly decrease our need for random numbers from potentially 380+ million random bits a day to 256 random bits every couple of months. This would allow us to use PRNGs to

meet our application needs in the short run and ultimately loosen the design constraints for an eventual hardware based TRNG that can be used on a PLC.

2.2. PLC Processing Speed

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

Modern computers have multi-core processors that run in the GHz range and typically do not have real-time performance requirements (those that do can always have dedicated cryptographic modules). PLCs on the other hand have strong real-time performance requirements and have processors that run in the MHz range due to the lack of active cooling. To illustrate this challenge a basic SHA-512 algorithm written completely in ladder logic was developed. Running that algorithm in RSEmulate on a standard PC running an Intel 2500K processor at 4.2 GHz for 104 bytes of data gave a scan time of 2ms. Running that same algorithm on a 1756-L8x series PLC gave a scan time of 26ms, over an order of magnitude higher. While this may not seem like a lot, consider that operations used in cryptography such as point multiplication and modular exponentiation are much more complex. Such operations could easily consume the entire available scan time of a PLC and leave no real time to actually perform control work.

Therefore a solution is required that would allow us to perform these complex operations without impacting scan time.

2.3. BigIntegers

A BigInteger is a data type that represents an arbitrarily large integer whose value has no upper or lower bound. This is distinguished from a DINT or even a LINT, which has an upper bound limited to the number of bits present in the data type (32 and 64 respectively). In RSA, a common cryptographic algorithm used to secure a wide range of systems, BigIntegers on the order of 2048 to 8192 bits are used. In previous work [1] it was discussed how Elliptic Curve Cryptography (ECC) helps reduce the scale of this problem by allowing us to use smaller BigIntegers to provide solutions of equivalent strength at reduced bit sizes. For example, to provide security at the 128-bit level (common security level for data considered "secret") the RSA BigIntegers need to contain 3072 bits according to NIST SP 800-57. ECC can provide us the same security using BigIntegers of 256 bits, which dramatically improves the scope of the problem but does not in itself provide us a solution to perform 256-bit math in a PLC.

3. Proposed Solution Overview

3.1. Background

In the original prototype, a ladder logic implementation was used that contained the following custom C function blocks using the OpenSSL library: MsgGen, KeyGen, KeyVerify, SigGen, and SigVerify. The prototype had each controller start with a private-public key pair and knowledge of the public keys of each controller it needed to talk to. The prototype then immediately changed its active key and used a self-signing algorithm to communicate the key change to each pair. The controllers would regularly use this same mechanism to update and communicate these keys (on an approximately every hour basis).

Messages were hashed in the prototype using SHA-512, but had a limitation of only being able to receive a very fixed range of input options and to extend the range of input MsgGen Blocks had to be chained together as shown in Figure 2 below. The messages were then digitally signed by the controller using ECDSA and then transmitted to the next controller. The next controller would receive the data and the signature. Then it would have to regenerate the hash from the message and verify the digital signature. If the signature was not verified it would generate an operator alarm.

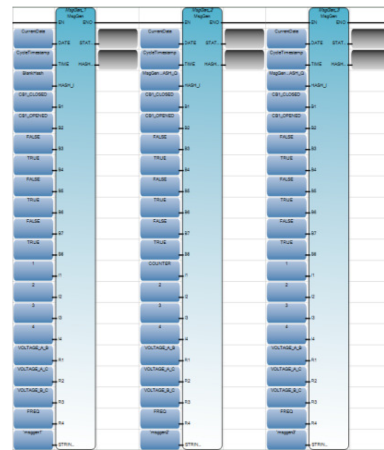

Figure 2. Three MsgGen Blocks Connected Together

This concept worked on the SoftPLC due to the increased power of the PLC, the ability to use OpenSSL's BigInteger data type, and the ability of the SoftPLC to generate random

numbers using the built-in RDRAND features of the Intel chipset. Discussion of the security of RDRAND itself and the potential backdoor compromise is beyond the scope of this paper, but for the sake of discussion we will assume it to be secure.

3.2. Concept

This section describes the high level concept of what the system needs to do in order to provide security. Most of what follows in this section is a fairly standard description of establishing a secure session across an unsecure channel.

