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

Dr. Ken Fischer, PhD

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

Abstract—*The challenge of securing industrial control systems is significant and previous work provided solutions to a number of these challenges 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 previous works the Variable-round Message Authentication Code (VMAC) algorithm was introduced for per-message data authentication, and a scheme was presented for two nodes to exchange a symmetric key for VMAC.*

This work expands upon previous work by introducing the Key Exchange Protocol (KEP), which allows for generation and distribution of symmetric keys for use in multicast implementations of VMAC. KEP is capable of being configured into multiple tree configurations to increase the efficiency of key distribution, but also provides for redundancy in case the root node is taken offline. This work also provides additional VMAC proof of security and VMAC implementation details.

A proof of concept for VMAC and KEP was then created and tested using four 1756-L83 Rockwell Automation processors. KEP was show to have an average scan time impact of 10ms during a key exchange with a minimum impact of less than 1ms when IDLE and a maximum impact of 20 ms if verifying and creating digital signed messages at the same time.

Index Terms—*PLC, HMI, ICS, multicast, authentication, cybersecurity*

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]. Implementing message authentication for Industrial Control System (ICS) data and commands would be a significant step forward in providing security for these systems, but the process is complicated by a number of factors which were described in [3]. Solutions were presented to these challenges, however those works were based around traditional ICS implementations which use unicast technologies. This paper expands upon the previous work by introducing a new Key Exchange Protocol (KEP) that will generate and distribute symmetric keys for use in the Variable-round Message Authentication Code (VMAC) [3] or equivalent algorithm for per-message data integrity verification. Section 2 of provides background information on previous work and the challenges related to multicast. Section 3 expands previous work [3] to provide additional details related to the security and implementation of VMAC. Section 4 introduces the KEP algorithm and provides details for its implementation and use. Section 5 describes future work.

2. BACKGROUND

Previous works provided general solutions to the underlying issues of ICS message authentication such as processing speed and how to perform BigInteger calculations on a PLC, which ultimately concluded with a theoretical way to perform Edwards-curve digital signature creation and verification on a PLC. It also presented a solution for securing communications between two devices, such as a PLC and a gas turbine's Full Authority Digital Controller (FADC) across an unsecure channel, in which a Elliptic Curve Diffie-Hellman (ECDH) Key Exchange would be performed to establish a shared symmetric key that then would be used by a Hashed Message Authentication Code (HMAC) algorithm to provide per message data authentication and verification. The work then described how the standard HMAC algorithm would run too slow for a PLC to be able to execute, and an improvement was required. The Variable-round Message Authentication Code (VMAC) algorithm was introduced as a replacement for VMAC, and the details of that algorithm are fully defined in [3].

Historically control systems data has been communicated between devices via unicast messages. Examples of this include an ICS consisting of just one PLC and one HMI, or a system consisting of multiple PLCs that unicast to a PC-based server node which then communicates to multiple

HMIs via unicast. However as systems have grown and become more redundant the use of multicast technologies has increased dramatically, particularly when there is a large number of different controllers communicating with a large number of consoles. The benefits of multicast are numerous from a networking perspective, particularly in being able to add nodes to an already established infrastructure. From a controls engineering perspective, this means that additional consoles can be added to a system without any impact to the end controllers, significantly reducing the testing and validation efforts.

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

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

Most secure multicast schemes follow some variant of the Iolus Framework for Secure Multicasting [4]. The framework essentially establishes a secure distribution tree, designed to communicate a symmetric key between the nodes which is used to secure the multicast data. It introduces the concept of different security subgroups, with each group having a Group Security Controller (GSC) which manages the top level subgroup and a Group Security Intermediary (GSI) which manages each of the other subgroups. Both GSCs and GSIs are known as Group Security Agents (GSA). The idea is that these entities form subgroups which then work together to deliver the multicast traffic to all of the subgroups in the overall group. At the root of the tree is the GSC which is ultimately responsible for the entire group. Typically these kinds of frameworks use a symmetric key called Kgroup to encrypt some new Kgroupnew, and then transmit the new key.

In many ways this approach is similar to the Group Key Management Protocol described in RFC 4046 [5], but since it is a tree it is potentially more scalable since all of the nodes will not have a reason to contact the root GSC at the same time which can cause an out-of-sync implosion. However, one of the issues with Iolus is that a node that wishes to join the group would have to first locate a GSA, particularly in a scenario where a GSA might be dynamic. Solutions that have been proposed to solve this problem primarily involve the node performing some kind of lookup against a directory service, however devices like PLCs do not have the processing power to perform such a lookup. The framework also requires that when a member joins or leaves a group a rekeying of the entire group will need to take place.

