2022 Seventh International Conference on Fog and Mobile Edge Computing (FMEC) | 979-8-3503-3452-4/22/$31.00 ©2022 IEEE | DOI: 10.1109/FMEC57183.2022.10062612

Fine-Tuned Access Control for Internet of Things

Luke Jakielaszek*, Xuening Xu†, Xiaojiang Du†, and E. Paul Ratazzi§

*Department of Computer & Information Sciences, Temple University, Philadelphia, PA, USA
†Department of Electrical & Computer Engineering, Stevens Institute of Technology, Hoboken, NJ, USA
§Air Force Research Laboratory, Information Directorate, Rome, NY, USA
Emails: luke.jakielaszek@temple.edu, xxu64@stevens.edu, dxj@ieee.org, edward.ratazzi@us.af.mil

***Abstract*—Security within the internet of things suffers from balancing power consumption and memory usage in devices. We propose two protocols that aim to reduce these strains while maintaining effective security through symmetric keys. Our first protocol takes a user-level approach for fine-tuned access control over advanced operations between a device and its installed software. Our second protocol allows for flexibility between the energy and memory tradeoff for network designers along with a dynamic bootstrapping mechanism for an already established network.**

***Index Terms*—IoT, access control, energy efficient**

I. Introduction

The internet of things (IoT) has seen great development in recent years and has been widely adapted throughout modern society for civil, health, and militaristic means [2]. One of the major applications of IoT has been the area of wireless sensor networks (WSNs), which has been studied over the years [5], [7], [21]. However, the devices within these networks are often limited in functionality due to their power and memory constraints. These constraints limit both the internal and external security of WSN devices, allowing for breaches in both the confidentiality and authenticity of information gathered from a WSN [14]. Several previous works designed protocols that consider the constraints and can improve the performance of WSNs [4], [6]. However, it is still a challenge regarding how to achieve security for resource-constraint IoT.

The limitations of these devices can negatively impact their internal security, reducing their ability to screen installed software. This limited evaluation allows software to abuse the device through unnecessary and dangerous permission ratification. These acquired permissions could lead to the potential access and dissemination of privileged information, causing serious harm to those depending on the confidentiality of the IoT device [22]. In addition, these limitations can adversely impact the external security of WSNs, allowing for the WSN to be vulnerable to both passive and active attacks, inhibiting the confidentiality and authenticity of communication between nodes [3].

To combat the burdens brought on by the power and memory constraints of these devices, we propose two symmetric protocols. The first deals with the internal security of a device and its software, while the second deals with external security between nodes in a WSN. Our internal software protocol dispatches a unique secret key, which is constructed from an approved set of permissions, to an installed software. The external protocol makes use of a set of three one-way key functions to securely bootstrap nodes into a WSN, while also providing secure communication within a WSN through a top-down methodology.

II. Design and Implementation

Both our single node and multi node protocols operate using symmetric keys in combination with the AES encryption algorithm and the SHA256 hashing algorithm. The AES encryption algorithm was chosen due to its low-memory overhead, relatively fast total encryption time, and robustness in security [19]. The SHA256 hashing algorithm was selected for both basic hashing as well as HMAC generation due to its computational efficiency and reliability in security [8].

A. Single Node Protocol

The single node protocol securely authorizes requested software permissions over the application layer. This protocol is designed as a user-space program to allow for more fine-tuned control over installed software and their actions. Our single node protocol was developed to function within the architecture of a microkernel, such as seL4, which relies on user-level system composition for functionality [18]. The operation for this protocol is outlined in a sequential manner within Table I. To avoid replay attacks all messages are encrypted with a timestamp, which is always checked for timeliness and redundancy.

The protocol assumes that both the installed software (agent) and dominant process (authorizer) share a secret key. In addition, the authorizer has its own master key. The agent first must initialize itself within a device by requesting a set of base permissions. This request is encrypted by the shared secret key and sent to the authorizer. Upon receiving the request, the authorizer decrypts the message and validates the requested permissions, timestamp of the message, and ID of the agent. The authorizer then creates an HMAC using SHA256 with its master key and the approved permission vector. This hash is then encrypted along with the current time. The authorizer stores the computed hash along with the approved permissions. The encrypted hash is sent as a response to the agent and is decrypted and stored for future encryption.

When the agent needs to perform an action requiring permissions, it sends an action request to the authorizer. The

TABLE I
SINGLE NODE EXECUTION ORDER

Time	Authorizer	Agent	Description
T_0	K_m, K_1	K_1	The authorizer and agent are initialized with shared secret keys. The authorizer also has a master key.
T_1	-	Req(AES_{K_1}(Perms, ID, time))	Each agent sends an encrypted permission request to the authorizer.
T_2	Perms, ID, time	-	Authorizer decrypts each request.
T_3	isValid(Perms)	-	Authorizer performs a validity check to see if the agent should be granted the requested permissions. The isValid() function is incomplete as it is based on our unfinished deep learning classifier.
T_4	$SHA256_{K_m}$(ID, Perms) = K_{App}	-	The authorizer generates a hash which serves as the secret key for Node. The authorizer then stores the requested permissions & corresponding K_{Appi}.
T_5	Response(AES_{K_1}(K_{App}, time))	-	The authorizer returns the generated secret key wrapped in encryption.
T_6	-	AES_{K_1}(K_{App}, time)	The agent receives the encrypted hash.
T_7	-	K_{App1}, time	The agent decrypts the wrapper to retrieve the generated hash.
T_8	-	(ID, MessageType, $AES_{K_{App}}$(time, action, $Perms_{desired}$), HMAC)	The agent sends a message using the generated hash to encrypt a requested action and the required permissions. An HMAC using $SHA256_{K_{App}}$ is generated to preserve the authenticity of the plain text.
T_9	ID, time, action, $Perms_{desired}$	-	The authorizer checks the authenticity of the HMAC and decrypts the message to retrieve the contents using its stored K_{App}.
T_{10}	$Perms_{desired} \leq Perms_{orig}$	-	The authorizer checks if the needed permissions are a subset of the approved initialization permissions.
T_{11}	Perform Action	-	If the permissions are valid, the root performs the requested action for the agent.
T_{12}	$AES_{K_{App1}}$(Approval, time)	-	The authorizer creates encrypted response to notify the agent that the requested action was approved.
T_{13}	-	Approval	The agent decrypts the response and checks whether approval was given.