Assume you have a Distributed Control System (DCS) that needs to interface with a gas turbine's Full Authority Digital Control (FADC) across an unsecure channel. The DCS has a Remote Terminal Unit (RTU) that will directly communicate to the FADC via Ethernet. As typical in an ICS development, an Interface Design Document (IDD) is developed that defines the parameters the RTU needs to send and receive from the FADC and how to obtain/send that data.

The first step in this concept is that we add to the IDD a new parameter, a 256-bit public key **A** for both the FADC and the RTU. The public keys are generated offline and loaded into the controllers, along with corresponding private keys **a** which are not shared and considered CONFIDENTIAL or SECRET. The private key **a** and the public key **A** are mathematically related to each other. The primary purpose of these keys is to allow control system nodes to be able to identify each other across unsecure channels and prevent Man-In-The-Middle (MITM) attacks.

In the proposed design, PLC messages are authenticated via a Hashed Message Authenticated Code (HMAC) algorithm, where the data communicated between the RTU and the FADC is combined with a shared secret session key **K** to produce a fixed length result. The session key will be generated using the Elliptic Curve Diffie-Hellman (ECDH) algorithm and a cryptographically secure PRNG. ECDH would have each controller in each pair of nodes generate a private number **q**. That number would be used in a point multiplication algorithm in which a publicly known base point **B** on an elliptic curve (also defined in the IDD) is multiplied by the pseudo-random number to generate a result **Q** that is made public.

Simultaneously the public number is digitally signed within each controller, using the Edwards-curve Digital Signature Algorithm (EdDSA) [4] and the private key **a**. EdDSA offers a number of improvements to standard ECDSA, such as the formulas being complete along the curve (no special exceptions that need to be handled) and the ability to generate secure digital signatures without the need for a random number. Additionally, the curve Ed25519 is optimized for performance while still providing 128-bit level security.

The digital signature **[R, S]**, and the public ECDH value **Q** is sent to the other controller. The two controllers each receive the other's public value and digital signature. The controllers then verify the digital signature using only the public key value **A**. Once verified the controllers would take the public number **Q** and multiply it with their own private number **q** to create the 256-bit key **K**, which is never sent across the unsecure channel. This process works because:

$$\mathbf{K = q1{*}Q2 = q1{*}q2{*}B = q2{*}q1{*}B = q2{*}Q1}$$

This entire operation would occur during power-up or when initiated by an operator. This means that the most complex mathematical operations would only need to occur rarely such as once a year, during planned maintenance availabilities. The HMAC itself is used to protect the system during operation, allowing the system to identify legitimate commands versus illegitimate commands, which can then be alarmed for operator's situational awareness.

The advantage of this approach is that it significantly reduces the workload of the controllers during normal operation compared to digital signatures. It also significantly reduces the scale of the random number problem, since now we only need one random number each time we change the session key (i.e. once every couple of months to a couple of years depending on the strength of the HMAC and the data rate). At that rate, a PRNG initialized with a truly random seed generated offline would be sufficient, effectively solving the random number problem without the use of a PLC hardware based TRNG.

The approach as described so far does not address the other two major concerns: how do you handle BigIntegers and even more importantly how do you get a PLC to do this kind of complex math without impacting control system performance?

4. Solution Details

4.1. Representing BigIntegers in Ladder Logic

Representing a BigInteger can be done a number of ways, with the most obvious being to simply take an array of 8 32-bit DINTs, which total 256-bits. The problem with this approach is that when you perform operations such as addition and multiplication you end up having to propagate the carries, and if you use an array of 8 32-bit DINTs you have to perform a lot of work to propagate the carries which turns into a very long sequence of bit operations. Bernstein on his blog [5] states "The standard NIST P-256 reduction procedure becomes even more painful if integers aren't represented in radix 2^{32} (or 2^{16} or 2^8): the word shuffling required for $T,S_1,S_2,S_3,S_4,D_1,D_2,D_3,D_4$ then turns into a long sequence of bit manipulations. The reason this is important is that radix 2^{32} isn't the best way to carry out big-integer arithmetic on most CPUs. Even on CPUs where the largest multiplier size is exactly 32 bits, it's almost always better to use a radix smaller than 2^{32}, so that carries can be delayed."

