Provenance and Authentication of Oracle Sensor Data with Block Chain

Lightweight Wireless Network Authentication Scheme for Constrained Oracle Sensors

By

Gilroy Gordon

A Thesis Submitted to

Saint Mary's University, Halifax, Nova Scotia

in Partial Fulfillment of the Requirements for

the Degree of MSc. In Computing and Data Analytics

January, 2017, Halifax, Nova Scotia

Approved:	Dr. Dawn Jutla Supervisor
Approved:	Dr. Stavros Konstantinidis Examiner
Approved:	Dr. Gord Agnew Examiner
Approved:	Dr. Srini Simpalli External Examiner
Date:	January 31, 2017

Acknowledgement

First, I would like to acknowledge God, as through Him, all things are possible. I would also like to express my gratitude to my thesis supervisor, Dr. Dawn Jutla who not only provided the necessary supervision but also assisted in the procurement of additional resources to implement and test the items proposed in this thesis. Moreover, these efforts would not have been possible without the gracious assistance of the Queen Elizabeth II Diamond Jubilee Scholarship programme.

Provenance and Authentication of Oracle Sensor Data with Block Chain

Lightweight Wireless Network Authentication Scheme for Constrained Oracle Sensors

by Gilroy Gordon

Abstract

With the significant increase in the dependence of contextual data from constrained IoT, the blockchain has been proposed as a possible solution to address growing concerns from organizations. To address this, the Lightweight Blockchain Authentication for Constrained Sensors (LBACS) scheme was proposed and evaluated using quantitative and qualitative methods. LBACS was designed with constrained Wireless Sensor Networks (WSN) in mind and independent of a blockchain implementation. It asserts the authentication and provenance of constrained IoT on the blockchain utilizing a multi-signature approach facilitated by symmetric and asymmetric methods and sufficient considerations for key and certificate registry management. The metrics, threat assessment and comparison to existing WSN authentication schemes conducted asserted the pragmatic use of LBACS to provide authentication, blockchain provenance, integrity, auditable, revocation, weak backward and forward secrecy and universal forgeability. The research has several implications for the ubiquitous use of IoT and growing interest in the blockchain.

Keywords: sensors, blockchain, authentication, oracle, provenance

January 31, 2017

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

This thesis aims to address the increasing provenance concerns with constrained IoT devices and decentralized blockchain technology. It will focus on provenance as identifying the source of data, through integrity, instead of utilizing only submitted metadata. Subsequent to providing a background, significance and the specific research goals in the Introduction, it will identify possible gaps, design considerations and threats within the Literature Review. With the gaps identified, a proposal was also made, implemented and analyzed. Finally, it will conclude the results arising from this analysis and recommendations for future work.

1.1 Thesis Objectives

1) Design an authentication scheme for IoT Blockchain oracles operating in constrained wireless sensor networks
2) Conduct a threat assessment for constrained IoT oracles utilizing the authentication scheme to achieve provenance on the blockchain
3) Identify the resource requirements of constrained oracle sensors utilizing the authentication scheme to achieve provenance on the blockchain

1.2 Background and Purpose of Study

The increasing interest and applicability of the "Internet of Things" (IoT), a paradigm describing interconnected entities (usually smaller devices) over the internet with greater autonomy (Tiburski, Amaral, Matos, & Hessel, 2015) has excited much research into its numerous applications and associated concerns. With applications in

domains such as healthcare, smart buildings, supply chain management and aerospace to name a few (Perera, Zaslavsky, Christen, & Georgakopoulos, 2014), the security associated with these implementations has become a primary concern. Not only have these devices been utilized to collect information in mission critical environments that are better supported by wireless sensor networks, but they have been entrusted as actuators which should act reliably within the constraints applied by their various negotiation algorithms (Yuanyuan Zeng, 2016). Furthermore, although these smart devices have been designed to be activity, policy and process-aware (Kortuem, Kawsar, Sundramoorthy, & Fitton, 2010), their hardware constraints make them an easier target for security breaches. According to the Ponemon's 2016 Cost of Data Breach Study: Global Analysis (Ponemon Institute, 2016) which surveyed 383 companies in 12 countries, the average cost per data breach was US $4 Million Dollars. Compounding this with the increased deployment of IoT devices, security and reliability remain a primary concern. Moreover, the costs to maintain a centralized IoT of secured yet constrained devices has encouraged the use of multiple network topologies, transitive trust authorities and most recently decentralized paradigms such as the blockchain. It has been argued that the blockchain offers a scalable, decentralized peer-to-peer level of trust for IoT and therefore should be explored (IBM Corporation, 2015). Current implementations of the blockchain however have proven themselves to be too resource-intensive to be managed by constrained IoT devices.

With the increasing dependence on reliable data from constrained IoT devices and the advent of the blockchain, this study will seek to exact provenance and integrity of wireless sensor data through authentication on the blockchain. Furthermore, it will assess

the associated threats and resource requirements of wireless sensor data authenticating on the blockchain.

1.3 Problem Statement

The increased use of contextual data from IoT within decision-making has made provenance a growing concern in organizations (Townend et al., 2013). This issue is compounded by the constrained nature of these devices, which inhibit their abilities to employ stronger cryptographic constructions to enforce the integrity of the provided data. Moreover, the network concerns inherited by Wireless Sensor Networks increase the likelihood of successful attacks on integrity such as man-in-the-middle and spoofing attacks (Chelli, 2015). Furthermore, the costs to maintain a centralized IoT of secured and constrained devices has encouraged the exploration of multiple network topologies and has stimulated discussions about the use of the blockchain as an alternative approach (IBM Corporation, 2015).

Within the relatively new context of the blockchain, this qualitative and quantitative study will explore the possibility of exacting provenance and integrity of authenticated, wireless sensor data on the blockchain.

1.4 Research Questions

This study aims to answer the following research questions:

1. How can the blockchain be used to authenticate wireless oracle sensor data?
2. What are the resource requirements for wireless oracle sensors authenticating on the blockchain?
3. What threats exist for wireless oracle sensors authenticating on the blockchain?

1.5 Significance of Study

Provenance and integrity of contextual data being used in decision-making is a growing concern for organizations, government or privately owned. With millions of deployed IoT devices still operating on constrained computational resources and the high costs to maintain their supporting topologies, there is room for additional research to address these concerns.

The importance of this research project is to explore the integration of decentralized blockchain technology to reinforce the integrity of constrained sensor data. The research will determine how the current resource intensive implementations of the blockchain may be integrated with constrained sensors to achieve integrity. It will also explore the resource requirements of constrained sensors authenticating with the blockchain and compare these to existing security schemes addressing the integrity of sensor data. Moreover, the study will evaluate the threats and vulnerabilities of these constrained sensor devices authenticating with the blockchain.

Achieving provenance through authentication on the blockchain would allow greater auditability, security, fault tolerance and cost effective deployments of sensor provenance networks.

Chapter 2: Literature Review

2.1 Introduction

This literature review will aim to provide an overview of the applicability of sensors and actuators in IoT while highlighting several design constraints and challenges. Furthermore, it will highlight the design goals, threat models, key management, distribution, communication and authentication in Wireless Sensor Networks (WSNs). Finally, a conceptual overview of the blockchain and oracles will be provided along with their current resource requirements and the implications for IoT.

2.2 Wireless Sensor Networks

Neubert in 1975, identified sensors as devices which transformed physical quantities into electrical output signals based on a scale defined by a predictable output-time, output-input relationship with an acceptable degree of accuracy under a specified set of environmental conditions (Schroeder, 2008). This broad definition considers devices used for measuring physical quantities such as temperature, movement, humidity, pressure, audio, proximity and light (Beigl et. al, 2014). Choosing the correct sensor for a respective task depends on the following classifications: detection methods, conversion mechanism, sensor materials and applications, performance specifications and stimuli or measurand (Schroeder, 2008). When considering the power availability constraints of sensors, they are optionally classified as active and self-generating if a fixed power source is accessible, or passive and modulating if battery-operated as is the case of some Radio Frequency Identification (RFID) tags.

Advances in sensor technology has yielded numerous applications such as continuous medical monitoring in health care (Alemdar and Ersory, 2010), habitat monitoring, weather forecasting, supply chain management, inventory tracking and military monitoring (Perera et. al, 2014). Many of these applications have been realized through live exchange of sensors feeds with other systems in Wireless Sensor Networks (WSNs). A Wireless Sensor Network may be considered as a group of wired and wireless network possessing sensor nodes with the ability to perform computations, sense some property along with a module capable of wireless functions deployed in a designated field (Choi, 2010). A closer look at the types of sensors will highlight the trend to utilize micro-sensors as opposed to macro- sensors. Macro-sensors have proved themselves viable options in the past especially while monitoring the extraction, exploration and refinement processes in the oil and gas industry. However, these wired macro-sensors have also proved themselves as costly deployments that are not suited or easily maintained for temporary assignments. They usually require large amounts of energy and are even more difficult to install in remote and hostile environs as they often include proximity requirements to guarantee sufficient monitoring (Adejo, et al, 2013).

A more pragmatic approach is to deploy multiple micro-sensors and actuators or Micro-electro-mechanical systems (MEMs) in a WSN to exact coverage over a specified area of interest. Although these deployments offset several of the aforementioned concerns with macro-sensors, these advancements are accompanied by their own design constraints, which correlate to their size and computing capacity.

2.2.1 Design Considerations for WSNs.

Several design constraints should be taken into consideration when designing WSNs, notably: fault tolerance, scalability, hardware constraints, network communication, environment, production costs and power consumption (Akyildiz, et al, 2002). These constraints provide a basis to compare different solutions to issues experienced in the development and management of WSNs.

Fault tolerance is a network's ability to remain functional despite the failure of network devices or nodes. The reliability of the network becomes more debatable with the limited power availability of battery operated nodes and harsh or unpredictable environments that cause physical damage. For example, smart home WSNs monitoring temperature and humidity may require less fault tolerance measures unlike sensors deployed in critical military surveillance environments. The latter requires additional strategies for fault tolerance due to the probable interference from the volatile environment in which they are deployed (Akyildiz, et al, 2002).

Scalability focuses on the number of sensor nodes deployed or node density in a WSN. This could involve approximately 300 nodes for a 5 x 5 m^2 area in factory machine diagnosis or 10 nodes per area in vehicle tracking. WSN schemes should be able to facilitate and utilize the node density to tackle constraints experienced by sensors, such as multi-hop routing networks for battery-operated nodes aiming to reduce power used for radio communication (Akyildiz et al, 2002).

Moreover, production costs should be kept at a minimum in the design of WSNs. Coupling the frequent deployments, density of regions of interest and the application needs of WSNs, sensors nodes may develop specific hardware needs and in the case of

software-defined sensors networks, software needs which may easily become impractical to fund. Juxtaposing these facts with the vulnerability of sensor nodes, which will be described later, it is important to ensure that the components of the WSN are minimized to lower the cost of replacements also.

Despite the advancements in Micro-Electro-Mechanical Systems (MEMs) enabling much more powerful System on a Chip (SoC) designs, battery powered micro-sensor nodes are still very constrained devices. Smart dust mode devices usually possess approximately 1J of energy while Wireless Integrated Network Sensors (WINS) powered by Lithium coin cells 1cm thick and 2.5cm in diameter operate on 30 μA (Akyildiz et al, 2002). Moreover, Table 1 details additional hardware constraints as a result of the required hardware size according to the application specifications.

Table 1: Comparison of Hardware Constraints

Platform	MCU	RAM	Program & Data Memory	Radio Chip
BTnode3	ATMega128	64 KB	128 - 180 KB	CC1000/Bluth
Cricket	ATMega128	4 KB	128 - 512 KB	CC1000
Imote2	Intel PXA271	256 KB	32 - MB	CC2420
MICA2	ATMega128	4 KB	128 - 512 KB	CC1000
MICAZ	ATMega128	4 KB	128 - 512 KB	CC2420
Shimmer	TI MSP 430	10 KB	48 KB - Up to 2 GB	CC2420/Bluth
TelosA	TI MSP 430	2 KB	60 -512 KB	CC2420
TelosB	TI MSP 430	10 KB	48 KB - 1 MB	CC2420
XYZ	ARM 7	32 KB	256 - 256 KB	CC2420

Note: Reprinted from Security Challenges in Wireless Sensor Networks, by Singh et al, (2016)

The use of a wireless medium, which is more open and accessible than wired communication, introduces security concerns for WSNs. With communication being more likely to be intercepted, replayed or modified, networks must aim to facilitate security issues in addition to network congestion, noise and packet collisions (Akyildiz, Su, Sankarasubramaniam & Cayirci, 2002). Juxtaposing these factors with battery-exhausted nodes that may be participating in multi-hop networks, which support specific concerns such as Application Specific WSNs (ASWSNs) or utilize other additional communication protocols as in Software Defined WSNs (SDWSNs), communication may be unreliable.

2.2.2 Communication and Transmission

Much Machine to Machine (M2M) communication on constrained devices are conducted using more lightweight communication protocols such as the Constrained Application Protocol (COAP) over the User Datagram Protocol (UDP). COAP, although similar to HTTP, is a less complex, stateless web protocol working at the application layer over UDP that provides Uniform Resource Identifiers (URI) and content types with a lower header overhead (Shelby, Hartke, & Bormann, 2014). IPV6 over Lower Power Wireless Personal Area Networks (6LoWPAN) is another protocol targeting constrained devices which leverages IEEE 802.15.4 (Gehrmann et al,2015). The protocol focuses on header compression, which reduces communication bandwidth, time and ultimately power consumption. It achieves this by splitting communication in “contexts” that share metadata including IPV6 addresses. For example, an abbreviated context identified by an identifier *0x10* may be used to represent a particular session and set of source and or

receiver addresses. Nodes may then optimize communication by agreeing on a context identifier. Specific adaptations of 6LowPAN have also been proposed to work with COAP and COAPs to optimize node communication based on the constraints highlighted in the previous section (Raza et al, 2012)

WSN nodes may also use the lightweight publish/subscribe transport protocol to transmit or listen to message topics as proposed in the OASIS Message Queuing Telemetry Transport (MQTT), which at the time of writing is at version 3.1. Designed with constrained Machine-to-Machine (M2M) and IoT communication in mind, the protocol leverages other protocols at the network layer such as TCP/IP to support multicast messages in an agnostic manner. Furthermore, the protocol supports various message delivery Quality of Service (QoS), such as the message being delivered at most once, duplicated, or once-and-once-only delivery. This allows the devices to perform an asynchronous push (publish) of messages without polling with bit-wise compressed headers of at least 2 bytes (OASIS MQTT Technical Committee, 2014).

Additional methods suitable for short range communication in WSNs yet operating at lower levels of the OSI stack include Bluetooth (IEEE 802.15.1), Zigbee (IEEE 802.15.4), Wi-fi (IEEE 802.11) and ultra-wideband (UWB) (IEEE 802.15.3b) (Lee, Su, & Shen , 2007). Bluetooth, usually used in computer peripherals in WPANs, uses a frequency band of 2.4GHz with a maximum signal rate of 1Mb/s over approximately 10m with a transmit power (TX power) between 0-10dBm. Although UWB has a similar range, it's frequency band spans from 3.1 to 10.6 GHz with a max signal rate of 110MB/s and TX power of -41.3 dBm/ MHz making it more suitable for indoor high-speed wireless communication. UWB also exceeds Zigbee's max signal rate

of 250 kb/s, but Zigbee's frequency bands may span from 868 MHz to 2.4 GHz while still possessing 10 times UWB's range with a TX power between 0 and -25 dBm. Wifi, the strongest of the four, has the same range as Zigbee with a frequency band of 2.4GHz or 5Ghz with signal rates growing up to 54 Mb/s and a TX power between 15 and 20 dBm. When considering encryption and authentication, Bluetooth protects its data using a 16-bit Cycle Redundancy Check (CRC), encrypted with the E0 Stream Cipher using a shared secret over a piconet or scatternet. The piconet is an adhoc network of no more than eight interconnected wireless devices synchronized to a common clock and multi-hop sequences. With transmissions over bluetooth, the network supports a master slave hierarchy until multiple piconets become interconnected to become a scatternet (Lee, Su, & Shen, 2007). A bit more secure is UWB which uses a 32 bit CRC after being encrypted with Advanced Encryption Standard (AES) in Counter Mode (CTR) and authenticating with the Cipher Block Chaining (CBC) Message Authentication Code (MAC) (CCM). The Zigbee protocol utilizes the 16-bit CRC similar to bluetooth with the encryption and authentication used by UWB. Finally, Wifi shares the CRC of UWB but encrypts with the Rivest Cipher 4 (RC4) protocol with Wired Equivalent Privacy (WEP) and AES before authenticating with Wifi Protected Access (WPA2). Moreover, Lee, Su, & Shen (2007) in their comparative study, sought to evaluate the data code efficiency (sum total bytes used in data transmission), protocol complexity (the number of supported MAC primitives and Host Controller Interface (HCI) events) and required power consumption. As it pertains to data code efficiency, they argued that Bluetooth and Zigbee were more efficient for smaller payload transmissions since they had less data fragmentation and their maximum payload data were 339, 102 bytes respectively followed by 2044 and

2312 for UWB and Wifi at the time of writing. Furthermore, Zigbee's low support for many primitives and events (only 48 as defined in IEEE 802.15.4) made it more suitable to be embedded in constrained WSN nodes. Bluetooth for example, supported the most primitives such as client service access point (SAP) and Logical Link Adaptation Protocol (L2CAP) followed by UWB and WIFI. Again, when power consumption was evaluated by comparing the normalized energy consumption (mJ/Mb) of collected TX and RX (receiving), Bluetooth and Zigbee were identified as most suitable for power consumption in low data rate applications as opposed to UWB and WIFI (Lee et al , 2007).

2.3 Securing Wireless Sensor Networks

2.3.1 Objectives and Threat Model

In addition to the aforementioned design considerations, specific design goals for securing WSNs include secure localization, time synchronization, self organization, data freshness, confidentiality, integrity, availability and authentication (Chelli, 2015). With sensor nodes possessing the ability to communicate highly sensitive data, it is important to ensure that data transmitted remains confidential between the sender and intended recipients.

When considering the threat model, an attack in a particular WSN may be considered as goal, performer or layer-oriented (Chelli, 2015). In passive goal-oriented attacks, the attack is usually directed to compromising data confidentiality through network monitoring and traffic analysis. Poorly encrypted communication and the exchange of keys enable the unwanted disclosure and possible re-use of information

already sent or the prediction of future exchanges. Unlike passive attacks, the active attacker aims to control segments or the whole network. These attacks include but are not limited to hello floods and denial of service or sleep (DOS) attacks where victim nodes are flooded with requests, which prevent them from accepting requests from legitimate nodes. Such attacks ultimately exhaust the node's battery power due to the excessive power consumption associated with processing requests. Furthermore, black hole, sink hole, worm hole, selective forwarding and false node attacks may also be used to divert or drop transmitted packets in the network (Hu, Perrig, & Johnson, 2006). These attacks may be difficult to detect, as a network monitor must deduce whether the packets are being lost because the network is being compromised or whether the loss was caused by network collisions or environmental conditions. Moreover, attackers may aim to falsify transmitted data using replay, fabrication, spoofing and man-in-the-middle-attacks (Chelli, 2015).

Performer oriented attacks usually describe the origin of the attack, i.e. whether they originated from inside the network or not. Attacks from outside the network are usually more passive, while inside attacks, including when an attacker garners the trust of other nodes are more disruptive and similar to the aforementioned active attacks (Chelli, 2015).

Layer-oriented attacks consider the various attacks applicable at different layers of the Open Systems Interconnection (OSI) model. At the lowest layer which is the physical layer, an attacker may opt to jam the radio channel of the sensor with continuous and spurious or high energy radio transmissions, resulting in DOS attacks. Moreover, at the link layer, an abstraction responsible for coordinating nodes by regulating the flow of

data, attackers may opt to drain node energy through the continuous disruption of packets with abused Media Access Controls (MACs). Cryptanalysis at this layer may also deduce patterns, even when transmissions are encrypted. However, at the network layer, attackers aim to disrupt packet routing with sinkholes, spoofing or routing replay attacks. Above this layer, the transport layer may become the victim of hello floods with numerous connection requests to constrained nodes. Additionally, at the application layer, attackers target applications by compromising executable code to incite malicious behavior, such as those executed by captured bots or corrupting data (Chelli, 2015).

Different approaches have been utilized to target the various security design considerations of WSNs. With sensors, especially in military applications or Body Area Networks (BAN), often displaced after being deployed, nodes are required to re-establish routing and communication or localization. Techniques used to localize nodes include symmetric or asymmetric Diffie Hellman key exchanges (Liu et al, 2003), which ensure authenticated communication even between vehicles moving between multiple sub-networks (Hossain & Mahmud , 2007). Furthermore, confidentiality and integrity becomes a primary concern especially while handling sensitive data. Modified versions of standards such as MPEG-21 or ISO/IEC 21000, that addresses sharing of digital content, has been used to embed authentication and confidentiality for Biomedical Sensor Networks (Leister et al, 2008) along with protocols such as IPSec and DTLS. Finally, additional strategies specific to authentication while also considering availability, whether the nodes will be able to use the available resources within the network, and data freshness which is concerned with replay attacks, forward and backward secrecy (Chelli, 2015) will be described further in this review.

2.3.2 Key management and distribution

Authentication is usually established on a shared premise or key. It is important to note that the choice of a key management scheme for a WSN is heavily biased towards the particular application and deployment concerns related to the WSN. Key distribution and management schemes whether they are random or deterministic include varieties of pairwise management, random key-chain based key pre-distribution, and network-wise key management schemes (Dustin et. al, 2007, Lee and Stinson, 2005). These schemes may choose Public Key Infrastructure (PKI) methods or different pre-distribution schemes (some of which will be outlined later) to develop a trusted means of communication. The latter is less resource intensive for nodes but requires a trusted authority (TA), often represented as the base centre or station, which if compromised will compromise the security of the entire network (Lee and Stinson,, 2005). Furthermore, for schemes considering the ad-hoc nature of WSNs, a shared key discovery mechanism which is deterministic may prove more resource friendly to sensor nodes since key derivation may be verified through an algebraic function of pre-defined parameters (Lee and Stinson, 2005). Moreover, these schemes may employ various re-keying techniques including batch, immediate, delayed and periodic re-keying to reinforce the integrity of the network (Hossain & Mahmud 2007). These re-keying strategies reinforce forward and backward secrecy, i.e. whether a member of the network has access to future or past information respectively, essentially creating a communication session that lasts for the duration of the confirmed key. Finally, when considering key recovery, one may consider self-healing and stateless key distribution techniques. Unlike self-healing techniques,

which allow lost session keys to be recovered by group members, stateless techniques allow nodes to acquire updated session keys even if they miss previous key update messages. (Liu et al ,2003)

All pair-wise (using single master key) and *All pair-wise (distributed pair-wise keys) or fully pair-wise shared key* schemes (Feng & Wenpeng ,2010) are two popular pair wise key management schemes. Unlike the single master key implementation that issues the same master key to all sensor nodes in the network, the distributed pair-wise key implementation deploys all nodes in the network with their own key and the keys of all network participants (Dustin et. al, 2007). Although the former is less resource intensive due to the storage needs of sensor nodes and easier to deploy new nodes, an attacker may easily compromise the entire network by only gaining access to one node.

The *random pair-wise key* scheme aims to improve the resilience of the single master key scheme, yet reduce the load of the distributed pair wise key scheme using a random distribution *p*. The Erdős–Rényi model encouraged the design that for a network of *N* nodes, each node in the network would have the probability of being assigned Np other keys (Dustin et. al, 2007). This design made it possible for a node to connect to Np other nodes that were in its radio range during the shared-key discovery phase, thus supporting larger WSNs, multi-hop network designs while requiring 2Np units of memory to store the entire key chain. The *closest pair-wise key* distribution scheme innovated further by utilizing location information to influence key distribution. However, this scheme required prior knowledge of an organized deployment of nodes, as each node would share pairwise keys with "c" of its closest neighbours. This made it possible to reduce the storage needs to 2c+1 units as each node within a region could

utilize a Pseudo-Random Function (PRF) to re-generate keys of its peers by only storing its peers IDs and a shared key with PRF(PeerId, SharedKey) (Dustin et. al, 2007).

Alternatively, the *ID based one-way function scheme* (IOS) was inspired by a star topology sub-network of an r-regular graph of nodes. With each node being the center of a different star sub-graph, the scheme would distribute pair wise keys and require memory for only *r+1* neighbours. The scalability and resilience of this design may be improved using *Multiple IOS*, which focuses on a one-to-one correspondence between sensor nodes to decrease memory usage by a factor of 1. In this design, a sensor from a sub-network *A* stores the common key for sub-network A in addition to a secret Hash (secret key from sub-network B (KB) | ID of a node in network A(IDA_i)) for each node in network A. A node from network B may then verify a node from network A, A_i by generating the secret key Hash($KB|AD_i$) (Dustin et. al, 2007).

Another scheme, *Broadcast Session Key negotiation protocol* (BROSK), initially pre-deploys a single master key to all nodes, which allows the entire network to be compromised if one node is compromised. The benefits associated with this scheme lie within the low required storage requirements since nodes will only require the nonce exchanged in post-deployment shared key discovery to generate the shared session key using PRF (master key || nonce of node 1 || nonce of node 2) (Dustin et. al, 2007).

Unlike BROSK, the *lightweight key management scheme leveraging initial trust*, proposed by Deutre et al, (2004), distributes group authentication keys (gak_i) for each deployed set of sensor nodes. The scheme relies on a previous trust relationship based on the fact that the each deployed set will include a set of possible secret keys (sk_i) that may be used to authenticate future deployed sets. A newly deployed set of nodes will possess

an additional generation key (gk_i) that may be used to self authenticate with previous generations. Authentication within a group would derive a session key, KA = PRF(gak_i, RNA, RNB) where RNA and RNB are randomly exchanged nonce values by sensor nodes A and B respectively. A node forms the newly deployed set, C, aiming to authenticate with node A from the first deployment, would first generate sk_i = PRF (gk_i | RNA) before generating KA= PRF (sk_i, RNA, RNC). Although a node would require approximately 4 + 2n units of key storage memory for n deployed sets of nodes, resilience remains low as an entire deployed set may be compromised if gak_i or sk_i is identified (Dustin et. al, 2007).

Other schemes utilize more randomized key-chain based key pre-distribution solutions. The *basic probabilistic key pre-distribution scheme* proposed in 2002, in its key setup phase, generates a pool of *KP* keys and associated ids, before distributing *k* randomly chosen keys (without replacement) to each sensor node's key chain. In the shared key discovery phase, the probability that a node is able to communicate with a neighbour is dependent upon *KP* and *k*, as each node will broadcast the identities in its key chain to neighbours, to determine node identities available in each node's key chain. Through the identification of shared identities, neighbouring nodes may utilize the keys associated with theses identities to communicate. The cluster key grouping scheme proposed by Hwang et. al. in 2004 utilizes a similar approach but splits key chains into clusters, each possessing a start key id. This reduces the initial message length to the size of the clusters multiplied by the size of the key id list since the start key id may be used to determine the remaining cluster keys (Dustin et. al, 2007).

In an effort to reduce the required key storage for each sensor node, the pair-wise key establishment protocol proposed by Zhu et. al in 2003 utilizes a PRF seeded with the unique ID of the sensor node to generate the necessary key IDS and reduce the shared key discovery broadcast length to size of one key id. The trade off here lies with the computation required for PRF(ID) for each neighbouring node (Dustin et. al, 2007). The *Q-composite random key pre-distribution* scheme goes a bit further to improve resilience, reducing the possibility that a node is captured from k/KP to kq/KPq where KP is the key pool size, k is key chain size assigned randomly to the node and q is the number of common keys shared between two nodes. It achieves this with the use of a secret key composed of a hash of all the common keys between two neighbours (Dustin et. al, 2007).

Another scheme, which increases key resilience, is the *multi-path key reinforcement scheme*. The scheme is CPU and power resource-intensive however, as it requires each sensor node to generate *n* random key updates that are sent through *n* disjoint secure paths which prompts each receiving node to generate a reinforced link key. These progressive updates to keys in post-deployment through indirect or disjoint nodes links reduces the possibility of successful traffic sniffing for the updated keys. In order to compromise the new key, the attacker would need to identify the *n* random key updates being routed through the *n* nodes and the PRF that is used to generate the new key (Chan et al, 2003). This is similar to the key management scheme proposed by Feng & Wenpeng (2010), which adds an additional storage overhead, as nodes will also store the path to the base station since the combined key path doubles as a session key.

The *pair-wise key establishment protocol* is similar to the *multi-path key reinforcement scheme* with the exception that it utilizes a "threshold secret sharing" for key reinforcement and a more resource intensive key update strategy. The threshold of *n-1* random key part updates are all XOR'd with the current node's key before being transmitted along the disjoint path (Dustin et. al, 2007). Similarly, the cooperative pair-wise key establishment protocol limits the shared key node set to a set of "cooperative nodes". Sensor node *A* will expect each cooperative node to generate a HMAC of (cooperative node key | B | IDA). Sensor node *A* will then utilize this HMAC to generate reinforced keys while also sharing the set of cooperative nodes with another sensor node *B* which will repeat the process to generate the same key. Although this practice reduces the possibility of compromising keys, it is associated with a high communication and power usage cost (Dustin et. al, 2007).

Another approach is to utilize block design techniques in combinatorial design theory to achieve the *combinatorial design based key pre-distribution scheme*. This design aims to allocate pre-defined key chains to nodes where each node will have at least one common key with another, thus reducing the key path to 1. Utilizing the number of sensor nodes to be deployed *n*, a prime power *p* is chosen where $2p+p+1 >= n$. The prime power is then used to generate *p-1* Mutually Orthogonal Latin Squares (MOLS), a square matrix where each of the *p* elements occurs only once in each row and column and each row column intersection is distinct. This *complete* set or order *p* may be used to generate an affine plane ($p^2, p,$ 1) or a projective plane ($p^2 + p + 1, p + 1, 1$) which are both symmetric in design. Although, these networks improves a node's ability to localize and reduce the network communication and power consumption used for shared key

discovery, they require a storage capacity of *n+1* keys for the keychain which becomes less feasible with larger WSNs. Furthermore, in an effort to increase scalability, other hybrid designs exist which reduces the necessary key storage by increasing the average key-path (Camtepe & Yener, 2007).

Unlike much of the previous schemes, key matrix-based dynamic key generation as utilized in the popular *Blom's scheme* is a public key cryptography scheme which utilizes the product of a public and private key matrix of size *N x N*, where *N* is the amount of nodes in the network. The approach guarantees that each node, when assigned a row (set of public-private key products) in the matrix, will be able to verify another node when the node multiplies its private key with the other nodes public key. The complexity associated with compromising the network then becomes dependent on the attacker's ability to attain the private key matrix. To increase scalability and reduce this risk, the *multiple space Blom scheme* divides the network into two equal sets which do not share a common key. Furthermore, the *multi-space pre-distribution scheme* improves on these schemes by utilizing several predetermined private matrices and assigning each node a row in each. The use of public key cryptography in these schemes increases each node's ability to communicate at the cost of excessive computation, power consumption and storage needs (Dustin et. al, 2007).

Another scheme utilizing public key cryptography, is the polynomial based key pre-distribution scheme which allows nodes to generate link keys with the assistance of partially evaluated polynomials. With each node possessing a polynomial with *p-1* coefficients, nodes are able to generate a link key from the evaluation of their polynomial against their neighbour's polynomial. Although scalability, storage and communication

are good, the repetitive evaluation of polynomials increases power consumption and is computationally intensive. The localized encryption and authentication protocol however, encourages the use of pair-wise keys and distributes an initial key, KI, in the key setup phase. A node, SA, with a unique identifier, IDA, will then be responsible for generating its master key, KA = Hash(KI, IDA). In the shared key discovery phase, a node, SB, broadcasting (IDB, RNB -- random nonce) will receive a response from its neighbour, SA, containing (IDA, MAC (KA | RNB | IDA)), allowing SB to generate the key Kv = Hash(KI + IDA) before both nodes generate the session key Ks = H(Kv). This scheme may also be adapted to devise multi-hop pair-wise keys similar to the multi-path key reinforcement scheme mentioned earlier. As such it inherits the same benefits and disadvantages.

Moreover, *network wise key management schemes*, such as master key based solutions, where the same key is issued for all nodes, are similar to *single master key pair-wise schemes*. However, multi-tiered security solutions are characterized by varying degrees of protection corresponding to the data available at the respective level. One such solution (Slijepcevic et. al., 2002) divided into three tiers, utilizes a master key and strong encryption algorithms to secure mobile codes being exchanged by sensors at level 1. At level 2, sensor groups defined by location are assigned another key specific to the group to encrypt location information. Finally, at level 3, application data is secured with a hash of the original master key. Although scalable with low storage requirements (for storing the master and location key with a PRF), the approach may be easily compromised through the capture of one node's keychain (Dustin et. al, 2007).

Time Efficient Stream Loss-tolerant Authentication (TESLA) solutions increase their robustness with delayed key disclosure techniques where the key required for a future message is computed and disclosed with an earlier message. The computational cost associated with this technique has encouraged u-TESLA (Dustin et. al, 2007).

Although, Hossain. & Mahmud (2007) analyse key distribution management schemes for more powerful multi-cast WSNs to be used in a Vehicular Software Designed Network comprised of different tiers of Central Managers, Regional Group Managers, Control Groups, Base Stations and finally vehicles, the proposal considers a key-update scheme for moving nodes. Since vehicles are able to move between multiple groups localized by a base station's range, a vehicle node may request entry into a new group by sending a message to the new base station (the id of the base station is being transmitted frequently) consisting of the hash of (vehicle's id | the old group id | new base id | old group key), which may be verified by the new base station by sending a confirmation request from the old base station. If verified, this new vehicle would receive the group key for the new base station.

While exploring efficient key management distribution techniques for P2P live streaming applications, with a focus on centralized schemes that are hierarchical tree (HTS) based distribution networks, Liu X et. al, (2007), highlight the increased key distribution performance gain in a single-hop tree delivery key distribution scheme over multi-cast mesh networks. Their proposed scheme, Efficient and scalable Key Management Distribution Scheme (EKMD) describes a P2P network, where each node only shares keys with its immediate neighbour and the base station similar to the aforementioned IOS scheme. The scheme's performance in storage, scalability and

communication benefits from the single-hop nature of the tree network and manages sessions using a PKI model in what they term as the local view. A new node may join the network and increase the local view of its neighbours by first authenticating with the base station in an attempt to receive a group certificate. This may then be utilized to verify that the new node is apart of the group and communicate with peers at the respective level of the tree (a binary tree was utilized in the study). Furthermore, in order to revoke keys, the base station may transmit the new certificate or key with each of its immediate peers who will be responsible for decrypting the new key and subsequently encrypting this key with their key before forwarding. This model increases the possibility for a man in the middle attack however as each node will have the ability to re-encrypt their own message as the new key.

2.3.3 Authentication and attestation

While considering "severely" constrained nodes in WSNs as detailed in Figure *1* below, a set of Secure Protocols for Sensor Networks (SPINS) which consolidated previous protocols such as the Secure Network Encryption Protocol (SNEP) and μTesla was proposed as a constrained solution to two party data authentication, authenticated broadcasts, data freshness and confidentiality (Perrig et al, 2002).

Characteristics of prototype SmartDust nodes.

CPU	8-bit, 4 MHz
Storage	8 Kbytes instruction flash
	512 bytes RAM
	512 bytes EEPROM
Communication	916 MHz radio
Bandwidth	10 Kbps
Operating system	TinyOS
OS code space	3500 bytes
Available code space	4500 bytes

Figure 1: Technical Specification for SmartDust node.

Reprinted from SPINS: Security Protocols for Sensor Networks. Wireless Networks by Perrig, et al, (2002)

The proposal critiques asymmetric algorithms at the time such as RSA which requires immense storage, 1024 bits, and computation along with bandwidth, approximately 50-1000 bytes/packet as infeasible for the resource constrained devices. Furthermore, it focuses primarily on direct communication between base stations and nodes (with an extension for node to node communication) and base station broadcasts to all nodes in what they coin as a routing forest (multiple base stations surrounded by multiple nodes). With a trust model based on the assumption that the base station is computationally resourceful and secure, the proposed SNEP, which adds 8 bytes to each message, also utilizes a non-transmitted message counter. Furthermore, they achieve semantic security (an attacker will be unable to derive information from encrypted plain text if multiple encryptions of the same plain text are studied (Goldwasser & Micali, 1984)) by prepending each counter and DES-CBC (cipher block chaining) encrypted message with a random bit string. The counters are stored by the nodes and not transmitted to reduce the communication overhead. Furthermore, although the

communicating nodes share a master key, a derived key is used to encrypt messages. The counter enforces semantic security, data freshness, replay protection. Moreover, to accommodate for the high communication overhead of 24 bytes, computationally expensive PKI model, extensive bandwidth usage attributed to key disclosure in each packet and storage needs of the one-way key-chain, an improvement of TESLA, uTesla was proposed. By utilizing loose time synchronization with each node cognizant of the upper bound of the time synchronization error, each MAC becomes verifiable with the assumption that the generated MAC is a result of the current epoch. (Perrig et al, 2002).