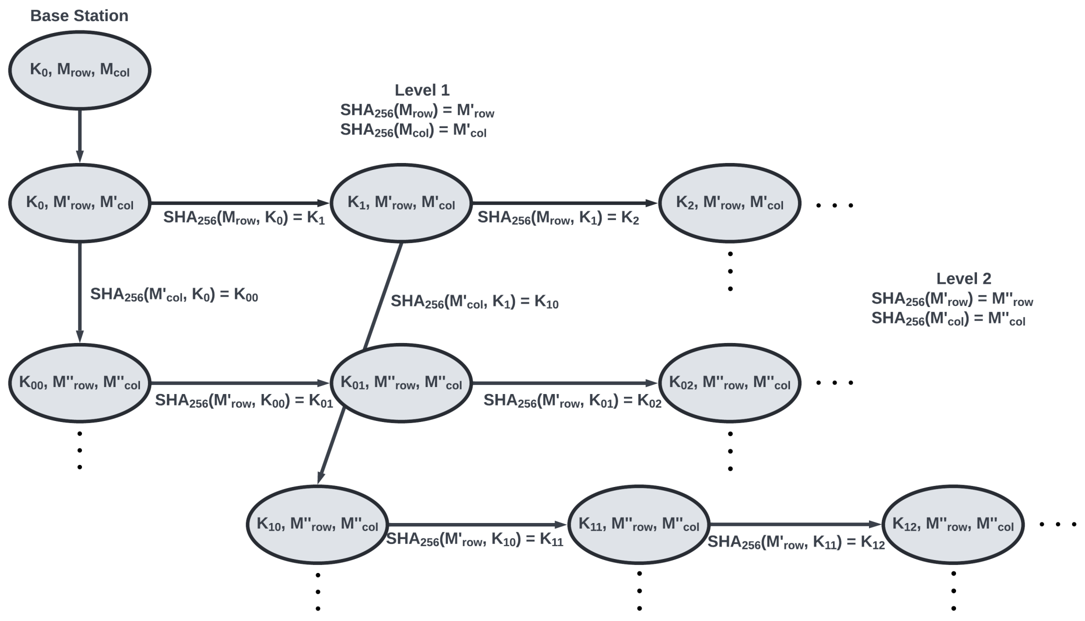


Fig. 1. Keychain generation visualized.

action request consists of the agent's ID and message type in cleartext along with the time, desired action, and needed permissions, which are encrypted using the stored hash and AES algorithm. This is wrapped using a computed HMAC to preserve authenticity of the clear text. This message is sent to the authorizer who then checks the authenticity and decrypts the message using the stored hash. The desired permissions are compared to the original approved permissions. If they are a subset, the authorizer performs the action for the agent or instructs a privileged process to perform the action. The authorizer then notifies the agent of the approval status of the action with an encrypted Boolean response and timestamp.

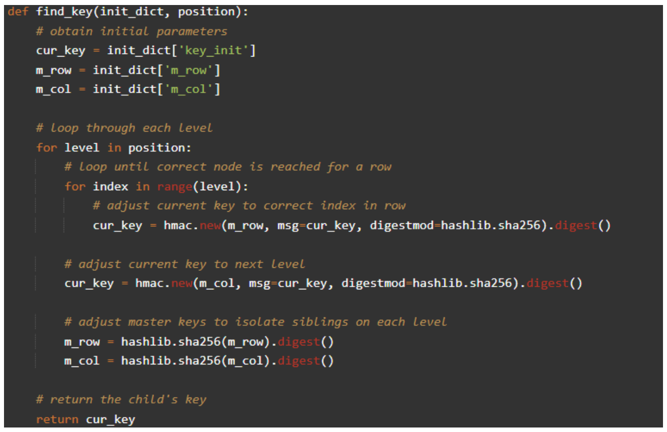
B. Multi Node Protocol

The multi node protocol dynamically bootstraps nodes into an established wireless sensor network with secure symmetric communication capabilities. This protocol is designed to take advantage of the common tiered structure of distributed WSNs i.e. clustering of nodes and formulated cluster heads [11]. This protocol allows for flexibility in the tradeoff between memory use and power consumption. This tradeoff can be shifted by restricting the number of child keys cached within a parent node. A step-by-step execution of the protocol can be seen in Table II.