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

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

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

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

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

4.2. Sequencing Complex Mathematical Operations

Here we discuss how to perform the more complex math operations, using the specific example of point multiplication which is at the heart of ECC. Elliptic curve point multiplication is the process of repeatedly adding a point along an elliptic curve to itself. This process creates a trapdoor function, and the security of ECC is based on the intractability of determining a multiplier **n** from the equation **Q = n*P** where **Q** and **P** are given points on the curve.

There are a number of approaches in performing point multiplication, such as the sliding-window method and Montgomery ladder. Discussion of the details of point multiplication is beyond the scope of this paper, but what is relevant is that it is an extremely complex algorithm to implement on a PLC. If you follow the approach in RFC 8032 [4] then a single point addition requires nine 256-bit multiplications, four 256-bit additions, and four 256-bit subtractions. Each of these operations must be repeated for each bit in the multiplier in order to perform the point multiplication at constant time and avoid a timing side channel attack.

The coding of such an algorithm into ladder logic is a challenge in of itself, but even the most efficient implementation will not be able to run on the PLC. This is because, as stated above, there is an already known optimally efficient way to store the 256-bit value, which results in a known optimal number of standard ladder logic math instructions such as ADD and MUL to perform a basic B256_ADD and B256_MUL. For example, just looking at the number of standard RSLogix5000 MUL instructions in one point multiplication gives the following result:

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

1 POINT_ADD = 9 B256_MUL

1 B256_MUL = 484 MUL

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

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

The solution to this problem is to break the point multiplication algorithm up into a series of sequential pieces, so that only a small portion of the total algorithm is run in any given scan. This is similar to a standard sequencer used in a batch control process, where the sequencer executes a set of commands in each step and waits for a feedback from the process indicating that step is complete before executing the next step. In the case of point multiplication, the most obvious solution is to break up the algorithm so that each point addition is performed on a separate scan. The sequencer waits for a confirmation that the point addition is completed, stores the result in a temporary variable, and then uses that result in the next scan for the next point addition. Appendix B shows a sequence chart for the point multiplication

Using this approach a point multiplication AOI was developed that can run on a Rockwell 1756-L83 processor with only a 10ms impact to total scan time. The same approach was used on all the other different pieces of the Ed25519 digital signature algorithm such as SHA-512. Combining all the pieces into a master sequencer produces a digital signature implementation capable of running on a PLC with a projected scan time of impact of less than 10ms. The prototype for the signature is still in development, however the projections are based on the known impact of the most computationally complex operations which are complete, multiplied by 2. The prototype AOI for the Ed25519 digital signature algorithm is shown in Figure 3. The AOI is designed to use a externally generated keys that can be generated to meet the requirements of RFC 8032 [4]. The private key portion is modified so that instead of saving the random number as the private key you save the scalar and the hash generated from the random number. This improves overall efficiency by removing the requirement to

repeatedly perform operations that could be performed once offline and then saved for future use.