Some proposals such as found in [6] involve creating a virtual binary tree from the root node and only require rekeying between the end node and the path towards the root node which improves the overall efficiency of the algorithm to O(log n). In general the primary problem that these works are attempting to solve is how to keep the group secure such that an old node can't just access data within the group without properly joining the group, and to effectively boot the node out of the group. As a result significant overhead must be assigned towards joining and leaving the group to ensure security before the nodes can talk. For an information technology system this makes sense, however for operational systems such as control systems this presents significant problems. A control system needs to be able to communicate as fast as possible when it powers on, and delays in that could have unintended operational consequences. Additionally, control system nodes do not just go on and offline like a typical information system node. The control systems are generally fairly static, and changes to them have to go thru rigorous testing processes for Installation Qualifications (IQ), Operational Qualifications (OQ), and finally Process Qualifications (PQ). Therefore if a node appears to be joining and leaving a group rapidly it is likely due to either equipment or power failure and control system operators will want to restore operation of the node as fast as possible.

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

The Iolus framework even includes an expiration for the keys in the framework, requiring that nodes reach out to a GSA to get an updated key.

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

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

3. VMAC – Additional Details

Previous work [3] provided the VMAC algorithm along with information on its security against collision and length extension attacks. However that work did not provide information on VMAC's security to key reversal attacks or implementation details with regards to what must be included in a VMAC to protect against various forms of replay attacks. This section expands that previous work to cover these gaps.

3.1. VMAC Key Reversal Attack

An analysis of the VMAC was performed to determine how difficult it would be to extract the key. The initial assessment involved looking at a single input data block at 16-rounds, which is much simpler since it avoids any of the data bit rotations in the SHA-256 message schedule. To simplify the analysis the input data that was used did not conform to the SHA-256 padding requirements, which are implemented in VMAC. In order to reverse out a round, first note that at the end of a round the intermediary hash values are available as shown below

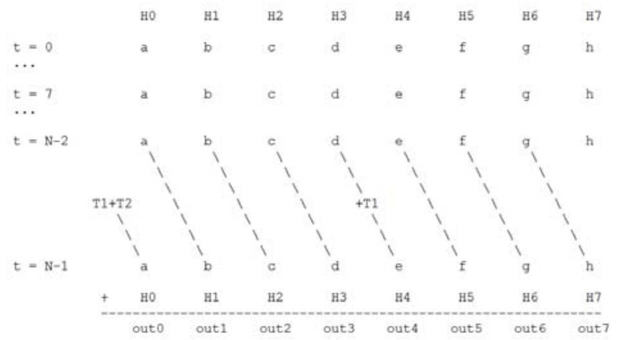

Figure 1. SHA Intermediary Hash Values

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

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

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

- $T1 = h + ConditionalFunction + Kt + Wt$
  $h = T1 - ConditionalFunction - Kt - Wt$

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

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

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

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

The next assessment was done looking at a single input block at 17-rounds. The idea was similar to the 16-round crack, where if you can determine h then you could unwind the hash one more stage and find all of the key bits. At 17

rounds though, the message schedule takes affect which means that W[16] is no longer a copied value with a trivial key mix, and the formula expands to:

$$V = h + (((D \text{ XOR } E) + X) \text{ XOR } Y)$$

where

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

and

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

If we can determine **h** then we could unwind the hash one more stage and find all of the key bits. **Data[9,1,0]** affect **h** and **X**, but changes in **D** do not change **h**, **E**, **X**, or **Y**, so flipping bit **i** of **D** will flip bit **i** of **V**. Although the change in bit **i** provides no information itself, the addition with **X** may flip carry bits, and bit **i+1** of ((**D** XOR **E**) + **X**) will change only if bit **i** of **X** is 1. However we can only observe **V**, and the addition with **h** may also flip carry bits. So if bit **i+1** of **V** changes then bit **i** of either **h** or **X** is 1, but we don't know which. Therefore we would have to continue without knowing **h** by running the approach for cracking 16 rounds for each of the possible 232 values of **h**. Each of those produces 28 candidate keys so the overall search space is O(240). For each value of **h**, Crack16 invokes VMAC twice for each of 256-8 = 248 bits, that is 2*248 = 496 invocations. At a scan rate of 100msec, i.e. 10 invocations per second, that would take over 6000 years:

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

Extrapolating to N=20 rounds, there would be 4 unknown h values associated with each candidate key, so the overall search space is 28*(232)4 = O(2136). At a scan rate of 100msec that would take over 5.35e32 years.