The multi node protocol depends on three separate one-way keychains in implementation. The main keychain, as specified in Table II by K_0, serves as the base secret key that will be transformed into all child secret keys. The secondary keychains, as denoted in Table II by M_{row} and M_{col}, allow for the isolated manipulation of the base secret key across children on different levels within the network. The keychain generated by M_{col} allows for a parent to generate secret keys

TABLE II
MULTI NODE EXECUTION ORDER

Time	Ancestor	Descendant	Description
T_0	K_0, M_{row}, M_{col}	K_0, M'_{row}, M'_{col}	The parent is initialized with the first key for each of the three keychains. Each successive node in the network receives their own initialization parameters that are determined by the hashing of the original chain and their location within the network.
T_1	-	SHA256(K_0, position, time) = $HMAC_1$	The child generates an HMAC using its secret key, position in the network, and a timestamp.
T_2	-	Validation(position, time, $HMAC_1$)	The child constructs a validation request for its parent containing its position and the timestamp with the HMAC appended.
T_3	Find_Key(position, K_0, M_{row}, M_{col}) = K_Gen$_0$	-	The parent uses its base parameters along with the disclosed position of the child to generate the child's secret key. The parent can choose to cache this key based on memory and power constraints.
T_4	SHA256(K_Gen$_0$, position, time) == $HMAC_1$	-	The parent checks the authenticity of the child by hashing the received parameters and validating that they match the supplied HMAC.
T_5	SHA256(K_Gen$_0$, validity, time) == $HMAC_2$	-	The parent generates an HMAC using the generated secret key, validity of the child's request, and a timestamp.
T_6	Validation(validity, time, $HMAC_2$)	-	The parent constructs a validation response for its child containing the determined validity and the timestamp with the HMAC appended.
T_7	-	SHA256(K_0, validity, time) == $HMAC_2$	The child checks the authenticity of the parent by hashing the received parameters and validating that they match the supplied HMAC. If they match, then the parent and child have now established a secure line of communication.
T_8	-	SHA256(K_0, data, time) = $HMAC_3$ OR AES_{K_0}(data, time) = E_message	The child creates a message from collected data using SHA256 hashing or AES encryption depending on the sensitivity of the data.
T_9	-	Report(data, time, $HMAC_3$) OR Report(E_message)	The child sends either the hashed or encrypted data as a report to its parent.
T_{10}	SHA256(K_Gen$_0$, data, time) == $HMAC_3$ OR Dec(E_message) == data, time	-	The parent receives and validates/decrypts the hashed/encrypted message using the cached key. The data is now authenticated and ready to be acted upon.
T_{11}	SHA256(K_Gen$_0$, OK, time) = $HMAC_4$	-	The parent constructs a HMAC using the child's secret key, indication of whether the child can continue sending, and a timestamp.
T_{12}	Continue(K_Gen$_0$, OK, time) = $HMAC_4$	-	The parent appends the HMAC to the message and sends it to the child when ready. This step is used for flow control as well as indicating to the child whether they should re-instantiate a connection with the parent to reestablish a secret key.
T_{13}	-	SHA256(K_0, OK, time) == $HMAC_4$	The child receives the message and validates the HMAC's authenticity. Based on the response, the child will send messages or reestablish its secret key.



```python
def find_key(init_dict, position):
    # obtain initial parameters
    cur_key = init_dict['key_init']
    m_row = init_dict['m_row']
    m_col = init_dict['m_col']

    # loop through each level
    for level in position:
        # loop until correct node is reached for a row
        for index in range(level):
            # adjust current key to correct index in row
            cur_key = hmac.new(m_row, msg=cur_key, digestmod=hashlib.sha256).digest()

        # adjust current key to next level
        cur_key = hmac.new(m_col, msg=cur_key, digestmod=hashlib.sha256).digest()

        # adjust master keys to isolate siblings on each level
        m_row = hashlib.sha256(m_row).digest()
        m_col = hashlib.sha256(m_col).digest()

    # return the child's key
    return cur_key
```

Fig. 2. Keychain generation code example.

corresponding to specific generations below them. In Figure 1, each generation is labeled as a separate level. The keychain generated by M_{row} allows for a parent to generate secret keys corresponding to siblings within the same generation. These siblings are within the same level as depicted in Figure 1. The combination of M_{row} and M_{col} isolate sibling nodes within a network, allowing only a direct ancestor to generate a descendant's key. To generate a descendant's key, the ancestor feeds its initialization parameters, along with the child's relative location, through the code snippet seen in Figure 2. The modified base keys for the three keychains are stored within

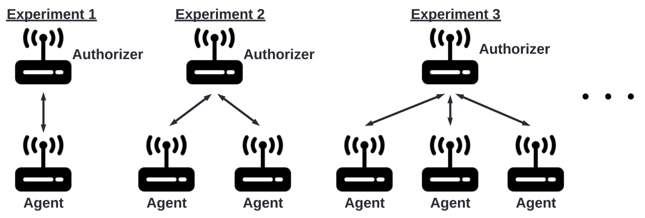


Fig. 3. Single node process architecture.

the descendant for the bootstrapping process. This protocol assumes that a third party has access to the initialization parameters of all three keychains within the base station. Therefore, this party can add new nodes to an established network by simply generating their initialization file.