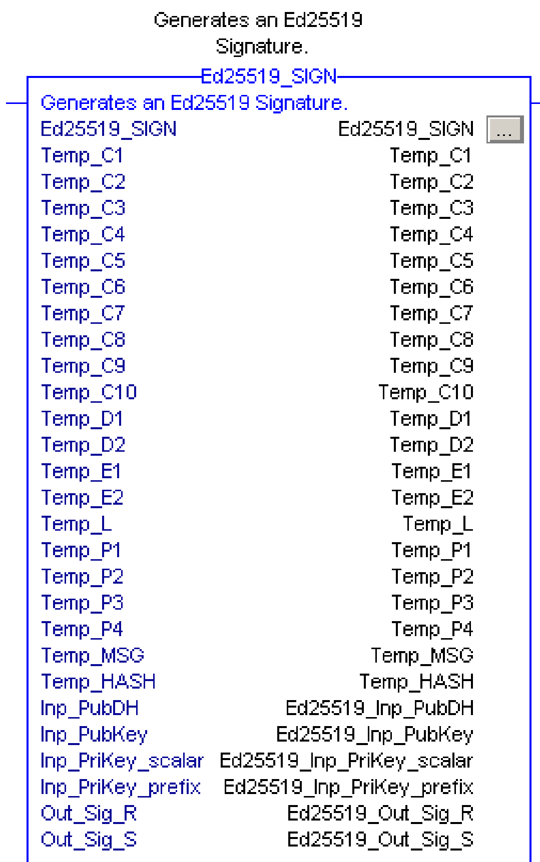

Figure 3. Ed25519_SIGN Add-On Instruction

There are downsides to this approach. The first is that the total time to produce a digital signature is significantly longer. Depending on the processor and the amount of control code that has to run in a scan (that is not related to cryptography) the entire operation could take a couple of minutes to complete. Testing on the point multiplication showed that it took less than a minute to complete on a 1756-L8x series processor. Regardless, the system itself will be able to run without impact during this time. The message traffic will continue to use the old key until negotiation and verification of the new key is complete. The process can also be performed during scheduled maintenance windows to reduce overall risk.

The second problem is that this approach increases the size of the program significantly since we are essentially “trading space for time”. This is unlikely to cause a concern on newer processors, which have significant memory storage capabilities. However for older devices or embedded devices this approach could be constrained.

The third problem is that extending this algorithm across multiple scans can potentially make the algorithm vulnerable to side channel attacks. To prevent this two steps must be taken:

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

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

4.3. PLC Message Authentication Code

The use of the HMAC for per-message security vs digitally signing each message is the biggest change to the work done in the prototype [2]. The HMAC algorithm must be designed to meet the following requirements:

- The algorithm would be fast enough to create the HMAC without negatively impacting overall control system operation.
- Any adversary with full knowledge of the software, the data, and matching hardware would not be able to produce the HMAC without knowing the secret session key
- Provide 128-bit level security (SHA-256 equivalent)
- For a single bit change in the input produce on average approximately 128 bit changes in the output and no less than 50 bit changes in the output in the worst case (equivalent of SHA-256)
- Protect against common attacks such as the length extension attack and collision attack

The standard HMAC algorithm for producing values at 128-bit security is HMAC-SHA-256 which is a $\Theta(n)$ algorithm. This algorithm was implemented and tested in the lab on a Rockwell 1756-L83 processor. The message size was 200 bytes. Initial results in the lab indicated that such an algorithm would have a scan time of 50-100ms when used in production. Considering that you would use two of them for both send and receive the combined scan time impact could reach up to 200ms for only 200 bytes. This would have detrimental impact on overall control system operation. Unfortunately, since the entire operation would have to be performed per message per scan there is no way to practically sequence the code in the same manner as was done for point multiplication or other complex operations. Therefore a new HMAC was designed as a

compromise between security and performance, using an alternative design approach than the Ed25519_SIGN AOI discussed previously.

The standard HMAC-SHA-256 algorithm follows the following construction as defined in RFC 2104 [3]:

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

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

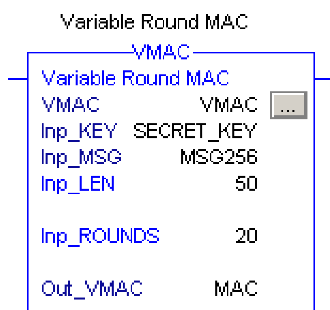

Figure 4. VMAC Add-On Instruction

The proposed Variable Round Message Authentication Code (VMAC) shown in Figure 4 was created as a reduced round derivative of SHA-256. Specifically the number of internal rounds of SHA-256 is reduced from 64 to between 20 and 64 rounds as defined by the parameter "*Inp_ROUNDS"* and the key is mixed into the scheduling of the message using the formulas below. Similar approaches are proposed in [6, 7], however this approach has a more advanced key indexing, rotation, and message scheduling scheme to reduce vulnerabilities to key recover attacks while at the same time allowing for reduced rounds.