3.2. VMAC Implementation Details

A secure implementation consists of two parts: the generation of outgoing messages and the processing of incoming messages. A secure implementation also addresses the following concerns:

1. Protection against the standard replay attack, where a message between two nodes is saved and then later played again
2. Protection against a variant of the replay attack where a message generated by one node intended for a second node is captured and played to a third node
3. Determination of the key used in the VMAC algorithm, either Ks or Ks', which is changed by the KEP and is used to provide a bumpless transfer during a key change

3.2.1. Outgoing Messages

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

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

	31..24	23..16	15...8	7...0	Comments
DINT0	Counter				Initialized to number of microseconds since epoch time on power up and incremented by 1 for each new VMAC
DINT1					
DINT2	Node ID				Represents the sender node
DINT3	Destination ID				Represents the destination node or multicast group
DINT4	KeyID (only used with KEP)				Represents the ID of the key used to generate the VMAC
DINT5					
DINT6	Data DINT1				Data
DINT7	Data DINT2				
DINT8	Data DINT3				
DINT9	Data DINT4				
DINT10	Data DINT5				
DINT11	Data DINT6				

Figure 2. VMAC Message Structure

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

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

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

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

3.2.2. Incoming Messages

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

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

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

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

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

that sometimes a system component might need to run in a "standalone" mode due to equipment damage, plant maintenance, cyber threats, or a number of other reasons.

4. Key Exchange Protocol

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

Section 2 described the main approaches to generating and handling the symmetric key in multicast groups, but these approaches have additional complications when applied to control systems. The limited processing power is the primary challenge which impacts each of the multicast solutions in a number of ways, primarily that they cannot process the number of steps that would be required to update keys each time group membership changes while at the same time meeting performance requirements. Additionally there are significant challenges that stem from how message structures and send rates work since most of these algorithms require that the system send out specialized "one-time" messages as part of the group join and leave processes. An implementation of these existing algorithms which must transmit the control messages used to exchange the symmetric key would likely suffer performance degradation.

Thankfully, the design of control systems does mean that the requirements for multicast messaging are more limited than the requirements for the generalized multicast group scenarios that are addressed in [4, 5, 6]. The rate of group membership changes is relatively low and is primarily the result of equipment or power failures where there is a desire to re-join the node to the group as speedily as possible. This eliminates the need to change the group key solely based on group membership changes which is a primary driver for the approaches taken in the literature. However this does add a unique challenge in that a node must be capable of being restored to the group in near instantaneous time even during the middle of a group key change. Even an algorithm capable of updating a key with an O(log n) efficiency is not going to be fast enough given the inherent limitations of processing speed. This is commonly referred to in control systems engineering as a "bumpless transfer" and is not addressed in the literature simply because it is not a requirement for an information system.

To solve these challenges the Key Exchange Protocol (KEP) for control systems was developed. The primary purpose of KEP is to securely generate and transmit a Ks', which serves as the primary shared symmetric key, along with the identity number of the Ks' and the Ks currently in use. These identity numbers are 64-bit numbers that represent the number of microseconds since epoch time at the time these were generated. During a key exchange a new Ks' is created and distributed, however the old Ks is used in VMAC until a node confirms that all nodes it communicates with have received Ks' via the identity number. The nodes then begin to transmit messages using Ks' vice Ks. When a node receives valid VMAC messages from each node, which were generated using Ks', the node sets Ks equal to Ks'.

KEP consists of two parts, the Listener and the Processor, that run in parallel with each other. The Listener is responsible for receiving messages from other nodes and verifying the messages of these nodes. The Processor is a state-machine, shown in Figure 3, which uses verified messages from the Listener along with internal data to perform the bulk of the KEP logic. KEP operates using a client-server model, however unlike the Iolus framework which uses GSAs, each node in KEP can act as both a client and a server depending on the state of the node. This is done by each node having an internal "priority table" for all of the other nodes. A node finds the highest priority node that is advertising that it has a Ks' available for distribution and selects that node as its server to obtain the key. It then goes through an Ed25519 ECDH process to generate a one-time symmetric key with the server node, with that key known as Kp. Kp is then used to encrypt a copy of Ks', with the encrypted key known as Ke. Ke is then transmitted from the server node to the client node and subsequently decrypted using the client's copy of Kp to obtain Ks'.