To bootstrap into the WSN, a child node constructs a HMAC, using the secret key K_i found in its initialization file, for a message containing its position in the network and a timestamp. This message and HMAC are sent to its suspected parent. Upon receiving the message, the parent ensures that the disclosed location of the child is valid and that it can be a descendant. Next, the parent will obtain its own secret keys, K_j, M^j_{row}, and M^j_{col} and use the child's relative disclosed location in the WSN to generate the child's secret key. This process of obtaining a child's key can be seen below in Figure 2. Once the child's key is generated, the parent validates the HMAC

TABLE III
SINGLE NODE EXECUTION RATES

	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6
Number of Processes	2	3	4	5	6	11
Average Execution (s)	0.004383611	0.005209468	0.007668551	0.010009681	0.012502742	0.024012095
Min Execution (s)	0.003637075	0.0037148	0.003685951	0.003690004	0.00365591	0.003652811
Max Execution (s)	0.026875973	0.032971859	3.156201839	7.183223009	3.151322126	7.218795061
Total Execution	46,000	79,000	75,000	76,000	67,500	80,000
Bits Transferred	117,129,984	201,139,256	190,934,528	193,525,296	171,896,000	203,718,648
Run Time (s)	202.5455861	206.593955	192.2491777	190.5893332	169.073019	192.2684983
Effective Throughput (b/s)	578,289.49	973,597.01	993,161.74	1,015,404.65	1,016,696.82	1,059,552.92
Executions Per Second	227.1093677	382.3926019	390.1187038	398.7631349	399.2357882	416.0848017

TABLE IV
SINGLE NODE ENERGY & MEMORY CONSUMPTION

	Processes	VIRT (KB)	RES (KB)	SHR (KB)	%CPU	ENERGY CONS. (W)	%MEM	MEM USAGE (KB)	TIME+ (S)
Exp 1	2	26,360	17,332	11,440	106.4394	0.612026806	1.8	17,069	206.08
Exp 2	3	40,260	26,313	17,142	198.7411	1.142761389	2.702222	25,625	380.99
Exp 3	4	54,160	35,233	22,709	211.0277	1.213409235	3.620169	34,330	384.72
Exp 4	5	68,060	44,168	28,448	220.2258	1.266298626	4.5	42,674	401.85
Exp 5	6	81,960	52,884	33,848	229.0062	1.316785426	5.4	51,208	401.41
Exp 6	11	151,460	97,336	61,789	251.2487	1.444679851	10.01999	95,020	442.44

TABLE V
SINGLE NODE PACKET STATISTICS

	Processes	Total Packets	Time (s)	Average PPS	Average packet size (B)	Total Bytes	Average kB/s	Average kb/s
Exp 1	2	477,793	192.816	2,478	97	46,165,970	239	1,915
Exp 2	3	755,238	190.938	3,955	98	74,103,494	388	3,104
Exp 3	4	759,423	187.617	4,048	100	76,192,395	406	3,248
Exp 4	5	780,231	193.766	4,027	100	78,317,587	404	3,233
Exp 5	6	763,282	186.297	4,097	100	76,634,012	411	3,290
Exp 6	11	829,648	203.487	4,077	101	83,530,582	410	3,283

of the child's message, proving the legitimacy of the child.

Next, the parent caches the child's secret key, and responds to the child with its own message and HMAC. This HMAC was generated using the child's key, so that the child may also validate the legitimacy of the parent. A secure line of communication has been established once both the parent and child have validated each other.

Now that both nodes have been authenticated, the descendant can forward collected sensor data to the parent. This data can be sent using either AES encryption or a SHA256 HMAC depending on the sensitivity of the collected information. Regardless of the form, the parent will always respond with a message and HMAC after receiving and processing the data. This response serves to notify the child of successful correspondence and to regulate the flow of incoming data from the parent's children.

III. PERFORMANCE EVALUATION

Both the single node protocol and the multi node protocols were evaluated on the Raspberry Pi 3B over a set of six experiments with varying network architectures. Linux's top command was used to measure statistics such as the total execution time, energy consumption, and memory usage. Wireshark, an open source packet capture application with advanced capabilities and statistical analyses of packet transfer, was used to gather various measurements regarding packet delivery and bitrate [15].

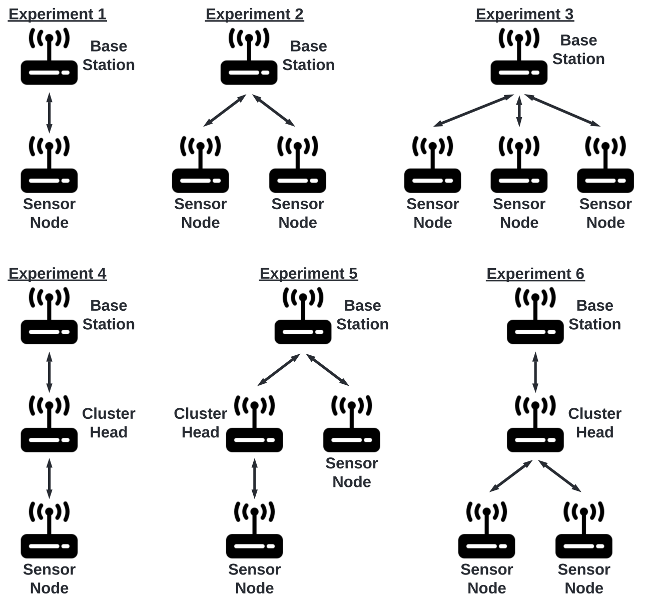


Fig. 4. Multi node network architecture.