Another approach, which addresses authentication at the link layer, is TinySec, a lightweight security architecture provided as a part of TinyOS operating system. It facilitates replay protection, message integrity and authentication with a MAC, confidentiality with encryption and semantic security through the use of initialization vectors (IV) (Karlof, Sastry, & Wagner, 2004). The architecture, aimed at supporting end-to-end security without crippling the ability of multi-hop networks to drop duplicate messages (since dense WSNs reporting to base stations may flood the base station and network communication with duplicates), supports two options, namely: authentication only (TinySec-Auth) and authentication and encryption (TinySec-AE), with the latter possessing the ability to encrypt the transmitted payload. Unlike, μTesla, Karlof et al (2004) argue the use of only the less computationally expensive Skipjack CBC-MAC with an encrypted IV instead of the counter mode since the latter is a stream cipher mode operation which implies that if the counter is repeated with a repeatable (small) IV, then more semantic security will be lost, unlike the block cipher which will only leak the difference in block length. It should be noted that block ciphers are not completely

impervious to attacks, some of which include the postfix equality check, last word and block decryption oracles to name a few (Vaudenay 2002). Again, because of the resource constraints of WSNs, TinySec utilizes a 4-byte MAC. Although shorter, the required brute force of 2^{31} attempts to receiving nodes on the network for verification over a 19.2 kb/s channel would take approximately 20 months to attack (Karlof, Sastry, & Wagner, 2004). The transmitted packet and associated field sizes described by the architecture is illustrated below in Figure *2*.

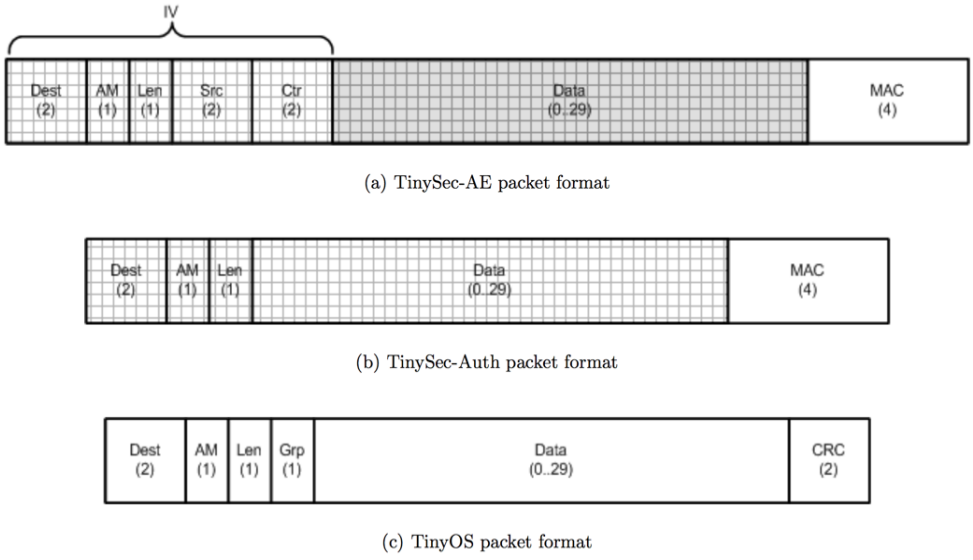

Figure 2: TinySec and TinyOS packet formats illustrating field size in bytes. Retrieved from TinySec: A Link Layer Security Architecture for Wireless Sensor Networks by Karlof et al, (2004)

Another protocol which aims to provide intrusion prevention, integrity, anti-replay, and authentication checks is the Authentication and Anti-replay Security Protocol (AASP). The protocol utilizes two (2) approaches, namely the Authentication Handshake

and the Last MAC Method (Gheorghe et al , 2010). The latter, utilizing a globally shared key, requires the communicating nodes to initially accept the first unauthenticated packet transferred by each node. Each subsequent message will then be verified with a MAC computed from the hash of the previous message and shared key. To tackle the initially unauthenticated packet from the last MAC method, the authentication handshake is used to present a Diffie-Hellman challenge (Gheorghe et al ,2010). The use of the last MAC challenge raises the issue of forward secrecy, however, even though it may assist with replay protection.

With the constrained nature of these devices, utilizing COAP over UDP is a more resourceful approach. In order to secure COAP and achieve end-to-end security, Datagram Transport Layer Security Protocol (DTLS), the TLS equivalent for COAP, was proposed as the main security protocol for COAP as COAPs (COAP with DTLS). The protocol was designed to secure application communication for lossy networks where handshake messages may not be delivered reliably or in sequence while retaining similar traits to TLS (Rescorla & Modadugu, 2012). DTLS achieves this with the use of two layers (illustrated in Figure *3*), the lower layer consisting of the record protocol and the upper layer, which may consist of the ChangeCipherSpec, handshake or alert protocols or application data. During the handshake process, the ChangeCipherSpec protocol is responsible for signaling to the record protocol layer that subsequent communication will use the newly negotiated cipher suite and keys. Furthermore, the alert protocol is used when transmitting error messages between peers and finally, the handshake protocol is used to negotiate compression methods, cipher suites and security keys. The lower record layer, contains a header which consists of the content-type, identifying the upper layer

protocol, and the fragment which contains the respective protocol data. With the assistance of its headers, the record protocol cryptographically protects the upper layers to achieve authenticity, confidentiality and integrity (Raza et. al ,2013). Moreover, to accommodate loss-insensitive messaging, explicit sequence numbers are added, to prevent anti-replay and facilitate message re-ordering. Furthermore, a re-transmission timer is used to resend messages for loss packets and a bitmap set of received records, similar to Internet Protocol Security (IPSec), is used to achieve replay detection. With UDP datagrams limited to approximately <1500 bytes, DTLS allows fragmentation for messages which in theory could become 2^{24}-1 bytes (Rescorla & Modadugu, 2012).

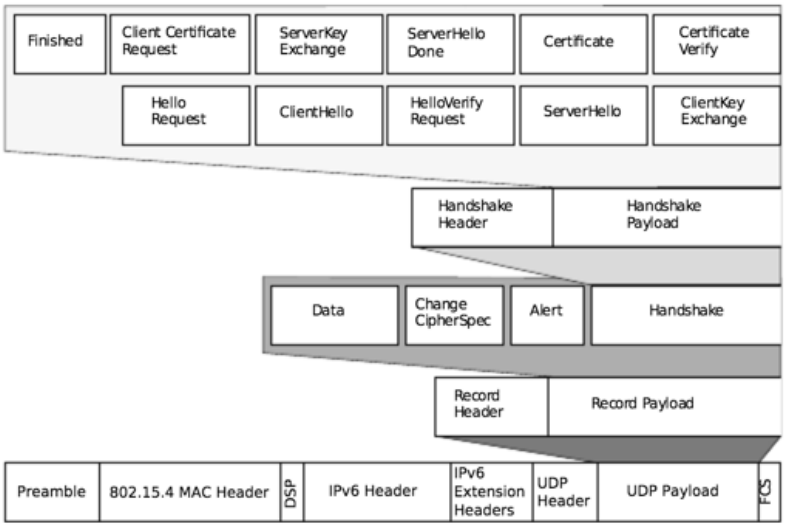

Figure 3: Layout of a packet secured with DTLS.

Retrieved from Lithe: Lightweight Secure CoAP for the Internet of Things by Raza, Shafagh, Hewage, Hummen, & Voigt (2013)

Lithe, is another proposed protocol, which compresses the recommended security for COAP communication, DTLS using 6LoWPAN Next Header Compression

(6LoWPAN-NHC). Raza, et. al. (2013) argues that computation is a less expensive use of a node's power than communication and shows that 6LoWPAN-NHC reduces the 13 bytes added by the DTLS record protocol and the 12 bytes added by its handshake to 5 and 3 bytes respectively. By compressing the initial handshake, subsequent transmissions will benefit from a reduced communication length since DTLS will encrypt these messages. Moreover, they explore the compression of the record and handshake with a single encoding (6LoWPAN-NHC-RHS) and the compression of the record protocol when the fragment contains application data (6LoWPAN-NHC-R).

An alternate approach to the previously described cryptographic schemes is the Implicit Security Authentication Scheme which utilizes unexpected node behaviour to identify malicious nodes (Chen et al ,2010). The approach considers a base station, watchdog stations and a fixed set of sensor nodes, which have previously recorded behaviour data before being deployed. The behavior recorded comprises of a set of vectors, each consisting of time, an event and a set of monitored parameters. These monitored parameters may be reported from various levels of the OSI stack such as the expected radio transmission range from the physical layer, error control and error rate occurrences at the data link layer, number of successful deliveries at the network layer or time synchronization activities at the application layer. Computationally competent watchdog nodes deployed in WSNs may then monitor current node behaviour and simultaneously compare it to previous data for the specific node to develop an authentication score from a learned machine model. The approach was inspired by the fact that a compromised node, may still be able to authenticate with its set of compromised keys (Chen et al,,2010). Although insightful, the approach may not be

feasible for many WSNs as harsh sensor environments may produce unpredictable node behaviour and the approach does not consider the addition of new nodes or replacements to the network to name a few.

The Short Message Authentication Code Check (SMACK) is a proposed security extension for COAP aiming to reduce DOS and battery exhaustion attacks with a verifiable 4 -byte HMAC inserted into the already available token field in COAP headers (Gehrmann et al 2015). The HMAC consists of a 2-byte request ID and 2-byte validity check. The validity check is the output of a HMAC seeded by three keys A, B and C where C changes every modulus of the session length. With the assistance of a pre-computed lookup table of 36 bytes, supporting Galois field sizes of 1 to 16 bytes for the MAC calculations used, keys are combined with the message id, request id, version, type, token length and code of the COAP header. While A and B are the first and last 16 bits of the generated session key, C is taken from Session Key J= PRF(Session Key). Similarly, the session key is the result of a PRF(Initial MID, Master Session key) given to all network devices. The master key and seed used to generate the master session key is maintained by the KDC or base station.

2.4 Blockchain

2.4.1 Overview

The blockchain represents a distributed and replicated structure shared, contributed to and verified by networked peers. With the exception of the genesis block, the starting block of transactions for the chain, each subsequent, time stamped block is identifiable by its cryptographic hash that links it to its predecessor and may be used to

validate the block using a consensus algorithm (Christidis & Devetsikiotis, 2016). With the advent of Bitcoin, the use of this data structure to manage its cryptocurrency has garnered widespread acceptance, speculation, and intrigued many researchers and professionals alike to identify the applications in different domains. Bitcoin, uses these verifiable blocks, each consisting of transactions, as a decentralized yet verifiable, peer maintained ledger, allowing each node to verify the status of any account in their network by perusing the history of the blockchain. At the time of writing, many alternative networks and approaches to interacting with the blockchain have been implemented utilizing different consensus and operating mechanisms with some promising domain specific concerns. Some of these include Bitcoin, Ethereum, Tendermint, Ripple, Hyperledger and Intel Ledger's Sawtooth to name a few.

In this decentralized network (no mediation by a middle or single trusted entity), each node aims to improve its global view of the network's state. A typical round in this iterative process includes peers, connected nodes (usually persons interested in interacting with the respective network), committing signed transactions (changes to the current state of the network or payments in terms of bitcoin) to the network. The networks utilize public key cryptography to attain authentication and integrity, requiring each transaction to be signed by its owner with his/her/its respective private key. This transaction is then broadcasted to neighbouring peers who validate the transaction before broadcasting the same to their peers. After an agreed time interval, participants group acquire transactions into candidate blocks to be verified using the network's consensus algorithm or to be "mined". Throughout this process, peers typically aim to validate the transactions in the respective block and the provided cryptographic hash or link to the

previously verified node. If successful, the block is committed to the trusted ledger of blocks and the process is repeated (Christidis & Devetsikiotis, 2016).

The inherent properties of the blockchain around decentralized trust and the ability to facilitate the popular cryptocrurrency has attracted much attention in several domains. Of the many implementations thus far, is a taxonomy which includes private and public networks, (Kosba, Miller, Shi, Wen, & Papamanthou, 2015), Unspent Transaction Outputs (UTXO), smart contracts within an account based model as seen in Ethereum (Christidis & Devetsikiotis, 2016) and Turing incomplete or quasi-turing complete (Popejoy, 2016) languages. Furthermore, distributed ledgers may be permissioned (Walport, 2016), privacy-preserving (Kosba, Miller, Shi, Wen, & Papamanthou, 2015) or utilize more relaxed consensus models to fit a particular domain in a trusted environment. Irrespective of the taxonomy, blockchains are able to provide a verifiable and auditable consensus on assets in a distributed and possible untrusted peer-to-peer (P2P) environment.

2.4.2 Consensus

As mentioned earlier, each participating node aims to achieve a single global view of the network through mining, however with each node possessing so much independence, it is possible to develop forks (branches) of the original chain. In the event where two nodes identify that their blockchain is different, the most trusted fork, usually the longest and most verifiable, is accepted as the main chain and the other discarded. This is useful as will be discussed later especially in cases where new nodes connect to the network or an existing node may have been dormant or disconnected for some time.

In an attempt to avert Sybil attacks, where a single entity with multiple identities may out vote the entire network on the legitimacy of an illegitimate block to attain control of the network, blockchain implementations such as Bitcoin has made mining computationally "expensive". This implies that it is possible for any node to have their assembled block be considered as the next mined block in the network given that they find a random nonce in the block's header that will allow the SHA-256 of the header to include the amount of leading zeroes expected by the network's difficulty level (Christidis & Devetsikiotis, 2016). This computational expensive process to achieve consensus is considered the proof-of-work (PoW) to create a legitimate block and may be easily verified by peer nodes since this is a one-way cryptographic hash. Furthermore, the consensus protocol in this implementation also declares that whenever forks are identified by peer nodes, the longest chain should be accepted as the correct chain as mentioned before, being that it would consist of the most blocks or the most PoW. Another crypto-currency, Litecoin utilizing blockchain as a backbone is similar to Bitcoin, with the exception of the possible hashing algorithms included such as Blake-256 (Henzen, Aumasson, Meier, & Phan, 2011) and scrypt (Percival,2009).

Proof of Stake (PoS) is an alternative consensus protocol that is less computationally intensive (leading to a faster network and mining process) but is heavily reliant on a node's overall balance (amount of cryptocurrency owned) or ownership of the network. By allowing the node with the greatest stake or balance in the network to make the most decisions as the trusted entity, issues such as the speed and storage requirements of the network may be reduced. However, this may open the network to

other attacks, especially if the malicious node has acquired enough of the currency to influence the network.

Another approach inheriting traits from PoS and PoW is Delegated Proof of Stake (DPoS). In DPoS, participants of the network are considered shareholders with the ability to delegate their vote of legitimate blocks to a rotating set of active participants similar to how shareholders are represented by board of directors. These directors would then be responsible for performing the PoW and communicating the results. With less voting members, the network inevitably increases its mining power and overall reduces the computational power required by all nodes ("Delegated Proof of Stake", 2016).

In distributed networks, byzantine fault tolerance, which may be solved using the Practical Byzantine Fault Tolerance algorithm (PBFT), is an important benchmark which evaluates how the network handles the byzantine general problem or the loss of a node or service in a manner which presents the same issues differently to each observer (AlZain, Soh, & Pardede, 2013). This is a critical issue since these decentralized networks aim to collaboratively achieve consensus through mining. Bitcoin aims to addresses this issue with the aforementioned PoW, however if legitimate transactions were contained in blocks on the shorter, rejected fork, the network would ultimately lose this history or part of the ledger. Alternatively, other crypto-ledgers, Juno and Tendermint utilize variants of the BFT or consensus such as Raft and Tangaroa respectively. PBFT is more suitable for faults where up to one-third of the nodes are faulty (f) and therefore will require a minimum of $3f + 1$ nodes. Tendermint capitalizes on this by dynamically rotating and varying the participants and validators in a round robin manner when more than a third of the network is loss. Ripple approaches this issue differently with the use of Unique Node

Lists (UNL). Instead of querying the entire network, nodes query there UNL or sub-network, thereby reducing latency and potentially increasing its tolerance to $5f + 1$. Similar to UNL, Multichain utilizes a whitelisted set of nodes operating on a consensus factor called "mining diversity". Mining diversity describes with leniency how many blocks a node should wait before attempting to mine again. If a node pre-empts this factor, their attempt to contribute is rejected (Christidis & Devetsikiotis, 2016).

Some blockchain implementations such as Ethereum and Kadena add additional properties to their implementations such as the ability to support smart contracts (Popejoy, 2016). Smart contracts, a concept introduced by Nick Szabo in 1994 are "a computerized transaction protocol that executes the terms of a contract'' (Szabo, 1994). Essentially, smart contracts enable parties (nodes or persons) to interact with contractual clauses embedded or stored in the respective blockchains (at an address to be executed later). This increases visibility and reduces the need for a trusted third party, potentially creating Decentralized Autonomous Organizations (DAO). Both implementations take different approaches to providing this functionality, for example, Ethereum enables a quasi-turing complete language encoded using Recursive Length Prefix (RLP) on its Virtual Machine (EVM) (Wood, 2016) while Bitcoin's scripting language and Kadena's implementation, Juno, utilizing Pact as its language, features a turing-incomplete language which they argue is more secure than Ethereuem since it presents more limits (Popejoy, 2016).

2.4.3 Oracles

Bitcoin defines oracles as external “servers possessing key pairs which are able to sign transactions upon request when a user-provided expression is true” ("Contract - Bitcoin Wiki", 2015). Oracles become necessary in this cryptographic consensus, as the blockchain is unable to perceive its environment or world events without the inputs of its stakeholders or human participants. In order to create a more autonomous operation, outside devices or servers are used to submit transactions to the blockchain. Instead of being completely independent, bitcoin proposes and utilizes several methods to attest and improve the level of trust these oracles contribute. A typical re-enactment of an oracle’s contribution may be identified in the pre-sale of crops in an automated green house. A sensor or oracle capable of monitoring crop yield may attest the availability of a crop by encoding it as a digital asset on the blockchain, thus allowing shoppers to pre-order their yield months in advance and monitoring the crop throughout its lifecycle. Oracle integrations tie the advancements in automated systems and IoT into the integrity the blockchain is able to provide.

Trust remains an issue in this model however, especially if the oracle, being used by a disbursement bureau, possesses knowledge of the owner’s assets and intends to withhold its signature in return for remuneration. Bitcoin has proposed the used of encrypted pre-conditions which only require that the oracle receives before hand, the hash of the pre-condition and the public key, which may be used in the future to verify a correct request. Moreover, the use of trusted hardware as discussed earlier in authentication or the use of multi-signature contracts have also been proposed, where a majority of oracles may vote on the pre-condition ("Contract - Bitcoin Wiki", 2015).

Oraclize is one such service aiming to serve as a “provable-honest oracle” that will enable smart contracts to access resources from the internet ("Oraclize API Reference", 2016). Since the execution of smart contracts occur on the blockchain or in virtual environments, disconnected from world events, Oraclize provides a smart contract which acts as a service provider that uses TLSNotary to prove its honesty. Other smart contracts may request data (from a HTTPS URL, Wolfram API to name a few) and optionally request a cryptographic proof using TLSNotary (Gibson, 2014) which may be accessed from a location stored on the InterPlanetary File System (IPFS). IPFS is a distributed P2P file system that provides a content-addressed, block storage model combined with a distributed hash table, self-certifying namespace and a block exchange that has been incentivized (Benet, 2016). This design enables IPFS to provide a decentralized, replicated, self-certified and version file system that allows users to enjoy access similar to bittorrent clients or the git version control system. A similar yet pre-mature service, Orisi, aims to provide multi-signature oracle confirmations utilizing Bitmessage as its underlying communication protocol, limiting oracles to also be participating blockchain nodes. Bitmessage utilizes a hash of the node’s public key as the node’s id (potentially obscuring IP addresses), allowing messages to be easily authenticated by verifying the sender’s id. Although messages are transferred similar to how blocks are exchanged in bitcoin, DOS attacks are further reduced since each message transferred would undergo an additional proof of work, with each participant in the network attempting to decode a message to prove that they are the recipient. This naturally raises scalability concerns, in which they propose to divide subsequent nodes and their communication into streams after a network threshold size has been achieved.

Streams of divided nodes would be divided into sub-trees which would then require nodes wishing to communicate with nodes in other sub-trees to broadcast up and down the tree as necessary (Warren, 2012).

2.4.4 Resource Constraints

With the increased computing power incorporated into IoT, incorporating the blockchain, smart contracts and oracles into the mix holds many possibilities. These could automate asset tracking, supply chain management, contract signing, fund disbursement with the assistance of sensors, servers and actuators (Christidis & Devetsikiotis, 2016). In the aforementioned consensus models, blockchain nodes require all or a delegated set of participants in PoW and DPoS respectively to perform computationally expensive mining to determine the legitimacy of new blocks, which are replicated on each node. This computational and storage constraint that requires a significant amount of power and network connectivity has already been identified as an issue with the blockchain's public network amassing over 75,043 MB as of July 9, 2016 as illustrated in Figure *4* below("Bitcoin Charts - Blockchain.info", 2016).

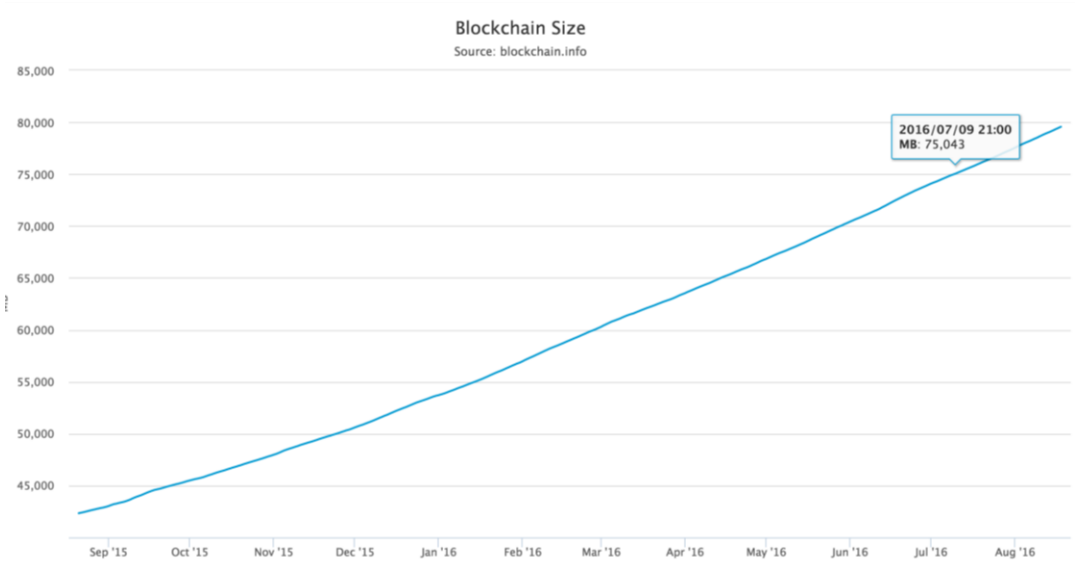

Figure 4: Blockchain Storage Size as of July 9, 2016.

Retrieved from Blockchain.info ("Bitcoin Charts - Blockchain.info", 2016)

Ethereum has run similar benchmarks to evaluate its block processing which considered their PoW (specific to their quasi-turing complete execution model), transaction signature checking, receipt verification, uncle validation and database insertion on 1,000,000 blocks on their public frontier network (first public Ethereum network). The test was conducted in Ubuntu 14.0.4 LTS on a virtual cloud server with 4GB RAM, 2 Core processors, 60 GB SSD and 4TB transfer rate (network) to yield the results in Table *2* below("Ethereum Benchmarks", 2016).

Table 2: Ethereum Benchmarks on Frontier Network for each client

-	Eth	EthereumJ	Geth	Parity
Time	4h 33m	7h 7m	8h 43m	2h 31m
CPU (avg)	123%	90%	70%	107%
Memory (avg)	921MB	3.168GB	1.5GB	365MB

Note. Reprinted from "Ethereum Benchmarks", retrieved from https://github.com/ethereum/wiki/wiki/Benchmarks Copyright 2016 Ethereum Foundation

Efforts to reduce the storage requirements include the use of Simple Payment Verification (SPV) Clients in bitcoin or the Light Ethereum Sub-protocol (LES) in Ethereum (not yet implemented at the time of writing). LES considers partially light clients which still participate in the consensus and fully light clients which do not participate at all. Low capacity nodes processing approximately 512Bytes/min may still verify the integrity of the chain by performing a reverse-hash lookup in their respective merkle trees ("Light Ethereum Subprotocol (LES)", 2016), however nodes tend to only retrieve block headers, similar to the eth/63 fast synchronization algorithm which downloads headers then verifies the set by verifying a random block from the newly downloaded set (Szilágyi, 2015). SPV clients operate in a similar fashion, however implementations such as bitcoinj tend to store the transactions related to the owner's wallet also as an additional step to verify the status of the owner's account. As a result, SPV clients are susceptible to Finney attacks where they may have accepted a double

spent transaction because they have trusted a mining node which isn't broadcasting several spent transactions (Hearn, 2016).

In essence the associated resource constraints of the decentralized trust attained through verifiable consensus in a fault tolerant network has made it infeasible to embed and reap the rewards of the technology in constrained devices (sensors and actuators). With the relatively young age of blockchain technology and the constraints of smaller IoT devices still deemed applicable, additional research is required in incorporating these devices into blockchain technology.

2.5 Conclusion

The ubiquitous use of micro-sensors has resulted in many variations as it pertains to resource constraints. Furthermore, when considering numerous micro-sensors, it has become more pragmatic to employ Wireless Sensor Networks as they are easier to deploy and maintain. These networks however require different design considerations, notably: fault tolerance, scalability, hardware constraints, network communication, environmental considerations, production costs and power consumption. Moreover, the use of a wireless medium increases the likelihood of the explored threats to occur such as replay attacks, session hijacking and worm-holes.

Due to device constraints, authentication and attestation schemes have been designed with the communication medium in mind and are often integrated with a particular layer of the OSI model. In particular, many sensors have incorporated wireless communication protocols such as 802.15.4, IPv6 and 6LoWPAN. While there are many concerns, the overlapping concerns of these authentication schemes have aimed to assert identity,

integrity, data freshness and replay protection with the assumption that the base station is computationally resourceful to enforce security. Moreover, while application specific, random or deterministic pairwise key management schemes are less computationally intensive than PKI methods for WSNs. These schemes however rely on a trusted authority or base station and use various re-keying techniques to reinforce the integrity of the network.

The literature review therefore highlights several concerns and opportunities. Notably, the cost (monetary, communication overhead) associated with key management and key updates especially in post deployment and the costs associated with securing base stations. In addition, the trust (i.e. to remain uncompromised) and fault tolerance level required for base stations connecting constrained sensors to aggregation networks is another area of concern. Moreover, the integrity of data is more likely to be lost while propagating the data to different participants in a network topology, resulting in the final recipient relying only on the trust of its closest peer. It is also evident that due to the design considerations for WSNs, one solution may not fit all and the applicability of the constrained authentication schemes may not be secure enough for more resource competent nodes. However, although current blockchain implementations are too resource intensive for constrained sensors, the blockchain addresses several issues such as byzantine fault tolerance, provenance, integrity and auditability. It is therefore an opportunity to explore the integration of existing security primitives, authentication schemes into a WSN that integrates with the blockchain.

Chapter 3: Proposal

With the increased usage of constrained devices to provide valuable yet critical information, it has become necessary to not only securely acquire data but prove its source. The Lightweight Blockchain Authentication for Constrained Sensors (L.B.A.C.S.) addresses this need for provenance of sensor data within highly distributed and semi-trusted environments that utilize blockchain technology as a backbone to achieve consensus. Blockchain technology, currently being researched and commercialized, possesses attractive attributes for untrusted decentralized environments but remains resource intensive as highlighted in the literature review. At the time of writing, the author is yet to identify proposals targeting the provenance of constrained oracle data on the blockchain, especially when most models addressing WSN concerns utilize hierarchical networks which completely trust the base stations that forward communication on behalf of the sensor nodes. The aforementioned model is useful for aggregation networks but also requires greater security from the base stations involved. This proposal will therefore define the scope, assumptions, threat model and security objectives while describing how LBACS addresses this issue.

LBACS is a lightweight authentication scheme for wireless constrained oracle sensors communicating with semi-trusted blockchain competent nodes. The scheme, operating above the transport layer of the OSI model, enables interoperability with existing schemes and blockchain implementations, while providing provenance and auditing in fault tolerant networks. Considering the limited power, storage and communication abilities of WSN nodes, the scheme proposes lightweight computations

to achieve data freshness, semantic security, source repudiation, forward and backward secrecy. Specifically, this proposal will include methods for key generation and distribution (pre and post deployment) for WSNs interacting with the blockchain, key revocation and authentication (peers, groups and multi-hop). Although possible, this proposal considers methods enabling encryption, packet routing and confidentiality as out of scope due to time constraints of this thesis.

3.1 Security Objectives

The Lightweight Blockchain Authentication for Constrained Sensors (LBACS) aims to achieve the following security objectives:

1. **Authentication:** A receiver should be able to verify a claim made by the source of a message forwarded through n communication links where $n \in N$. (N represents the set of natural numbers)
2. **Blockchain Provenance**: An interested party, without access to a constrained oracle sensor or its network, may prove that data published on the blockchain originated from the oracle sensor.
3. **Integrity:** The generated authentication tag should attest to the contents of the transmitted message.
4. **Replay Protection:** The scheme should facilitate the identification and rejection of replay attacks.

5. **Weak Backward Secrecy:** Without the permission of the key distribution centre, previously used pairwise keys should not be discoverable by an adversary or newly deployed members of the network
6. **Weak Forward Secrecy:** Future keys will be inaccessible to revoked peer or group members.
7. **Universal Forgeability**: An adversary, not apart of the network, should not be able to generate a correct authentication tag for all messages.
8. **Identity Revocation:** Any participating sensor node, semi-trusted base station or trusted authentication entity may have their identity revoked by a valid certificate authority.
9. **Auditable:** Communication between trusted authentication entities, semi-trusted base stations and sensors should be available for examination.

3.2 Notation

The following notation will be used throughout this chapter to abbreviate the concepts, terms and identifiers utilized within the LBACS scheme.

Table 3: LBACS Notation

Notation	Definition
CA	Certificate Authority
TAE	Trusted Authentication Entity
STBS	Semi-Trusted Base Station and Blockchain Node
BN	Blockchain Node
SN	Sensor Node
KDF	Key Derivation Function
PRNG	Pseudo-Random Number Generator
$PK_i(A, B)$	Pairwise key "i" between A and B where: "i" $\in N$ "i" represents the current iteration of the key A and B are entities (either TAE, STBS, SN) where

Notation	Definition
	A ≠ B. $PK_i(A, B) = PK_i(B, A)$
$RS_i(A, B)$	Random seed "i" shared between A and B where: "i" ∈ *N* "i" represents the current iteration of the key A and B are entities (either TAE, STBS, SN) where A ≠ B. $RS_i(A, B) = RS_i(B, A)$
PPK(A)	Public/private key pair assigned to A where A is an entity from the set of STBS and TAE apart of the system (valid/invalid)
PNK_A	Private Node Key for A where A is a TAE or Sensor node
$PRNG_A$	Pseudo Random Number Generator being used by A where A is a TAE or Sensor node
No(A)	Number of A where A will be defined
MAC	Message Authentication Code
HMAC	Keyed Hash Message Authentication Code
INTENTION_BIT_SIZE	Size in bytes of the intention bit used in the LBACS authentication tag
SESSION_SIZE	Size of an LBACS authenticated session
LBACS_TOKEN_SIZE	Size in bytes of the LBACS authentication tag/token
HMAC_SIZE	Size in bytes of either HMAC within the LBACS authentication tag
HMAC_PIECES_COUNT	Number of pieces of the generated HMAC before truncation (implementation specific)

3.3 Overview and Assumptions

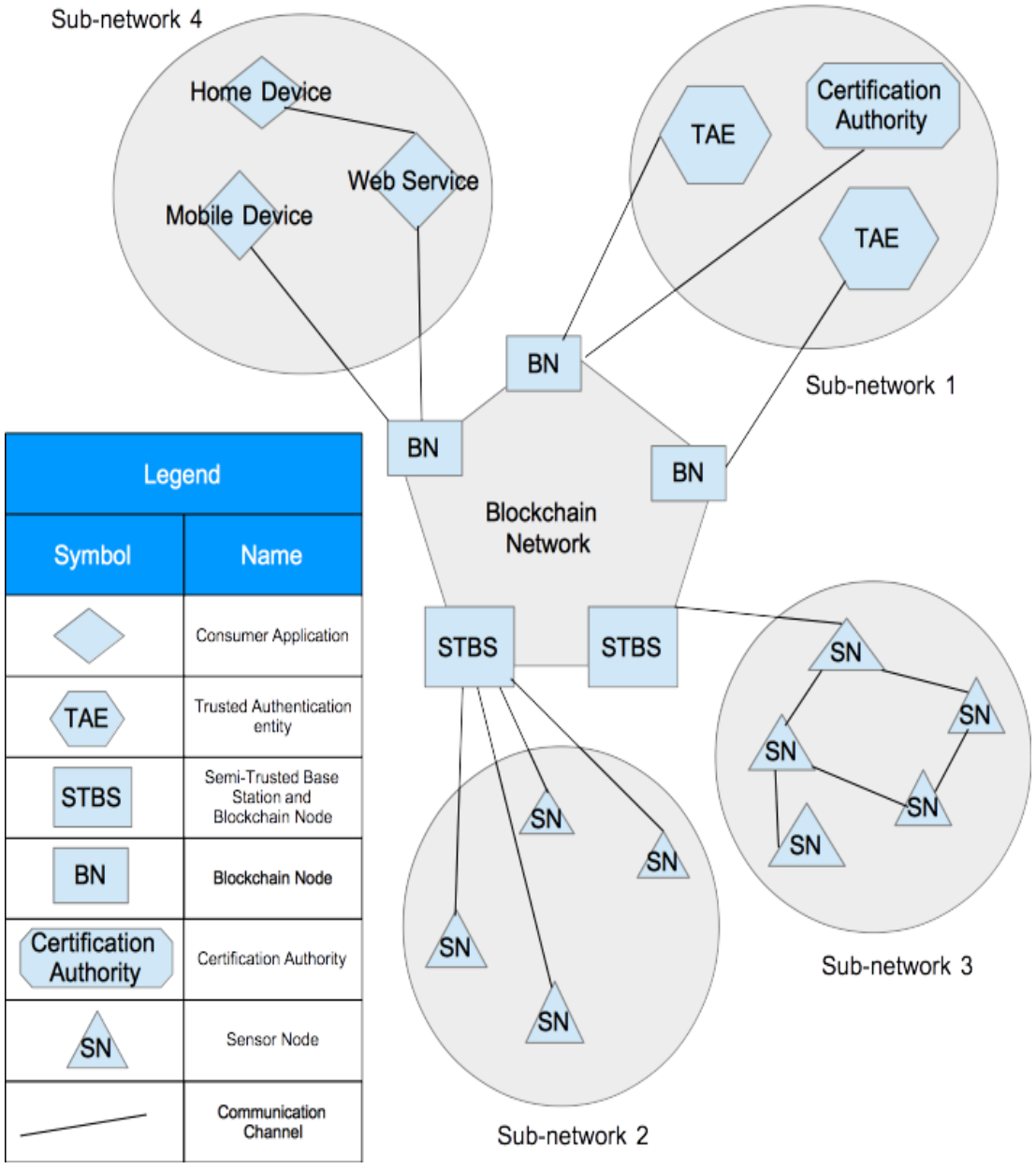

Figure 5: LBACS Network Overview

Figure 5 above illustrates four types of sub-networks connected by a larger blockchain network. In operation, multiple types of each sub-network may co-exist on the

same blockchain or form side-chains with other blockchains based on the respective blockchain implementation. Each sub-network has access to the blockchain network through a Blockchain Client Node (BN_{1-4}) which also acts as Semi-Trusted Base Stations (STBS) for sub-networks 2 and 3.