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

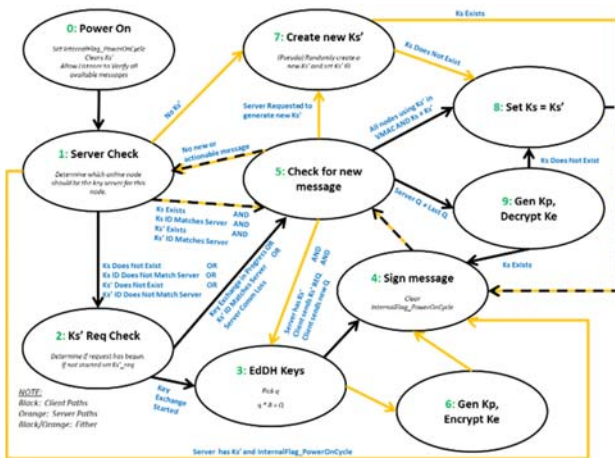

Figure 3. KEP State Machine

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

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

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

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

5. Results and Future Work

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

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

Ultimately the biggest scan time impacts come from VMAC itself. Previous work [3] provided the timing results for VMAC using various rounds. For the proof-of-concept 20 rounds were used with a message length of 100 DINTs, or 400 bytes. Produce/consume tags have a maximum length of 500 bytes so the VMAC data consumed 400 of the 500 bytes. The remaining 100 bytes was reserved for some overhead such as the VMAC itself (32 bytes), flags indicating the number of rounds and the length of the

message (8 bytes total), and information on the connection status which is a generic part of using produce/consume tags in a Rockwell PLC. With each node generating one VMAC message and processing 3 incoming messages the total scan time of the program (including all the other I/O handling for none-VMAC data and all the implementation details described in Section 4.4) was between 51.7 ms and 52.1 ms.

This result is interesting because the results indicate that running four VMACs containing 400 bytes each should take a total of approximately 51.6 ms, indicating that the overwhelming majority of the run time is spent running VMAC itself. Therefore the implementation details provided in Section 3.2 have a negligible impact on overall system performance but are critical for providing overall security. The result also suggests that if additional alarms or monitoring capability were to be added that capability would have a negligible impact on performance.

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

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

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

REFERENCES

[1] 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] K. Fischer, "Advancements in Control System Data Authentication and Verification." Presented at ASNE Intelligent Ships Symposium 2017, May 25-25, 2017, Philadelphia, PA.

[4] S. Mittra, "Iolus: A Framework for Scalable Secure Multicasting." (http://conferences.sigcomm.org/sigcomm/1997/papers/p113.pdf) Accessed: 7 March 2018

[5] M. Baugher, R. Canetti, F. Lindholm, "RFC 4046: Multicast Security (MSEC) Group Key Management Architecture" (https://www.ietf.org/rfc/rfc4046.txt) Accessed: 7 March 2018

[6] L. Dondeti, S. Mukherjee, A. Samal, "DISEC: A Distributed Framework for Scalable Secure Many-to-many Communication" (http://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1030&context=cseconfwork) Accessed: 7 March 2018

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.

Kenneth A. Fischer, received a BS in Chemical Engineering from the University of Delaware, and an MS and PhD in Computer Engineering from Villanova University. Dr. Fischer has over 15 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 – CRACKING VMAC 16-ROUNDS

/***

File: crack16.py

***/

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

import sha, random

N = 16

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

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

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

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

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

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

# R will hold the h bits
#
R = 0

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

# masks to set bit 0 of Data high or low
#
ones  = (1 << 512) - 1 # 11...11
mask1 = 1              # 00...01
mask0 = ones ^ mask1   # 11...10

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

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

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

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

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

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

  V0 = (V0 - R) & sha._mask

  V1 = (V1 - R) & sha._mask

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

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

  # get r = bit n of h, and k = bit n of Key0
  #
  # bits: i = input data, k = key, (v1,v0) = (carry, V bit n)
  #
```

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

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

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

    k = v0 ^ r # key bit

    # print( "k =", k)

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

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

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

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

  # sample runs:
  #
  #  Key0 = 10010101111110101000100010001 0011
  #    K0 = 00010101111110101000100010001 0011
  # hprev = 10010001110011101110001100001011
  #     R = 00010001110011101110001100001011
  #
  #  Key0 = 11001100000110000011101111101 1010
  #    K0 = 01001100000110000011101111101 1010
  # hprev = 00001011000100000001101010010000
  #     R = 00001011000100000001101010010000
  #
  #  Key0 = 00110100000010110000100010010010
  #    K0 = 00110100000010110000100010010010
  # hprev = 11100011110111001011001101110100
  #     R = 01100011110111001011001101110100
  # --------^
  # this bit is not determined here
```