Table III and Table VI display the execution rates for their corresponding protocol. For our single node protocol, an execution is measured after the bootstrapping phase, i.e. timestep eight to thirteen within Table I. For the multi node protocol, an execution is also measured after bootstrapping:

TABLE VI
MULTI NODE EXECUTION RATES

	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6
Number of Processes	2	3	4	3	4	4
Average Execution (s)	0.009273741	0.009830033	0.00991102	0.009742714	0.011328294	0.011487879
Min Execution (s)	0.003154039	0.003211975	0.003166914	0.003123045	0.00323391	0.00324297
Max Execution (s)	0.875487089	0.930810213	0.979777098	0.913099051	1.417691946	1.611480951
Total Execution	25,500	54,000	66,000	44,000	66,000	66,000
Bits Transferred	29,532,592	62,551,848	763,762,24	50,948,240	76,456,792	76,453,344
Run Time (s)	236.7227559	265.6658971	218.2473479	214.5581549	249.4259726	252.9443073
Effective Throughput (b/s)	124,756.03	235,453.06	349,952.59	237,456.55	306,531.00	302,253.67
Executions Per Second	107.7209494	203.2628222	302.409173	205.0726061	264.607568	260.9270029

TABLE VII
MULTI NODE ENERGY & MEMORY CONSUMPTION

	NODE	VIRT (KB)	RES (KB)	SHR (KB)	%CPU	ENERGY CONS. (W)	%MEM	MEM USAGE (KB)	TIME+
Exp 1	**bm_1**	41,620	8,892	5,648	14.1511111111	0.08	0.9	8,535	0:32.52
	b_1m	17,212	9,564	6,104	18.2172222222	0.10	1.0	9,483	0:40.09
Exp 2	**b_12m**	17,208	9,640	6,180	17.3916666667	0.10	1.0	9,483	0:39.55
	bm_12	50,840	8,828	5,584	26.1216666667	0.15	0.9	8,535	0:56.98
	b_1m2	17,212	9,584	6,124	17.0011111111	0.10	1.0	9,483	0:44.80
Exp 3	**bm_123**	60,060	8,924	5,680	40.8422222222	0.234842778	0.9	8,535	1:30.61
	b_123m	17,212	9,660	6,200	17.135	0.09852625	1.0	9,483	0:49.23
	b_12m3	17,212	9,524	6,064	17.7438888889	0.102027361	1.0	9,483	0:43.74
	b_1m23	17,212	9,488	6,028	17.3172222222	0.099574028	1.0	9,483	0:48.07
Exp 4	**bm_1_1**	41,620	8,868	5,624	13.885	0.07983875	0.9	8,535	0:28.36
	b_1_1m	17,212	9,568	6,108	17.4394444444	0.100276806	1.0	9,483	0:43.91
	b_1m_1	44,872	9,108	5,808	30.1994444444	0.173646806	1.0	9,483	1:10.36
Exp 5	**bm_12_1**	50,840	8,952	5,708	23.6883333333	0.136207917	0.9	8,535	0:52.72
	b_12_1m	17,208	9,548	6,084	16.4366666667	0.094510833	1.0	9,483	0:43.64
	b_1m2_1	44,872	9,068	5,768	23.6472222222	0.135971528	1.0	9,483	0:59.30
	b_12m_1	17,212	9,524	6,064	14.5077777778	0.083419722	1.0	9,483	0:39.23
Exp 6	**b_1_12m**	17,212	9,664	6,204	16.0627777778	0.092360972	1.0	9,483	0:41.23
	b_1m_12	54,088	9,588	6,052	33.0027777778	0.189765972	1.0	9,483	1:18.57
	b_1_1m2	17,208	9,648	6,184	14.6272222222	0.084106528	1.0	9,483	0:37.58
	bm_1_12	41,620	8,952	5,708	11.2972222222	0.064959028	0.9	8,535	0:27.63

TABLE VIII
MULTI NODE PACKET STATISTICS

	Total Packets	Time (s)	Average PPS	Average packet size (B)	Total Bytes	Average kB/s	Average kb/s
Exp 1	50,075	232.386	215.5	204	10,239,951	44	352
Exp 2	106,631	259.668	410.6	204	21,796,497	83	671
Exp 3	176,728	291.176	606.9	204	36,130,952	124	992
Exp 4	71,637	200.905	356.6	204	14,637,014	72	582
Exp 5	144,060	269.001	535.5	204	29,439,035	109	876
Exp 6	139,761	264.624	528.1	204	28,547,547	107	863

timestep eight to thirteen in Table II. The effective throughput within these tables is a measurement of the user-level bits sent over a socket without the added headers of lower-level network protocols.

Table IV and Table VII show the energy consumption and memory usage for each protocol. To compute these values, Linux's Top command was used for 180 iterations with one second splits. Only one device is monitored in the single node protocol, therefore, authorizers and agents had their measurements averaged and summed together for total usage on the device. For the multi node protocol, each node is on a separate device, therefore the device averages are independent of each other. Our %CPU measurement was converted to a measurement of energy consumption by obtaining the difference in energy consumption (2.3W) between a Raspberry Pi 3B with 400% CPU Load (3.7W) and an idle PI (1.4W) [1]. This 2.3W difference was then multiplied by the ratio of average %CPU over maximum %CPU (400%) to obtain an energy consumption value. To calculate memory usage, the average %MEM was multiplied by the free memory of the corresponding Pi which is listed using the Free Linux command.

Table V and Table VIII show the bit rate during each experiment. These values were obtained through Wireshark and factor in for the added headers from lower level communication protocols.