Sub-network 1 (SbN_1) is a critical component of LBACS' architecture. It is regarded as the most trusted and secured sub-network and as a result, possesses the most authority in the decentralized group. It consists of the Key Generation and Distribution Centre and Trusted Authentication Entities (TAEs) along with the elements necessary to facilitate the PKI model to be discussed later such as a Certification Authority (CA) and Registration Authority (RA). Primarily this network will be responsible for the following activities:

1. maintaining a directory of valid public keys of STBS relaying sensor messages
2. verifying MAC of sensor messages
3. provenance of sensor data submitted to the blockchain
4. generation and distribution of symmetric keys to STBS and constrained oracle sensors
5. generation and distribution of private-public keys to STBS (dependent on the PKI model)

On the other hand, sub-network 4 (SbN_4) consists of consumer applications and services (web applications, servers, other IoT, etc.) that require information provided by oracles. This network is typically untrusted and only awaits the non-repudiation of messages submitted by STBSs from a member of SbN_1.

Both Sub-network 2 (SbN_2) and Sub-network 3 (SbN_3) are networks consisting of constrained oracle sensors (SN). However, they have been separated to illustrate the

pragmatic use of various topological schemes in various types of WSNs. Specifically, SbN_2 represents WSNs where each sensor node is able to communicate directly with the base station (hub and spoke model) while nodes in SbN_3 use routing mechanisms to forward or broadcast packets until they arrive at the STBS (daisy-chained model). Sensor nodes that are a part of these networks are assumed to be constrained devices with resources similar to those identified in Figure *1* on page 27 where it may be impractical to involve significant computation for authenticating packets such as the more heavyweight computations of PKI models. Finally, these networks consist of more computationally resourced base stations, which are responsible for relaying sensor data to the blockchain. Due to the limited constraints of the participants of these networks, it is assumed that it is also economical to deploy multiple base stations supporting the same WSN. However, these base stations may experience the same environmental factors as the sensor nodes and with less constraints in an untrusted environment, it is also assumed that these base stations may be targets of attack or harsh environmental conditions as discussed earlier in the literature review.

3.4 Key Management

3.4.1 Pre-Distribution

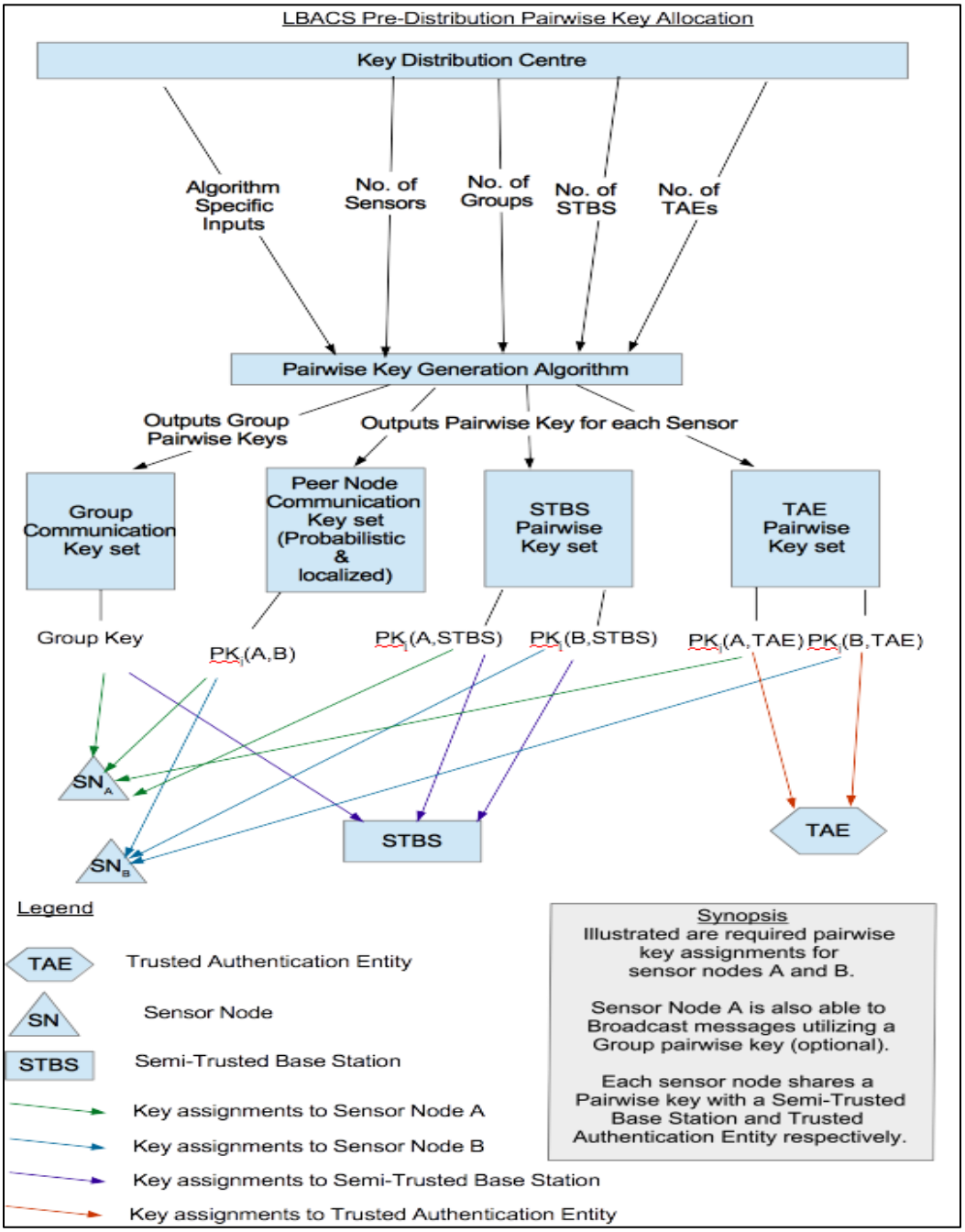

Figure 6: LBACS Pre-Distribution Pairwise Key Allocation

LBACS utilizes a combination of pairwise symmetric and asymmetric key approaches to provide provenance of constrained oracles sensors on the blockchain. Prior to deploying base stations and WSN nodes in SbN_2, as illustrated in Figure 6 on page 53, the KDC issues pairwise keys between these parties to enable each to authenticate communicated packets. These pairwise keys are accompanied by a random seed value for a PRNG that will provide an expected nonce capable of reducing the communication overhead and power consumption when transmitting the MAC as will be discussed further. Similarly, the KDC will issue a keyset consisting of a pairwise key and seed between Trusted Authentication Entities (TAE) in SbN_1 and sensor nodes as these TAEs will be responsible for authenticating sensor data published on the blockchain. It should be noted that the STBS does not share the same keys shared between the Trusted Authentication Entities (TAE) and oracle sensors. This additional storage requirement reduces the possibility of man-in-the-middle attacks as a result of a compromised STBS, a common risk in WSNs. In P2P or multi-hop networks represented by SbN_3, each sensor would receive a random set of N pairwise keys and seeds to communicate with *N* peers with a probability of *p* assignments (where *p* is rational number). To further optimize a node's ability to communicate with nearby peers, the chosen set of keys would be picked randomly from expected nearby nodes or the expected sensor group for multi-cast networks. The resulting sensor node in these networks would require storage for the keyset of *Np* peers in addition to the TAE in order to provide provenance on the chain. This is similar and reaps the benefits of the earlier discussed *probabilistic pairwise key distribution scheme* proposed by Hwang et. al. in 2004 (Dustin et. al, 2007) but with an additional keyset for provenance on the blockchain. As with many scalable networks, it

may not be feasible to physically inject the necessary keyset(s) into the participant nodes. Instead, the KDC, within a trusted environment, may inject a key setup symmetric key (to be discarded after key setup phase) and broadcast the necessary keyset to each node to bootstrap mass deployment. Finally, each node would receive an additional key, which would allow the node to derive future pairwise keys broadcasted from the STBS using a Key Derivation Function (KDF) seeded with the key and other parameters. An example of the required key storage for a sensor node, A, with the ability to communicate with two additional peers, namely sensor node B and C has been illustrated in Figure 7 below.

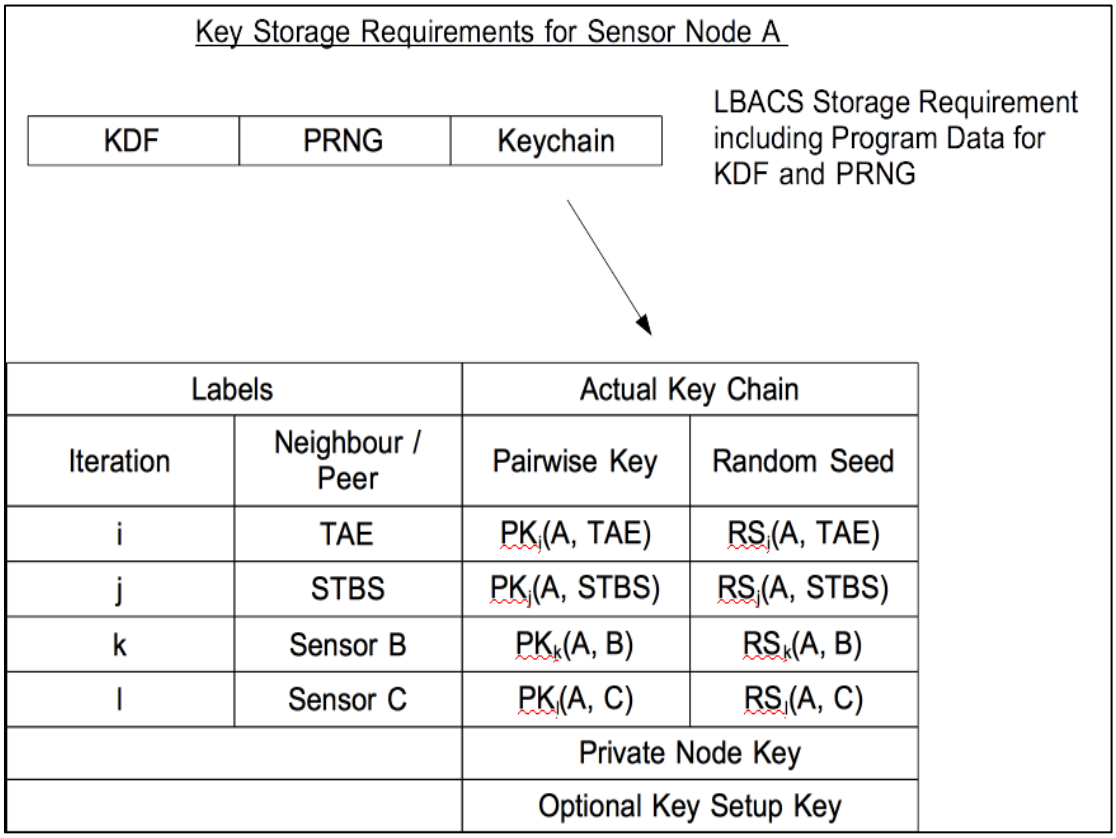

Figure 7: Required Key Storage for LBACS Sensor Node

In order to facilitate signing and signature verification, STBS and TAE, assumed to be computationally competent, would generate their public-private key pair and submit to their public keys to a certificate authority for approval as illustrated in Figure 8 below. Alternatively, but less secure, if a STBS is not computationally competent to generate its key pair, it may be issued from the KDC. Furthermore, in order to enable a more fault tolerant network that is auditable and thus less likely to be compromised, the blockchain would be used as the decentralized directory storing a list of valid public keys for respective TAE, SBTS and registration or certificate authorities who would be responsible for publishing updates to this list. By publishing this directory on the blockchain, any STBS, TAE or consumer application in SbN_4 would be capable of verifying signed transactions in the network and dynamically updating communication partners where necessary in the case of sybil or byzantine attacks.

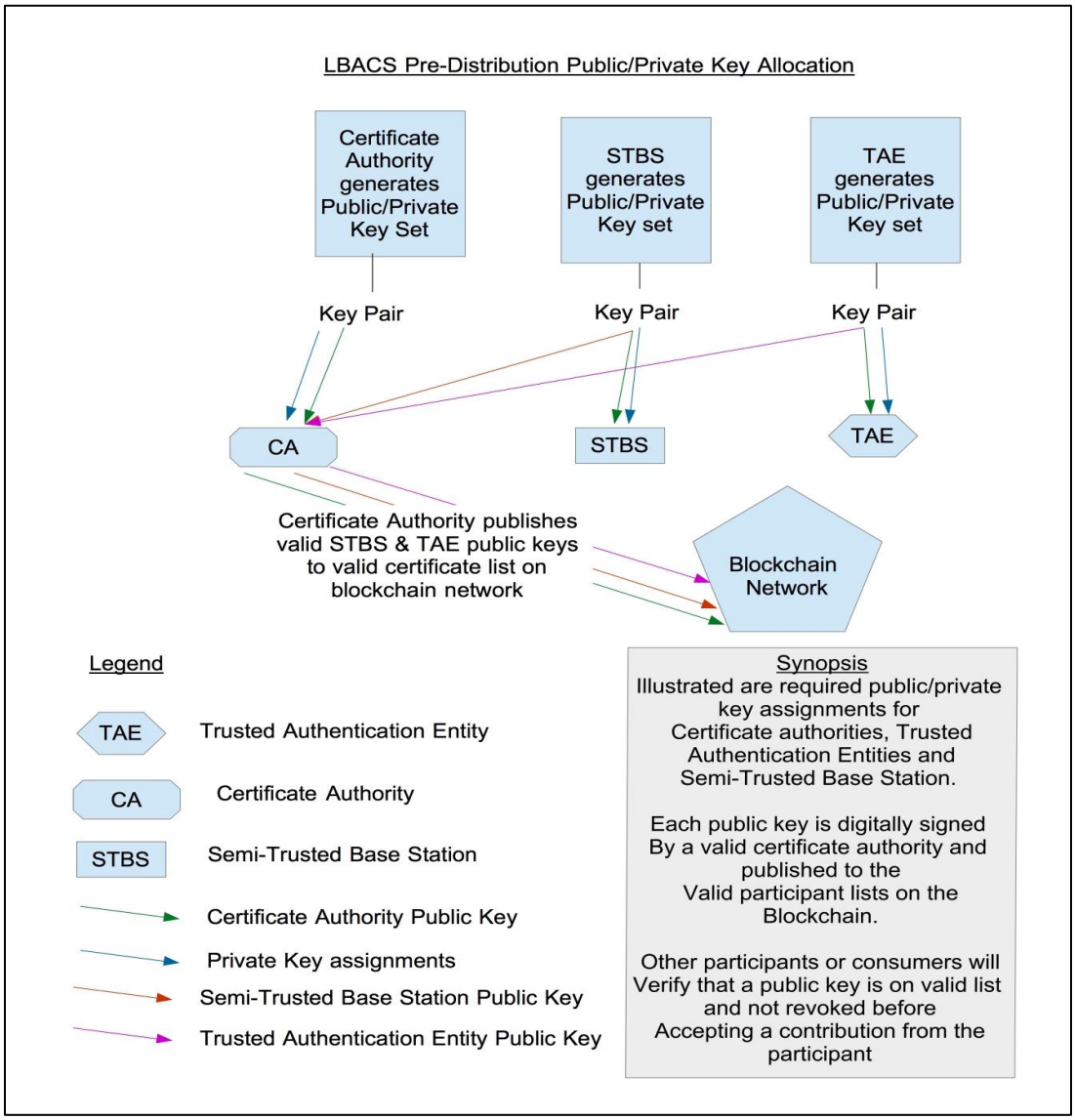

Figure 8: LBACS public/private key pre-distribution

3.4.2 Post Deployment

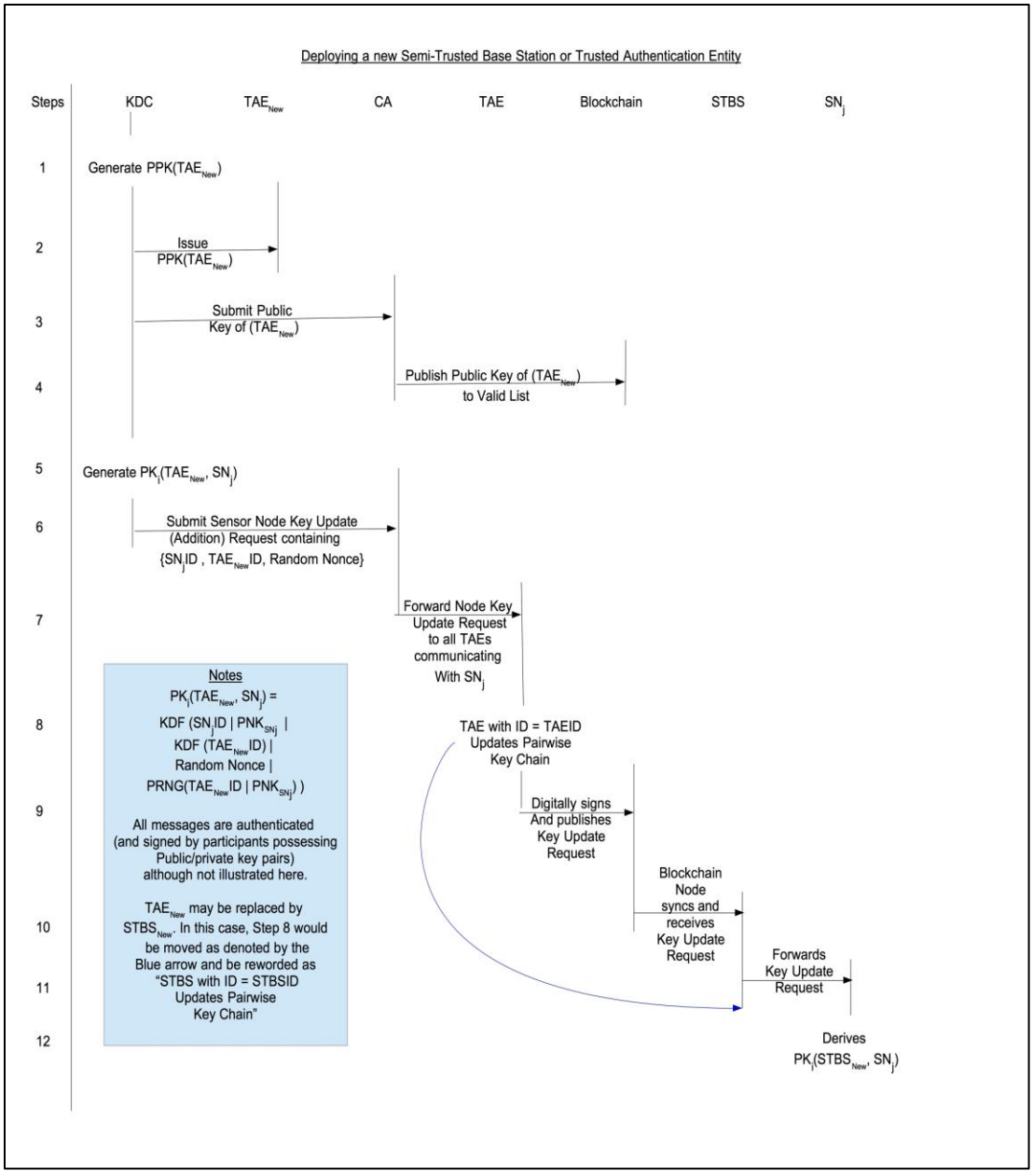

Figure 9: LBACS Post Deployment: Addition of a STBS or TAE

In order to add a TAE post deployment, the certification authority would update the valid TAE registry on the blockchain with the new TAE's public key. Furthermore, the KDC would assign a set of symmetric keysets (pairwise key and seed) derived using the KDF (illustrated in the notes section of

Figure 9 above) to the TAE. Finally, to update existing WSN nodes, the KDC with the assistance of TAEs, publishes the pseudo-id of the new TAE along with a random nonce on the blockchain. Subsequent to verifying the signature on the TAE's request, the STBS broadcasts this information to WSN nodes. WSN nodes then derive the new pairwise key using KDF (SN_jID | PNK_{SNj} | KDF ($TAE_{New}ID$) | Random Nonce | PRNG($TAE_{New}ID$ | PNK_{SNj})) . A STBS would be added similarly to TAEs but with less pairwise keys based on the expected number of WSN nodes the base station would be communicating with. The addition of a new TAE or STBS has been simplified in

Figure 9 above.

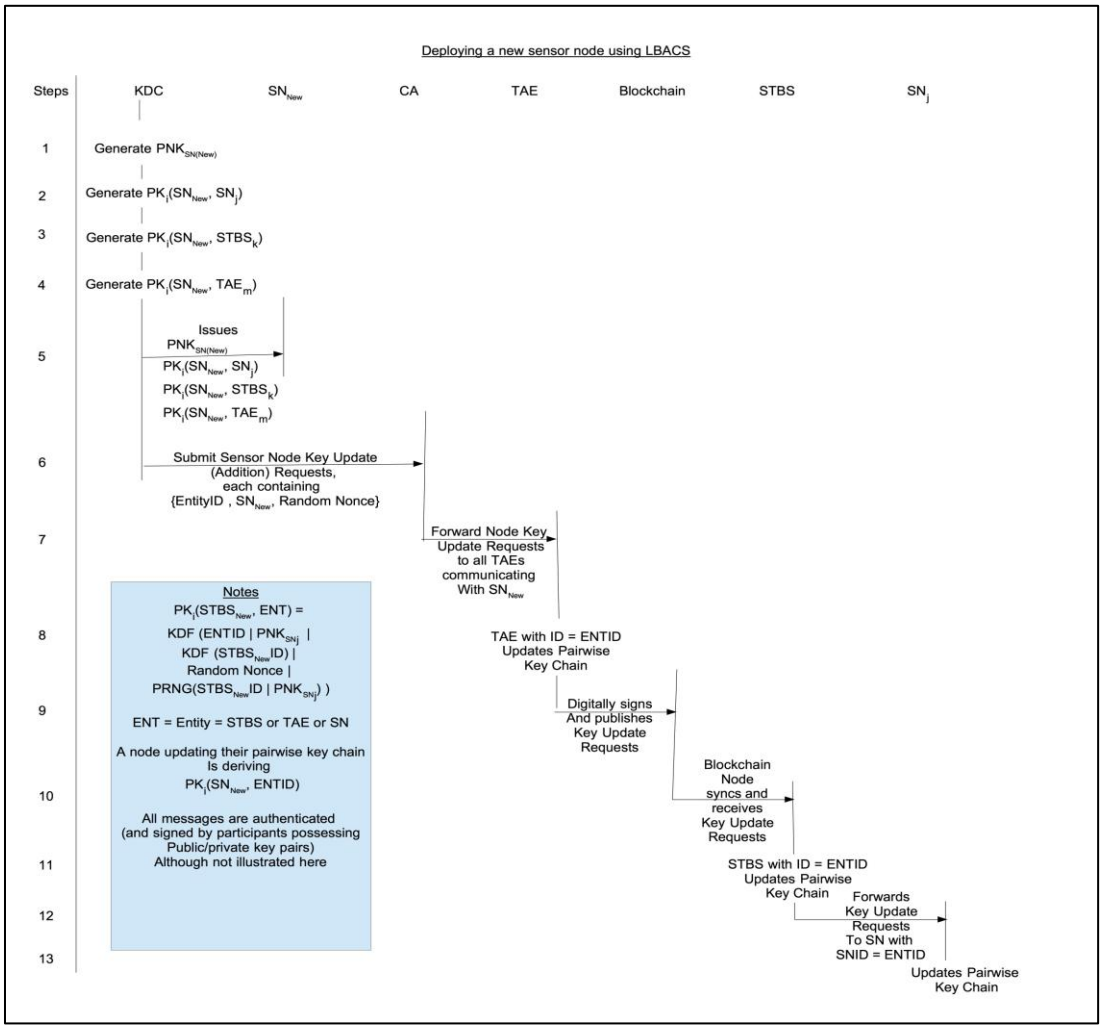

Figure 10: LBACS new Sensor Node Deployment

To add a new sensor node to the network, the KDC would first inject the new sensors keychain before deploying the node into its sub-network. This deployment would be accompanied with an authenticated broadcast to other participants of the new node id and random nonce. This would allow other participants to derive keys using the KDF to establish communication with this new node. The required key generation and necessary broadcasts have been illustrated in Figure 10 above. It should be noted that even though any deployed node or an attacker may receive this broadcast message of a new node, the

node or attacker will not be able to authenticate with the new node if the new node was not assigned the pairwise key generated by the already deployed node.

3.4.3 Session Management

A session in LBACS between a sensor node, A, and another peer (sensor node, TAE or STBS) is characterized by the use of a shared pairwise key and random seed derived by the *i*th iteration of a shared KDF and shared PRNG where $i \in N$. The length of a session, **SESSION_LENGTH**, is defined by the finite amount of requests (each characterized by a request ID) that may be made by a single party, **NO_OF_REQUESTS_IN_SESSION**, *plus* one (1) additional request to that may be used to signify a session change, **SESSION_CHANGE_REQUEST**.

SESSION_LENGTH = NO_OF_REQUESTS_IN_SESSION + SESSION_CHANGE_REQUEST

With each request in a session, sent by a party A, characterized by a request ID, a **SESSION_CHANGE_REQUEST** may be characterized by the last available request ID within the session.

SESSION_CHANGE_REQUEST_ID= SESSION_LENGTH – 1

OR

SESSION_CHANGE_REQUEST_ID = NO_OF_REQUESTS_IN_SESSION + 1

After deployment, nodes utilize the first session, $i = 1$. Nodes may continue to communicate within this session until they have exhausted

NO_OF_REQUESTS_IN_SESSION or prematurely initiate a new session by transmitting a message utilizing the **SESSION_CHANGE_REQUEST_ID**. Each participant of the session, receiving a message containing the **SESSION_CHANGE_REQUEST_ID** will then be responsible for performing the *i+1* update of their pairwise key and random seed to continue communication. The *i*+1 session generation has been illustrated below for two peers, A and B.

$$\mathbf{PK_{i+1}(A, B) = KDF\ (PRNG\ (RS_i\ (A, B))\ |\ PK_i(A, B)\ )}$$

$$\mathbf{RS_{i+1}(A, B) = PRNG\ (RS_i(A, B)\ )}$$

It should also be noted that an identifier identifying the current session iteration or past pairwise keys and random seeds should not be stored by participants. Not only does this improve the key storage requirements of the scheme but reduces the associated risks to the scheme's security objectives such as backward and forward secrecy, replay protection and others as discussed in the "Threat Analysis" section of this proposal.

3.4.4 Revocation

A certificate authority may revoke an existing TAE at any time by updating the revocation list of TAEs on the blockchain. Since all consumer applications and STBS check this list while verifying the integrity of a signature, all parties will be able to identify the signature of the revoked TAE. The process is similar for a STBS; however, additional information will have to be propagated to WSN nodes utilizing this STBS. Assuming that a second STBS ($STBS_2$) has access to a WSN where the first STBS

($STBS_1$) is being revoked; $STBS_2$ would communicate an authenticated revocation packet, characterized by the use of an intention bit (the intention will prompt the constrained device to verify the TAE as will be discussed later) and the associated TAE's MAC, to all sensor nodes in the WSN. Since $STBS_1$ does not share the same pairwise key as the TAE and sensor nodes within the WSN, it is unable to fabricate this message even if it has access to the published information on the blockchain.

Since the focus is the provenance of sensor data, sensor revocation not only has to be at the sub-network level in the WSNs but also at the base station level and global level (blockchain) where other STBS and TAEs may have the ability to authenticate transmissions provided by an invalid sensor node. To remove a sensor node, the certificate authority may communicate the pseudo-id (for privacy) of the revoked sensor nodes on the blockchain in a revocation list or to other TAEs via another trusted medium. TAEs would then be responsible for creating the revocation message with their HMAC to be published on the blockchain where their STBS will have access. A STBS possessing a node identified by this pseudo-id may then forward the authenticated revocation packet to the WSN sub-network. In these cases, the use of an additional group key has proven useful as discussed earlier in the key management and distribution section of the literature review. In particular, polynomial regression techniques has not only proved to provide confidentiality (Ozdemir, Peng, & Xiao, 2013) but have also reduced communication overhead (Chen & Xie, 2014) in constrained Self-Healing Group Key Distribution (SGKD) by only transmitting coefficients of a polynomial that may be evaluated by WSN nodes.

This revocation strategy is suitable for nodes to update their personal key chains by revoking the identified nodes, but may not be suitable when the WSN communicates with a group pairwise key. To address this issue of group key revocation, the STBS would forward a key update message to $N=v+r$ (where v are valid nodes) individual nodes within the network. N messages are required as the STBS is transmitting a message M that when applied to a PRNG and KDF utilizing the individually assigned key to the specific node, will generate the next group key. The set of revoked nodes (r) are also included but their M will result in the generation of an incorrect group key. Assuming the compromised nodes are still acting within the scheme key possession constraints i.e. only storing one key at a time, the revoked nodes will no longer be able to communicate and will lose the last valid group key.

By including revocation, the network owner is further able to optimize sensor node storage at their discretion as a revoked STBS or peer node, may mean freeing the storage used to keep the keys for this participant.

3.5 Authentication

3.5.1 Tag Format

LBACS utilizes a MAC or multi-signature packet comprised of an intention bit, request id and ordered Hash Message Authentication Codes (HMACs) illustrated in Figure 11 below. Since LBACS operates above the transport layer, the generated MAC may then be included in the request, such as the Token field in the COAP header or preceding the MQTT message payload.

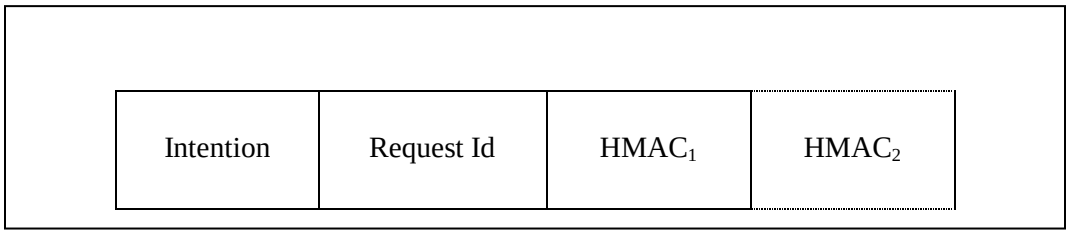

Figure 11: LBACS Authentication Packet

The intention bit is a flag parameter that describes the intent of the message that is being authenticated and may be as small as 2 bits. Depending on the network, the following intention bits may be useful as described in Table 4 below. Based on the use of intention bits described in Table 4 below it is clear, that the packet size is optimally reduced (by removing $HMAC_2$) based on the communication intent in the network. This is a desirable trait as WSNs aim to reduce the resource (computational, power, communication) overhead in each network activity. By increasing the bit length, the intents and expectations of participating nodes may be adjusted to accommodate Application Specific or SDWSNs.

Table 4: LBACS Intention Bits

Bit mask	Intention	Description
00	Peer Authentication	Used by a WSN node when authenticating with the base station or other immediate peer in multi-hop or routed network. $HMAC_1$ will be used to authenticate this message.
01	Group Authentication	Used by a WSN node when authenticating with the group or during multi-cast. $HMAC_1$ will be used to authenticate this message.
10	Key Revocation	A forwarded message will utilize $HMAC_1$ to verify the node or base station which forwarded the packet and use $HMAC_2$ to verify the TAE since a locally revoked sensor node must also be revoked globally.
11	Key Recovery	

A Request Id (ReqId) is used (size is implementation or application specific) to assist in replay attack detection in each communication session. A constrained node receiving an authentication packet with a ReqId that has already been used will

immediately be able to halt processing of the authentication packet or message, thus saving resources similar to SMACK (Gehrmann et al, 2015). Since communication with an oracle is presented as a single global view that is auditable; when a STBS communicates with a WSN node (even for sub-network management), the STBS will be responsible for publishing this request (at most the ReqId on the blockchain). If a TAE attempting to communicate with the same node is unable to authenticate with a sensor node because it has utilized a ReqId that is no longer valid, it implies that the STBS may not be following the LBACS scheme. This is a precaution to highlight possible issues, since the Base Station is not entirely trusted and may be compromised or not working effectively (software or hardware issues).

This implies that if a STBS is compromised and decides to communicate excessively with nodes but not publish these updates to the chain, a subsequent failed request from a TAE (because a TAE has utilized a ReqID the oracle has already received and thus marks it as invalid) to a sensor node within that sub-network will highlight that there is a rogue STBS. This also works in cases where the STBS continues to publish ReqId updates to the chain, as network auditors would have the ability to audit suspicious activity on the blockchain network. This optimized use of the single ReqId and the already involved participation of STBS blockchain nodes, includes some of the desired properties of the Implicit Security Authentication Scheme (Chen et al. ,2010) such as audited communication and network behavior analysis without the additional overhead of pre-behavioural analysis, additional watchdog node deployments and model development of participating network nodes.

Each HMAC enables and enforces forward and backward secrecy, replay protection, source and destination repudiation, message integrity and authentication. An HMAC is generated using HMAC (Pairwise Key | Integrity Parameters) and further optimized in size with the use of *r* concatenated set of *h* HMAC pieces chosen at "random" by the use of a shared PRNG(seed) that was initially shared between peers. Integrity parameters include the packet source, destination, request id, intention bit and message and therefore allow the generated HMAC to act as a signature for the request. Moreover, the use of the "randomly" chosen pieces of the generated HMAC allows network participants to utilize longer keys to maintain HMAC integrity while reducing the communication overhead. A constrained oracle would then improve the used semantic security by using longer keys, eg. 160 or 256 bits in Keccak, and reduce the possibility of a hacker guessing the keys used from cipher text attacks (stream or block). The attack problem is further compounded since the nodes use a random set of pieces of the generated HMAC to reduce their communication overhead. Furthermore, pairwise keys are updated each session length (related to the max id possible in request ids) using a KDF (PRNG(seed), current Pairwise Key). By updating the pairwise key, an attacker compromising the node will be unable to use the current key to effectively assist in key attempts for previous messages, thus promoting backward secrecy. Although the processes outlined may enable these properties, it does not guarantee these properties as the desired use of a HMAC, KDF and PRNG based on specific network constraints and applications may be prone to some side-channel attacks. To reduce this possibility and localization issues in WSNs, the scheme has refrained from the use of time dependent parameters.

3.5.2 Certificate Registry Storage Considerations

Based on the blockchain implementation and network use, storage and search issues in large registries for valid or invalidated entities during the verification process may become a concern. This concern is then propagated to lighter blockchain clients (clients that selectively participate in the mining process or only store a subset of the entire block chain). To reduce the storage needs, any request by a verified entity may be accompanied by a signature from the certificate authority (also referred to as the verification authority). This will enable a more involved certificate authority and is similar to the Online Certificate Status Protocol RFC2560 (Myers et. al, 1999). With the use of the blockchain however, the expected communication overhead changes from:

1. Sender sends signature request to verification authority
2. Verification authority checks verification list
3. Verification authority signs message if sender is not revoked
4. Verification authority responds to sender with signed message
5. Sender appends signature of verification authority to message to identify it as verified
6. Sender sends verified message to recipient
7. Recipient checks for verification authority's signature and therefore accepts message as not from a sender in the revocation list

to a more optimized approach:

1. Sender publishes message on blockchain with verification request
2. Certificate authority (having access to the blockchain) receives verification request

3. Verification authority checks whether sender is revoked
4. If sender is valid, verification authority publishes signature for verification request.
5. Recipient (having access to the blockchain) acknowledges the verification authority's signature of the sender's request and therefore accepts message as not from a sender in the revocation list

In the second scenario, the blockchain is utilized as the communication medium and as such reduces the number of needed requests to attest to whether the sender is revoked. This provides several benefits which include:

1. Less independent requests for verification
2. Lighter clients storing a subset or only the most recent updates from the blockchain may operate with the same level of verification from the verification authority
3. The revocation list is still verifiable by any blockchain node capable of storing and searching it and as a result averts sybil and other related attacks
4. Multiple verification authorities may be utilized to distribute the search and verification cost of the entire revocation list since they have access to the same data (requests for verification, verification authority responses)
5. Light blockchain clients only need to maintain an active view of valid certificate authorities

3.5.3 Communication Flows

The communication flows presented in this section will highlight common message exchanges, composition of the LBACS authentication tag and the verification of the created tag by the intended recipient. These message exchanges will include oracle sensors and their peer sensor nodes, Semi-Trusted Base Stations (STBS) and Trusted Authentication Entities (TAEs). Moreover, for each scenario, it will be assumed that a communication session has already been initiated. As described in the "Session Management" section of this proposal, a session may be initiated by sending a message with the **SESSION_CHANGE_REQUEST_ID,** prompting the peer to update their pairwise key and random seed to the next valid set. Furthermore, each flow will be described as a scenario as an example of a context in which the authenticated exchanges would take place.