Step1: Select one of the eight 32-bit portions of the 256-bit key to become the *IndexedKey* using the algorithm below. Note that if each round from 0 to 63 is designated by the parameter *n*, then the selected portion of the key is:

KeyIndex = n for n=0..7
KeyIndex = n-5 for n=8..12
KeyIndex = 1 for n=13
KeyIndex = 2 for n=14
KeyIndex = 0 for n=15
KeyIndex = (63-n) mod 8 for n=16..63
IndexedKey = Inp_KEY(KeyIndex)

Step2: Calculate the *RotationIndex* of the key, which causes a different rotation amount to be applied each time an IndexedKey value is used:

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

Step3: The key is then mixed into the scheduling process of the message using the following formula:

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

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

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

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

Figure 5. VMAC Timing Results (in ms)

4.3.1. VMAC Vulnerability to Length Extension Attack

The length extension attack works when you hash a data set that looks like:

H (Key || Message) = Hash1

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

H (Hash1 || FakeMessage) = Hash2

This is because

Key || Message || FakeMessage

Is indistinguishable from

Key || SomeDifferentLongerMessage

This assumes that you properly calculate enough padding at the end of the real message before you append the fake data so that you fall on the next block. Thus the construction of most HMACs based on the SHA2 family need to use two hashes that essentially bound the length of the message.

Note that SHA3 avoids this limitation by using a sponge construction and avoids blocks altogether.

VMAC avoids this attack vector because it does not prefix the key as part of the first block of the message, it injects the key into every block of the message. Thus adding an extension to the method would not generate the same result because the FakeMessage above would not have been processed in the context of the key.

4.3.2. VMAC Vulnerability to Collision Attack

Reducing the number of rounds does make the algorithm weaker than a typical HMAC because it increases the chance of collisions. For starters the first 16 rounds of SHA-256 do not utilize the message expansion contained in the remaining rounds, so doing only 20 rounds means that you are only using 4 rounds of expansion versus 48. Therefore testing was conducted on reduced round SHA-256 to determine how reducing the rounds affected the output result when you change one bit in the input. Python code used to test SHA-256 can be found in Appendix A. Standard NIST messages were used as the input, and the table below gives results for changing each bit in the input sequentially over a range of 512 trials per message per test.

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

Figure 6. SHA256 Reduced Round Test Results

The results of the test show that in general you will always have the possibility of changing a large number of bits in the output on average, however you need to use at least 20 rounds to guarantee you will have a significant change every time. Going past 20 rounds you get an exponential decrease in returns with a linear increase in time. However during the message scheduling process the number of input data bits which affect a single bit of W[19] varies from 19 to 26, whereas for W[23] that is increased significantly to 81 to 106. Therefore even for 20 rounds it may be feasible to do a differential input attack trying 2^{19} to 2^{26} inputs, but 2^{81} is likely to be infeasible.

Additionally a number of papers have been published detailing how to create a collision with reduced round SHA-256, some up to 46 rounds. It should be noted that these papers are using standard SHA-256 without a secret key which increases computation complexity assuming the key is generated securely. Therefore, at this time, the author is recommending at least between 24 to 32 rounds as a compromise to meet both security and performance requirements, with the acknowledgement that key changes may need to be more frequent to maintain security. The algorithm does allow for the use of only 20 rounds, but that is for extreme cases only where the choice is between nothing and or 20 rounds and should only be used with caution and frequent key changes.

Reference [6] also notes that when working with a key-less hash function an advisory can work in finding collisions independently of any user or key, which makes brute force attacks more feasible. For a keyed-hash function the attacker needs to get "examples" of the messages from a legal user who already knows the key.

The AOI does include a parameter called "Inp_ROUNDS" that allows a user to increase the number of rounds from a minimum of 20 (minimum of 24 recommended) up to 64 rounds. This will allow for additional flexibility in the future as the computational power increases with minimum impacts to the PLC code. Since all HMACs run in $\Theta(n)$ time an increase to the number of rounds will linearly increase the running time of the algorithm and impact performance.