A. Single Node

Our single node protocol was evaluated through a set of six experiments on a single device. Figure 3 illustrates the architecture of each experiment. There is one authorizer for all experiments and the gradual introduction of additional agents. However, experiment six increases substantially in the number of agents at eleven.

Table III shows that both the execution rate and the effective throughput increased substantially from experiment two to experiment three (one agent to two). However, each successive experiment had only a slight increase in execution rate and effective throughput. This large jump from experiment one to two is most likely due to the underuse and idling of the Raspberry Pi. Whereas the gradual increase from experiment 2 and onwards more closely reaches the Raspberry Pi's processing limit. Table IV supports this theory as there is a large jump from 106% CPU to 198% CPU from experiment one to two but only a small raise of around 10% for each successive experiment. Table V also shows a similar pattern with an increase in bit rate of 1000 k from experiment one to two while maintaining approximately the same bit rate of 3200 k for subsequent experiments.

This pattern indicates that the Raspberry Pi 3B approached near maximum effective throughput at around two agents continuously sending messages while simultaneously running Wireshark and the top command. Therefore, the maximum effective throughput can be estimated to be around 1 MB with around 450 protocol executions per second. In addition, under heavy load, the protocol consumes around 1.5W and uses around 100KB of memory. Finally, when analyzing the bitrate listed within Table V in comparison with the effective throughput from Table III, we can see that around a third of the actual bits transferred come directly from our protocol. The other two-thirds of bits are added on as headers by the lower level protocols.

B. Multi Node

Our multi node protocol was evaluated through six different experiments with varying WSN architectures. Figure 4 graphically depicts these designs. Experiments one through three only consist of the base station and direct connections to sensor nodes. Experiments four through six observe the impact of a cluster head on the network's performance. For all experiments, we set message transmission to consist of forty percent encrypted messages and sixty percent message with computed HMAC.

According to Table VI, the effective throughput and execution rate between experiments one to three increase linearly at around 100,000 b/s and 100 executions per second. Similarly, we can see that experiment four has rates almost equal to experiment two. However, experiments five and six have slightly slower effective throughput (50,000 bps less) and execution rates (40 executions per second less) despite having the same number of nodes within the network. This indicates that WSNs using our protocol should favor bushy forests rather than deep, tree-like, architectures to achieve higher message rate.

When viewing Table VII, we can see statistics for individual nodes within an experiment. Each row corresponds a specific node in the architecture. The active node for the row is indicated by an m to its right. For example, in experiment two, bm_12 indicates the row corresponds to the base station, whereas b_1m2 indicates the row corresponding to the left sensor node as seen in Figure 4. Interestingly, throughout all experiments we see a consistent memory usage of around 9000KB. In addition, for experiments one through three we see that the base station's CPU Load increases steadily at around ten to fifteen percent per added sensor node. Like Table VI, we also see that experiment four has similar energy consumption to experiment two. However, we also see that experiment six matches four and two in energy consumption. Experiment five has very low energy consumption for all nodes in the network and is on par with experiment one. Therefore, we see an inverse relation that suggests deep, tree-like, architectures favor a lower energy consumption over bushy forests for this protocol.

Once again, we see a similar pattern in the performance metrics of Table VIII. Experiments one through three increase in bit rate at a steady pace of around 300 bps. In addition, experiment four has a similar bit rate to experiment two and experiments five and six are comparable to experiment three in performance. However, although similar, four, five, and six are all around 100K bps slower than their counterparts. This once again supports our earlier theory that a bushier architecture will increase network traffic. When comparing Table VIII to Table VI, we once again see that one-third of the transferred bits come directly from our protocol, while the other two-thirds are added on by lower level protocols.

IV. Related Work

An overview of symmetric authentication protocols within wireless sensor networks (WSNs) is given within [9], highlighting the effectiveness of each technique as well as their corresponding pitfalls. These protocols exemplify the various functionality of one-way keychains within WSNs which we expand upon within our multi node protocol.

Timed Efficient Stream Loss-Tolerant Authentication (TESLA) is identified by [9] as one of the first symmetric broadcast authentication protocol, basing itself on the use of a symmetric MAC within a message and the delayed broadcast of a secret key. This secret key is constructed using a one-way keychain, allowing nodes to easily verify that the broadcasted key and message were authentic. Messages are buffered at the receiver until the delayed broadcast of the secret key. TESLA makes use of digital signatures to initially bootstrap a node to the network, which can prove to be too computationally intensive for nodes within a WSN.

[9] also introduces an expansion upon TESLA in which messages can be immediately authenticated by buffering packets at the sender rather than the receiver. Therefore, the hash value of a message can be verified ahead of time by a receiver allowing the instant authentication of the message.

Multiple-TESLA, which is also introduced by [9], instantiates multiple one-way keychains at the base station. These messages are broadcasted at different time intervals, allowing for the support of several receivers at varying distances.

The μTESLA protocol was developed to enable broadcast communication within resource constrained WSNs [9], [16]. To do this, μTESLA unicasts initial key commitments rather than authenticating initial packets with digital signatures, reducing overhead in terms of both memory and processing requirements [16].

Multi-level µTESLA improves upon the scalability of µTESLA by having multiple levels of keychains where higher keychains validate lower-level keychains and the lowest-level serves to authenticate broadcasted messages [9], [11]. High-level keychains also represent longer time-intervals than the lower keychains, reducing memory requirements of storing keychains whilst maintaining the quick release of low-level keychains [11]. DoS attacks are also combatted by the periodic and random retransmission of commitment distribution messages [11]. In effect, a more secure and efficient µTESLA is created to allow for expansion into larger networks.