Oracle Sensor Node → Semi-Trusted Base Station → Trusted Authentication Entity

The following scenario describes a sensor node that intends to publish information globally or on the blockchain. The LBACS MAC generated by the sensor node will be transmitted over a WSN to the STBS, which will verify the request before publishing this information on the blockchain. The sensor's message, forwarded by the STBS to the blockchain, will then be verified by the TAE. Consumer applications will then be able to attest that the data published originated from the oracle sensor. It should be noted that this scenario is applicable whether the oracle sensor node uses a pairwise

key or a group key shared with the STBS. This flow has also been illustrated in Figure 12 on page 76.

1. The sensor node creates a message to be sent, for e.g. monitored temperature data.
2. The sensor node prepends the intention bit (00) for peer authentication, the request Id (currently shared with this TAE and STBS) of REQUEST_ID_SIZE_BITS to the LBACS MAC.
3. The sensor node then appends the $HMAC_1$ to be verified by the STBS.
    a. $PRE\text{-}HMAC_1$ = HMAC (Pairwise (Group) Key for STBS | (Packet source = Sensor Node Id | Packet destination = STBS Id | Request id | Intention bit = 00 | message = monitored temperature data))
    b. Divide $PRE\text{-}HMAC_1$ in HMAC_PIECES_COUNT of size HMAC_PIECE_SIZE
    c. RANDOM_INDEX = PRNG (seed shared with STBS | *i*th required index) mod HMAC_PIECE_SIZE
    d. $HMAC_1$ reduced in size to optimize transmission = Concatenation of every RANDOM_INDEX of $PRE\text{-}HMAC_1$ pieces.
    e. If the session has ended, update pairwise keys/group key using KDF and randomization seed using, new seed = PRNG (current seed).
4. The sensor node then appends the $HMAC_2$ to be verified by the TAE. The $HMAC_2$ is required in this case since the sensor intends to prove the origin of the message on the blockchain. Furthermore, the addition of an additional HMAC reduces the possibility of a man-in-the-middle (MITM) attack by the STBS.

   a. PRE-$HMAC_2$ = HMAC (Pairwise Key for TAE | (Packet source = Sensor Node Id | Packet destination = TAE Pseudo Id | Request id | Intention bit = 00 | message = monitored temperature data))
   b. Divide PRE-$HMAC_2$ in HMAC_PIECES_COUNT of size HMAC_PIECE_SIZE
   c. RANDOM_INDEX = PRNG (seed shared with STBS | *i*th required index) mod HMAC_PIECE_SIZE
   d. $HMAC_2$ reduced in size to optimize transmission = Concatenation of every RANDOM_INDEX of PRE-$HMAC_2$ pieces.
   e. If session has ended update pairwise keys using KDF and randomization seed using, new seed = PRNG (current seed).
5. The sensor node transmits the message and MAC to the STBS
6. The STBS verifies the $HMAC_1$ by repeating Step 3. If it is invalid, the STBS discards the message as not authenticated.
7. The STBS then removes the $HMAC_1$
8. The STBS generates a signed message with its private key that may be verified with its public key.
   a. Signed message = SIGN (MAC (excluding $HMAC_1$) | Pseudo ID for WSN oracle node | message)
9. The STBS then publishes the signed message, the updated MAC (excluding $HMAC_1$), and the pseudo-id for the source WSN oracle sensor on the blockchain.
   a. By removing $HMAC_1$ the STBS reduces the possibility that future communication between it and the sensor node may be compromised by a

malicious member who has access to the blockchain network and who is willing/able to perform the necessary cryptanalysis. Furthermore, this reduces the storage requirements for the blockchain network by not publishing information that will not be useful.

10. A TAE that has access to the network through a blockchain node, checks whether any transactions within blocks contains requests from a STBS.
11. The TAE acknowledges that a block contains requests from the STBS and verifies that its public key or ID is not in the revocation list available on the blockchain.
12. If the public key of the STBS is valid (not in the revocation list), the signature of the STBS is verified.
13. If the STBS' signature is verified, the TAE verifies $HMAC_2$ by attempting to derive $HMAC_2$ utilizing the following steps:
    a. Verify that the Request Id has not already been used. If the request Id has already been used, the message is considered as invalid.
    b. Create PRE-$HMAC_2$ = HMAC (Pairwise Key for TAE | (Packet source = Sensor Node Id | Packet destination = TAE Pseudo Id | Request id | Intention bit = 00 | message = monitored temperature data))
    c. Divide PRE-$HMAC_2$ in HMAC_PIECES_COUNT of size HMAC_PIECE_SIZE
    d. RANDOM_INDEX = PRNG (seed shared with STBS | *i*th required index) mod HMAC_PIECE_SIZE
    e. $HMAC_2$ reduced in size = Concatenation of every RANDOM_INDEX of PRE-$HMAC_2$ pieces.

f. If the generated $HMAC_2$ matches the $HMAC_2$ submitted by the STBS, the message is authenticated.

g. If the session has ended, update pairwise keys using KDF and randomization seed using, new seed = PRNG (current seed).

14. If both signatures (STBS submitted signature verifiable with its public key and $HMAC_2$) are verified, the TAE then submits a signature for the message (confirmation signature) published by the STBS on the blockchain.
15. Consumer applications syncing transactions would receive the confirmation signature for the TAE. If they are interested in the data published by this oracle sensor (identified by the Pseudo ID) they may consume the data.

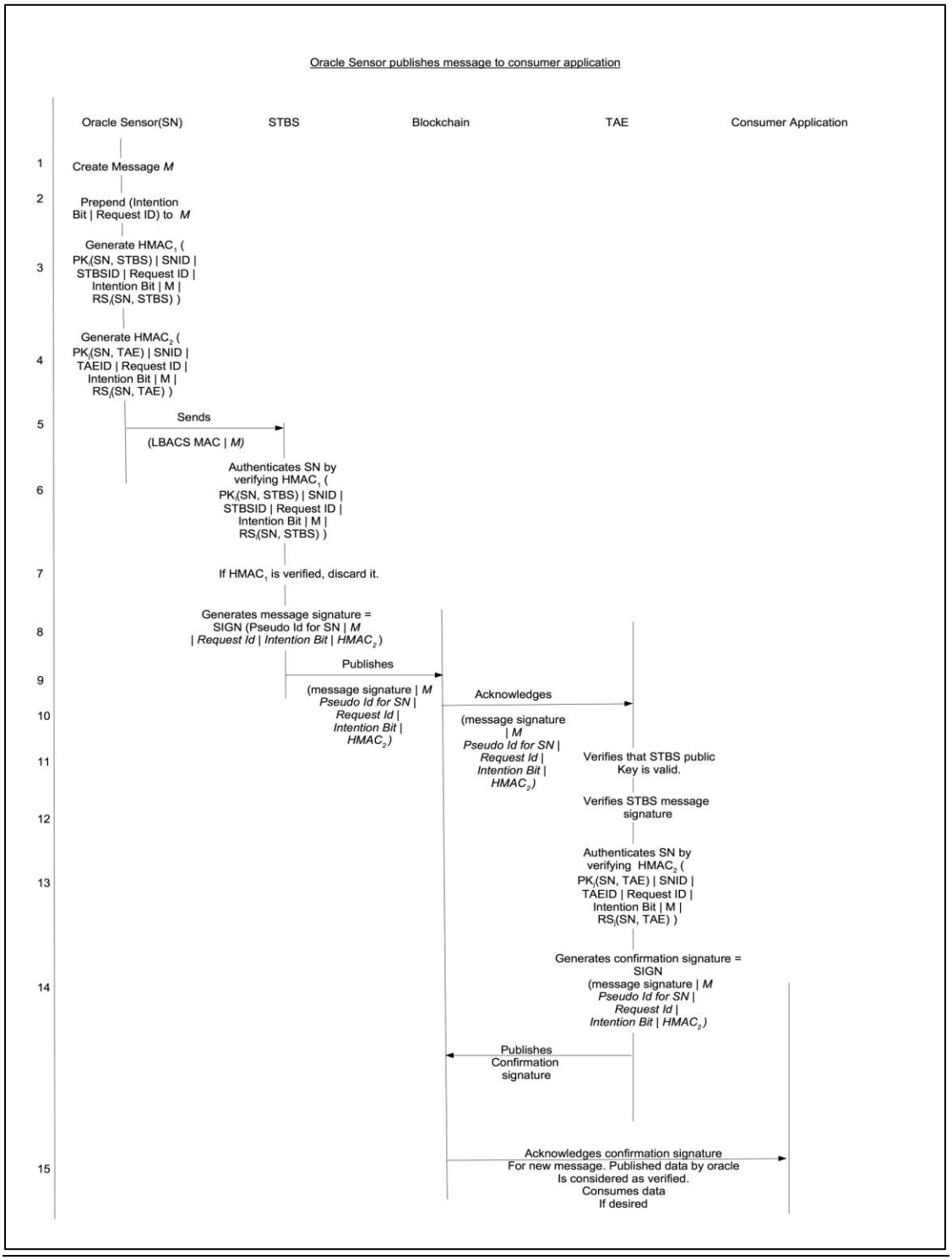

Figure 12: LBACS Oracle Sensor publishes data for consumer application

Sensor node(j) → Sensor Node (j+1) → Semi-Trusted Base Station → Trusted Authentication Entity

This scenario is similar to the aforementioned scenario with the exception that the sensor node is unable to establish direct communication with the STBS such as nodes in sub-network 3 in Figure 5. In this case, where a node must route its communication through a peer, the node would generate $HMAC_1$ with the pairwise key for its linked node (modifying Steps 3 and 5). The linked node, sensor node (j+1), would then be able to forward the same request by replacing the $HMAC_1$ with its pairwise key for the STBS. The communication and authentication flow for the other steps would be the same. This has been illustrated in Figure 13 on page 78.

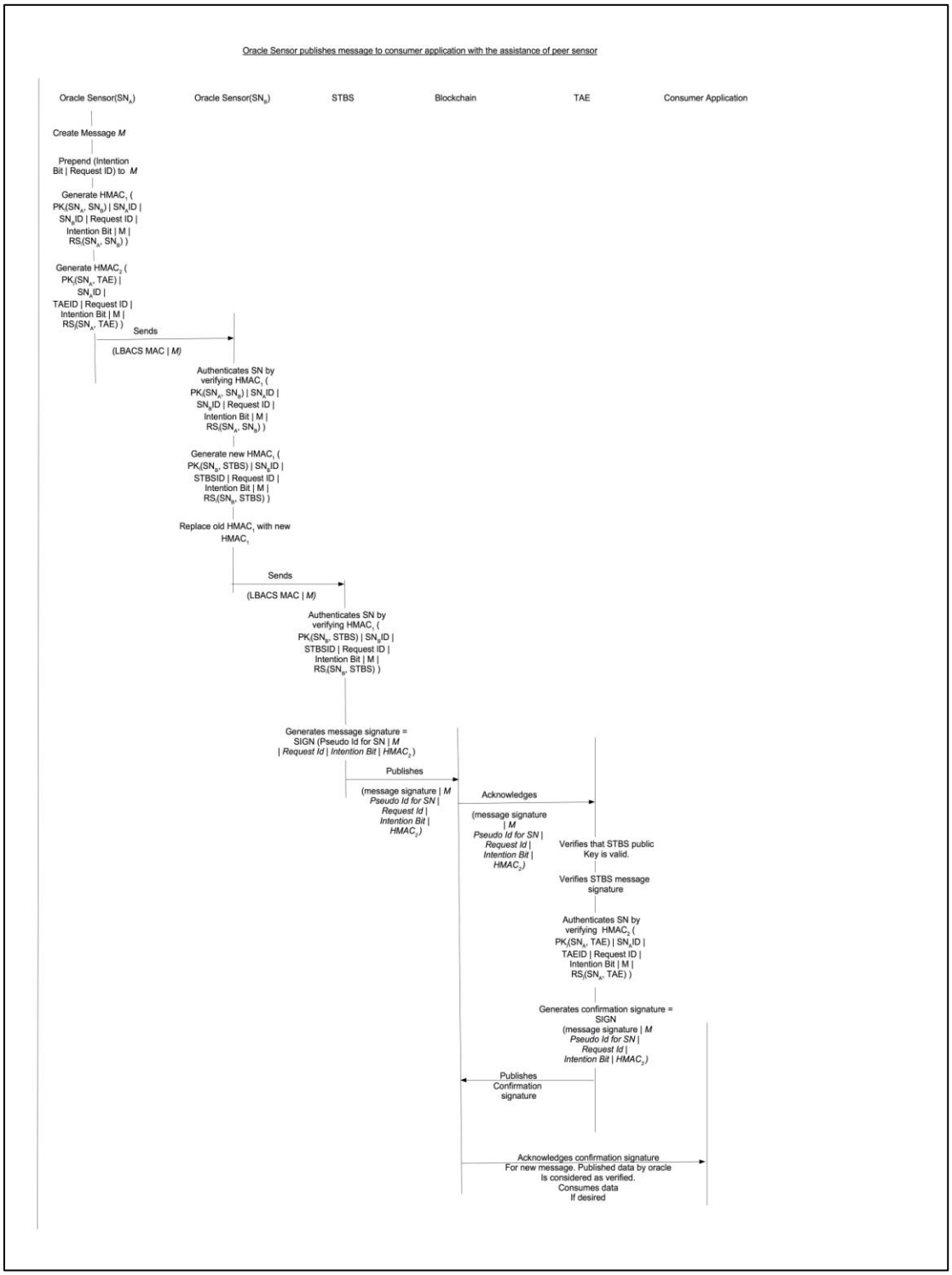

Figure 13: LBACS Oracle Sensor publishes data for consumer application via peer sensor

Trusted Authentication Entity → Semi-Trusted Base Station →Sensor Node

The following scenario describes a TAE that wishes to communicate with an oracle sensor node with the assistance of a STBS, which has access to both networks (WSN and blockchain). The flow will denote how the LBACS MAC generated by the TAE is verified by the oracle sensor and ultimately how a message sent by the TAE is authenticated by a STBS before being forwarded to a sensor.

1. The TAE creates a message. For example, the TAE has received a request from the certification authority to revoke Nodes 1 and 2 from Sub-network 3.
2. The TAE prepends the intention bit (10) for key revocation, the request Id (currently shared with this oracle sensor and STBS) of REQUEST_ID_SIZE_BITS to the LBACS MAC.
3. The TAE generates $HMAC_2$ to be verified by the oracle sensor
    a. PRE-$HMAC_2$ = HMAC (Pairwise Key for oracle sensor | (Packet source = TAE Pseudo Id | Packet destination = Oracle sensor ID | Request id | Intention bit = 10 | message = key revocation message))
    b. Divide PRE-$HMAC_2$ in HMAC_PIECES_COUNT of size HMAC_PIECE_SIZE
    c. RANDOM_INDEX = PRNG (seed shared with STBS | *i*th required index) mod HMAC_PIECE_SIZE

   d. $HMAC_2$ reduced in size to optimize transmission = Concatenation of every RANDOM_INDEX of PRE-$HMAC_2$ pieces.
   e. If the session has ended update pairwise keys using KDF and randomization seed using, new seed = PRNG (current seed).

4. The TAE generates a signed message with its private key that may be verified with its public key
   a. Signed message = SIGN (MAC | message | Pseudo Id for Sub-network with oracle sensor)
5. A STBS that has access to the network, receives and confirms a new blockchain block with additional transactions.
6. The STBS acknowledges that it contains requests from the TAE and verifies that its public key or ID is not in the revocation list available on the blockchain
7. If the STBS has access to the sub-network identified by the Psudeo Id, the STBS injects the $HMAC_1$
   a. PRE-$HMAC_1$ = HMAC (Pairwise Key for oracle sensor | (Packet source = TAE Pseudo Id | Packet destination = Oracle Sensor | Request id | Intention bit = 10 | message = revocation message))
   b. Divide PRE-$HMAC_1$ in HMAC_PIECES_COUNT of size HMAC_PIECE_SIZE
   c. RANDOM_INDEX = PRNG (seed shared with STBS | *i*th required index) mod HMAC_PIECE_SIZE
   d. $HMAC_1$ reduced in size to optimize transmission = Concatenation of every RANDOM_INDEX of PRE-$HMAC_1$ pieces.

e. If session has ended update pairwise keys using KDF and randomization seed using, new seed = PRNG (current seed).

8. The STBS transmits the message with the updated MAC to the oracle sensor which will be able to verify the $HMAC_1$ and $HMAC_2$ by repeating the procedures outlined in Step 7 and 3 respectively.

Semi-Trusted Base Station →Sensor Node

This would be similar to the aforementioned scenario with the exception that the source of the message would be the STBS. This scenario describes a STBS communicating with an actuator which is a thermostat, the temperature should be changed to 21 ℃.

1. The STBS creates the message to be sent.
2. The STBS prepends the intention bit (00) for peer authentication, the request Id (currently shared with the oracle sensor) of REQUEST_ID_SIZE_BITS to the LBACS MAC.
3. The STBS then appends $HMAC_1$ to be verified by the oracle sensor.
    a. $PRE\text{-}HMAC_1$ = HMAC (Pairwise Key for STBS | (Packet source = STBS Id | Packet destination = sensor Id | Request id | Intention bit = 00 | message = temperature update request))
    b. Divide $PRE\text{-}HMAC_1$ in HMAC_PIECES_COUNT of size HMAC_PIECE_SIZE

c. RANDOM_INDEX = PRNG (seed shared with STBS | *i*th required index) mod HMAC_PIECE_SIZE

d. $HMAC_1$ reduced in size to optimize transmission = Concatenation of every RANDOM_INDEX of PRE-$HMAC_1$ pieces.

e. If session has ended update pairwise keys using KDF and randomization seed using, new seed = PRNG (current seed).

4. The STBS transmits the message and MAC to the oracle sensor which will be able to verify $HMAC_1$ by repeating the steps in Step 3
5. Because the STBS is audited, it signs and publishes the message + request + sensor Pseudo Id on the blockchain
6. A TAE, receiving this updated block, which corresponds with that sensor node and STBS may then update their request Id for future communication

Chapter 4: Implementation

To further highlight the feasibility of the Lightweight Blockchain Authentication Scheme for Constrained oracle Sensors (LBACS), it was implemented with tests on several devices with varying resource constraints. These devices, including a Zolertia Z1 mote, Raspberry PI 3 and laptop, assured the provenance of sensor data and authentication of peers in the LBACS network. This implementation of LBACS utilized the Ethereum blockchain network in addition to the Keccak Sponge Family and the secp256k1 elliptic curve to achieve the security objectives and benefits of the LBACS scheme outlined in Chapter 3. This section will therefore highlight the aims of the implementation, specifications of the apparatus used, particular configuration values used, procedure to replicate the implementation along with issues faced and the solutions or decisions made. Observations and analysis of results garnered are highlighted in Chapter 4.

4.1 Objective

The overall objective of this first iteration was to implement a buoy monitoring system using LBACS to authenticate sensor communication and achieve provenance of sensor data. Sensors would be responsible for communicating the accelerometer readings from the simulated buoys and TAEs would verify and authenticate the sensor data forwarded by the STBSs. Furthermore, with the assistance of the already integrated blockchain, consumer applications would be able to attest to the provenance of sensor data.

4.2 Aims

The implementation had the following aims:

1. Implement LBACS for MSP430 Processors
2. Implement LBACS keychain store for Contiki devices
3. Implement LBACS for ARMv7 and ARMv8 Processors
4. Implement LBACS for x86-64 Processors
5. Implement Node.js add-on for LBACS
6. Implement a Trusted Authentication Entity (TAE) for x86-64
7. Implement a Semi Trusted Base Station (STBS) for Raspberry PI 3
8. Implement a buoy accelerometer sensor on Contiki Z1 motes

4.3 Configuration

LBACS was designed to allow the provenance and authentication of constrained sensor data. As described in the literature review, these networks consider a varied combination of devices, technologies and strategies to achieve their aims. This section will outline the configurable parameters included in LBACS and the rationale for the inclusion of each.

The configurable parameters specific to this implementation have been summarized in Table 5 below. It should be noted that capitalized parameters included in Table 5 are referenced from the notations listed in Table 3 on page 48.

Table 5: LBACS Configuration for Buoy Monitoring Implementation

Parameter	Configuration
INTENTION_BIT_SIZE	1 byte
SESSION_SIZE	12
LBACS_TOKEN_SIZE	8 bytes
HMAC_SIZE	3 bytes
HMAC_PIECES_COUNT	3
HMAC_PIECES_SIZE	32
PRNG	Keccak-f[1600, c=256, r=1344] SHA3
KDF	Keccak-f[1600, c=256, r=1344] SHA3
Private/Public Key Generation	secp256k1 elliptic curve
Signature generation and verification	ECDSA using secp256k1 elliptic curve
Blockchain Implementation	Private Ethereum Blockchain Network

4.3.1 Constrained Application Protocol

The Constrained Application Protocol (COAP) is a specialized machine-to-machine (M2M) communication protocol for constrained devices operating within Low Power and Lossy Networks (LLNs). The protocol, quite similar to the Hypertext Transfer Protocol (HTTP), provides a request-response interaction, Uniform Resource Identifiers (URIs), media types, status codes in addition to multicast support and a low overhead which is required in these constrained networks (Shelby et. al, 2014).

COAP was utilized to provide a uniform communication model between the sensors, actuators and the base station within the WSN. This ensured greater interoperability and a standard way to parse for the message payload and generated LBACS authentication tag. As illustrated in the COAP message format in Figure 14 below, the 4 bit unsigned Token Length (TLK) specifies a maximum available token size of 8 bytes.

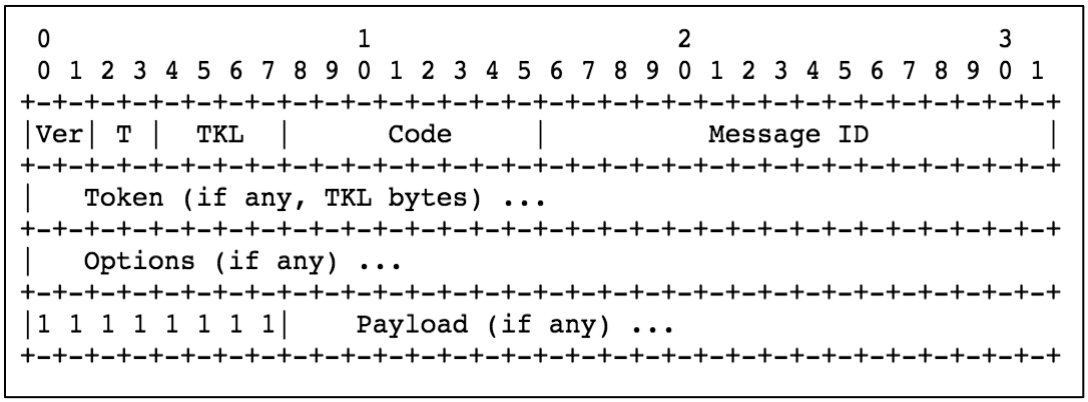

Figure 14: COAP Message Format

Furthermore, the size allocation for the authentication tag included in the token field has been illustrated below in Figure 15.

Intention	Request Id	$HMAC_1$	$HMAC_2$
1 byte	1 byte	3 bytes	3 bytes

Figure 15: LBACS authentication tag size allocations

4.3.2 *Keccak-f*[1600, c=256, r=1344]

To reduce the required storage and memory needs of the constrained Z1 motes, the same KDF and PRNG was used for all peer nodes. Each peer, still maintained their independent set of shared random seeds and pairwise keys. The KDF, PRNG and HMAC used were all derived from the Keccak family of functions using the *Keccak-f*[1600, c=256, r=1344] permutation. The implementation used has been included in Appendix I.

The Federal Information Processing Standards (FIPS) Publication 202 of the National Institute of Standards and Technology (NIST) published the adopted Keccak family of cryptographic permutations in 2015. The Keccak family of functions are a set of cryptographic permutations based on the sponge construction. Each of the seven (7) *Keccak-f* permutations denoted as *Keccak-f*[*b*] where $b \in \{25, 50,100,200, 400, 800, 1600\}$ are defined as a progression of operations on a three (3) dimensional state *a* where *a* is a Galois Field of elements (GF(2)) (Bertoni, Daemen, Peeters, & Van Assche, 2011). The relationship between the Keccak-f permutation and its sponge construction has been illustrated below in Figure 16 below.

$$\text{KECCAK}[r, c] \triangleq \text{SPONGE}[\text{KECCAK-}f[r + c], \text{pad10*1}, r]$$

where

r = bitrate

c = capacity and c= b – r where $b \in \{25, 50,100,200, 400, 800, 1600\}$

Figure 16: Keccak relation to sponge construction

As a cryptographic primitive, *Keccak-f* derives many of the capabilities of the sponge and duplex constructions which were instrumental to this implementation. It should be noted that while both sponge and duplex constructions accept a variable-input length and yield an arbitrary output length based on a bitrate, *r*, padding rule and fixed-length permutation, a duplex construction maintains its previous state through each use. Moreover, unlike previous primitives, the claims for security strength are not based on the output length making it ideal for this constrained implementation. Furthermore, the provably secure primitive has undergone much cryptanalysis and performance tests on hardware varying in resource constraints.

4.3.3 Secp256k1 Elliptic Curve Digital Signature Algorithm

LBACS requires that resource competent entities such as the Semi-Trusted Base Stations (STBS), Trusted Authentication Entities (TAEs) and the Certificate Authority (CA) utilize asymmetric cryptography to enforce digital integrity of transmitted messages. This implementation utilized an existing library implementing the Elliptic Curve Digital Signature Algorithm (ECDSA) using the recommended Secp256k1 curve parameters (Certicom Research, 2010).

The use of the elliptic curve variant of the digital signature algorithm has garnered much attention due to the advantages of Elliptic Curve Cryptography (ECC) when compared with the widely used Rivest-Shamir-Adleman (RSA) algorithm. Notably, ECC offers the same level of security as RSA using smaller key sizes, less computational effort and power consumption. ECC is based on the relative intractability of solving the discrete logarithm problem for a random elliptic curve element with a publicly known

base point (Khalique, Singh, & Sood, 2010). The Standards for Efficient Cryptography (SECP) recommended Koblitz curve parameters for secp256k1 defined by the sextuplet T = (p, a, b, G, n, h) are included below (Certicom Research, 2010).

Here the finite field, F_p, is defined as:

$$\begin{aligned} p &= \texttt{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F} \\ &= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1 \end{aligned}$$

Furthermore, the curve E: $y^2 = x^3 + ax + b$ is defined with constants

$$\begin{aligned} a &= \texttt{00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000} \\ b &= \texttt{00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007} \end{aligned}$$

an uncompressed base point G

$$G = \texttt{04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8}$$

an order *n* and cofactor *h*

$$\begin{aligned} n &= \texttt{FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141} \\ h &= \texttt{01} \end{aligned}$$

4.3.4 Ethereum and Solidity

This implementation utilized a private Ethereum blockchain network to record sensor data. As mentioned earlier in the literature review, Ethereum is a blockchain implementation similar to bitcoin but uses a quasi-turing-complete model on its virtual machine. Each execution still requires gas and is verified by a compute-intensive Proof-of-Work (PoW) consensus algorithm (Wood, 2016). Solidity is one high level language

that allows developers to interact with the Ethereum blockchain network using smart contracts and Externally Owned Accounts (EOA). The implementation used has been included in Appendix B. Other languages available to developers include Serpent and LLL.

4.4 Apparatus

4.4.1 Devices

Four (4) types of devices were included in the current implementation, notably the Zolertia Z1 mote, Raspberry PI 3 a windows PC and Mac OS X laptop. Device specifications have been listed below.

Zolertia Z1 Mote

- 2nd generation MSP430TM 16-bit MCU
- 92KB Flash
- 8KB RAM
- 2.4 GHz IEEE 802.15.4 Transceiver
- 3-axis Digital Accelerometer
- Low Power Digital Temperature sensor
- 16 Mbit, 100 Cycles Serial Flash
- USB/2xAA/coin cell power options
- 52-pin expansion connector
- Contiki Operating System

Raspberry PI 3 Model B

Processor	Broadcom BCM2387 chipset. 1.2GHz Quad-Core ARM Cortex-A53 802.11 b/g/n Wireless LAN and Bluetooth 4.1 (Bluetooth Classic and LE)
GPU	Dual Core VideoCore IV® Multimedia Co-Processor. Provides Open GL ES 2.0, hardware-accelerated OpenVG, and 1080p30 H.264 high-pro le decode.
Memory	1GB LPDDR2
ROM	16GB
Dimensions	85 x 56 x 17mm
Power	Micro USB socket 5V1, 2.5A
Operating System	Ubuntu 15.10 (GNU/Linux 4.1.18-v7+ armv7l)

Windows PC

Operating System	Windows 8
RAM	8 GB
ROM	500GB
Processor	Intel(R) Core(TM) i3-3227U CPU @ 1.90 GHZ x64

Mac OS X

Operating System	OS X El Capitan version 10.11.6
RAM	8 GB
ROM	128GB
Processor	2.7GHz Intel Core i5 x64

4.4.2 Software and Libraries

Most of the development was completed with a text editor since the primary languages in use were C and JavaScript. Additional software was used in simulations, testing and management. The versions and source of each software and library used have been included in Table 6 and Table 7 below. How each was used will be described later in this chapter.

Table 6: Software used in implementation

Software	Version	Acquired From
Git	2.11.0	https://git-scm.com
Mocha Test Framework	3.2.0	https://github.com/mochajs/mocha
Contiki-OS	3.0	http://www.contiki-os.org
MSP430 tool chain	5.3	http://www.ti.com/tool/msp430-gcc-opensource

Software	Version	Acquired From
Cooja Simulator	3.0	http://www.contiki-os.org
Tunslip6	3.0	http://www.contiki-os.org
Truffle	2.1.0	https://github.com/ConsenSys/truffle
Geth	1.4.5	https://geth.ethereum.org
Docker	1.12.1	https://www.docker.com/
Node.js	4.4.7	https://nodejs.org/en/
Copper	1.0.0	https://addons.mozilla.org/en-US/firefox/addon/copper-270430/

Table 7: Libraries used in implementation

Libraries	Version	Acquired From
web3.js	0.17.0-beta	https://github.com/ethereum/web3.js
node-coap	0.18.0	https://github.com/mcollina/node-coap
node-ffi	2.1.0	https://github.com/node-ffi/node-ffi
node-ref	1.3.2	https://github.com/TooTallNate/ref

Libraries	Version	Acquired From
Elliptic	6.3.2	https://github.com/indutny/elliptic
SHA3IUF	Commit 772553b	https://github.com/brainhub/SHA3IUF
Keccak Code Package	Commit e39f89a	https://github.com/gvanas/KeccakCodePackage

4.5 Design

4.5.1 Network Overview

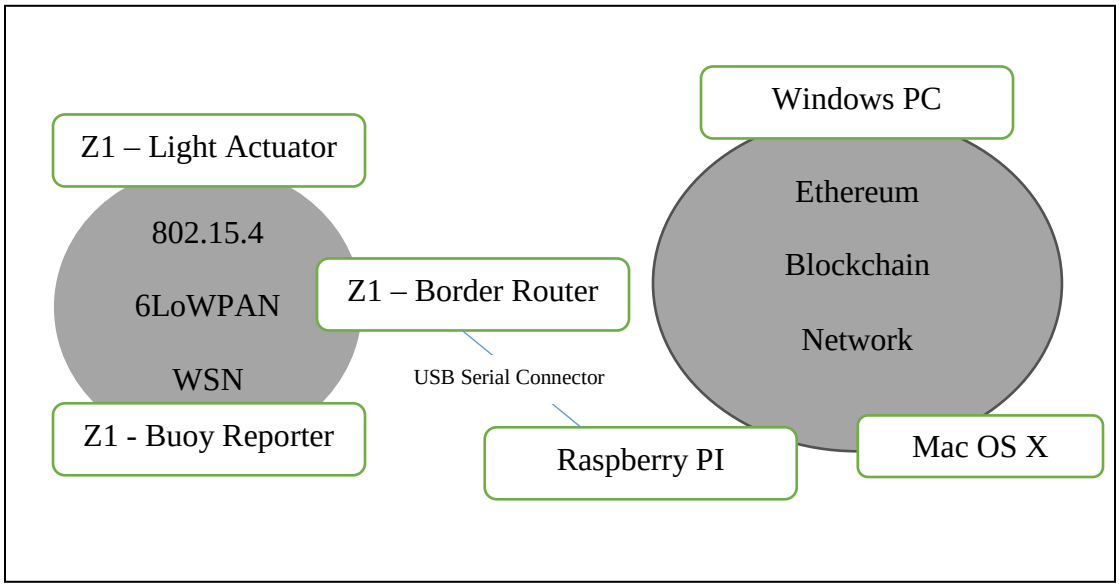

Figure 17: Implementation Network Overview Diagram

Figure 17 above illustrates how each device was connected to recreate the LBACS network. Three (3) Z1 motes were a part of the LBACS WSN sub-network. It

should be noted that one mote was used as a RPL border router for the WSN, doubling as a transceiver for the Raspberry PI (Semi-Trusted Base Station) connected to the Z1's USB-Serial Port. This allowed the STBS on the Raspberry Pi to communicate with the sensors in the WSN. Moreover, the Raspberry PI, Mac OSX and Windows PC were able to sync communication over the Ethereum blockchain network with the assistance of Geth client nodes.

4.5.2 Technology Stacks

Illustrated below are the various application stacks used on each device.

Z1 – Buoy Reporter

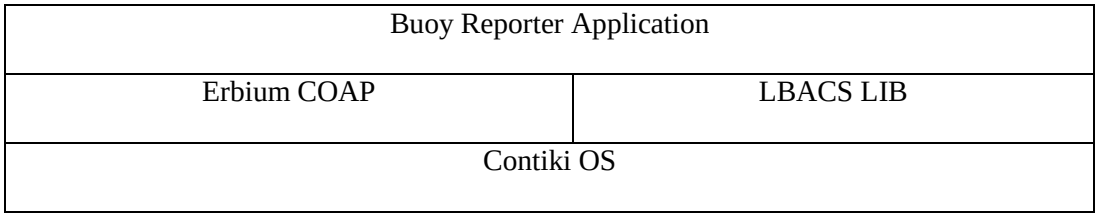

Figure 18 – LBACS Z1 Technology Stack

Raspberry PI – Semi Trusted Base Station

<table>
<tr><td></td><td colspan="4">Restful COAP + HTTP Semi-Trusted Base Station Server</td></tr>
<tr><td></td><td colspan="2">lbacsjs</td><td>elliptic</td><td>node-coap</td><td>web3</td></tr>
<tr><td rowspan="2">tunslip6</td><td>node-ffi</td><td rowspan="2" colspan="3">Node.js</td><td rowspan="2">Geth</td></tr>
<tr><td>LBACS
Shared LIB</td></tr>
<tr><td colspan="6">Ubuntu 15.10 (GNU/Linux 4.1.18-v7+ armv7l)</td></tr>
</table>

Figure 19 - LBACS Raspberry Pi Technology Stack

Windows PC – Trusted Authentication Entity

<table>
<tr><td colspan="5">Restful HTTP Trusted Authentication Entity Server</td></tr>
<tr><td colspan="2">lbacsjs</td><td>elliptic</td><td colspan="2">web3</td></tr>
<tr><td>node-ffi</td><td rowspan="2" colspan="2">Node.js</td><td rowspan="2">Geth</td></tr>
<tr><td>LBACS
Shared LIB</td></tr>
<tr><td colspan="5">Windows 8 – i3 x64</td></tr>
</table>

Figure 20 - LBACS Trusted Authentication Entity Technology Stack

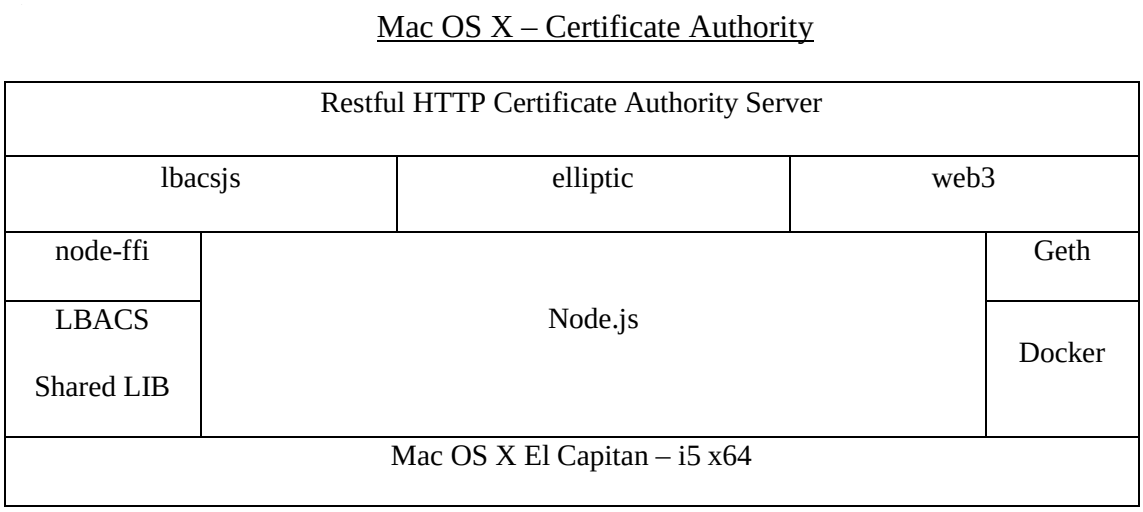

Figure 21- LBACS Certificate Authority Technology Stack

4.6 Procedure

As seen in Appendix A, similar application programming interfaces were available for the pairwise portion of LBACS, however Figure 27 and Figure 28 illustrate a few of the variations across device platforms. Variations took into consideration the various platform architectures, memory, storage and network constraints for each device platform. Furthermore, additional considerations and strategies were implemented to facilitate Ethereum blockchain interactions required by the TAE, STBS and CA on their respective devices. The work done to implement the aims identified above on each platform will be described below.