4.3.3. VMAC Vulnerability to Replay Attacks

A replay attack is an attack in which a valid message to a device is captured and later a copy of the message is sent to the target device. The goal is to take a command or data that was legitimate at the time it was sent, such as a command to stop a motor which was sent intentionally, and be able to replay that command to cause the device (in this case a motor) to respond to that command again at an unexpected time. The traditional defense against a replay attack is to include a timestamp as part of the data used to generate the HMAC, and then ensure that when a message is received it is newer than the last previously received message before even bothering to validate the HMAC.

Control systems typically cannot rely on using time, normally because they don't have access to a time synchronization device and even if they do it creates a security dependency on the device. Therefore it is recommended to have the first two DINTs of the data to be hashed represent a 64-bit number, called a nonce. The nonce is then incremented with each message and represents a time stamp. It is critical that the start value of the nonce always be greater than any previous start value incremented by some pseudo-random number, and that the nonce value is included in the calculation of the VMAC. This is action would not be performed by the VMAC AOI itself, but be performed outside of the VMAC and included as part of the data in Inp_MSG[0] and Inp_MSG[1].

Having the nonce in the beginning of the message, versus some other location in the message, is critical as well. The

SHA-2 family in general works such that data towards the beginning of the message has a greater impact on the final hash then data towards the end. The VMAC algorithm amplifies this with the way the key is used recursively. A control system operating in a steady state condition would, ideally, have very little changes and thus the message itself would have little to no changes in the data. Having a nonce value at the beginning of the message, which changes all the time, would maximize the apparent randomness of the final result.

There is a concern that if the value of the nonce is reset to zero or some value lower than the previous value all subsequent messages will be rejected by other PLCs as a replay attack. This can happen in one of three ways:

1. Someone intentionally resets the nonce
2. Someone re-downloads a program into the PLC
3. Power loss

PLC passwords and operational procedures must be used to prevent someone from resetting the nonce either intentionally or thru a download. The later can easily be prevented by viewing the current value of the nonce before the download and making sure that a value greater than the current value is downloaded to the PLC. Preventing a nonce reset due to a power loss will vary depending on the vendor, however in RSLogix5000 we have found that certain instructions such as accumulators retain their previous values on a power loss.

4.3.4. VMAC Vulnerability to Key Recovery

The heart of the VMAC is a reduced round SHA-256, which can be reversed when the data only consists of one block (i.e. less than 448 bits of data, since padding is always added). This property is a primary reason why implementations of SHA-256 with less than 24 rounds are vulnerable to attacks. The goal of this approach is to generate valid hashes by reversing the algorithm back a number of rounds and then replacing the end of the message with false data, then hashing forward to generate a valid result.

This analysis does not use a key, so additional analysis was performed to determine the vulnerability of VMAC to reversing. The first 16 rounds of the VMAC, and of SHA-256 in general, do not use any message scheduling. Therefore it is easy to reverse the process and potentially recover the key. Testing on a single “non-standard” message block (i.e. a block without the normal padding present in SHA-256 and VMAC) showed that all but 8 bits of the key could be reversed engineered, with the remaining 8 bits found by exhaustive search. While this analysis is limited, it does indicate that it is the rounds past 16 that provide the most security benefit.

When the same approach is performed on rounds past 16 for a single block, the analysis becomes significantly more difficult. The analysis becomes even more complex when multiple blocks of data are being processed. Since padding is always added to the end of a message the message would have to be less than 56 bytes to only consist of only one block. A large message, such as the 800 byte message, would consist of twelve 512-bit blocks plus a smaller 13th block of 256 bits with the remaining being padding.

4.3.5. VMAC Recommendations and Final Thoughts

At the time of this writing a method is not known to recover the key when using at least 24 rounds for a single data block. Therefore it is recommended that at least 32 rounds or more be used for a single block of data, which has an estimated scan time impact of less than 5ms (based on the results in Figure 5 and the fact that the algorithm is $\Theta(n)$). For large messages such as the 800 byte message it is recommended to use 24 rounds or more. There does appear to be some extra security when multiple blocks are involved mainly because of the unknown intermediary stage outputs. However those outputs are used in a simplistic way in SHA-256 to initialize the subsequent stage and then just added on to the result at the end. Therefore it may be theoretically possible that they could be attacked guessing one bit at a time at the output, but at this time we do not know of an actual attack that exists.