Scalable µTESLA introduces the use of a Merkle hash tree to distribute initial parameters, effectively increasing the number of transmitters within a network and providing immunity to DoS attacks along with immediate authentication [9]. All transmitters within the network regularly broadcasts its initialization parameters [9].

[13] proposes a similar protocol, X-µTESLA, which uses a bloom filter to aggregate commitment keys during a time interval and expedite the authentication of messages within receiving nodes. Although the use of a bloom filter can lead to false positives, it also increases detection rates of forged packets and lowers memory and processing requirements [13].

Regular Predictable TESLA (RPT) allows for the immediate and regular authorization of messages. However, to do this, RPT sends the generated MAC in advance, requiring the message to be known before its needed transmission [9].

Batch-Based Broadcast Authentication (BABRA) makes use of independent keys to broadcast batches of messages within a time interval [9]. Due to the independence of keys, the advantages of no time-synchronization requirements and infinite key generation are present, however, packet loss cannot be handled if the lost packet contains an authentication key [9]. Like BABRA, the unbounded one-way chain method was also developed to overcome limitations in the set length of one-way key chains. However, this technique still has the drawback of time-synchronization evident in TESLA [9]. Long-duration TESLA also aims to overcome the bounded nature of one-way key chains by proposing a hierarchical key chain, like that which is used in multi-level µTESLA, that can be infinite in length [9].

TESLA++ expands upon the original TESLA protocol by initially sending a computed MAC along with an index number during the transmission phase, buffering it at the receiver, then sending the message and key together during the validation phase [9], [17]. This technique reduces the memory overhead within receivers as the buffering of MACs is a much smaller load than entire messages [17].

DoS Resilient TESLA combines ideas from TESLA++ and multi-level µTESLA to address computational and memory-based DoS attacks [9], [17]. A sender will first send its computed MAC and then send then send the corresponding key after the transmission period. In addition, the key is generated using a two-level key chain when the higher level validates the lower and the lower validates broadcasted messages [17].

Localized Tesla (L-TESLA) is identified within [9] as useful for authenticated broadcast transmission in larger networks. L-TESLA is implemented by constructing a distributed network based on hierarchical (trusted) nodes. These trusted nodes broadcast and rebroadcast to nodes within their subset of the network [9].

Extendable Tesla (X-TESLA) uses two-level key chains with different time intervals which cross authenticate one another. X-TESLA also provides for commitment hopping for networks with infrequent messaging saving energy within resource constrained devices [9].

[10] proposes a broadcasting technique that uses a one-way key chain in combination with a hierarchical ring structure to promote responsiveness after detection of malicious or downed nodes. The protocol described in [10] organizes all users within a network into a list. This list divides the users into intervals where revoked users mark the end of an interval. Users within each interval are then assigned a corresponding key in which only non-revoked users can use to decrypt the session key [10].

MultiMAC uses multiple MACs within a single message to reduce the number of required keys [9]. Each node is initialized with a key ring and a receiver verifies the authenticity of a message by checking the validity of each MAC generated by the individual keys on its ring [9].

[12] incorporates ideas from both µTESLA and LEAP++ to generate an E-LEAP++ protocol which reduces the authentication delay for messages and increases responsiveness to detected malicious nodes. E-LEAP++ calculates delays throughout the network and releases the commitment key earlier during the packet transfer process reducing delay between authenticating packets and reducing the threat of DoS attacks [12]. Additionally, if a malicious node is detected, a revocation mechanism is employed in which the base station, or cluster head, broadcasts a packet to all nodes in the network with instructions to stop processing packets from the detected node [12]. In E-LEAP++, nodes are only accepted into the network during the initial discovery phase to further prevent effects of malicious nodes.

Broadcast Authentication using Cryptographic Puzzles (BAP) achieves immediate authentication by disclosing symmetric keys within a cryptographic puzzle prior to broadcasting a message [9]. The MAC and message are then sent at a later period where the receiver can immediately verify the authenticity of the message on arrival. However, the verification of authenticity within BAP is dependent on the receiver's ability to receive and solve the cryptographic puzzle prior to receiving the message and its computed MAC [9].

[20] exploits the heterogenous nature of WSNs through a Multiple-Tier Remote Attestation (MTRA) protocol to verify the legitimacy of nodes within a WSN. MTRA assumes that the base station, along with cluster heads, are TPM-enabled devices that can manage their non-TPM-enabled one-hop neighbors. The MTRA base station legitimizes TPM-enables devices within the network through a hardware attestation procedure in which a random region of flash memory is hashed using a key generated from a one-way key chain [20]. Non-

TPM-enabled devices are legitimized by their TPM-enabled cluster head in the same way except a nonce is used in replacement of a TPM [20].

V. Future Work

Due to time constraints, we were unable to develop a machine learning algorithm that could effectively identify acceptable sets of permissions for software based on categorical analysis of an application. Therefore, to fully access the fine-tuned access control functionality of our single node protocol, a robust and efficient access authorization granting scheme should be developed.

In addition, the multi node protocol was tested on a relatively small scale despite being designed for large, distributed, WSNs. Therefore, large-scale experimentation for this protocol should be conducted to get a full picture of its performance. Experimentation should be further conducted on the effects of bushy vs. deep architectures on network performance when using the multi node protocol.