4.6.1 Buoy Node Reporter and Light Actuator

A lightweight version of LBACS capable of generating and authenticating LBACS tags was developed for Contiki-OS. The implementation was abstracted into a pluggable Contiki app module after forking the official Contiki-OS repository (https://github.com/contiki-os/contiki). Although, the official Keccak implementations are provided at https://github.com/gvanas/KeccakCodePackage, a minimal implementation of the Keccak-f[1600, c=256, r=1344] SHA3, located at https://github.com/brainhub/SHA3IUF, was chosen and modified (as included in

Appendix I) to become more platform independent. This decision was a result of the required program data used by the official implementation. When compiled for Z1 devices, the program data including the buoy functions performed by the Z1 motes exceeded the maximum program data size by approximately 94 kb. Furthermore, the LBACS implementation on Z1 motes was more static as it relied less on dynamic memory allocation. Although this decision increased program data size, it allowed the motes to have better networking capabilities and reduced program crashes due to memory constraints. Additional work was also done to create bootstrap functions used in the pre-distribution of pairwise keys and seeds for the keychain.

The buoy transmitter utilized the Contiki Erbium COAP implementation to transmit a JSON message containing the x, y, and z accelerometer readings of the transmitter every nine (9) seconds. The communication overhead for the message was less than or equal to 24 bytes. Since the generated LBACS token used was 8 bytes, the total communication overhead was less than or equal to 32 bytes. Similarly, the buoy light actuator used the Contiki Erbium COAP implementation to parse incoming messages and LBACS to verify the received token before toggling the light sensors. Both applications were tested on actual hardware and simulated with the assistance of the Cooja Simulator.

Finally, one Z1 mote was connected to the Raspberry PI via a USB-Serial Connector and tunslip6 and used as a RPL Border Router to allow the Raspberry PI to operate as a Semi-Trusted Base Station (STBS) for the IPV6 WSN network of Z1 motes.

This was necessary as the Raspberry PI did not have a IEEE 802.15.4 transceiver capable of interacting with the WSN.

4.6.2 Blockchain Network

The blockchain network was created by running geth Ethereum blockchain clients on the Raspberry PI and laptops as illustrated in the Design section of this chapter. This private blockchain network was created by initializing each Ethereum client with the same genesis block as illustrated in Figure 22 below in addition to running each client with the same network id of "2255346".

```
{
    "nonce": "0x0000000000000042",
    "timestamp": "0x0",
    "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "extraData": "0x0",
    "gasLimit": "0x8000000",
    "difficulty": "0x400",
    "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "coinbase": "0x71f78aef60041aa2f49980b9d63b80bb334b1553",
    "alloc": {
            "0x71f78aef60041aa2f49980b9d63b80bb334b1553":{
                    "balance":"99999999999999999999999999"
                },
                "0x7fe49395b20d603801df7b9f7226a081c2a50c18":{
                    "balance":"99999999999999999999999999"
                },
                "0xd309e071601307f54411bf527eb358de359c1f75":{
                    "balance":"99999999999999999999999999"
                },
                "0x478b5dc34d34729489cc0882a8c20e7b3d4872c2":{
                    "balance":"99999999999999999999999999"
                },
                "0x2cb171408f9ae9d8bebe499b807c3554e7b98a1b":{
                    "balance":"99999999999999999999999999"
                },
                "0xb71fd46dd19a6b08c9264a271c59035e920abe9a":{
                    "balance":"99999999999999999999999999"
                }
    }
}
```


Figure 22: LBACS Ethereum Genesis Block

Interactions with the Ethereum blockchain were performed with the assistance of compiled solidity contracts. These contracts illustrated in

Appendix B were used to store the current state of the LBACS blockchain network. With the assistance of various features available in Solidity additional constraints and interactions were seamlessly integrated in the autonomous servers. Solidity events were used by new entities (Semi-Trusted Base Stations, Trusted Authentication Entities and Certificate Authorities) to notify and request approval of created public keys in the Certificate Registry. Furthermore, newly published buoy data, denoted in Figure 39 on page 164, requiring authentication by Trusted Authentication Entities (TAEs) also utilized events to notify consumers and TAEs respectively. Moreover, access modifiers were used to restrict updates to the Certificate Registry, denoted in Figure 36 on page 161 , to only approved Certificate Authorities. Also included in

Appendix B are the contracts used to store and access ECC public keys and signatures and various entity types.

4.6.3 Base Station, Authentication Entity and Certificate Authority

Although different platform specific implementations, the Raspberry PI and both laptops shared similar software and ran servers using similar code bases. In order for devices to gain access to the private blockchain network, devices utilized Geth (Go Ethereum client) enabled with a JSON RPC API to provide access to the private Ethereum blockchain network. Furthermore, servers residing on these devices as illustrated in the "Technology Stacks" section were implemented in Node.js to take advantage of the well maintained web3.js package used to interact with the Ethereum blockchain clients. Although the Raspberry PI could not actively participate in the mining process due to memory constraints, the client was still able to achieve the global state of the network by syncing with the more resource competent nodes in the network.

Moreover, since the servers on both devices were implemented in Node.js, a Node.js add-on, named *lbacsjs*, was created to equip the servers with the ability to authenticate and generate LBACS tags for each message. The C-implementation of LBACS for the respective platform was packaged as a Node.js add-on with the assistance of the Node.js foreign function interface package, node-ffi. Tests conducted with the mocha testing framework verified LBACS token generation and authentication. The *lbacsjs*, *elliptic (*used for ECDSA signature generation and verification*)* and web3.js packages were then integrated to realize LBACS. Each server was then able to generate

their respective public/private key pairs and request approval in the certificate registry from the Certificate Authority.

With the assistance of the compiled *tunslip6* available with Contiki-OS and the node-coap package, the Raspberry PI hosting the Semi-Trusted Base Station was able to actively participate in the WSN network and forward LBACS authenticated packets to and fro as required.

4.7 Issues and Resolutions

Throughout the implementation, a few challenges were experienced and concerns raised. This section will list a few of these challenges in addition to resolutions made to realize the current implementation.

1. The prominent Keccak implementations from the *Keccak Code Package* for multiple platforms being utilized by LBACS as a PRNG, KDF and HMAC had to be replaced for a smaller Keccak implementation (SHA3IUF) as LBACS had exceeded the max storage for program data on Z1 devices by approximately 94 kb. This was completed successfullyand verified by the unit tests.
2. Although within program data storage requirements, the SHA3IUF implementation was not platform independent, resulting in the output generated by the Z1 devices not to match the output on other platforms. The implementation was modified to be more platform independent to realize a uniform KDF, PRNG and HMAC construction (also included in Appendix I).
3. Although it resulted in a smaller program data size, Z1 motes experienced numerous network and memory issues with dynamic memory allocation. As a

result, a static implementation and different application programming interface was adapted for Contiki LBACS. The static implementation has a predefined amount of peers, currently two (2) in this implementation.

4. Although proximity was not an issue in the WSN, many packets were still lost in transmission. As a result, the Erbium COAP implementation was modified to prevent the device from waiting on a response to each message. The time used to wait on responses lost in transmission prevented the Z1 mote from networking and carrying out application needs as expected.
5. More recent versions such as Geth versions 1.4.10 and 1.5.4 had syncing issues and did not allow the Raspberry Pi client to sync transactions to the other clients on the network. Since modifying the codebase could have several domino implications, Geth 1.4.5 was used instead.
6. The Semi-Trusted Base Station server on the Raspberry PI 3 is often interrupted when submitting new transactions to the Ethereum blockchain network with Geth 1.4.5. This issue has been attributed to the frequent account locks experienced on the Geth client which seem to occur in periods of inactivity (not submitting new transactions to the blockchain network) or approximately every five (5) minutes. This prevents the seamless forwarding of LBACS authenticated sensor messages to the blockchain network. A temporary solution was implemented to detect and unlock the default account, however, some transactions are still lost in the time used to unlock accounts.

Chapter 5: Analysis

Throughout the implementation, several metrics were recorded alongside various tests to assist in the evaluation of the proposed scheme, LBACS. These tests, including unit, incremental integration and end-to-end tests, assisted the development and verification of the buoy monitoring prototype using LBACS. This chapter will therefore describe and detail the results of these metrics and tests in addition to providing a qualitative threat assessment of the scheme.

5.1 Aims

Specifically, this analysis intends to fulfill the following objectives:

1. Perform a qualitative threat analysis for LBACS
2. Identify program data size for LBACS on Contiki Z1 motes
3. Identify keychain size for LBACS on Contiki Z1 motes
4. Identify WSN COAP communication overhead between Contiki Z1 motes and Raspberry PI 3 Semi-Trusted Base Station
5. Identify Power consumption for LBACS authentication tag generation and verification on Contiki Z1 motes
6. Identify duration for LBACS authentication tag generation and verification on Contiki Z1 motes
7. Perform end-to-end tests for authenticated data transmission from Z1 mote to TAE
8. Test Replay Packet attacks on LBACS peers
9. Test MITM fake authentication attacks with LBACS peers

5.2 Threat Analysis

A qualitative threat analysis was performed to evaluate the associated risk of various attacks on critical sub-components of the LBACS scheme. The methodology used and accompanying assessment has been included below.

5.2.1 Methodology

The European Telecommunications Standards Institute (ETSI) published a threat assessment methodology in 2003 which utilized a likelihood and impact estimate to rank risk (Kheirabadi, Kulkarni, & Shaligram, 2011). Bardeau (2005) expanded on this to include the difficulty level and motivation required to observe the threat. The resulting risk evaluation grid has been denoted in Table 8 below.

Table 8: Risk Evaluation Grid

		Rationale		
Criteria	Cases	Difficulty	Motivation	Rank
Likelihood	Unlikely	Strong	Low	1
	Possible	Solvable	Reasonable	2
	Likely	None	High	3
		User	System	
Impact	Low	Annoyance	Very limited outages	1
	Medium	Loss of service	Limited outages	2
	High	Long time loss of service	Long time outages	3
Risk	Minor	No need for countermeasures		1, 2
	Major	Threat need to be handled		3, 4
	Critical	High priority		6, 9

The likelihood describes the availability of theoretical and practical knowledge that may be utilized to compromise the scheme. A threat that is considered as *likely* is assigned the highest rank of three (3), implying that all of the required knowledge to compromise the scheme is easily available. On the other hand, if some critical components of the system are available, the threat is assigned a rank of 2 and considered

as *possible.* With the lowest rank of 1, the required or critical information necessary to compromise the scheme is assumed unavailable and therefore considered as *unlikely*. Likelihood may be decomposed further into its technical difficulty and motivation required or potential gain of the attacker. Bardeau considered three levels for technical difficulty, namely *none*, *solvable* and *strong,* which was defined similar to the proposal made by ETSI. *None* implies that the attack has been successful before, *solvable* alludes to theoretical possibility of the attack while *strong* describes a scenario where required theoretical and practical elements are missing. Similarly, three levels where considered for motivation, namely, *high*, *moderate* and *low*. High implies that the expected gain from the attack is significant; moderate implies that there may be service disruption only or the attacker will only break-even with the expected gain and effort (time, resources and cost of the attack). Finally, low implies that the expected gain is minimal.

The impact of a threat may assist in identifying the severity of the scenario in which the scheme becomes compromised and further rationalize the motivation of an attacker interesting in crippling the system using the scheme. Again, three ranks are used from 1 – 3 (named Low, Medium and High respectively) with the counting order determining the severity of impact. Low represented by 1 implies that the network and its participants will experience little to no if an attack occurs while high represented by 3 implies serious consequences and long term outages.

Based on the aforementioned estimates, risk may then be deduced from the product of associated numerical values for likelihood (technical difficulty and motivation) and impact. The possible products may also be categorized based on their ranges to achieve a single risk evaluation metric. These include *minor* (1,2,3) which

identifies the threat as a low priority; *major* (4), implying that the threat should be addressed in the near future and *critical* (6,9) threats as threats which should be addressed immediately with countermeasures and sufficient risk mitigation strategies.

The threat analysis will therefore assess the likelihood and impact of various attacks on critical sub-components of the scheme and the various affected security objectives.

5.2.2 Sub-components

The network that requires provenance of sensor data may be sub-divided into six (6) major components as illustrated in Figure 5 on page 50. These components include:

- Constrained Oracle Sensor
- Wireless Sensor Network (WSN)
- Semi-Trusted Base Station (STBS)
- Blockchain Network
- Trusted Authentication Entity (TAE)
- Consumer Applications

Although consumer applications will not be actively participating in the provenance of data or authentication of participating entities, it should be noted that an adversary may utilize consumer applications or servers as entry points to exploit available vulnerabilities in the respective blockchain network. An exploited vulnerability in the blockchain network as will be discussed later would be an additional threat for the scheme. This threat analysis however will focus on the critical components required to establish the scheme (sensors, WSN, STBS and the blockchain network) with the exception of Trusted

Authentication Entities. It will be assumed that the Trusted Authentication Entity is secure and the adversary in question is not motivated to attack the entity directly.

5.2.3 Threat Assessment

For the purpose of this analysis, a threat may be defined as a possible violation of security arousing from an event or capability to cause harm (Shirey, 2000). An adversary, *A*, will then be identified as an entity, individual or program that is a threat to the system (Shirey, 2000) utilizing the LBACS scheme. It will be assumed that the adversary may have physical access to a subset of sensor devices in a particular sub-network and may or may not have physical access to a semi-trusted base stations. Furthermore, it is assumed that the adversary is highly motivated to compromise the provenance of data on the blockchain. As a result, only the likelihood and impact of each threat will be discussed since the adversary has an assumed motivation of *high* (3) as outlined in the aforementioned methodology. The analysis will therefore describe each threat and it's impact on the sub-components involved (with the exception of Trusted Authentication Entities) and before assigning a rating denoted in the "Risk Evaluation Grid" in Table 8 on page 106 Table 8. Furthermore, each threat's impact on the outlined security objectives of the scheme will be discussed along with the scheme's ability to secure these objectives.

A highly motivated adversary with physical access to a subset of sensor nodes within the WSN is very likely to tamper, capture or attempt to replicate sensor devices. Ideally, a sensor node should incorporate a framework to ensure integrity, confidentiality of data, keys and computations to minimize the effect of unintended physical access or

injected malware. Such capabilities may be included with the assistance of Trusted Platform Modules (TPM). A TPM is microcontroller capable of securely storing artifacts or platform measurements necessary to ensure the integrity of the device such as keys and certifications (Trusted Computing Group, 2015). However, considering a context where a constrained sensor has not implemented this framework or the framework has been compromised then the attacker would gain access to the set of pairwise keys, shared seeds (used to randomize and reduce communication overhead), PRNG and KDF used by the sensor to communicate with each pair within the WSN (including semi-trusted base stations) and the TAE responsible for authenticating the sensor. In the worst case scenario, the adversary may then utilize this key to submit invalid data to the blockchain on behalf of this sensor node. The adversary would have then successfully compromised authentication, integrity, universal forgeability and auditing for this sensor but not the entire network (WSN or blockchain). Since the scheme does not share pairwise keys or seeds amongst other participants (sensor nodes, STBS or TAE) the impact of the attack would be reduced to communication between the compromised sensor, the semi-trusted base station sharing a pairwise key with the sensor and the TAE sharing a pairwise key with the sensor. Possessing a sensor's pairwise keys, shared seeds, PRNG and KDF does not allow an attacker to perform replay attacks since each sensor tracks each request's id sent with the authentication tag. Furthermore, since identity revocation may only be sanctioned from a TAE, they would be unable to revoke identities since they would require the pairwise keys shared between a TAE and other sensor nodes within the WSN and subsequently the STBS shared pairwise key for each sensor node. To compromise N other sensor nodes, the attacker would need to obtain, N STBS shared pairwise keys to

forge $HMAC_1$ of the LBACS authentication tag and N TAE pairwise keys to forge $HMAC_2$ of the LBACS tag. LBACS has therefore increased the complexity to spoof identity revocation even for a highly motivated attacker. Finally, LBACS encourages the dismissal/deletion of pairwise keys and seeds used after each session change, therefore an attacker would be unlikely to compromise weak backward secrecy. Although an adversary, compromising a sensor node, would then be able to generate the next set of keys (K_{i-n}) that would be used to communicate with a peer in an attempt to compromise forward secrecy. If this sensor node has been identified as compromised, LBACS facilitates the revocation of network participants, which allows the network to recover and ignore future attempts to authenticate by this compromised node. As a result, these attacks have been considered as *likely* (3) but with *medium* (2) impact to achieve a risk rating of *critical* (6). These attacks become more severe in the case of a compromised STBS as the attacker not only has access to all the pairwise keys and seeds used in the WSN but has the ability to interact with the blockchain, an attack that will be discussed later. Since the STBS is assumed more computationally resourceful (possessing enough memory, power, storage to sufficiently implement a framework to reduce the impact of physical access), the difficulty considers it as *possible* (2) with a *high* (3) impact and *critical* (6) risk rating.

LBACS aims to secure constrained devices communicating over a WSN which makes it an easier target for traffic analysis and monitoring because of the wireless medium. These passive attacks are therefore easier to accomplish and would aim to infer from the transmitted tag and data the shared pairwise keys, shared seeds (used to randomize and reduce communication overhead), KDF and PRNG. Assuming that the

attacker is aware of the use of the LBACS scheme, KDF and PRNG, the problem of deducing the pairwise key and shared seed becomes more difficult since they are changed each session. Furthermore, the scheme's use of the HMAC construction coupled with the PRNG seed reduction, securely generates a MAC while reducing the available information that would assist in compromising backward secrecy or forward secrecy on each message exchange. These attacks should still be considered as pragmatic due to the possibility of side channel attacks which are implementation specific. As a result, these attacks have been assigned a likelihood of *possible* (2) and an impact of *low* (1) resulting in a *minor* (2) risk rating.

More active WSN threats targeting the physical, link and network layers of the OSI model include jamming (constant, random, deceptive or reactive), radio interference, network collisions, denial of sleep/service and flooding attacks impede wireless communication by inhibiting the transmission and reception of messages on the communication channel (Wenyuan et. al, 2006). Furthermore, they result in battery exhaustion as the sensor's receiver must remain in an active mode (instead of periodic sleep) which requires more power and ultimately reduces the battery lifetime of the sensor. These attacks directly impact identity revocation, since the STBS will be unable to forward a revocation message over a flooded channel. However, due to the lightweight nature of LBACS, battery exhaustion attempts will be less effective as a sensor will cease processing any message containing an invalid authentication tag. These threats have therefore been considered *possible* (2) with a *medium* (2) impact due to the loss of services and a resulting *major* (4) risk.

Other WSN threats affecting message transmission include sinkhole, black hole, wormhole, blackmail, jelly fish, gray hole and selective forwarding attacks. These threats aim to drop, selectively forward or change the route of transmitted messages and are usually initiated through the fabrication of routing requests advertising shorter routes or fake addresses (Bhargava & Goyal, 2014). Implementing LBACS at a lower level of the OSI model (link or network layer) would allow the sensor to ignore unauthenticated network participants sending fabricated routing requests. However, a compromised node still possessing valid keys may still submit authenticated messages to disrupt the networks routing operation. Although, LBACS (a lightweight authentication scheme securing provenance of sensor data on the blockchain) does not cover routing, it's security objectives are still met during these attacks since it was designed for loss-tolerant and time insensitive networks. As illustrated in Figure 11 on page 65, the format for the LBACS authentication tag includes a Request ID which is included in the integrity check and verification step of the authentication process. As such, a receiving node will be able to identify which messages were received thus achieving the objectives of authentication, integrity and being auditable. These attacks may also aim to subvert session change requests and thus put the communicating peers out of sync. This may be ratified with multiple re-transmissions, additional checks (especially for numerous failed transmissions) and post deployment strategies that allows peers to recover shared keys and random seeds as discussed in the proposal section. This tracked Request Id may also assist other techniques utilized to identify and secure networks against these attacks such as watchdog, ACK-based, reputation-based and incentive-based schemes (Bhargava & Goyal, 2014). The Request ID also assists in replay protection as these attacks may replay

valid authentication tags in an attempt to fabricate false routes messages when trying to gain entry to the network. Moreover, even if a node has been compromised, the scheme should sustain its premise on backward and forward secrecy along with universal forgeability as discussed above for threats that possess physical access to devices. These threats have therefore been considered *possible* (2) with a *low* (1) impact to the scheme's objectives and a resulting *minor* (2) risk.

A Man-in-the-middle (MITM) attack occurs when two legitimate parties communicate through an adversary without their knowledge or approval. With the adversary's control of the communication channel they may observe, modify, re-order, insert or drop transmitted packets. This attack is also the basis for session hijacking, in which the attacker hijacks a legitimate session, often flooding the user's radio or ignoring their messages, in order to masquerade as the user to the other communicating party (Bharti & Chaudhary, 2013). As mentioned in the last paragraph about packet dropping threats, the layer of the OSI model at which LBACS is implemented may allow the adversary to initiate a more passive (observe, re-order or drop packets) MITM attack. However, LBACS prevents the insertion or modification of packets which also includes active session hijacking threats where the adversary attempts to masquerade as the sensor. In the worst case scenario, the adversary will designate itself as the message recipient by compromising routing behavior at the link or network layer, but will still be unable to masquerade as the intended recipient without the ability to generate the correct authentication tag. Session attacks are usually possible because critical session details are communicated over the insecure communication medium, such as the session id in a cookie or query parameter (Bharti & Chaudhary, 2013). This attack becomes more likely

when this session identifier remains constant. The LBACS session is characterized by pairwise key and a seed generated by a shared KDF and PRNG respectively. In addition to changing these parameters (pairwise key and shared seed) each session, the authentication tag also utilizes a changing Request ID to generate and verify it's modified HMAC (see Communication Flows section for additional information on generation and verification). Even if the same message is sent repeatedly be legitimate participants within a session, the generated authentication tag changes to maintain authentication or the sender, integrity of the message, replay protection, auditing and universal forgeablity. The MITM threat has therefore been considered as *possible* (2) with a *low* (1) impact to the scheme's objectives and a resulting *minor* (2) risk, while session hijacking has been considered as *unlikely* (1) with a *high* (3) impact to the scheme's objectives and a resulting *major* (3) risk.

Although WSN deployments are usually designed with their environment in mind, highly volatile environments, as are some cases (military, weather monitoring), should always be considered as a threat. These threats would disable the WSN by immobilizing nodes, resulting in a loss of service. Unless captured by an adversary, this individual threat does not pose a significant risk to the security objectives. As a result, the threat has been considered as *possible* (2) with a *low* (1) impact to the scheme's objectives and a resulting *minor* (2) risk.

Internal threats such as malicious employees or error prone organization procedures are critical threats. Although the likelihood varies based on the maturity of the organization's policies and execution of such policies, issues propagating from these policies are likely to reveal much theoretical and practical knowledge of the network's

implementation. These could reveal useful designs and artifacts to an adversary such as certificates, keys or key generation procedures. These threats may be considered as *likely* (3) with a *high* (3) impact to the scheme's objectives and a resulting *critical* (3) risk.

As discussed in the "Blockchain" section of the literature review, utilizing the blockchain provides many security properties such as decentralized autonomy, auditability, replication and integrity. A more objective threat analysis of this sub-component would be more comprehensive and if a specific implementation was considered. Since LBACS is independent of a particular blockchain implementation, this will be considered as an area of future work and the sub-component would be assessed generally as a software implementation and as such would inherit the vulnerabilities associated with all software implementations.

At this time however, it is possible to identify how the failure of this sub-component may affect the scheme since it formalizes the trust backbone of LBACS. In the worst case, the complete failure of a blockchain node would disconnect the sub-network (the types illustrated in Figure 5 on page 50) that utilizes the blockchain node to achieve a global state of the network. If the blockchain node utilized by a Trusted Authentication Entity (TAE) fails (and there is no other TAE using another blockchain node), the network would be unable to sufficiently prove the source of sensor data, ultimately relying only on the authenticated contribution published by the Semi-Trusted Base Station (STBS). Furthermore, the network would be unable to revoke participants, since revocation is an action sanctioned from the trusted sub-network with the assistance of a TAE. In essence, the network could continue to maintain itself until another TAE is able to connect to the network (since a blockchain node will first sync with its peers to

achieve a global state). The most affected participants would be consumer applications requiring complete provenance (not solely relying on the STBS) and revocation. LBACS may also utilize the chain of message submissions stored within the blockchain for implicit security analysis.

Although, not a critical component of the scheme, it highlights the loss of some auditability achieved through the shared use of the blockchain. Similarly, if a STBS loses its ability to interact with the blockchain, it would lose its ability to secure provenance of its forwarded sensor data and the ability to receive revocation messages from TAEs. Post deployment procedures, such as the deployment of new nodes would also be negatively affected. These threats may be considered as *possible* (2) with a *medium* (2) impact to the scheme's objectives and a resulting *major* (4) risk.

Although the blockchain network may be prone to sybil, byzantine attacks or peer nodes who have found it possible to subvert the consensus protocol in their favor, LBACS requires that all interactions on the network be digitally signed and verified utilizing public/private key pairs. This verification process increases the integrity of the global network. Moreover, software vulnerabilities and side channel attacks may expose even more implementation vulnerabilities. These vulnerabilities may be classified under many taxonomies such as the *Comprehensive, Lightweight Application Security Process* v1.0 (Viega & Secure Software Inc, 2005)

the *Seven Pernicious Kingdoms* (Tsipenyuk, Chess, & McGraw, 2005) or even a simpler taxonomy of system vulnerabilities as illustrated in Figure 23 below to name a few. These threats may be considered as *possible* (2) with a *medium* (2) impact to the scheme's objectives and a resulting *major* (4) risk.

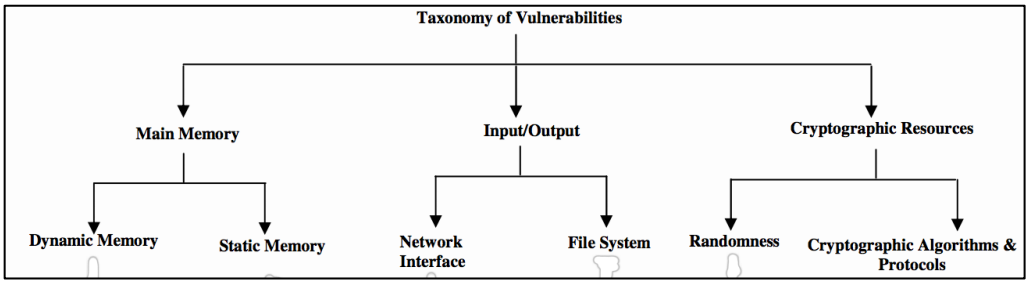

Figure 23: Taxonomy of System Vulnerabilities (Hansen & Hansen, 2010)

The following tables assign pseudo-identifies for the earlier identified security objectives (Table 9) and summarizes the LBACS threat assessment (Table 10) respectively. It should be noted that Table 9 above was used to encode the LBACS security objectives with identifiers that would be used in Table 10: LBACS Threat Assessment Summary on page 119. Table 10, contains the list of vulnerabilities/threats including MITM attacks and how they affected each security objective (encoded in the table) based on the threat assessment methodology. The table also summarizes the likelihood and impact of the threat or vulnerability when using LBACS.

Table 9: Security Objective Key for LBACS Threat Assessment Summary Table

Identifier	Security Objective
1	Authentication
2	Integrity
3	Replay protection
4	Forward Secrecy
5	Backward Secrecy
6	Universal forgeability
7	Identity Revocation
8	Auditable
ALL	All security objectives

Table 10: LBACS Threat Assessment Summary

Component	Sensor		WSN		STBS		Blockchain	
Threat	Difficulty or Likelihood / Impact / Risk | Objectives							
Device Tampering	3/2/6	1248			2/3/6	1248		
Device Capture	3/2/6	1248			2/3/6	1248		
Jamming	3/1/3	1248			2/3/6	1248		
Node Replication	3/1/3	1248			2/3/6	1248		
Traffic Analysis			2/1/2	456				
Traffic Monitoring			2/1/2	456				
Network Collisions			2/2/4	17				
Noise			2/2/4	17				
Denial of Service			2/2/4	17				
Denial of Sleep			2/2/4	17				
Flooding			2/2/4	17				
Sinkhole			2/1/2	37				
Blackhole			2/1/2	37				
Wormhole			2/1/2	37				
Blackmail			2/1/2	37				
Jellyfish			2/1/2	37				
Grayhole			2/1/2	37				
Selective forwarding			2/1/2	37				
Man-in-the-middle	2/1/2	1234678			1/3/3	1234678		
Session Hijacking	2/1/2	1234678			1/3/3	1234678		
Natural Environment	2/1/2	7			2/1/2	7		
Replay Attack			1/3/3	1236				
Insufficient Security Policies	3/3/3	ALL	3/3/3	ALL	3/3/3	ALL	3/3/3	ALL
Malicious Employees	3/3/3	ALL	3/3/3	ALL	3/3/3	ALL	3/3/3	ALL

Component	Sensor		WSN		STBS		Blockchain	
Threat	Difficulty or Likelihood / Impact / Risk | Objectives							
STBS Blockchain node failure							2/2/4	1278
TAE Blockchain node failure							2/2/4	1278
Side Channel	2/2/4	ALL			2/2/4	ALL	2/2/4	ALL
Software Vulnerabilities	2/2/4	ALL			2/2/4	ALL	2/2/4	ALL

5.3 LBACS Comparison

The following table highlights the vulnerabilities identified in the literature review and subsequent threat assessment that LBACS directly protects sensors against.

Table 11: Vulnerabilities prevented by LBACS

Vulnerability	With LBACS	Without LBACS
Jamming	Y	N
Denial of Service	Y	N
Denial of Sleep	Y	N
Man-in-the-middle	Y	N
Session Hijacking	Y	N
Replay Attack	Y	N

As highlighted in the literature review, securing WSNs with lightweight authentication schemes has been an active research area. Schemes and protocols such as:

- Secure Protocols for Sensor Networks (SPINS) (consolidating Secure Network Encryption Protocol (SNEP) and the “micro” version of the Timed, Efficient, Streaming, Loss-tolerant Authentication Protocol (μTesla))
- Short Message Authentication CheckK (SMACK) for battery exhaustion
- TinySec

- Authentication and Anti-replay Security Protocol (AASP)
- Datagram Transport Layer Security Protocol (DTLS) with COAP
- Lithe
- Implicit Security Authentication Scheme

all address authentication and other security concerns in different ways. The remainder of this section will be dedicated to comparing the proposed Lightweight Blockchain Authentication for Constrained Sensors (LBACS) scheme to these existing schemes.

5.3.1 SPINS

Although both SPINS and LBACS facilitate peer and group authentication in a WSN, SPINS also provides data confidentiality and data freshness based on loose time synchronization. LBACS is time independent however, instead using a PRNG to enforce integrity. Moreover, while both utilizes an iterative key chain, SPINS assumes the base station is computationally resourceful and secure while LBACS does not as it utilizes a Trusted Authentication Entity connected by a fault tolerant and auditable blockchain network. Furthermore, both utilize a non-transmitted message counter, preventing replay attacks. In addition, SPINS uses a derived key to compute its authentication tag to protect its master key, while LBACS reduces the key storage requirements by using a key reduction mechanism and PRNG to reduce the transmitted authentication tag and information available for differential cryptanalysis.

5.3.2 TinySec

Similar to the TinySec architecture included in the TinyOS providing replay protection, message integrity and authentication, LBACS also asserts these security

objectives. However, by operating at the link-layer, TinySec also provides confidentiality and aims to support end-to-end security. Unlike TinySec which does not provide replay protection, LBACS includes a request ID to address these concerns. Furthermore, to achieve semantic security TinySec utilizes Initialization Vectors (IV) which are added to the communication overhead. LBACS however relies on the HMAC reduction, changing request ID, PRNG and KDF to reduce the information available for cryptanalysis.

5.3.3 Authentication and Anti-replay Security Protocol

Similar to the AASP, LBACS also provides integrity, intrusion detection, anti-replay and authentication. It should be noted that AASP utilizes two approaches. The first approach utilizes an authenticated handshake similar to the diffie hielman exchange for new peers which presents an additional communication overhead. LBACS requires an authenticated peer message from the KDC, distributed via the blockchain to add a new peer to its keychain. This provides greater control and auditability. Interestingly, the second approach, the last MAC method, provides authentication by hashing the shared key and previous message. This raises concerns in lossy and low power networks (LLNs) with high transmission rates as messages may be lost. LBACS enforces integrity by including the message in the HMAC but iterates the shared keys and PRNG thus facilitating these types of networks. Although this last MAC method assists with replay protection, LBACS utilizes a request id that is a part of the session size to accomplish the same.

5.3.4 DTLS and Lithe

Unlike AASP, COAP over DTLS (COAPs) and LBACS was designed with LLNs in mind, where handshake messages may not be suitable. However, unlike LBACS, DTLS provides confidentiality. Similar to LBACS which includes a request ID, DTLS uses sequence numbers to facilitate message reordering and replay protection. Furthermore, unlike DTLS, LBACS does not accommodate for fragmented messages. Although, DTLS has more communication overhead dependent on the chosen HMAC size for LBACS, Lithe utilizes 6LoWPAN to further compress DTLS.

5.3.5 Implicit Security Authentication Scheme

Similar to the Implicit Security Authentication Scheme which identifies malicious node behavior to authenticate nodes, LBACS monitors node behaviour with the assistance of the blockchain. However, in LBACS, the monitored parameters need not be benchmarked before deployment allow easier to deployment. For example, if a rogue base station using LBACS, has managed to spoof an authenticated transmission from a trusted authentication entity, this attempt may be discovered when the Trusted authentication entity fails to elicit a response from the sensor because of a failed authentication attempts.

5.3.6 Short Message Authentication Check

Unlike many of the schemes and authentication methods identified in the literature review, LBACS is implemented above the network layer of the OSI model. Although, this may seem as an additional overhead, the scheme benefits from the well-

researched and secure design considerations of lower layers of the OSI model while providing authentication and the optional additional provenance on the blockchain. Another security scheme operating above the transport layer but only considering the immediate WSN is SMACK. Although SMACK includes the COAP version, code, token length, message type, message id and request id in the integrity check, it does not include the payload or the sender and recipients. LBACS, while acting independent of a particular protocol above the network layer of the OSI model, also focuses on the integrity of the payload, recipient and sender. Another concern with SMACK is the ability of an attacker to perform a replay attack by submitted an accepted message from the last session (Gehrmann, C., Tiloca, M., & Hoglund, R. 2015). LBACS prevents this occurrence as each session change iterates on the shared key and random seed generated by the KDF and PRNG respectively.