Ultimately VMAC represents a compromise between security and performance. The balance between the two is non-trivial, since a PLC’s ultimate job is to control a system and significant scan time impacts might create an unstable control situation. Future PLCs may become fast enough to use the full 64 rounds.

5. RECOMMENDATIONS FOR PLC VENDORS

The bulk of this work was dedicated to creating a cryptographic solution that could run on low power devices like PLCs with high performance requirements. The implementation itself as a proof of concept was developed in ladder logic. However it would be more ideal for the ideas presented in this paper to be pushed down to a firmware implementation. The objective is to create a solution so that the user only has to load the controller with the master public/private key pair and the public keys of the different nodes we want to communicate securely with.

In order to increase the efficiency of future cryptographic algorithms implemented in PLCs, the second recommendation is that vendors add 64-bit processors. The inability to do 64-bit math natively becomes very limiting, especially in the design of an effective HMAC. The ability to perform 64-bit functions would effectively double the amount of data that can be hashed in the same time frame. This would be very helpful for large systems with high throughput requirements.

The third recommendation is for PLC vendors to develop a couple of different TRNGs capable of fitting in an PLC chassis. The vendor will want to offer at least three different options based on different technologies. If vulnerabilities are discovered in one option it might not be possible to correct that vulnerability with a firmware update.

6. FUTURE WORK

The purpose of this work was to be able to protect and verify data integrity between control system nodes. However this work has not defined what an operator should do if a problem is detected. Such decisions are often based on a wide range of factors that require further study with the goal of producing an automated system that can respond intelligently to detected cyber threats. Further work is required to map out these threat profiles and corresponding decision trees, and ultimately to develop the response solution.

7. CONCLUSION

In this paper it has been demonstrated that PLC data authentication and verification is possible without impacting overall scan time. Further work is now required to implement this solution on ICS and to define decision trees to perform automated responses to detected cyber threats.

REFERENCES

[1] K. Fischer, "Control System Data Authentication and Verification Using Elliptic Curve Digital Signature Algorithm." Presented at ASNE Intelligent Ships Symposium X, May 22-23, 2013, Philadelphia, PA.

[2] K. Fischer, "Results and Code for a Software-based Implementation of ECDSA for Control System Data Authentication and Verification." Presented at ASNE Electric Machines Technology Symposium, May 28-29, 2014, Philadelphia, PA.

[3] H. Krawczyk, M. Bellare, R. Canetti, "Request for Comments: 2104, HMAC: Keyed-Hashing for Message Authentication." http://www.ietf.org/rfc/rfc2104.txt Last accessed: February 12, 2017

[4] S. Josefsson, I. Liusvaara, "Request for Comments: 8032, Edwards-Curve Digital Signature Algorithm (EdDSA)." https://tools.ietf.org/html/rfc8032 Last accessed: February 12, 2017

[5] D.J. Bernstein, "The cr.yp.to blog: How to Design an elliptic-curve signature system" https://blog.cr.yp.to/20140323-ecdsa.html Last accessed: February 12, 2017

[6] M. Bellare, R. Canetti, H. Krawczyk, "Keying Hash Functions for Message Authentication." http://cseweb.ucsd.edu/~mihir/papers/kmd5.pdf Last accessed: May 7, 2017

[7] B. Preneel, P.C. van Oorschot., "MDx-MAC and building fast MACs from hash functions." Crypto'95, Springer LNCS vol.963, 1995.

ACKNOWLEDGEMENTS

The author would like to thank Dr. Richard Perry, Villanova University, Associate Professor of Electrical and Computer Engineering, for his support, technical review, and mentorship in this work. The author would also like to thank Paul Gucciardi (NSWCPD C511) for his support in testing the algorithms.

Kenneth A. Fischer, received a BS in Chemical Engineering from the University of Delaware, an MS in Computer Engineering from Villanova University, and is a PhD candidate at Villanova University. Mr. Fischer has over 14 years of automation and controls experience in pharmaceutical, power generation, food and beverage, specialty chemical, and naval applications. He is currently employed with the Naval Surface Warfare Center, Philadelphia Division and is the Lead Engineer for DDG 1000 Machinery Control Systems.