Finally, current speculation on revocation mechanisms for the multi node architecture are to revoke the highest-level node within a subtree that was detected to be malicious. However, this would eliminate the entire subtree under the malicious node and require a technician to correct the affected nodes. Therefore, an area of interest is in the methods of detection for a malicious node within the network and the revocation mechanisms that are employed. The aims of these detection and revocation methods are to reduce the impact on the network and the affected subtree.

VI. Conclusion

The internet of things is a rapidly expanding field consisting of low-powered and memory-constrained devices. Due to this, IoT devices often sacrifice security in favor of functionality. We propose two separate protocols that aim to harden the security of these IoT devices while maintaining effective communication and function of the device. These methods both employ symmetric keys in combination with AES encryption and SHA256 HMAC generation to ensure confidentiality and authenticity. Our first protocol takes a user-level approach on the internal security between a device and its installed software. This single node protocol allows for fine-tune access control for advanced operations required by software. Our second protocol allows for WSN designers to adapt their networks and selectively tradeoff between memory use and power consumption in a sensor network. In addition, this protocol allows for efficient, low-cost, and dynamic bootstrapping of nodes into an established symmetric network.

References

[1] "Power consumption benchmarks," 2022, https://www.pidramble.com/wiki/benchmarks/power-consumption.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A survey on sensor networks," *IEEE Communications magazine*, vol. 40, no. 8, pp. 102–114, 2002.

[3] G. Cerullo, G. Mazzeo, G. Papale, B. Ragucci, and L. Sgaglione, "Iot and sensor networks security," in *Security and Resilience in Intelligent Data-Centric Systems and Communication Networks*. Elsevier, 2018, pp. 77–101.

[4] X. Du and F. Lin, "Designing efficient routing protocol for heterogeneous sensor networks," in *PCCC 2005. 24th IEEE International Performance, Computing, and Communications Conference, 2005*. IEEE, 2005, pp. 51–58.

[5] ——, "Improving sensor network performance by deploying mobile sensors," in *PCCC 2005. 24th IEEE International Performance, Computing, and Communications Conference, 2005*. IEEE, 2005, pp. 67–71.

[6] X. Du and D. Wu, "Adaptive cell relay routing protocol for mobile ad hoc networks," *IEEE Transactions on Vehicular Technology*, vol. 55, no. 1, pp. 278–285, 2006.

[7] X. Du and Y. Xiao, "Energy efficient chessboard clustering and routing in heterogeneous sensor networks," *International Journal of Wireless and Mobile Computing*, vol. 1, no. 2, pp. 121–130, 2006.

[8] H. Gilbert and H. Handschuh, "Security analysis of sha-256 and sisters," in *International workshop on selected areas in cryptography*. Springer, 2003, pp. 175–193.

[9] K. Grover and A. Lim, "A survey of broadcast authentication schemes for wireless networks," *Ad Hoc Networks*, vol. 24, pp. 288–316, 2015.

[10] N.-S. Jho, J. Y. Hwang, J. H. Cheon, M.-H. Kim, D. H. Lee, and E. S. Yoo, "One-way chain based broadcast encryption schemes," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2005, pp. 559–574.

[11] D. Liu and P. Ning, "Multilevel μtesla: Broadcast authentication for distributed sensor networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 4, pp. 800–836, 2004.

[12] B. Mbarek, A. Meddeb, W. B. Jaballah, and M. Mosbah, "A secure authentication mechanism for resource constrained devices," in *2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA)*. IEEE, 2015, pp. 1–7.

[13] ——, "A broadcast authentication scheme in iot environments," in *2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*. IEEE, 2016, pp. 1–6.

[14] M. Muneebahmdhyiddeen, R. Mohan, B. L. Anupama, and A. V. Nair, "Recent survey on security in wireless sensor network," *Wireless Communication*, vol. 8, no. 7, pp. 270–273, 2016.

[15] V. Ndatinya, Z. Xiao, V. R. Manepalli, K. Meng, and Y. Xiao, "Network forensics analysis using wireshark," *International Journal of Security and Networks*, vol. 10, no. 2, pp. 91–106, 2015.

[16] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler, "Spins: Security protocols for sensor networks," *Wireless networks*, vol. 8, no. 5, pp. 521–534, 2002.

[17] N. Ruan and Y. Hori, "Dos attack-tolerant tesla-based broadcast authentication protocol in internet of things," in *2012 International Conference on Selected Topics in Mobile and Wireless Networking*. IEEE, 2012, pp. 60–65.

[18] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein, "sel4 enforces integrity," in *International Conference on Interactive Theorem Proving*. Springer, 2011, pp. 325–340.

[19] G. Singh, "A study of encryption algorithms (rsa, des, 3des and aes) for information security," *International Journal of Computer Applications*, vol. 67, no. 19, 2013.

[20] H. Tan, G. Tsudik, and S. Jha, "Mtra: Multi-tier randomized remote attestation in iot networks," *Computers & Security*, vol. 81, pp. 78–93, 2019.

[21] M. Zhang, X. Du, and K. Nygard, "Improving coverage performance in sensor networks by using mobile sensors," in *MILCOM 2005-2005 IEEE Military Communications Conference*. IEEE, 2005, pp. 3335–3341.

[22] Z.-K. Zhang, M. C. Y. Cho, C.-W. Wang, C.-W. Hsu, C.-K. Chen, and S. Shieh, "Iot security: ongoing challenges and research opportunities," in *2014 IEEE 7th international conference on service-oriented computing and applications*. IEEE, 2014, pp. 230–234.