5.4 Results

The following metrics and tests results were recorded throughout the investigation.

5.4.1 Storage

Since LBACS aims to allow constrained oracles to achieve authentication and provenance on larger blockchain networks, storage needs, a common concern for constrained devices were recorded. Specifically, the size of the program data and key storage requirements were recorded. Key storage requirements were captured at run-time using the *sizeof* function in *C programming language used*, while program data size was identified as the difference of the compiled mote program with and without LBACS. It should be noted that program data size for LBACS will be considered with the ability to

retrieve and update the keychain stored using the Contiki file system in addition to token generation for two peers (Semi-Trusted Base Station and Trusted Authentication Entity). Furthermore, the key sizes used were 32 bytes each as seen in Appendix A. These results have been highlighted in the table below.

Table 12: LBACS Storage Requirements

Metric	Size (bytes)
LBACS Keychain (see Figure 7 on page 55)	180
Z1 Buoy Reporter without LBACS	49,561
Z1 Buoy Reporter with LBACS	58,997
LBACS program data	9,436

5.4.2 Communication Overhead

Independent of the varying message size (no more than 24 bytes from sensors), LBACS utilized 8 bytes for its authentication tag as specified in the configuration parameters in the Configuration section of the previous Implementation Chapter.

5.4.3 Power Consumption

Power consumption for LBACS on the Z1 mote was also recorded and compared with the same mote not using LBACS. The CPU time recorded over 50 intervals with the assistance of the Contiki Energest module was used to calculate power consumption using the following formula:

$$Power\ Consumption\ (mW) = \frac{CPU\ Time\ x\ Current\ x\ Voltage}{RTIMER\_SECOND\ x\ INERVAL\ TIME}$$

Although the CPU time was recorded, the remaining parameter values for the formula were acquired from the Z1 data sheet based on the mote's use. In this instance, calculations used a 3.6V (Voltage), 0.5mA (Current) with the frequency of the internal clock (RTIMER_SECOND) to be 32768 ticks at a 10 second interval (INTERVAL TIME). The average power difference illustrated in Figure 24 below was 0.077 ± 0.01 mW using a 95% confidence interval. The CPU time for each observation along with additional metrics such as low power mode, transmit, listen and clock time have been included in Appendix D.

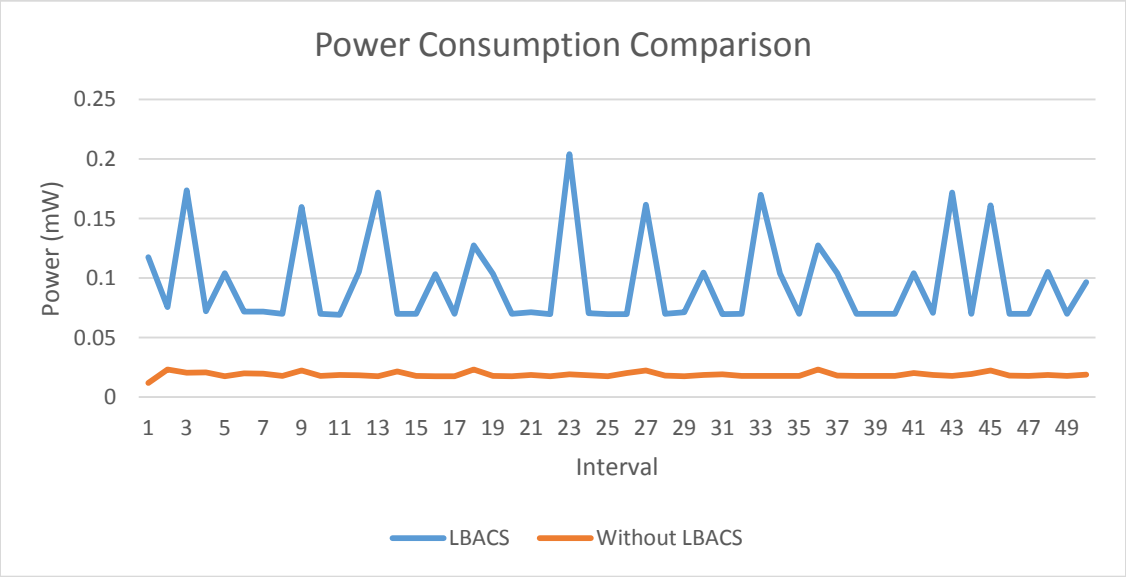

Figure 24: Power Consumption Comparison with LBACS

5.4.4 Time

Moreover, the time recorded to generate the LBACS authentication tag for two (2) peers (STBS and TAE) on the Z1 mote was recorded with the assistance of the Contiki Energest module. As illustrated in Figure 25 below, LBACS token generation on the Z1 mote averaged 38 milliseconds (ms) with the exception of session changes which

averaged 107 ms due to the combined use of the KDF and PRNG. Time was recorded over 388 tag generation attempts which included 35 session changes. The data recorded has been included in Appendix C.

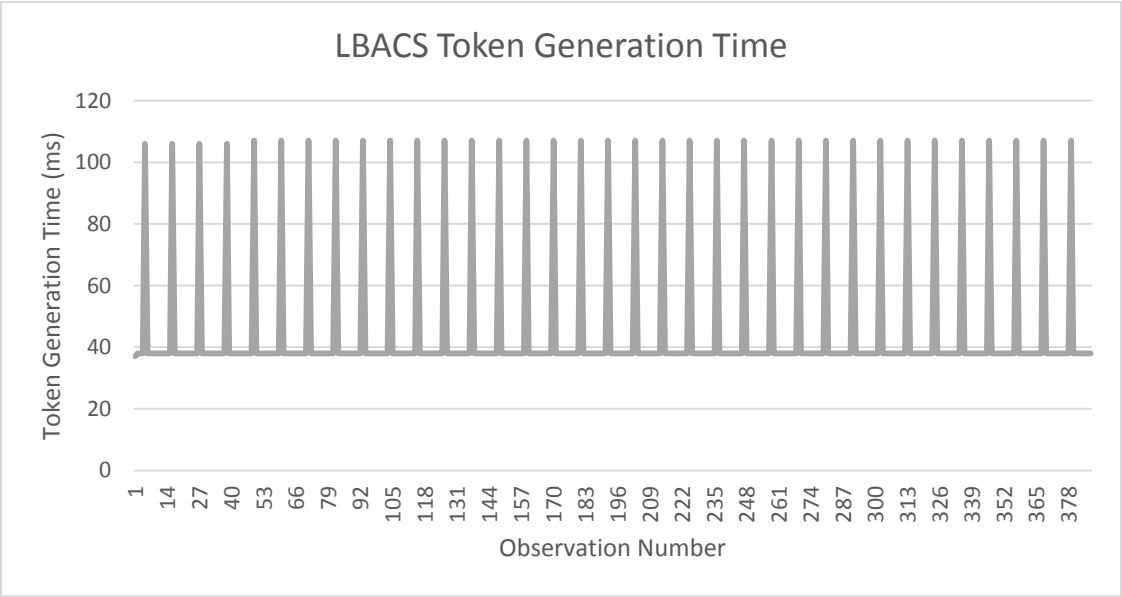

Figure 25:LBACS Token Generation Time on Z1

5.4.5 Functional Test

In addition to unit and incremental testing to assert the correctness of the implementation, a complete end-to-end test was performed. In this test, the session size was reduced to 12 available requests and up to 400 requests were sent from the Z1 mote to the Semi-Trusted Base Station (STBS) and to the Trusted Authentication Entity (TAE) via the blockchain. The Firefox Copper COAP Client and a simpler COAP application written in Node.js was also used to inject replay and spoof attacks into the WSN which

were rejected by network peers. Sampled serial output from the Z1 for request id 8 has been included in Appendix F.

Although many requests were successful, due to the Low Power and Lossy Network (LLN), some packets were loss in transmission. As a result, the session change request was re-transmitted to ensure that the base station would receive this request. The TCP dump showing network packets acquired from the Raspberry PI's slip (USB) connection to the router has also been included in Appendix E. Moreover, log output from the STBS denoting the LBACS authentication, session change, signature generation and publishing of the buoy data has also been included in Appendix G. Similarly, logs from the TAE, acknowledging newly published buoy data from the blockchain, verifying the STBS in the Certificate Registry, verifying the signature of the STBS and subsequently authenticating the sensor mote using LBACS and approving the message on the blockchain has been included in Appendix H. It should be noted that logs from the STBS and the TAE were retrieved from an asynchronous implementation and therefore will not always follow the order in which messages were received especially due to the latency from interacting with the blockchain implementation. The Certificate Authority (CA) was also implemented with the ability to approve or remove entities (STBS, TAE or CA) from the certificate registry using a restful endpoint.

5.4.6 Hardware Overhead

The hardware overhead on a constrained device contributed by a LBACS implementation is directly dependent on the hardware requirements of the cryptographic

constructions (KDF, PRNG, HMAC) and application needs. The hardware requirements of LBACS has been modeled using the block diagram in Figure 26 which describes LBACS as an integrated circuit.

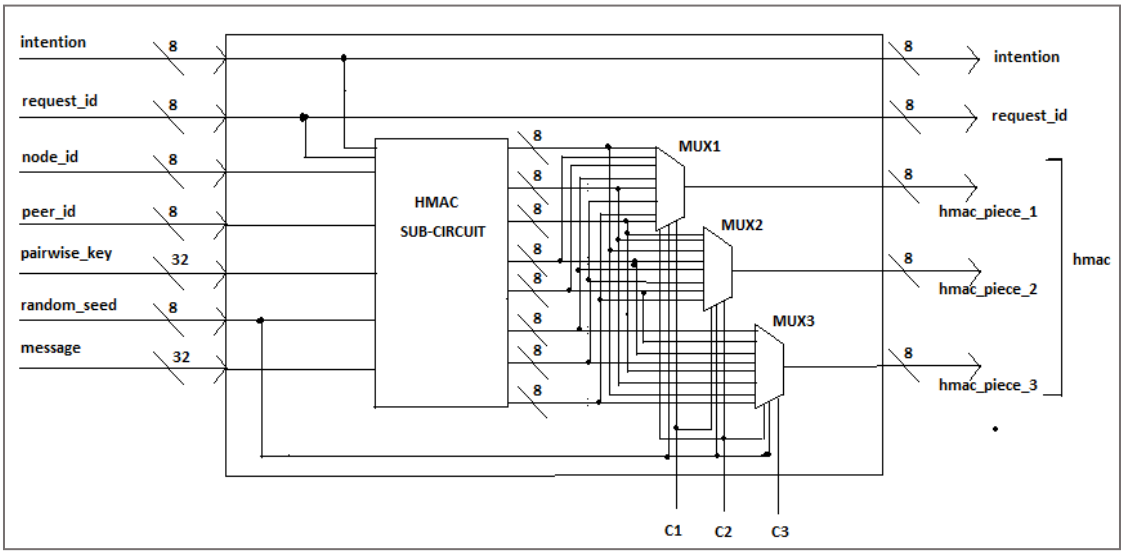

Figure 26: LBACS Authentication Hardware Block Diagram

The block diagram illustrates an authentication module, similar to the one described in the Implementation chapter, for a LBACS Integrated Circuit (IC) that uses a HMAC component and three multiplexers to achieve the HMAC reduction as described in the proposal. Specifically, the module uses mainly eight (8) bit data paths to accept the authentication parameters (intention, request id, node id, peer id random seed) except for the pairwise key and message which are thirty-two (32) or four times eight (4 x 8) data paths to reduce the required design area for low core micro processor and System on a Chip (SoC) designs. Both the intention bits and request id are included in the IC's output. In order to generate the HMAC, LBACS uses the HMAC specified in its configuration parameters by the system designer as highlighted in the proposal. The HMAC output, illustrated as eight eight bit data paths is fed into three 8-1 multiplexers (MUX1, MUX2

and MUX3) which reduce the HMAC output based on the inputted data select lines. In this implementation, the data select lines are three constants (C1, C2, C3), and the shared PRNG random seed.

This design was implemented in Very High Speed Integrated Circuit Hardware Description Language (VHDL) IEEE1076 in Appendix K. VHDL IEEE1076 provides a standard formal notation that assists in the development, synthesis, testing and communication of hardware designs ("IEEE Standard VHDL Language Reference Manual", 2009). This therefore allows this design to be easily replicated, communicated and tested for further research and implementations.

In this implementation, the Keccak-f permutation was chosen as the variable size, output sponge construction allowed it to be used as a suitable PRNG, KDF and HMAC. Many contributions have been made to optimize the area, storage and memory requirements on low core processors used in constrained devices as proposed in the Keccak Implementation Overview (Bertoni et. al, 2011). Variants of these contributions have even targeted very constrained RFID devices (Pessl and Hutter, 2013) (Kavun and Yalcin, 2010) and Fixed Programmable Gated Array implementations (FPGA) (Provelengios et. al, 2012).

Chapter 6: Conclusion

With the research objectives met, this section will discuss the findings of the research project, specifically answer the research questions and highlight possible areas for future work.

6.1 Discussion

The review of existing literature highlighted several concerns and opportunities as it pertains to achieving provenance of sensor data on the blockchain. Concerns arising from the design considerations and review of existing WSN authentication schemes included:

- Costs (financial and communication overhead) associated with key management
- Costs to deploy secure base stations
- Fault tolerance
- Loss of provenance through network topologies
- One solution does not fit all
- Inability to deploy secure cryptographic constructions on constrained sensors to communicate with more resource competent nodes.

Although the blockchain addresses several issues such as byzantine fault tolerance, provenance, integrity and auditability, current implementations are too resource intensive for constrained devices.

To address this gap in the research, the Lightweight Blockchain Authentication for Constrained Sensors (LBACS) scheme was proposed. With the assistance of a lightweight multi-signature authentication tag, symmetric and asymmetric schemes, it

allows constrained sensor devices to authenticate with resource competent peers on the blockchain through semi-trusted base stations. The integration of the blockchain and the use of several cryptographic constructions allows the scheme to observe the various design considerations of sensors networks while providing the following security objectives:

1. Authentication
2. Blockchain Provenance
3. Integrity
4. Replay Protection
5. Weak Backward Secrecy
6. Weak Forward Secrecy
7. Universal Forgeability
8. Identity Revocation
9. Auditable

When using the LBACS configuration parameters as described in Table 5 on page 85, the resource requirements for a constrained sensor authenticating and providing provenance through integrity on the blockchain has been included in Table 3 below.

Table 13: LBACS Resource Requirements Summary

Metric	Value
LBACS Keychain (see Figure 7 on page 55)	180 bytes
LBACS program data	9,436 bytes
Communication Overhead	8 bytes
Power Consumption	0.077 ± 0.01 mW
Token Generation Time (general)	38ms
Token Generation Time (during session change)	107ms

Similarly, the hardware requirements for a sensor device utilizing LBACS or implementing LBACS as an independent Integrated Circuit were highlighted and illustrated in a block diagram in Figure 26 on page 129 and also in VHDL in Appendix K.

The functional tests conducted attested to the qualitative thread and risk assessment conducted as participating nodes successfully rejected replay and spoofed attacks. Specifically, LBACS protects nodes from vulnerabilities such as jamming, denial of service, denial of sleep, replay, session hijacking and man-in-the-middle attacks. Although, routing related attacks may result in packet loss, LBACS incorporates post-deployment key management methods to allow node and session recovery if necessary. Furthermore, when compared to existing authentication schemes such as DTLS, TinySec, Lithe, AASP, SPINS and SMACK, LBACS possessed many qualities of the compared schemes with the exception of confidentiality. Although similar, LBACS possessed several attractive traits such as being time independent when compared to SPINS, preventing replay attacks when compared to TinySec, a small communication overhead, fault tolerance, maintenance of provenance and the ability to integrate with the blockchain. When compared to a node with no security, LBACS directly provides protection against the following vulnerabilities:

1. Jamming
2. Denial of Service
3. Denial of Sleep
4. Man-in-the-middle
5. Session Hijacking

6. Replay Attack

Based on the results arising from the quantitative and qualitative study, the Lightweight Blockchain Authentication for Constrained Sensors (LBACS) scheme has proven to provide a lightweight means for constrained sensors to authenticate and achieve provenance of sensor data on the blockchain while providing several other benefits when compared to other authentication schemes.

Although, the use case discussed in this thesis pertained to buoy monitoring, it should be noted that the applicability of LBACS extends to all IoT. This includes areas such as health, mining, supply chain management, aerospace to name a few. The configuration parameters presented by LBACS increases its interoperability as devices with varying computational resources may use a different KDF, PRNG, HMAC based on the computational resources and needs of the network. This allows devices with hardware enhanced modules such as the Texas Instruments CC2538 that has an AES128/256 SHA2 Hardware Encryption Engine (Gehrmann, Tiloca, & Hoglund, 2015) or mobile phones with accelerated GPUs to utilize cryptographic primitives which are best suited for their hardware and the network's needs. Similarly, the more competent blockchain nodes need not be constrained to the Elliptic Curve implementation used in the buoy monitoring implementation and may use other methods such as variants of RSA algorithm.

6.2 Future Work

Due to the time constraints of this thesis submission and the applicability and need for sensor provenance, there is room for additional scope and work with LBACS. Additional tests with other sensors or motes, Fixed Programmable Gated Arrays,

blockchain implementations and LBACS configuration parameters may also identify additional concerns or opportunities. Furthermore, several concerns such as confidentiality, routing and closer integration with layers of the OSI model may be explored. Furthermore, existing approaches to provenance which focus more on the metadata collected such as the Open Provenance Model may be explored as LBACS only asserts the integrity of data transmitted (which may include metadata) but does not specify any meta data requirements.

References

Adejo, A., Onumanyi, A., Anyanya, J., & . Oyewobi, S. (2013). OIL AND GAS PROCESS MONITORING THROUGH WIRELESS SENSOR NETWORKS: A SURVEY. *Ozean Journal Of Applied Sciences*, *6*(2), 39-43. Retrieved from http://ozelacademy.com/ojas.v6.i2-1.pdf

Akyildiz, I., Su, W., Sankarasubramaniam, Y., & Cayirci, E. (2002). Wireless sensor networks: a survey.*Computer Networks*, *38*(4), 393-422. http://dx.doi.org/10.1016/s1389-1286(01)00302-4

Alemdar, H. & Ersoy, C. (2010). Wireless sensor networks for healthcare: A survey. *Computer Networks*, *54*(15), 2688-2710. http://dx.doi.org/10.1016/j.comnet.2010.05.003

AlZain, M., Soh, B., & Pardede, E. (2013). A Byzantine Fault Tolerance Model for a Multi-cloud Computing. *2013 IEEE 16Th International Conference On Computational Science And Engineering*. http://dx.doi.org/10.1109/cse.2013.30

Atzori, L., Iera, A., & Morabito, G. (2010). The Internet of Things: A survey. *Computer Networks*,*54*(15), 2787-2805. http://dx.doi.org/10.1016/j.comnet.2010.05.010

Arkko, J., Devarapalli, V., & Dupont, F. (2004). Using IPsec to Protect Mobile IPv6 Signaling Between Mobile Nodes and Home Agents. http://dx.doi.org/10.17487/rfc3776

Barbeau, M. (2005). WiMax/802.16 threat analysis. *Proceedings Of The 1St ACM InternationalWorkshop On Quality Of Service & Security In Wireless And Mobile Networks - Q2swinet '05*. http://dx.doi.org/10.1145/1089761.1089764

Beigl, M., Krohn, A., Zimmer, T., & Decker, C. (2014). Typical sensors needed in ubiquitous and pervasive computing. *Proceedings Of First International Workshop on Networked Sensing Systems (INSS)*, 22-23. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.4.195&rep=rep1&type=pdf

Benet, J. (2016). *IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3)* (1st ed., pp. 1-11). IPFS. Retrieved from https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf

Bertoni, G., Daemen, J., Peeters, M., & Van Assche, G. (2011). *Cryptographic sponge functions* (1st ed.). Retrieved from http://sponge.noekeon.org/CSF-0.1.pdf

Bertoni, G., Daemen, J., Peeters, M., & Van Assche, G. (2011). *The Keccak reference* (3rd ed., pp. 7-10, 31-65). Retrieved from http://keccak.noekeon.org/Keccak-reference-3.0.pdf

Bhargava, K. & Goyal, D. (2014). PACKET DROPPING ATTACKS IN MANET: A SURVEY. *Journal Of Advanced Computing And Communication Technologies*, *2*(3), 14-18. Retrieved from http://www.jacotech.org/uploads/1403889878__64718502.pdf

Bharti, A. & Chaudhary, M. (2013). Detection of Session Hijacking and IP Spoofing Using Sensor Nodes and Cryptography. *IOSR Journal Of Computer Engineering (IOSR-JCE), 13*(2), 66-73. Retrieved from http://www.iosrjournals.org/iosr-jce/papers/Vol13-issue2/K01326673.pdf?id=4013

Bitcoin Charts - Blockchain.info. (2016). *Blockchain.info*. Retrieved 2 August 2016, from https://blockchain.info/charts/blocks-size

Camtepe, S. & Yener, B. (2007). Combinatorial Design of Key Distribution Mechanisms for Wireless Sensor Networks. *IEEE/ACM Transactions On Networking*, *15*(2), 346-358. http://dx.doi.org/10.1109/tnet.2007.892879

Certicom Research,. (2010). *Standards for Efficient Cryptography 2 (SEC 2)* (1st ed., p. 9). Standards for Efficient Cryptography. Retrieved from http://www.secg.org/sec2-v2.pdf

Chan, H., Perrig, A., & Song, D. (2003). Random Key Predistribution Schemes for Sensor Networks. *Proceedings Of The 2003 IEEE Symposium On Security And Privacy*, 197. Retrieved from http://dl.acm.org/citation.cfm?id=830566

Chen, H., Chen, X., & Niu, J. (2010). Implicit Security Authentication Scheme in Wireless Sensor Networks. *2010 International Conference On Multimedia Information Networking And Security*. http://dx.doi.org/10.1109/mines.2010.170

Chen, H. & Xie, L. (2014). Improved One-Way Hash Chain and Revocation Polynomial-Based Self-Healing Group Key Distribution Schemes in Resource-Constrained Wireless Networks. *Sensors*, *14*(12), 24358-24380. http://dx.doi.org/10.3390/s141224358

CHELLI, K. (2015). Security Issues in Wireless Sensor Networks: Attacks and Countermeasures.*Proceedings Of The World Congress On Engineering 2015*, *1*(1), 519-524. Retrieved from http://www.iaeng.org/publication/WCE2015/WCE2015_pp519-524.pdf

Choi, Y., Jeon, Y., & Park, S. (2010). A study on sensor nodes attestation protocol in a

Wireless Sensor Network. In Advanced Communication Technology (ICACT), 2010 The 12th International Conference on (pp. 574 - 579). Phoenix Park: IEEE. Retrieved from http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=5440398

Christidis, K. & Devetsikiotis, M. (2016). Blockchains and Smart Contracts for the Internet of Things. *IEEE Access, 4*, 2292-2303. http://dx.doi.org/10.1109/access.2016.2566339

Contract - Bitcoin Wiki. (2015). *Bitcoin Wiki*. Retrieved 2 July 2016, from https://en.bitcoin.it/wiki/Contract

Delegated Proof of Stake. (2016). *Delegated Proof of Stake*. Retrieved 2 August 2016, from http://docs.bitshares.org/bitshares/dpos.html

Duetre, B., Cheung, S. and Levy, J. (2004) *Lightweight Key Management in Wireless Sensor Networks by Leveraging Initial Trust*, Contract F30602-02-C-0212, April, Menlo Park, CA: SRI International.

Ethereum Benchmarks. (2016). *GitHub*. Retrieved 4 July 2016, from https://github.com/ethereum/wiki/wiki/Benchmarks

Gehrmann, C., Tiloca, M., & Hoglund, R. (2015). SMACK: Short message authentication check against battery exhaustion in the Internet of Things. *2015 12Th Annual IEEE International Conference On Sensing, Communication, And Networking (SECON)*, 274 - 282. http://dx.doi.org/10.1109/sahcn.2015.7338326

Gheorghe, L., Rughinis, R., Deaconescu, R., & Tapus, N. (2010). Authentication and Anti-replay Security Protocol for Wireless Sensor Networks. *2010 Fifth International Conference On Systems And Networks Communications*. http://dx.doi.org/10.1109/icsnc.2010.9

Gibson, A. (2014). *TLSnotary - a mechanism for independently audited https sessions* (1st ed., pp. 1-10). Retrieved from https://tlsnotary.org/TLSNotary.pdf

Goldwasser, S. & Micali, S. (1984). Probabilistic encryption. *Journal Of Computer And System Sciences*, *28*(2), 270-299. http://dx.doi.org/10.1016/0022-0000(84)90070-9

Hansen, J. & Hansen, N. (2010). A taxonomy of vulnerabilities in implantable medical devices. *Proceedings Of The Second Annual Workshop On Security And Privacy In Medical And Home-Care Systems - SPIMACS '10*. http://dx.doi.org/10.1145/1866914.1866917

Hearn, M. (2016). *Understanding the bitcoinj security model*. *Bitcoinj.github.io*. Retrieved 3 July 2016, from https://bitcoinj.github.io/security-model

Henzen, L., Aumasson, J., Meier, W., & Phan, R. (2011). VLSI Characterization of the Cryptographic Hash Function BLAKE. *IEEE Transactions On Very Large Scale Integration (VLSI) Systems*, *19*(10), 1746-1754. http://dx.doi.org/10.1109/tvlsi.2010.2060373

Heinzelman, W., Chandrakasan, A., & Balakrishnan, H. (2002). An application-specific protocol architecture for wireless microsensor networks. *IEEE Transactions On Wireless Communications*,*1*(4), 660-670. http://dx.doi.org/10.1109/twc.2002.804190

Hossain, I. & Mahmud, S. (2007). Analysis of Group Key Management Protocols for Secure Multicasting in Vehicular Software Distribution Network. *Third IEEE International Conference On Wireless And Mobile Computing, Networking And Communications (Wimob 2007)*. http://dx.doi.org/10.1109/wimob.2007.4390819

Hu, Y., Perrig, A., & Johnson, D. (2006). Wormhole attacks in wireless networks. *IEEE J. Select. Areas Commun., 24*(2), 370-380. http://dx.doi.org/10.1109/jsac.2005.861394

IBM Corporation,. (2015). *Device democracy: Saving the future of the Internet of Things* (pp. 1-23). NY: IBM Corporation. Retrieved from http://www-935.ibm.com/services/us/gbs/thoughtleadership/internetofthings/

IEEE Standard VHDL Language Reference Manual. (2009). http://dx.doi.org/10.1109/ieeestd.2009.4772740

Jooyoung Lee and D. R. Stinson, "A combinatorial approach to key predistribution for distributed sensor networks," *IEEE Wireless Communications and Networking Conference, 2005,* 2005, pp. 1200-1205 Vol. 2.

Karlof, C., Sastry, N., & Wagner, D. (2004). TinySec. *Proceedings Of The 2Nd International Conference On Embedded Networked Sensor Systems - Sensys '04*, 162-175. http://dx.doi.org/10.1145/1031495.1031515

Kavun, E., & Yalcin, T. (2010). A Lightweight Implementation of Keccak Hash Function for Radio-Frequency Identification Applications. *Radio Frequency Identification: Security And Privacy Issues*, 258-269. http://dx.doi.org/10.1007/978-3-642-16822-2_20

Khalique, A., Singh, K., & Sood, S. (2010). Implementation of Elliptic Curve Digital Signature Algorithm. *International Journal Of Computer Applications*, *2*(2), 21-27. http://dx.doi.org/10.5120/631-876

Kheirabadi, S., Kulkarni, N., & Shaligram, A. (2011). WIRELESS SENSOR

NETWORK BASED TRAFFIC MONITORING; OVERVIEW AND THREATS TO ITS SECURITY. *Journal Of Global Research In Computer Science*, *2*(8), 60-66. Retrieved from http://www.rroij.com/open-access/wireless-sensor-network-based-traffic-monitoring- overview-and-threats-to-its-security-60-66.pdf

Kortuem, G., Kawsar, F., Sundramoorthy, V., & Fitton, D. (2010). Smart objects as building blocks for the Internet of things. *IEEE Internet Computing*, *14*(1), 44-51. http://dx.doi.org/10.1109/mic.2009.143

Kosba, A., Miller, A., Shi, E., Wen, Z., & Papamanthou, C. (2015). Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. *IACR Cryptology Eprint Archive*, *2015*, 675. Retrieved from https://eprint.iacr.org/2015/675.pdf

Lange, F. & Trón, V. (2015). *RLPx Encryption*. *GitHub*. Retrieved 4 August 2016, from https://github.com/ethereum/go-ethereum/wiki/RLPx-Encryption

Lee, J., Su, Y., & Shen, C. (2007). A Comparative Study of Wireless Protocols: Bluetooth, UWB, ZigBee, and Wi-Fi. *IECON 2007 - 33Rd Annual Conference Of The IEEE Industrial Electronics Society*, 46 - 51. http://dx.doi.org/10.1109/iecon.2007.4460126

Leister, W., Fretland, T., & Balasingham, I. (2008). Use of MPEG-21 for Security and Authentication in Biomedical Sensor Networks. *2008 Third International Conference On Systems And Networks Communications*. http://dx.doi.org/10.1109/icsnc.2008.24

Light Ethereum Subprotocol (LES). (2016). *GitHub*. Retrieved 13 July 2016, from https://github.com/zsfelfoldi/go-ethereum/wiki/Light-Ethereum-Subprotocol-%28LES%29

Liu, D., Ning, P., & Sun, K. (2003). Efficient self-healing group key distribution with revocation capability. *Proceedings Of The 10Th ACM Conference On Computer And Communication Security - CCS '03*, 231-240. http://dx.doi.org/10.1145/948109.948141

Liu Feng, & Zhang Wenpeng,. (2010). The study on key distribution and management mechanisms in Wireless Sensor Networks. *2010 The 2Nd International Conference On Industrial Mechatronics And Automation*. http://dx.doi.org/10.1109/icindma.2010.5538067

Lopez, J., Roman, R., & Alcaraz, C. (2009). Analysis of Security Threats, Requirements, Technologies and Standards in Wireless Sensor Networks. *Foundations of Security Analysis And Design V*, 289-338. http://dx.doi.org/10.1007/978-3-642-03829-7_10

OASIS Message Queuing Telemetry Transport (MQTT) TC,. (2014). *MQTT Version 3.1.1*. *Docs.oasis-open.org*. Retrieved 5 August 2016, from http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

Oracalize API Reference. (2016). *Docs.oraclize.it*. Retrieved 2 July 2016, from http://docs.oraclize.it/

Ozdemir, S., Peng, M., & Xiao, Y. (2013). PRDA: polynomial regression-based privacy-preserving data aggregation for wireless sensor networks. *Wirel. Commun. Mob. Comput.*, *15*(4), 615-628. http://dx.doi.org/10.1002/wcm.2369

Percival, C. (2009). Stronger key derivation via sequential memory-hard functions. *Self-published*, 1-16. http://www.tarsnap.com/scrypt/scrypt.pdf

Perera, C., Zaslavsky, A., Christen, P., & Georgakopoulos, D. (2014). Context Aware Computing for The Internet of Things: A Survey. *IEEE Communications Surveys & Tutorials, 16*(1), 414-454. http://dx.doi.org/10.1109/surv.2013.042313.00197

Perrig, A., Szewczyk, R., Tygar, J., Wen, V., & Culler, D. (2002). SPINS: Security Protocols for Sensor Networks. *Wireless Networks*, *8*(5), 521-534. http://dx.doi.org/10.1023/a:1016598314198

Pessl, P., & Hutter, M. (2013). Pushing the Limits of SHA-3 Hardware Implementations to Fit on RFID. *Cryptographic Hardware And Embedded Systems - CHES 2013*, 126-141. http://dx.doi.org/10.1007/978-3-642-40349-1_8

Ponemon Institute LLC,. (2016). 2016 Cost of Data Breach Study: Global Analysis (p. 1). IBM. Retrieved from http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=SEL03094WWEN

Popejoy, S. (2016). *The Pact Smart-Contract Language* (1st ed., pp. 1-16). Kadena LLC. Retrieved from http://kadena.io/docs/Kadena-PactWhitepaper-Aug2016.pdf

Provelengios, G., Kitsos, P., Sklavos, N., & Koulamas, C. (2012). FPGA-based Design Approaches of Keccak Hash Function. *2012 15Th Euromicro Conference On Digital System Design*. http://dx.doi.org/10.1109/dsd.2012.63

Raza, S., Shafagh, H., Hewage, K., Hummen, R., & Voigt, T. (2013). Lithe: Lightweight Secure CoAP for the Internet of Things. *IEEE Sensors J.*, *13*(10), 3711-3720. http://dx.doi.org/10.1109/jsen.2013.227765

Raza, S., Trabalza, D., & Voigt, T. (2012). 6LoWPAN Compressed DTLS for CoAP. *2012 IEEE 8Th International Conference On Distributed Computing In Sensor Systems*, 287 - 289. http://dx.doi.org/10.1109/dcoss.2012.55

Rescorla, E. & Modadugu, N. (2012). Datagram Transport Layer Security Version 1.2. http://dx.doi.org/10.17487/rfc6347

Schroeder, N. (2008). Sensors as Information Transducers. *arXiv preprint arXiv:0804.0814*.

Shelby, Z., Hartke, K., & Bormann, C. (2014). The Constrained Application Protocol (CoAP). http://dx.doi.org/10.17487/rfc7252

Shirey, R. (2000). Internet Security Glossary. http://dx.doi.org/10.17487/rfc2828

Singh, R., Singh, J., & Singh, R. (2016). SECURITY CHALLENGES IN WIRELESS SENSOR NETWORKS. *International Journal Of Computer Science And Information Technology & Security*, *6*(3), 1-6. Retrieved from http://ijcsits.org/papers/vol6no32016/1vol6no3.pdf

Szabo, N. (1994). *Smart Contracts*. *Nick Szabo's Essays, Papers, and Concise Tutorials*. Retrieved 2 August 2016, from http://szabo.best.vwh.net/smart.contracts.html

Szilágyi, P. (2015). *eth/63 fast synchronization algorithm*. *GitHub*. Retrieved 6 July 2016, from https://github.com/ethereum/go-ethereum/pull/1889

Tiburski, R., Amaral, L., Matos, E., & Hessel, F. (2015). The importance of a standard security archit ecture for SOA-based iot middleware. *IEEE Commun. Mag.*, *53*(12), 20-26. http://dx.doi.org/10.1109/mcom.2015.7355580

Townend, P., Webster, D., Venters, C., Dimitrova, V., Djemame, K., Lau, L., Jie Xu, Fores, S., Viduto, V., Dibsdale, C., Taylor, N., Austin, J., Mcavoy, J. and Hobson, S.

(2013). Personalised Provenance Reasoning Models and Risk Assessment in Business Systems: A Case Study. *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*.

Trón, V. & Fikki, R. (2016). *Account Management — Ethereum Homestead 0.1 documentation.Ethdocs.org*. Retrieved 4 August 2016, from http://ethdocs.org/en/latest/account-management.html

Trusted Computing Group. (2015). *Guidance for Securing IoT Using TCG Technology* (1st ed..pp. 1-25). Trusted Computing Group. Retrieved from http://www.trustedcomputinggroup.org/wp-content/uploads/TCG_Guidance_for_Securing_IoT_1_0r21.pdf

Tsipenyuk, K., Chess, B., & McGraw, G. (2005). Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors. *IEEE Security And Privacy Magazine*, *3*(6), 81-84. http://dx.doi.org/10.1109/msp.2005.159

Vaudenay, S. (2002). Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ... In *EUROCRYPT '02 Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology* (pp. 534-546). London, UK: Springer-Verlag.