"The views expressed herein are the personal opinions of the author and are not necessarily the official views of the Department of Defense or any military department thereof."

APPENDIX A – PYTHON SOURCE CODE FOR TESTING REDUCED ROUND SHA-256

Section A-1 – sha.py

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

import random, zlib

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

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

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

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

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

# ~x produces negative value

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

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

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

def Sigma1(x):
```

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

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

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

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

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

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

  N >= 16 is required.  N < 64 represents a reduced-round SHA-256.
  """
  W = [0 for t in range(N)]
  # unpack M into array of 16 32-bit values
  for t in range(15,-1,-1):
    W[t] = M & 0xFFFFFFFF
    M >>= 32
  for t in range(16,N):
    W[t] = (sigma1(W[t-2]) + W[t-7] + sigma0(W[t-15]) + W[t-16]) & _mask
  a = _H0[0]; b = _H0[1]; c = _H0[2]; d = _H0[3]
```

```
  e = _H0[4]; f = _H0[5]; g = _H0[6]; h = _H0[7]
  for t in range(N):
    T3 = sigma1(W[t])
    T1 = (h + Sigma1(e) + Ch(e,f,g) + _K[t] + W[t]) & _mask
    T2 = (Sigma0(a) + Maj(a,b,c)) & _mask
    h = g; g = f; f = e; e = (d + T1) & _mask
    d = c; c = b; b = (a + T3) & _mask; a = (T1 + T2) & _mask
  H =            (a + _H0[0]) & _mask
  H <<= 32;  H |= (b + _H0[1]) & _mask;
  H <<= 32;  H |= (c + _H0[2]) & _mask;
  H <<= 32;  H |= (d + _H0[3]) & _mask;
  H <<= 32;  H |= (e + _H0[4]) & _mask;
  H <<= 32;  H |= (f + _H0[5]) & _mask;
  H <<= 32;  H |= (g + _H0[6]) & _mask;
  H <<= 32;  H |= (h + _H0[7]) & _mask;
  return H

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

  measure randomness by size of compressed hash (Z)

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

```
  return (C,Cmin,Cmax,Cavg,Zmin,Zmax,Zavg)

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

# some test inputs and outputs from SHA256ShortMsg.rsp
# http://csrc.nist.gov/groups/STM/cavp/documents/shs/shabytetestvectors.zip
#
_MSG_0 = [ 0x80000000, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
           0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 ]
MSG_0 = block(_MSG_0)
MD_0 = 0xe3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855

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

MSG_440 =
0x3ebfb06db8c38d5ba037f1363e118550aad94606e26835a01af05078533cc25f2f39573c04b6
32f62f68c294ab31f2a3e2a1a0d8c2be5180000000000000001b8
MD_440 = 0x6595a2ef537a69ba8583dfbf7f5bec0ab1f93ce4c8ee1916eff44a93af5749c4

del _MSG_0, _MSG_8
```

Section A-2 – test.py

```
from sha import *

tests = [ "MSG_0, 64", "MSG_8, 64", "MSG_440, 64",
          "MSG_0, 32", "MSG_8, 32", "MSG_440, 32",
          "MSG_0, 24", "MSG_8, 24", "MSG_440, 24",
          "MSG_0, 16", "MSG_8, 16", "MSG_440, 16",
          "MSG_0, 17", "MSG_8, 17", "MSG_440, 17",
          "MSG_0, 18", "MSG_8, 18", "MSG_440, 18",
          "MSG_0, 19", "MSG_8, 19", "MSG_440, 19",
          "MSG_0, 20", "MSG_8, 20", "MSG_440, 20",
          "MSG_0, 21", "MSG_8, 21", "MSG_440, 21",
          "MSG_0, 22", "MSG_8, 22", "MSG_440, 22",
          "MSG_0, 23", "MSG_8, 23", "MSG_440, 23" ]

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

APPENDIX B – POINT MULTIPLICATION SEQUENCE CHART

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