Viega, J. & Secure Software Inc. (2005). *The CLASP Application Security Process* (1st ed.). Secure Software Inc. Retrieved from https://www.ida.liu.se/~TDDC90/literature/papers/clasp_external.pdf

Walport, M. (2016). *Distributed ledger technology: beyond block chain* (pp. 17-19). London, U.K.: U.K. Government Office Science. Retrieved from

https://www.gov.uk/government/publications/ distributed-ledger-technology-blackett-review

Warren, J. (2012). *Bitmessage: A Peer-to-Peer Message Authentication and Delivery System* (1st ed., pp. 1-5). Retrieved from https://bitmessage.org/bitmessage.pdf

Wenyuan Xu, Ke Ma, Trappe, W., & Yanyong Zhang. (2006). Jamming sensor networks: attack and defense strategies. *IEEE Network, 20*(3), 41-47. http://dx.doi.org/10.1109/mnet.2006.1637931

Wood, G. (2016). *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER* (1st ed., pp. 1-32). Retrieved from http://gavwood.com/Paper.pdf

Zeng, Y., Xiang, K., & Li, D. (2012). Monitoring Technologies in Mission-Critical Environment by Using Wireless Sensor Networks. *Wireless Sensor Networks - Technology And Applications*. http://dx.doi.org/10.5772/48185

Appendix

Appendix A

Included are the various LBACS shared library interfaces utilized in C.

```
/*
 * @copyright Gilroy Gordon 2016
 * @overview LBACS C Implementation for Constrained Devices
 */

#ifndef LBACS_H_
#define LBACS_H_

#include <stdint.h>

// Intention Bits
#define LBACS_INTENTION_BIT_SIZE            0x02 //bits
#define LBACS_INTENTION_PEER_AUTHENTICATION  0x00   //0000 - binary
#define LBACS_INTENTION_GROUP_AUTHENTICATION 0x01   //0001 - binary
#define LBACS_INTENTION_KEY_REVOCATION       0x02   //0010 - binary
#define LBACS_INTENTION_KEY_RECOVERY         0x03   //0011 - binary


#define LBACS_SESSION_SIZE 12//0x7F
#define LBACS_SESSION_CHANGE LBACS_SESSION_SIZE // 127 requests / max for 8 bits
#define LBACS_MAX_REQUEST_ID (LBACS_SESSION_SIZE-1)//0x7E // 126 requests

#define LBACS_TOKEN_TYPE uint8_t*
#define LBACS_MAX_TOKEN_SIZE 8
#define LBACS_HMAC_SIZE 3
#define LBACS_HMAC_PIECES_COUNT 32
#define LBACS_HMAC_PIECE_SIZE 8

#define LBACS_MAX_PEERS 2
#define LBACS_MAX_PAIRWISE_KEY_SIZE 32

#ifndef NULL
#define NULL 0
#endif

#ifndef bool
#define bool uint8_t
#endif

#ifndef true
#define true 1
#endif
```

```
#ifndef false
#define false 0
#endif

/*
 * @function KDF
 * @description Key derivation function
 * @param seed uint8_t - PRNG seed
 * @param key uint8_t* - key array
 * @param keysize uint8_t - key array size
 * @returns uint8_t derived key
 */
typedef uint8_t* (* LBACS_KDF)(const uint8_t seed,
                        const uint8_t* key_array,
                        const uint8_t key_array_size);

typedef uint8_t (* LBACS_PRNG)(void **prngStore, const uint64_t seed);

typedef struct BIT_ {
   uint8_t value : 1;
   //char value;
} BIT;

typedef struct LBACS_SESSION_CTX_ {
   uint8_t my_request_id;
   uint8_t pairwise_key[LBACS_MAX_PAIRWISE_KEY_SIZE];
   uint8_t pairwise_key_size;
   uint8_t random_seed;
   BIT received_request_ids[LBACS_SESSION_SIZE];
} LBACS_SESSION_CTX;

typedef struct LBACS_KEYCHAIN_NODE_ {
   uint8_t peer_id;


   LBACS_SESSION_CTX session;
   //------ Functions used
   /*
    * @function PRNG
    * @param seed uint8_t - PRNG seed
    * @returns uint8_t PRNG number
    */
   LBACS_PRNG prng;
   void *prngStore;

   //struct LBACS_KEYCHAIN_NODE_ *next;
   void * next;
} LBACS_KEYCHAIN_NODE ;

typedef struct LBACS_CTX_
{
   //------ Node Specific Properties
   uint8_t node_id;
```

```
    uint8_t node_private_key[LBACS_MAX_PAIRWISE_KEY_SIZE];

    uint8_t node_private_key_size;

    uint8_t no_peers; // keychain length

    //------ Peer Related Propeties

    /*
    - keychain linkedlist
        > peer Id | pairwise key | random seed
    */
    LBACS_KEYCHAIN_NODE keychain[LBACS_MAX_PEERS];

    /*
     * @function KDF
     * @description Key derivation function
     * @param seed uint8_t - PRNG seed
     * @param key uint8_t* - key array
     * @param keysize uint8_t - key array size
     * @returns uint8_t derived key
     */
    LBACS_KDF kdf;
    //uint8_t (*KDF) (uint8_t, uint8_t*, uint8_t);
} LBACS_CTX;


//----------------------------------------
// Private Methods

//----------------------------------------
// Public Methods

static LBACS_CTX * lbacs_ctx_instance = NULL;

//creates key store context from static variable
void lbacs_init_context(
    const uint8_t node_id,
    const uint8_t* node_private_key,
    const uint8_t node_private_key_size
);

void lbacs_clear_context(void);

void lbacs_init_context_with_kdf(
    const uint8_t node_id,
    const uint8_t* node_private_key,
    const uint8_t node_private_key_size,
    const LBACS_KDF kdf
);

void lbacs_init_context_with_context(LBACS_CTX *ctx);

//gets lbacs context
LBACS_CTX * get_lbacs_ctx();
```

```
uint8_t lbacs_get_node_id();

//gets key store context from static variable
LBACS_KEYCHAIN_NODE * get_lbacs_keychain();

LBACS_KEYCHAIN_NODE * add_lbacs_peer(
    const uint8_t peer_id,
    const uint8_t* pairwise_key,
    const uint8_t pairwise_key_size,
    const uint8_t random_seed
);

LBACS_KEYCHAIN_NODE * add_lbacs_peer_with_prng(
    const uint8_t peer_id,
    const uint8_t* pairwise_key,
    const uint8_t pairwise_key_size,
    const uint8_t random_seed,
    const LBACS_PRNG prng,
    void *prngStore
);

LBACS_KEYCHAIN_NODE * get_lbacs_peer(
    const uint8_t peer_id
);

uint8_t lbacs_remove_peer_by_id(
    const uint8_t peer_id
);

//returns token/lbacs mac?
bool lbacs_authenticate(
    const uint8_t intention,
    uint8_t request_id,
    const uint8_t* message,
    const uint64_t message_size,
    const uint8_t* peer_ids,
    const uint8_t peer_id_size,
    const LBACS_TOKEN_TYPE token,
    const uint8_t token_size
);

LBACS_TOKEN_TYPE lbacs_generate_token(
    LBACS_TOKEN_TYPE token,
    const uint8_t intention,
    uint8_t* request_id,
    const uint8_t* message,
    const uint64_t message_size,
    const uint8_t* peer_ids,
    const uint8_t peer_id_size,
    uint8_t from_peer_id,
    const bool isAuthRequest
);

uint8_t* lbacs_generate_hmac(
    const uint8_t intention,
```

```
    const uint8_t request_id,
    const uint8_t* message,
    const uint64_t message_size,
    const uint8_t from_peer_id,
    const uint8_t to_peer_id,
    LBACS_KEYCHAIN_NODE * peer,
    uint8_t* hmac_size
);

bool lbacs_truncate_hmac(
    uint8_t* hmac,
    uint8_t* hmac_size,
    const uint8_t random_seed,
    const uint8_t expected_size
);

void lbacs_update_session_after_request(LBACS_KEYCHAIN_NODE * peer);
void lbacs_init_session(
    LBACS_SESSION_CTX * session,
    const uint8_t* pairwise_key,
    const uint8_t pairwise_key_size,
    const uint8_t random_seed
    );

//------------------
// Utility Functions
LBACS_CTX* loadLBACSContext(void);
int saveLBACSContext(LBACS_CTX * ctx);
int removeLBACSContextStore();
void print_lbacs_context(const LBACS_CTX * ctx);
void print_lbacs_peer(const LBACS_KEYCHAIN_NODE * peer);
//-----------
// Utility Functions Debug

#endif
```

Figure 27: LBACS C Interface for Constrained Devices

```
/*
 * @copyright Gilroy Gordon 2016
 * @overview LBACS C Implementation for Resource Competent Devices
 */

#ifndef LBACS_H_
#define LBACS_H_

#include <stdint.h>

// Intention Bits
#define LBACS_INTENTION_BIT_SIZE            0x02 //bits
#define LBACS_INTENTION_PEER_AUTHENTICATION  0x00   //0000 - binary
#define LBACS_INTENTION_GROUP_AUTHENTICATION 0x01   //0001 - binary
#define LBACS_INTENTION_KEY_REVOCATION       0x02   //0010 - binary
#define LBACS_INTENTION_KEY_RECOVERY         0x03   //0011 - binary


#define LBACS_SESSION_SIZE 12//0x7F
#define LBACS_SESSION_CHANGE LBACS_SESSION_SIZE // 127 requests / max for 8 bits
#define LBACS_MAX_REQUEST_ID (LBACS_SESSION_SIZE-1)//0x7E // 126 requests

#define LBACS_TOKEN_TYPE uint8_t*
#define LBACS_MAX_TOKEN_SIZE 8
#define LBACS_HMAC_SIZE 3
#define LBACS_HMAC_PIECES_COUNT 32
#define LBACS_HMAC_PIECE_SIZE 8

#ifndef NULL
#define NULL 0
#endif

#ifndef bool
#define bool uint8_t
#endif

#ifndef true
#define true 1
#endif

#ifndef false
#define false 0
#endif

/*
 * @function KDF
 * @description Key derivation function
 * @param seed uint8_t - PRNG seed
 * @param key uint8_t* - key array
 * @param keysize uint8_t - key array size
 * @returns uint8_t derived key
 */
typedef uint8_t* (* LBACS_KDF)(const uint8_t seed,
                        const uint8_t* key_array,
                        const uint8_t key_array_size);

typedef uint8_t (* LBACS_PRNG)(void **prngStore, const uint64_t seed);

typedef struct BIT_ {
   uint8_t value : 1;
} BIT;

typedef struct LBACS_SESSION_CTX_ {
   uint8_t my_request_id;
   uint8_t* pairwise_key;
   uint8_t pairwise_key_size;
   uint8_t random_seed;
```

```
    BIT received_request_ids[LBACS_SESSION_SIZE];
} LBACS_SESSION_CTX;

typedef struct LBACS_KEYCHAIN_NODE_ {
    uint8_t peer_id;


    LBACS_SESSION_CTX * session;
    //------ Functions used
    /*
     * @function PRNG
     * @param seed uint8_t - PRNG seed
     * @returns uint8_t PRNG number
     */
    LBACS_PRNG prng;
    void *prngStore;

    struct LBACS_KEYCHAIN_NODE_ *next;
} LBACS_KEYCHAIN_NODE ;

typedef struct LBACS_CTX_
{
    //------ Node Specific Properties
    uint8_t node_id;

    uint8_t* node_private_key;

    uint8_t node_private_key_size;

    uint8_t no_peers; // keychain length

    //------ Peer Related Propeties

    /*
    - keychain linkedlist
        > peer Id | pairwise key | random seed
    */
    LBACS_KEYCHAIN_NODE *keychain;

    /*
     * @function KDF
     * @description Key derivation function
     * @param seed uint8_t - PRNG seed
     * @param key uint8_t* - key array
     * @param keysize uint8_t - key array size
     * @returns uint8_t derived key
     */
    LBACS_KDF kdf;
    //uint8_t (*KDF) (uint8_t, uint8_t*, uint8_t);
} LBACS_CTX;


//----------------------------------------
// Private Methods

//----------------------------------------
// Public Methods

static LBACS_CTX * lbacs_ctx_instance = NULL;

//creates key store context from static variable
void lbacs_init_context(
    const uint8_t node_id,
```

```
    const uint8_t* node_private_key,
    const uint8_t node_private_key_size
);

void lbacs_clear_context(void);

void lbacs_init_context_with_kdf(
    const uint8_t node_id,
    const uint8_t* node_private_key,
    const uint8_t node_private_key_size,
    const LBACS_KDF kdf
);

//gets lbacs context
LBACS_CTX * get_lbacs_ctx();

uint8_t lbacs_get_node_id();

//gets key store context from static variable
LBACS_KEYCHAIN_NODE * get_lbacs_keychain();

LBACS_KEYCHAIN_NODE * add_lbacs_peer(
    const uint8_t peer_id,
    const uint8_t* pairwise_key,
    const uint8_t pairwise_key_size,
    const uint8_t random_seed
);

LBACS_KEYCHAIN_NODE * add_lbacs_peer_with_prng(
    const uint8_t peer_id,
    const uint8_t* pairwise_key,
    const uint8_t pairwise_key_size,
    const uint8_t random_seed,
    const LBACS_PRNG prng,
    void *prngStore
);

LBACS_KEYCHAIN_NODE * get_lbacs_peer(
    const uint8_t peer_id
);

uint8_t lbacs_remove_peer_by_id(
    const uint8_t peer_id
);

//returns token/lbacs mac?
bool lbacs_authenticate(
    const uint8_t intention,
    uint8_t request_id,
    const uint8_t* message,
    const uint64_t message_size,
    const uint8_t* peer_ids,
    const uint8_t peer_id_size,
    const LBACS_TOKEN_TYPE token,
    const uint8_t token_size
);

LBACS_TOKEN_TYPE lbacs_generate_token(
    LBACS_TOKEN_TYPE token,
    uint8_t* token_size,
    const uint8_t intention,
    uint8_t* request_id,
    const uint8_t* message,
    const uint64_t message_size,
    const uint8_t* peer_ids,
    const uint8_t peer_id_size,
    uint8_t from_peer_id,
    const bool isAuthRequest
```

```
);

uint8_t* lbacs_generate_hmac(
    uint8_t* hmac,
    uint8_t* hmac_size,
    const uint8_t intention,
    const uint8_t request_id,
    const uint8_t* message,
    const uint64_t message_size,
    const uint8_t from_peer_id,
    const uint8_t to_peer_id,
    LBACS_KEYCHAIN_NODE * peer
);

bool lbacs_truncate_hmac(
    uint8_t* hmac,
    uint8_t* hmac_size,
    const uint8_t random_seed,
    const uint8_t expected_size
);

void lbacs_update_session_after_request(LBACS_KEYCHAIN_NODE * peer);

void lbacs_init_session(
    LBACS_SESSION_CTX ** session,
    const uint8_t* pairwise_key,
    const uint8_t pairwise_key_size,
    const uint8_t random_seed
    );

//-----------------
// Utility Functions
void print_lbacs_context(const LBACS_CTX * ctx);
void print_lbacs_peer(const LBACS_KEYCHAIN_NODE * peer);
//-----------
// Utility Functions Debug

#endif
```

Figure 28: LBACS C Interface for Resource Competent Devices

Appendix B

Included are the various Solidity contracts used by LBACS to interact with the Ethereum Blockchain network.

```
pragma solidity ^0.4.0;
/*
  @author Gilroy Gordon
  @copyright Gilroy Gordon 2016. All Rights Reserved.
*/

contract ECDSASignature {
    string public signature;
    function ECDSASignature(string _signature){
        signature = _signature;
    }
}
```

Figure 29: LBACS Solidity ECDSASignature Contract

```
pragma solidity ^0.4.0;
/*
  @author Gilroy Gordon
  @copyright Gilroy Gordon 2016. All Rights Reserved.
*/


contract ECPublicKey {
    string public pubKey;
    function ECPublicKey(string _pubKey){
        pubKey = _pubKey;
    }
}
```

Figure 30: LBACS Solidity Public Key Contract

```
pragma solidity ^0.4.0;
/*
  @author Gilroy Gordon
  @copyright Gilroy Gordon 2016. All Rights Reserved.
*/


library LBACSLIB {
    enum EntityType {
        CertificateAuthority,
        TrustedAuthenticationEntity,
        SemiTrustedBaseStation,
        Sensor
    }
}
```

Figure 31: LBACS Solidity Library Contract

```
pragma solidity ^0.4.0;
/*
  @author Gilroy Gordon
  @copyright Gilroy Gordon 2016. All Rights Reserved.
*/

import "./../libraries/LBACSLIB.sol";

contract LBACSEntity {
    address public publicKey;
    address public owner;
    LBACSLIB.EntityType public entityType;

    function LBACSEntity(address pubKey,LBACSLIB.EntityType _entityType){
        owner = msg.sender;
        publicKey = pubKey;
        entityType = _entityType;
    }
}
```

Figure 32: LBACS Solidity Abstract Entity Contract

```
pragma solidity ^0.4.0;
/*
  @author Gilroy Gordon
  @copyright Gilroy Gordon 2016. All Rights Reserved.
*/

import "./LBACSEntity.sol";
import "./../libraries/LBACSLIB.sol";

contract LBACSCertAuthority is LBACSEntity {

    event CARequiresVerification(address entity);

    function LBACSCertAuthority(address pubKey)
        LBACSEntity(pubKey, LBACSLIB.EntityType.CertificateAuthority)
    {
        CARequiresVerification(this);
    }
}
```

Figure 33: LBACS Solidity Certificate Authority Contract

```
pragma solidity ^0.4.0;
/*
  @author Gilroy Gordon
  @copyright Gilroy Gordon 2016. All Rights Reserved.
*/

import "./LBACSEntity.sol";
import "./../libraries/LBACSLIB.sol";

contract LBACSTrustedAuthEntity is LBACSEntity {

    event TAERequiresVerification(address entity);

    function LBACSTrustedAuthEntity(address pubKey)
        LBACSEntity(pubKey, LBACSLIB.EntityType.TrustedAuthenticationEntity)
    {
        TAERequiresVerification(this);
    }
}
```

Figure 34: LBACS Solidity Trusted Authentication Entity Contract

```
pragma solidity ^0.4.0;
/*
  @author Gilroy Gordon
  @copyright Gilroy Gordon 2016. All Rights Reserved.
*/

import "./LBACSEntity.sol";
import "./../libraries/LBACSLIB.sol";

contract LBACSBaseStation is LBACSEntity {

    event STBSRequiresVerification(address entity);

    function LBACSBaseStation(address pubKey)
        LBACSEntity(pubKey, LBACSLIB.EntityType.SemiTrustedBaseStation)
    {
        STBSRequiresVerification(this);
    }
}
```

Figure 35: LBACS Solidity Semi-Trusted Base Station Contract

```
pragma solidity ^0.4.0;
/*
  @author Gilroy Gordon
  @copyright Gilroy Gordon 2016. All Rights Reserved.
*/

contract LBACSCertificateRegistry {


    mapping (address => bool) public certAuthorities;

    mapping (address => bool) public baseStations;

    mapping (address => bool) public authEntities;

    modifier isCertificateAuthority(address sender){
        if(!certAuthorities[msg.sender])throw;
        _;
    }

    function LBACSCertificateRegistry(){
        //assign creator as valid cert authority
        certAuthorities[msg.sender]=true;
    }

    function approveCertAuthority(address entity, bool approval)
        isCertificateAuthority(msg.sender)
    {
        certAuthorities[entity] =  approval;
    }

    function approveBaseStation(address entity, bool approval)
        isCertificateAuthority(msg.sender)
    {
        baseStations[entity] =  approval;
    }

    function approveAuthEntity(address entity, bool approval)
        isCertificateAuthority(msg.sender)
    {
        authEntities[entity] =  approval;
    }

}
```

Figure 36: LBACS Solidity Certificate Registry Contract

```
pragma solidity ^0.4.0;
/*
  @author Gilroy Gordon
  @copyright Gilroy Gordon 2016. All Rights Reserved.
*/

contract LBACSMessage {

    uint8 public fromPeerId;
    uint8 public toPeerId;
    uint8 public intention;
    uint8 public requestId;
    bytes5 public hmac;
    address public signature;
    address public authorEntity;

    function LBACSMessage(
        uint8 _fromPeerId,
        uint8 _toPeerId,
        uint8 _intention,
        uint8 _requestId,
        bytes5 _hmac,
        address _signature,
        address _authorEntity
        ){
        fromPeerId=_fromPeerId;
        toPeerId=_toPeerId;
        intention=_intention;
        requestId=_requestId;
        hmac= _hmac;
        signature= _signature;
        authorEntity = _authorEntity;
    }

}
```

Figure 37: LBACS Solidity Generic Message Contract

```
pragma solidity ^0.4.0;
/*
  @author Gilroy Gordon
  @copyright Gilroy Gordon 2016. All Rights Reserved.
*/

contract LBACSAuthenticatedData{
    //will only be verified by TAEs
    address public verifiedBy;
    address public item;
    address public submittedBy;

    function LBACSAuthenticatedData(address _item, address entity){
        item = _item;
        verifiedBy = entity;
        submittedBy = msg.sender;
    }

}
```

Figure 38: LBACS Solidity Authenticated Data Message

```
pragma solidity ^0.4.0;
/*
  @author Gilroy Gordon
  @copyright Gilroy Gordon 2016. All Rights Reserved.
*/

import "./LBACSMessage.sol";


contract BuoyData is LBACSMessage {
    int32 public x;
    int32 public y;
    int32 public z;

    event BuoyDataPublished(address bouyData, int32 x, int32 y, int32 z);

    function BuoyData(
        uint8 _fromPeerId,
        uint8 _toPeerId,
        uint8 _intention,
        uint8 _requestId,
        bytes5 _hmac,
        address _signature,
        address _authorEntity,
        int32 _x,
        int32 _y,
        int32 _z
        )
        LBACSMessage(
        _fromPeerId,
        _toPeerId,
        _intention,
        _requestId,
        _hmac,
        _signature,
        _authorEntity
        )
        {
            x = _x;
            y = _y;
            z = _z;
            BuoyDataPublished(this,x,y,z);
        }
}
```

Figure 39: LBACS Solidity Buoy Data Contract

```
pragma solidity ^0.4.0;
/*
  @author Gilroy Gordon
  @copyright Gilroy Gordon 2016. All Rights Reserved.
*/

contract LBACSRevocationMessage {
    address public entity;
    address public submittedBy;
    /*
     * Shoud be Certificate Authority*
    */
    address public revokedBy;

    function LBACSRevocationMessage(
        address _entity,
        address _revokedBy
        ){
        revokedBy = _revokedBy;
        entity = _entity;
        submittedBy = msg.sender;

    }
}
```

Figure 40: LBACS Solidity Revocation Message

Appendix C

Table 14: LBACS Token Generation Time Observations

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
1,313	1276	37
2,449	2411	38
3,601	3563	38
4,753	4715	38
5,973	5867	106
7,057	7019	38
8,209	8171	38
9,361	9323	38
10,513	10475	38
11,665	11627	38
12,817	12779	38
13,969	13931	38
15,121	15083	38
16,273	16235	38
17,425	17387	38
18,645	18539	106
19,729	19691	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
20,881	20843	38
22,033	21995	38
23,185	23147	38
24,337	24299	38
25,489	25451	38
28,116	28078	38
28,166	28128	38
28,945	28907	38
30,097	30059	38
31,317	31211	106
32,401	32363	38
33,553	33515	38
34,705	34667	38
35,857	35819	38
37,009	36971	38
38,161	38123	38
39,313	39275	38
40,465	40427	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
41,617	41579	38
42,769	42731	38
43,989	43883	106
45,073	45035	38
46,225	46187	38
47,377	47339	38
48,529	48491	38
49,681	49643	38
50,833	50795	38
51,985	51947	38
53,137	53099	38
54,289	54251	38
55,441	55403	38
56,662	56555	107
57,745	57707	38
58,897	58859	38
60,049	60011	38
61,201	61163	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
62,353	62315	38
63,505	63467	38
64,657	64619	38
65,809	65771	38
66,961	66923	38
68,113	68075	38
69,334	69227	107
70,417	70379	38
71,569	71531	38
72,721	72683	38
73,873	73835	38
75,025	74987	38
76,177	76139	38
77,329	77291	38
78,481	78443	38
79,633	79595	38
80,785	80747	38
82,006	81899	107

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
83,089	83051	38
84,241	84203	38
85,393	85355	38
86,545	86507	38
87,697	87659	38
88,849	88811	38
90,001	89963	38
91,153	91115	38
92,305	92267	38
93,457	93419	38
94,678	94571	107
95,761	95723	38
96,913	96875	38
98,065	98027	38
99,217	99179	38
100,369	100331	38
101,521	101483	38
102,673	102635	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
103,825	103787	38
104,977	104939	38
106,129	106091	38
107,350	107243	107
108,433	108395	38
109,585	109547	38
110,737	110699	38
111,889	111851	38
113,041	113003	38
114,193	114155	38
115,345	115307	38
116,497	116459	38
117,649	117611	38
118,801	118763	38
120,022	119915	107
121,105	121067	38
122,257	122219	38
123,409	123371	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
124,561	124523	38
125,713	125675	38
126,865	126827	38
128,017	127979	38
129,169	129131	38
130,321	130283	38
131,473	131435	38
132,694	132587	107
133,777	133739	38
134,929	134891	38
136,081	136043	38
137,233	137195	38
138,385	138347	38
139,537	139499	38
140,689	140651	38
141,841	141803	38
142,993	142955	38
144,145	144107	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
145,366	145259	107
146,449	146411	38
147,601	147563	38
148,753	148715	38
149,905	149867	38
151,057	151019	38
152,209	152171	38
153,361	153323	38
154,513	154475	38
155,665	155627	38
156,817	156779	38
158,038	157931	107
159,121	159083	38
160,273	160235	38
161,425	161387	38
162,577	162539	38
163,729	163691	38
164,881	164843	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
166,033	165995	38
167,185	167147	38
168,337	168299	38
169,489	169451	38
170,710	170603	107
171,793	171755	38
172,945	172907	38
174,097	174059	38
175,249	175211	38
176,401	176363	38
177,553	177515	38
178,705	178667	38
179,857	179819	38
181,009	180971	38
182,161	182123	38
183,382	183275	107
184,465	184427	38
185,617	185579	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
186,769	186731	38
187,921	187883	38
189,073	189035	38
190,225	190187	38
191,377	191339	38
192,529	192491	38
193,681	193643	38
194,833	194795	38
196,054	195947	107
197,137	197099	38
198,289	198251	38
199,441	199403	38
200,593	200555	38
201,745	201707	38
202,897	202859	38
204,049	204011	38
205,201	205163	38
206,353	206315	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
207,505	207467	38
208,726	208619	107
209,809	209771	38
210,961	210923	38
212,113	212075	38
213,265	213227	38
214,417	214379	38
215,569	215531	38
216,721	216683	38
217,873	217835	38
219,025	218987	38
220,177	220139	38
221,398	221291	107
222,481	222443	38
223,633	223595	38
224,785	224747	38
225,937	225899	38
227,089	227051	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
228,241	228203	38
229,393	229355	38
230,545	230507	38
231,697	231659	38
232,849	232811	38
234,070	233963	107
235,153	235115	38
236,305	236267	38
237,457	237419	38
238,609	238571	38
239,761	239723	38
240,913	240875	38
242,065	242027	38
243,217	243179	38
244,369	244331	38
245,521	245483	38
246,742	246635	107
247,825	247787	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
248,977	248939	38
250,129	250091	38
251,281	251243	38
252,433	252395	38
253,585	253547	38
254,737	254699	38
255,889	255851	38
257,041	257003	38
258,193	258155	38
259,414	259307	107
260,497	260459	38
261,649	261611	38
262,801	262763	38
263,953	263915	38
265,105	265067	38
266,257	266219	38
267,409	267371	38
268,561	268523	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
269,713	269675	38
270,865	270827	38
272,086	271979	107
273,169	273131	38
274,321	274283	38
275,473	275435	38
276,625	276587	38
277,777	277739	38
278,929	278891	38
280,081	280043	38
281,233	281195	38
282,385	282347	38
283,537	283499	38
284,758	284651	107
285,841	285803	38
286,993	286955	38
288,145	288107	38
289,297	289259	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
290,449	290411	38
291,601	291563	38
292,753	292715	38
293,905	293867	38
295,057	295019	38
296,209	296171	38
297,430	297323	107
298,513	298475	38
299,665	299627	38
300,817	300779	38
301,969	301931	38
303,121	303083	38
304,273	304235	38
305,425	305387	38
306,577	306539	38
307,729	307691	38
308,881	308843	38
310,102	309995	107

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
311,185	311147	38
312,337	312299	38
313,489	313451	38
314,641	314603	38
315,793	315755	38
316,945	316907	38
318,097	318059	38
319,249	319211	38
320,401	320363	38
321,553	321515	38
322,774	322667	107
323,857	323819	38
325,009	324971	38
326,161	326123	38
327,313	327275	38
328,465	328427	38
329,617	329579	38
330,769	330731	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
331,921	331883	38
333,073	333035	38
334,225	334187	38
335,446	335339	107
336,529	336491	38
337,681	337643	38
338,833	338795	38
339,985	339947	38
341,137	341099	38
342,289	342251	38
343,441	343403	38
344,593	344555	38
345,745	345707	38
346,897	346859	38
348,118	348011	107
349,201	349163	38
350,353	350315	38
351,505	351467	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
352,657	352619	38
353,809	353771	38
354,961	354923	38
356,113	356075	38
357,265	357227	38
358,417	358379	38
359,569	359531	38
360,790	360683	107
361,873	361835	38
363,025	362987	38
364,177	364139	38
365,329	365291	38
366,481	366443	38
367,633	367595	38
368,785	368747	38
369,937	369899	38
371,089	371051	38
372,241	372203	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
373,462	373355	107
374,545	374507	38
375,697	375659	38
376,849	376811	38
378,001	377963	38
379,153	379115	38
380,305	380267	38
381,457	381419	38
382,609	382571	38
383,761	383723	38
384,913	384875	38
386,134	386027	107
387,217	387179	38
388,369	388331	38
389,521	389483	38
390,673	390635	38
391,825	391787	38
392,977	392939	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
394,129	394091	38
395,281	395243	38
396,433	396395	38
397,585	397547	38
398,806	398699	107
399,889	399851	38
401,041	401003	38
402,193	402155	38
403,345	403307	38
404,497	404459	38
405,649	405611	38
406,801	406763	38
407,953	407915	38
409,105	409067	38
410,257	410219	38
411,478	411371	107
412,561	412523	38
413,713	413675	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
414,865	414827	38
416,017	415979	38
417,169	417131	38
418,321	418283	38
419,473	419435	38
420,625	420587	38
421,777	421739	38
422,929	422891	38
424,150	424043	107
425,233	425195	38
426,385	426347	38
427,537	427499	38
428,689	428651	38
429,841	429803	38
430,993	430955	38
432,145	432107	38
433,297	433259	38
434,449	434411	38

Clock Time After (ms)	Clock Time Before (ms)	Difference (ms)
435,601	435563	38
436,822	436715	107
437,905	437867	38
439,057	439019	38
440,209	440171	38
441,361	441323	38
442,513	442475	38
443,665	443627	38
444,817	444779	38
445,969	445931	38

Appendix D

Table 15: Z1 mote metrics without LBACS

observation	clock_time	cpu	lpm	transmit	listen	Power (mW)
0	1384	2192	325508	69	137	0.012041016
1	2664	4247	323435	283	327398	0.023329468
2	3944	3742	323935	275	327402	0.02055542
3	5224	3786	323892	284	327394	0.020797119
4	6504	3214	324464	116	327563	0.017655029
5	7784	3629	324050	116	327564	0.019934692
6	9064	3581	324097	283	327393	0.019671021
7	10344	3225	324453	116	327563	0.017715454
8	11627	4059	324459	233	328285	0.022296753
9	12904	3250	323589	116	326721	0.017852783
10	14184	3394	324285	219	327460	0.018643799
11	15464	3363	324315	116	327562	0.018473511
12	16744	3214	324464	116	327562	0.017655029
13	18024	3912	323767	288	327390	0.021489258
14	19304	3226	324452	116	327561	0.017720947
15	20584	3210	324469	116	327562	0.017633057
16	21864	3219	324460	116	327562	0.017682495
17	23147	4202	324319	232	328287	0.023082275

observation	clock_time	cpu	lpm	transmit	listen	Power (mW)
18	24424	3253	323584	116	326719	0.017869263
19	25704	3220	324459	116	327562	0.017687988
20	26984	3401	324278	219	327460	0.018682251
21	28264	3214	324464	117	327561	0.017655029
22	29544	3492	324187	224	327454	0.019182129
23	30824	3362	324316	116	327562	0.018468018
24	32104	3216	324462	117	327561	0.017666016
25	33384	3710	323969	181	327497	0.020379639
26	34667	4064	324455	233	328285	0.022324219
27	35944	3279	323560	116	326722	0.018012085
28	37224	3221	324457	116	327562	0.017693481
29	38504	3371	324308	116	327563	0.018517456
30	39784	3497	324181	224	327454	0.019209595
31	41064	3238	324440	116	327561	0.017786865
32	42344	3247	324432	116	327562	0.017836304
33	43624	3234	324445	116	327562	0.017764893
34	44904	3243	324436	116	327562	0.017814331
35	46187	4225	324296	232	328287	0.023208618
36	47464	3277	323560	116	326720	0.018001099

observation	clock_time	cpu	lpm	transmit	listen	Power (mW)
37	48744	3244	324435	116	327561	0.017819824
38	50024	3247	324432	116	327562	0.017836304
39	51304	3234	324445	116	327562	0.017764893
40	52584	3663	324016	181	327497	0.02012146
41	53864	3383	324296	116	327563	0.018583374
42	55144	3237	324441	116	327561	0.017781372
43	56424	3516	324163	224	327454	0.019313965
44	57707	4088	324430	233	328284	0.022456055
45	58984	3276	323563	116	326722	0.017995605
46	60264	3243	324436	116	327561	0.017814331
47	61544	3383	324296	116	327562	0.018583374
48	62824	3236	324442	116	327561	0.017775879

Table 16: Z1 Mote metrics with LBACS

observation	clock_time	cpu	lpm	transmit	listen	Power (mW)
0	1384	21419	306275	68	137	0.117658081
1	2664	13760	313922	284	327398	0.075585938
2	3944	31606	296072	739	326935	0.173616943
3	5224	13104	314575	219	327460	0.071982422
4	6504	18938	308739	180	327496	0.104029541

observation	clock_time	cpu	lpm	transmit	listen	Power (mW)
5	7784	13098	314581	116	327564	0.071949463
6	9064	13073	314604	283	327394	0.071812134
7	10344	12717	314961	117	327562	0.069856567
8	11664	29093	308928	232	337788	0.159812622
9	12904	12719	304616	117	317219	0.069867554
10	14184	12608	315071	0	327679	0.069257813
11	15464	19127	308553	219	327459	0.105067749
12	16744	31274	296404	573	327102	0.171793213
13	18024	12714	314964	116	327562	0.069840088
14	19304	12737	314942	116	327562	0.069966431
15	20584	18799	308879	116	327562	0.103265991
16	21864	12716	314963	117	327562	0.069851074
17	23184	23197	314825	234	337788	0.127424927
18	24424	18848	298487	117	317219	0.103535156
19	25704	12714	314965	116	327562	0.069840088
20	26984	12980	314699	224	327455	0.07130127
21	28264	12705	314973	117	327562	0.069790649
22	29544	37149	290530	464	327213	0.204065552
23	30824	12849	314830	116	327563	0.070581665

observation	clock_time	cpu	lpm	transmit	listen	Power (mW)
24	32104	12707	314971	117	327562	0.069801636
25	33384	12707	314972	116	327563	0.069801636
26	34704	29397	308625	337	337685	0.161482544
27	35944	12755	304580	116	317219	0.070065308
28	37224	12984	314694	224	327453	0.071323242
29	38504	19029	308649	117	327562	0.104529419
30	39784	12705	314973	116	327562	0.069790649
31	41064	12719	314960	117	327562	0.069867554
32	42344	30924	296755	349	327328	0.169870605
33	43624	18923	308756	117	327562	0.103947144
34	44904	12726	314953	117	327562	0.069906006
35	46224	23214	314808	234	337788	0.127518311
36	47464	18959	298376	117	317219	0.104144897
37	48744	12724	314955	116	327563	0.06989502
38	50024	12726	314953	117	327562	0.069906006
39	51304	12722	314957	116	327562	0.069884033
40	52584	18971	308708	116	327562	0.104210815
41	53864	12871	314808	116	327562	0.070702515
42	55144	31290	296388	572	327103	0.171881104

observation	clock_time	cpu	lpm	transmit	listen	Power (mW)
43	56424	12727	314951	117	327562	0.069911499
44	57744	29337	308685	233	337788	0.161152954
45	58984	12738	304597	117	317219	0.069971924
46	60264	12727	314952	117	327562	0.069911499
47	61544	19157	308522	116	327562	0.105232544
48	62824	12716	314962	117	327562	0.069851074

Appendix E

The TCP dump showing network packets (between Zolertia Z1 mote and Raspeberry PI Semi-Trusted Base Station) acquired from the Raspberry PI's slip (USB) connection to the router has also been included below.

```
13:37:00.923576 IP6 133:33:1033:1033:c30c::66.51 > 133:33:1033:1033:1633:1633:1633:1633.5683: UDP, length 45
        0x0000:  6000 0000 0035 1115 0133 0033 1033 1033  `....5...3.3.3.3
        0x0010:  c30c 0000 0000 0066 0133 0033 1033 1033  .......f.3.3.3.3
        0x0020:  1633 1633 1633 1633 0033 1633 0035 ae78  .3.3.3.3.3.3.5.x
        0x0030:  4802 0008 0308 4eb4 978e 8f48 b361 7069  H.....N....H.api
        0x0040:  0462 756f 79ff 7b22 7822 3a2d 372c 2279  .buoy.{"x":-7,"y
        0x0050:  223a 352c 227a 223a 2d32 3530 7d         ":5,"z":-250}
13:37:02.145325 IP6 133:33:1033:1033:c30c::66 > 133:33:1033:1033:1633:1633:1633:1633: ICMP6, destination
unreachable, unreachable port, 133:33:1033:1033:c30c::66 udp port 51, length 97
        0x0000:  6000 0000 0061 3a3f 0133 0033 1033 1033  `....a:?.3.3.3.3
        0x0010:  c30c 0000 0000 0066 0133 0033 1033 1033  .......f.3.3.3.3
        0x0020:  1633 1633 1633 1633 0104 2d5b 0000 0000  .3.3.3.3..-[....
        0x0030:  6000 0000 0031 113f 0133 0033 1033 1033  `....1.?.3.3.3.3
        0x0040:  1633 1633 1633 1633 0133 0033 1033 1033  .3.3.3.3.3.3.3.3
        0x0050:  c30c 0000 0000 0066 1633 0033 0031 6e67  .......f.3.3.1ng
        0x0060:  6845 0008 0308 4eb4 978e 8f48 ff43 6f6e  hE....N....H.Con
        0x0070:  6669 726d 6564 2042 756f 7920 4461 7461  firmed.Buoy.Data
        0x0080:  2052 6563 6569 7665 64                    .Received
```

Appendix F

LBACS sensor output for Z1 mote has been included below.

```
Message : {"x":-6,"y":6,"z":-251}
--Requesting api/buoy--

LBACS Context is not NULL. No of peers : 2. Size : 180
LBACS Generating Token. Is Auth Request N from peer 102 with peer count 2
Checking if peer 50 is LBACS Peer
Found Peer With Id 50
About to update next request id 7
Updating next request id to 8. Session Request Id 8
Allocating memory to token
Assigned 16 bytes to token
Intention bit assigned 3
Getting first peer
Intention(3) requestId(8) from_peer_id(102) to_peer_id(50) message(
7b2278223a2d362c2279223a362c227a223a2d3235317d) pairwise_key(328ce2e8574)
Hex Hash : cbbe604358eacaf065e6e4c627b5203194bc39d159cff9c82833a370f37d27
Truncation | hmac_size L 32
here 1 = expected size : 3 | random_seed : 91 | hmac_size : 32
reducedBy : 0
reducedBy : 19
reducedBy : 6
Checking LBACS Session update for peer id : 50
Peer(50) Session does not need to be updated :
Assigning c @ token[2] to hmac1 generated using from_peer_id(102) to peer_id(50) with request_id(8)
Assigning 39 @ token[3] to hmac1 generated using from_peer_id(102) to peer_id(50) with request_id(8)
Assigning ea @ token[4] to hmac1 generated using from_peer_id(102) to peer_id(50) with request_id(8)
Generated and assgned hmac1
Checking if peer 51 is LBACS Peer
Skipping LBACS Peer With Id 50 to get to end of keychain
Found Peer With Id 51
Intention(3) requestId(8) from_peer_id(102) to_peer_id(51) message(
7b2278223a2d362c2279223a362c227a223a2d3235317d) pairwise_key(776f6e646572)
Hex Hash : 7597e694e7eec8271ab259134a824cf4e9dd7bc44ba85890663ac7cb688f0
Truncation | hmac_size L 32
here 1 = expected size : 3 | random_seed : 12 | hmac_size : 32
reducedBy : 0
reducedBy : 12
reducedBy : 24
Checking LBACS Session update for peer id : 51
Peer(51) Session does not need to be updated :
Assigning 75 @ token[5] to hmac2 using  from_peer_id(102) to peer_id(51) with request_id(8)
Assigning 13 @ token[6] to hmac2 using  from_peer_id(102) to peer_id(51) with request_id(8)
Assigning 90 @ token[7] to hmac2 using  from_peer_id(102) to peer_id(51) with request_id(8)
Generated and assgned hmac2
Current Token in Hex :  3 -  8 -  c -  39 -  ea -  75 -  13 -  90 -
Token Address : 0x30b4
Setting token
Token Set (8) bytes. Token Hex : 38c39ea751390
```

Appendix G

Output from Semi-Trusted Base Station running on Raspberry PI.

```
Received Request :  { method: 'POST',
  coapCode: '0.02',
  url: '/api/buoy',
  rsinfo:
   { address: '133:33:1033:1033:c30c::66',
     family: 'IPv6',
     port: 51,
     size: 45 },
  options:
   [ { name: 'Uri-Path', value: <Buffer 61 70 69> },
     { name: 'Uri-Path', value: <Buffer 62 75 6f 79> } ],
  payload: '{"x":-7,"y":5,"z":-250}',
  token: '\u0003\bN����H',
  headers: {} }
[34mdebug: [39mRouting Request [POST] -api/buoy
[34mdebug: [39mraw data {"x":-7,"y":5,"z":-250}
[34mdebug: [39mraw data after {"x":-7,"y":5,"z":-250}
[34mdebug: [39mtoken undefined
[34mdebug: [39mtoken hex 03084eb4978e8f48
[34mdebug: [39mhexToBytes 03084eb4978e8f48
[34mdebug: [39mhexToBytes start 03084eb4978e8f48
[34mdebug: [39mhexToBytes end [ '3', '8', '4E', 'B4', '97', '8E', '8F', '48' ]
[34mdebug: [39mProcessing message id : 03084eb4978e8f48-102-50-{"x":-7,"y":5,"z":-250}
bytesToHex start [ '3', '8', '4E', 'B4', '97', '8E', '8F', '48' ]
bytesToHex end 03084eb4978e8f48
[34mdebug: [39mauth { message: '{"x":-7,"y":5,"z":-250}',
  intention: 3,
  requestId: 8,
  token: [ 3, 8, 78, 180, 151, 142, 143, 72 ],
  hmac: '03084eb4978e8f48',
  peerIds: [ 50 ],
  fromPeerId: 102,
  toPeerId: 51,
  x: -7,
  y: 5,
  z: -250,
  status: 'Pending' }
hexToBytes start 03084eb497
hexToBytes end [ '3', '8', '4E', 'B4', '97' ]
Authenticating request id 8 fom 2 peers
Checking if peer with id 102 exists
Checking if peer 102 is LBACS Peer
Found Peer With Id 102
checking for replay attacks
Request id is a valid request id
Checking if peer with id 50 exists
Skipping id 50 since this is node id
LBACS Generating Token. Is Auth Request Y from peer 102 with peer count 1
Generating token from Peer (102) to Peer Ids : [ 50 ,]
Checking if peer 102 is LBACS Peer
Found Peer With Id 102
Allocating memory to token
Assigned 8 bytes to token
Intention bit assigned 3
Getting first peer
Intention(3) requestId(8) from_peer_id(102) to_peer_id(50) message(
7b2278223a2d372c2279223a352c227a223a2d3235307d) pairwise_key(6d617276656c)
Hex Hash : 4ed89757de7d23f25de8222da494e48d8b45392168d932084ec652d49b2784e
Truncation | hmac_size L 32
```

```
here 1 = expected size : 3 | random_seed : 9 | hmac_size : 32
reducedBy : 0
reducedBy : 17
reducedBy : 2
Checking LBACS Session update for peer id : 102
Peer(102) Session does not need to be updated :
Assigning 4e @ token[2] to hmac1 generated using from_peer_id(102) to peer_id(50) with request_id(8)
Assigning b4 @ token[3] to hmac1 generated using from_peer_id(102) to peer_id(50) with request_id(8)
Assigning 97 @ token[4] to hmac1 generated using from_peer_id(102) to peer_id(50) with request_id(8)
Generated and assgned hmac1
Current Token in Hex :  3 -  8 -  4e -  b4 -  97 -  0 -  0 -  0 -
Token Address : 0x2b55c40
Marking request id  8 as received for peer 102
Checking LBACS Session update for peer id : 102
Peer(102) Session does not need to be updated :
authentication successful
[34mdebug: [39mPairwise Auth Successful
[34mdebug: [39mretrieved entity { publicKey:
'04f1412dad768cbf62374430e463ad11485bf3bb0a3cae9a580564edb4541ef64c9388134e5e726d6d747d98d8c2dd99413dece0d
99d40601246053f0b882ed6f5',
  pubKey: '0x698806fdc850b0316aae21d2d6c1fe14859fa6fc',
  address: '0x57a2bb7f4781102460f98776672a8383328814bc',
  entityType: 2 }
bytesToHex start [ 3, 8, 142, 143, 72 ]
bytesToHex end 030814214372
pairwise signature included, complete params { intention: 3,
  message: '{"x":-7,"y":5,"z":-250}',
  toPeerId: 51,
  peerIds: [ 50 ],
  fromPeerId: 102,
  hmac: '030814214372',
  requestId: 8,
  hmacBytes: '030814214372' }
data message to be signed {"intention":3,"requestId":8,"fromPeerId":102,"toPeerId":51,"message":"{\"x\":-7,\"y\":5,\"z\":-
250}","hmac":"030814214372"}
Current Gas Cost :  4712377
new
signature(30450220651d689508acbec0c52d23e1d697f63af0f067b653ca4b5efa745f77467b0baf022100dd86e00fb1a5ede289e0d
8ca8a0336bf8e5339a84ed35441facd3eb89572922f) created at address 0xe0eae27e64dab0e7e5730781cfcf2a2892f7dea4
signature with address result { requestId: 8,
  hmac: '030814214372',
  hmacBytes: '030814214372',
  signature:
'30450220651d689508acbec0c52d23e1d697f63af0f067b653ca4b5efa745f77467b0baf022100dd86e00fb1a5ede289e0d8ca8a033
6bf8e5339a84ed35441facd3eb89572922f',
  signatureAddress: '0xe0eae27e64dab0e7e5730781cfcf2a2892f7dea4' }
[34mdebug: [39mbuoyData created at  0x2251e2e51123db527e0fb3a92e63b501308fd4b1
```

Appendix H

Output from Trusted Authentication Entity running on Mac OS X has been included below.

```
rr { address: '0x2251e2e51123db527e0fb3a92e63b501308fd4b1',
  blockHash: '0x0d12f060042e491783f363c475de7baa33465e5128d6f5d2e37b0ba361d58be6',
  blockNumber: 3884,
  data:
'0x000000000000000000000000002251e2e51123db527e0fb3a92e63b501308fd4b1ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffff8000000000000000000000000000000000000000000000000000000000000007ffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffff04',
  logIndex: 0,
  topics: [ '0x250749d98ef32bb38fd1f63d529c2947cecd0039e0a06285800b7abfd66dab08' ],
  transactionHash: '0x6d3bcde971d9e1126b9581079c138bb3ff667a950c070b7ece5309bc86bf1aa3',
  transactionIndex: 0 }
BuoyDataPublishedListener logs [ { address: '0x2251e2e51123db527e0fb3a92e63b501308fd4b1',
    blockHash: '0x0d12f060042e491783f363c475de7baa33465e5128d6f5d2e37b0ba361d58be6',
    blockNumber: 3884,
    logIndex: 0,
    transactionHash: '0x6d3bcde971d9e1126b9581079c138bb3ff667a950c070b7ece5309bc86bf1aa3',
    transactionIndex: 0,
    event: 'BuoyDataPublished',
    args:
     { bouyData: '0x2251e2e51123db527e0fb3a92e63b501308fd4b1',
       x: [Object],
       y: [Object],
       z: [Object] } } ]
Processing buoy data at 0x2251e2e51123db527e0fb3a92e63b501308fd4b1
Processing message id : 0x2251e2e51123db527e0fb3a92e63b501308fd4b1
Current Gas Cost :  4712377
Current Gas Cost :  4712377
saved event { logIndex: 0,
  transactionIndex: 0,
  transactionHash: '0x6d3bcde971d9e1126b9581079c138bb3ff667a950c070b7ece5309bc86bf1aa3',
  blockHash: '0x0d12f060042e491783f363c475de7baa33465e5128d6f5d2e37b0ba361d58be6',
  blockNumber: 3884,
  address: '0x2251e2e51123db527e0fb3a92e63b501308fd4b1',
  type: null,
  event: 'BuoyDataPublished',
  args:
   { bouyData: '0x2251e2e51123db527e0fb3a92e63b501308fd4b1',
     x: { s: -1, e: 0, c: [Object] },
     y: { s: 1, e: 0, c: [Object] },
     z: { s: -1, e: 2, c: [Object] } },
  id: 104,
  createdAt: '2016-12-14T17:38:32.652Z',
  updatedAt: '2016-12-14T17:38:32.652Z' }
Current Gas Cost :  4712377
retrieved
signature(30450220651d689508acbec0c52d23e1d697f63af0f067b653ca4b5efa745f77467b0baf022100dd86e00fb1a5ede289e
0d8ca8a0336bf8e5339a84ed35441facd3eb89572922f) at address 0xe0eae27e64dab0e7e5730781cfcf2a2892f7dea4
LBACSAuthService:: Authenticating: { signature:
'30450220651d689508acbec0c52d23e1d697f63af0f067b653ca4b5efa745f77467b0baf022100dd86e00fb1a5ede289e0d8ca8a0
336bf8e5339a84ed35441facd3eb89572922f',
  entityAddress: '0x57a2bb7f4781102460f98776672a8383328814bc',
  intention: 3,
  requestId: 8,
  hmac: '0x03088e8f48',
  fromPeerId: 102,
```

```
  toPeerId: 51,
  message: '{"x":-7,"y":5,"z":-250}' }
Current Gas Cost :  4712377
Verifying BaseStation in Registry On Chain using :  0x57a2bb7f4781102460f98776672a8383328814bc
Current Gas Cost :  4712377
BaseStation(0x57a2bb7f4781102460f98776672a8383328814bc) approval :  true
Retrieveing public key of entity :  0x57a2bb7f4781102460f98776672a8383328814bc
Retrieved public key address(0x698806fdc850b0316aae21d2d6c1fe14859fa6fc) of
entity(0x57a2bb7f4781102460f98776672a8383328814bc)
retrieved public
key(04f1412dad768cbf62374430e463ad11485bf3bb0a3cae9a580564edb4541ef64c9388134e5e726d6d747d98d8c2dd99413dec
e0d99d40601246053f0b882ed6f5) at address 0x698806fdc850b0316aae21d2d6c1fe14859fa6fc

LBACSAuthService:: Received public key:
04f1412dad768cbf62374430e463ad11485bf3bb0a3cae9a580564edb4541ef64c9388134e5e726d6d747d98d8c2dd99413dece0d
99d40601246053f0b882ed6f5
data message to be signed {"intention":3,"requestId":8,"fromPeerId":102,"toPeerId":51,"message":"{\"x\":-7,\"y\":5,\"z\":-
250}","hmac":"0x03088e8f48"}
attempting to verify message with public key
04f1412dad768cbf62374430e463ad11485bf3bb0a3cae9a580564edb4541ef64c9388134e5e726d6d747d98d8c2dd99413dece0d
99d40601246053f0b882ed6f5
attempting to verify message with signature
30450220651d689508acbec0c52d23e1d697f63af0f067b653ca4b5efa745f77467b0baf022100dd86e00fb1a5ede289e0d8ca8a03
36bf8e5339a84ed35441facd3eb89572922f
key from public key <Key priv: null pub: <EC Point x:
f1412dad768cbf62374430e463ad11485bf3bb0a3cae9a580564edb4541ef64c y:
9388134e5e726d6d747d98d8c2dd99413dece0d99d40601246053f0b882ed6f5> >
verifyMessage result true
LBACSAuthService:: Verified signaure:
04f1412dad768cbf62374430e463ad11485bf3bb0a3cae9a580564edb4541ef64c9388134e5e726d6d747d98d8c2dd99413dece0d
99d40601246053f0b882ed6f5
hexToBytes start 03088e8f48
hexToBytes end [ '3', '8', '8E', '8F', '48' ]
Authenticating pairwise params { message: '{"x":-7,"y":5,"z":-250}',
  intention: 3,
  requestId: 8,
  token: [ '3', '8', '8E', '8F', '48' ],
  peerIds: [ 51 ],
  fromPeerId: 102 }
LBACSAuthService:: Authenticated Pair Message:
04f1412dad768cbf62374430e463ad11485bf3bb0a3cae9a580564edb4541ef64c9388134e5e726d6d747d98d8c2dd99413dece0d
99d40601246053f0b882ed6f5
Retrieved Author Entity and Created Signature for Approved Message :
304602210080cdd5f683c7367b757b28c5888d6e8d0c3dfd56f875f59607da1d373f39521c022100e7a42b4eb0d7df8a7246ebba0
831f5e14541435fe60d13ee5d6cf65ff01dc745
Current Gas Cost :  4712377

new
signature(30450220487aaddc02bba75ab9e9de67b0b6e7e572c42e76e26c14fc8c07f79cfd3c4095022100f6938673ad84b64d13d
594c2d9eabde9b06e4721911e4250bf69669dbcc8f6d6) created at address 0x94b6bb02a1e4ecfab77679212cf43525c54e16cd
```

Appendix I

The modified SHA3IUF implementation used as the KDF, PRNG and HMAC has been included below.

```
 /* --------------------------------------------------------------------
* Works when compiled for either 32-bit or 64-bit targets, optimized for
* 64 bit.
*
* Canonical implementation of Init/Update/Finalize for SHA-3 byte input.
*
* SHA3-256, SHA3-384, SHA-512 are implemented. SHA-224 can easily be added.
*
* Based on code from http://keccak.noekeon.org/ .
*
* I place the code that I wrote into public domain, free to use.
*
* I would appreciate if you give credits to this work if you used it to
* write or test * your code.
*
* Aug 2015. Andrey Jivsov. crypto@brainhub.org
* Oct 2016  Modified by Gilroy Gordon. gilroygordon@gmail.com
* ---------------------------------------------------------------------- */

#ifndef KECCAK_SHA_3_H_
#define KECCAK_SHA_3_H_

#include <stdint.h>


#define SHA3_ASSERT( x )
#if defined(_MSC_VER)
#define SHA3_TRACE( format, ...)
#define SHA3_TRACE_BUF( format, buf, l, ...)
#else
#define SHA3_TRACE(format, args...)
#define SHA3_TRACE_BUF(format, buf, l, args...)
#endif

#define SHA3_USE_KECCAK 1
/*
 * Define SHA3_USE_KECCAK to run "pure" Keccak, as opposed to SHA3.
 * The tests that this macro enables use the input and output from [Keccak]
 * (see the reference below). The used test vectors aren't correct for SHA3,
 * however, they are helpful to verify the implementation.
 * SHA3_USE_KECCAK only changes one line of code in Finalize.
 */

#if defined(_MSC_VER)
#define SHA3_CONST(x) x
#else
#define SHA3_CONST(x) x##L
#endif

/* The following state definition should normally be in a separate
 * header file
 */

/* 'Words' here refers to uint64_t */
#define SHA3_KECCAK_SPONGE_WORDS \
```

```
            (((1600)/8/*bits to byte*/)/sizeof(uint64_t))
typedef struct sha3_context_ {
    uint64_t saved;             /* the portion of the input message that we
                        * didn't consume yet */
    union {                     /* Keccak's state */
        uint64_t s[SHA3_KECCAK_SPONGE_WORDS];
        uint8_t sb[SHA3_KECCAK_SPONGE_WORDS * 8];
    };
    uint32_t byteIndex;         /* 0..7--the next byte after the set one
                        * (starts from 0; 0--none are buffered) */
    uint32_t wordIndex;         /* 0..24--the next word to integrate input
                        * (starts from 0) */
    uint32_t capacityWords;     /* the double size of the hash output in
                        * words (e.g. 16 for Keccak 512) */
} sha3_context;

#ifndef SHA3_ROTL64
#define SHA3_ROTL64(x, y) \
            (((x) << (y)) | ((x) >> ((sizeof(uint64_t)*8) - (y))))
#endif

static const uint64_t keccakf_rndc[24] = {
    SHA3_CONST(0x0000000000000001UL), SHA3_CONST(0x0000000000008082UL),
    SHA3_CONST(0x800000000000808aUL), SHA3_CONST(0x8000000080008000UL),
    SHA3_CONST(0x000000000000808bUL), SHA3_CONST(0x0000000080000001UL),
    SHA3_CONST(0x8000000080008081UL), SHA3_CONST(0x8000000000008009UL),
    SHA3_CONST(0x000000000000008aUL), SHA3_CONST(0x0000000000000088UL),
    SHA3_CONST(0x0000000080008009UL), SHA3_CONST(0x000000008000000aUL),
    SHA3_CONST(0x000000008000808bUL), SHA3_CONST(0x800000000000008bUL),
    SHA3_CONST(0x8000000000008089UL), SHA3_CONST(0x8000000000008003UL),
    SHA3_CONST(0x8000000000008002UL), SHA3_CONST(0x8000000000000080UL),
    SHA3_CONST(0x000000000000800aUL), SHA3_CONST(0x800000008000000aUL),
    SHA3_CONST(0x8000000080008081UL), SHA3_CONST(0x8000000000008080UL),
    SHA3_CONST(0x0000000080000001UL), SHA3_CONST(0x8000000080008008UL)
};

static const uint32_t keccakf_rotc[24] = {
    1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 2, 14, 27, 41, 56, 8, 25, 43, 62,
    18, 39, 61, 20, 44
};

static const uint32_t keccakf_piln[24] = {
    10, 7, 11, 17, 18, 3, 5, 16, 8, 21, 24, 4, 15, 23, 19, 13, 12, 2, 20,
    14, 22, 9, 6, 1
};

/* For Init or Reset call these: */
void sha3_Init256(void *priv);

void sha3_Update(void *priv, void const *bufIn, size_t len);

/* This is simply the 'update' with the padding block.
 * The padding block is 0x01 || 0x00* || 0x80. First 0x01 and last 0x80
 * bytes are always present, but they can be the same byte.
 */
void const * sha3_Finalize(void *priv);

#endif

#ifndef SHA3_USE_KECCAK
#define SHA3_USE_KECCAK 1
#endif


/* generally called after SHA3_KECCAK_SPONGE_WORDS-ctx->capacityWords words
```

```
 * are XORed into the state s
 */
static void
keccakf(uint64_t s[25])
{
   int i, j, round;
   uint64_t t, bc[5];
#define KECCAK_ROUNDS 24

   for(round = 0; round < KECCAK_ROUNDS; round++) {

      /* Theta */
      for(i = 0; i < 5; i++)
         bc[i] = s[i] ^ s[i + 5] ^ s[i + 10] ^ s[i + 15] ^ s[i + 20];

      for(i = 0; i < 5; i++) {
         t = bc[(i + 4) % 5] ^ SHA3_ROTL64(bc[(i + 1) % 5], 1);
         for(j = 0; j < 25; j += 5)
            s[j + i] ^= t;
      }

      /* Rho Pi */
      t = s[1];
      for(i = 0; i < 24; i++) {
         j = keccakf_piln[i];
         bc[0] = s[j];
         s[j] = SHA3_ROTL64(t, keccakf_rotc[i]);
         t = bc[0];
      }

      /* Chi */
      for(j = 0; j < 25; j += 5) {
         for(i = 0; i < 5; i++)
            bc[i] = s[j + i];
         for(i = 0; i < 5; i++)
            s[j + i] ^= (~bc[(i + 1) % 5]) & bc[(i + 2) % 5];
      }

      /* Iota */
      s[0] ^= keccakf_rndc[round];
   }
}

/* ******************************** Public Inteface ********************************* */

/* For Init or Reset call these: */
void
sha3_Init256(void *priv)
{
   sha3_context *ctx = (sha3_context *) priv;
   memset(ctx, 0, sizeof(*ctx));
   ctx->capacityWords = 2 * 256 / (8 * sizeof(uint64_t));
}



void
sha3_Update(void *priv, void const *bufIn, size_t len)
{
   sha3_context *ctx = (sha3_context *) priv;

   /* 0...7 -- how much is needed to have a word */
   uint32_t old_tail = (8 - ctx->byteIndex) & 7;
```

```
    size_t words;
    uint32_t tail;
    size_t i;

    const uint8_t *buf = bufIn;

    SHA3_TRACE_BUF("called to update with:", buf, len);

    SHA3_ASSERT(ctx->byteIndex < 8);
    SHA3_ASSERT(ctx->wordIndex < sizeof(ctx->s) / sizeof(ctx->s[0]));

    if(len < old_tail) {        /* have no complete word or haven't started
                                 * the word yet */
        SHA3_TRACE("because %d<%d, store it and return", (uint32_t)len,
                (uint32_t)old_tail);
        /* endian-independent code follows: */
        while (len--)
            ctx->saved |= (uint64_t) (*(buf++)) << ((ctx->byteIndex++) * 8);
        SHA3_ASSERT(ctx->byteIndex < 8);
        return;
    }

    if(old_tail) {              /* will have one word to process */
        SHA3_TRACE("completing one word with %d bytes", (uint32_t)old_tail);
        /* endian-independent code follows: */
        len -= old_tail;
        while (old_tail--)
            ctx->saved |= (uint64_t) (*(buf++)) << ((ctx->byteIndex++) * 8);

        /* now ready to add saved to the sponge */
        ctx->s[ctx->wordIndex] ^= ctx->saved;
        SHA3_ASSERT(ctx->byteIndex == 8);
        ctx->byteIndex = 0;
        ctx->saved = 0;
        if(++ctx->wordIndex ==
                (SHA3_KECCAK_SPONGE_WORDS - ctx->capacityWords)) {
            keccakf(ctx->s);
            ctx->wordIndex = 0;
        }
    }

    /* now work in full words directly from input */

    SHA3_ASSERT(ctx->byteIndex == 0);

    words = len / sizeof(uint64_t);
    tail = len - words * sizeof(uint64_t);

    SHA3_TRACE("have %d full words to process", (uint32_t)words);

    for(i = 0; i < words; i++, buf += sizeof(uint64_t)) {
        const uint64_t t = (uint64_t) (buf[0]) |
                ((uint64_t) (buf[1]) << 8 * 1) |
                ((uint64_t) (buf[2]) << 8 * 2) |
                ((uint64_t) (buf[3]) << 8 * 3) |
                ((uint64_t) (buf[4]) << 8 * 4) |
                ((uint64_t) (buf[5]) << 8 * 5) |
                ((uint64_t) (buf[6]) << 8 * 6) |
                ((uint64_t) (buf[7]) << 8 * 7);
#if defined(__x86_64__ ) || defined(__i386__)
        SHA3_ASSERT(memcmp(&t, buf, 8) == 0);
#endif
        ctx->s[ctx->wordIndex] ^= t;
        if(++ctx->wordIndex ==
                (SHA3_KECCAK_SPONGE_WORDS - ctx->capacityWords)) {
```

```
            keccakf(ctx->s);
            ctx->wordIndex = 0;
        }
    }

    SHA3_TRACE("have %d bytes left to process, save them", (uint32_t)tail);

    /* finally, save the partial word */
    SHA3_ASSERT(ctx->byteIndex == 0 && tail < 8);
    while (tail--) {
        SHA3_TRACE("Store byte %02x '%c'", *buf, *buf);
        ctx->saved |= (uint64_t) (*(buf++)) << ((ctx->byteIndex++) * 8);
    }
    SHA3_ASSERT(ctx->byteIndex < 8);
    SHA3_TRACE("Have saved=0x%016" PRIx64 " at the end", ctx->saved);
}

/* This is simply the 'update' with the padding block.
 * The padding block is 0x01 || 0x00* || 0x80. First 0x01 and last 0x80
 * bytes are always present, but they can be the same byte.
 */
void const *
sha3_Finalize(void *priv)
{
    sha3_context *ctx = (sha3_context *) priv;

    SHA3_TRACE("called with %d bytes in the buffer", ctx->byteIndex);

    /* Append 2-bit suffix 01, per SHA-3 spec. Instead of 1 for padding we
     * use 1<<2 below. The 0x02 below corresponds to the suffix 01.
     * Overall, we feed 0, then 1, and finally 1 to start padding. Without
     * M || 01, we would simply use 1 to start padding. */

#ifndef SHA3_USE_KECCAK
    /* SHA3 version */
    ctx->s[ctx->wordIndex] ^=
            (ctx->saved ^ ((uint64_t) ((uint64_t) (0x02 | (1 << 2)) <<
                    ((ctx->byteIndex) * 8))));
#else
    /* For testing the "pure" Keccak version */
    ctx->s[ctx->wordIndex] ^=
            (ctx->saved ^ ((uint64_t) ((uint64_t) 1 << (ctx->byteIndex *
                            8))));
#endif

    ctx->s[SHA3_KECCAK_SPONGE_WORDS - ctx->capacityWords - 1] ^=
            SHA3_CONST(0x8000000000000000UL);
    keccakf(ctx->s);

    /* Return first bytes of the ctx->s. This conversion is not needed for
     * little-endian platforms e.g. wrap with #if !defined(__BYTE_ORDER__)
     * || !defined(__ORDER_LITTLE_ENDIAN__) || \
     * __BYTE_ORDER__!=__ORDER_LITTLE_ENDIAN__ ... the conversion below ...
     * #endif */
    {
        uint32_t i;
        for(i = 0; i < SHA3_KECCAK_SPONGE_WORDS; i++) {
            const uint32_t t1 = (uint32_t) ctx->s[i];
            const uint32_t t2 = (uint32_t) ((ctx->s[i] >> 16) >> 16);
            ctx->sb[i * 8 + 0] = (uint8_t) (t1);
            ctx->sb[i * 8 + 1] = (uint8_t) (t1 >> 8);
            ctx->sb[i * 8 + 2] = (uint8_t) (t1 >> 16);
            ctx->sb[i * 8 + 3] = (uint8_t) (t1 >> 24);
```

```
            ctx->sb[i * 8 + 4] = (uint8_t) (t2);
            ctx->sb[i * 8 + 5] = (uint8_t) (t2 >> 8);
            ctx->sb[i * 8 + 6] = (uint8_t) (t2 >> 16);
            ctx->sb[i * 8 + 7] = (uint8_t) (t2 >> 24);
        }
    }

    SHA3_TRACE_BUF("Hash: (first 32 bytes)", ctx->sb, 256 / 8);

    return (ctx->sb);
}
```

Appendix J

A section of the LBACS authentication service implemented in nodeJS utilizing the custom pairwise implementation of LBACS, blockchain and elliptic curve library has been included below.

```
/*
  @author Gilroy Gordon
  @copyright Gilroy Gordon 2016. All Rights Reserved.
*/

function LBACSAuthService(props) {
    props = props || {};
    var _self = this;

    _self.pairwiseContext = null;
    _self.pkiContext = new PKIService();

    _self.getNodeId = function() {
        return lbacsConfig.nodeId;
    };


    /**
     * Verify Signature Authenticity and Authenticate
     * Pairwise Message from EthereumContract
     */
    _self.authenticateLBACSMessage = function(params, callback) {
        callback = typeof callback === 'function' ? callback : function() {};
        params = params || {};
        params.getMessageFromJSON = typeof params.getMessageFromJSON == 'function' ?
            params.getMessageFromJSON : function(m) {
                return JSON.stringify(m.message)
            };
        logger.debug(LBACSAuthService.name + ":: Authenticating LBACS Message:", params);
        params.Contract
            .at(params.messageAddress)
            .toJSON(myEthConfig.getDefaults(), function(err, _messageJSON) {
                if (err) {
                    logger.debug("Retrieved from json error", err);
                }
                logger.debug("Retrived message json", _messageJSON);

                LBACSChainUtility.retrieveSignature(_messageJSON.signature, function(signatureErr, signature) {
                    if (signatureErr) {
                        return callback(signatureErr);
                    }
                    params.message = params.getMessageFromJSON(_messageJSON);
                    _self.authenticate({
                        intention: _messageJSON.intention,
                        message: params.message,
                        peerIds: [_messageJSON.toPeerId],
                        toPeerId: _messageJSON.toPeerId,
                        fromPeerId: _messageJSON.fromPeerId,
                        entityAddress: _messageJSON.authorEntity,
                        signature: signature,
                        requestId: _messageJSON.requestId,
```

```
                hmac: _messageJSON.hmac.replace("0x", "")
            }, callback);
        });

    });


};

/**
 * Verify Signature Authenticity using ECDSA and Authenticate Pairwise Message with LBACS
 */
_self.authenticate = function(params, callback) {
    callback = typeof callback === 'function' ? callback : function() {};
    params = params || {};
    logger.debug(LBACSAuthService.name + ":: Authenticating:", params);
    //check valid public key
    LBACSChainUtility.retrieveAndVerifyPublicKeyOfEntity(
        params.entityAddress,
        function(err, publicKey) {
            if (err) {
                return callback(err);
            }

            logger.debug(LBACSAuthService.name + ":: Received public key:", publicKey);
            //verify signature

            if (_self.pkiContext.verifyMessage(
                    LBACSChainUtility.buildDataMessageToBeSigned(params),
                    params.signature,
                    publicKey
                )) {
                logger.debug(LBACSAuthService.name + ":: Verified signaure:", publicKey);
                //pairwise authentication with sensor
                var pairwiseAuthParams = {
                    message: params.message,
                    intention: params.intention,
                    requestId: params.requestId,
                    token: hexToBytes(params.hmac, true),
                    peerIds: [params.toPeerId],
                    fromPeerId: params.fromPeerId
                };
                logger.debug("Authenticating pairwise params", pairwiseAuthParams);
                if (_self.pairwiseContext.authenticate(pairwiseAuthParams)) {
                    logger.debug(LBACSAuthService.name + ":: Authenticated Pair Message:", publicKey);
                    return callback(null, true);
                } else {
                    return callback(new Error('Peer Authentication Failure'));
                }
            } else {
                return callback(new Error('Signature Could not be verified'));
            }
        })
};

/**
 * Authenticate Pairwise Message with LBACS
 */

_self.authenticatePairwiseMessage = function(params) {
    params = params || {};
```

```
    return _self.pairwiseContext.authenticate({
        message: params.message,
        intention: params.intention,
        requestId: params.requestId,
        token: hexToBytes(params.hmac, true),
        peerIds: Array.isArray(params.peerIds) ? params.peerIds : [params.toPeerId],
        fromPeerId: params.fromPeerId
    });

};

/**
 * Sign Message with ECDSA
 */

_self.signMessage = function(params) {
    params = params || {};
    params.message = params.message || '';
    if (typeof params.message == 'function') {
        params.message = params.message();
    }
    if (typeof params.message === 'object') {
        params.message = JSON.stringify(params.message);
    }

    var pairwiseResult = _self.signMessageForPairwiseAuth(params);
    if (pairwiseResult) {
        params.requestId = pairwiseResult.requestId;
        pairwiseResult.hmac = params.hmac = bytesToHex(pairwiseResult.token.slice(0, 5));
        pairwiseResult.hmacBytes = pairwiseResult.token.slice(0, 5);
        logger.debug("pairwise signature successful, complete params", params);
        var messageToBeSigned = LBACSChainUtility.buildDataMessageToBeSigned(params);
        pairwiseResult.signature = _self.pkiContext.signMessage(messageToBeSigned);
        return pairwiseResult;
    }
    return null;
};


/**
 * Sign Message with ECDSA and create signature on Ethereum blockchain
 */
_self.signMessageAndCreateSignatureOnChain = function(params, callback) {
    callback = noop(callback);
    var result = _self.signMessage(params);
    if (!result) {
        return callback(new Error("Unable to sign message"), result);
    }
    LBACSChainUtility.createSignature(result.signature, function(err, signatureAddress) {
        if (err) {
            return callback(err, result);
        }
        result.signatureAddress = signatureAddress;
        return callback(null, result);
    });
};

/**
 * Sign Message with ECDSA and create signature on Ethereum blockchain
 */
```

```
_self.forwardSignedMessageAndCreateSignatureOnChain = function(params, callback) {
    callback = noop(callback);
    params = params || {};
    params.message = params.message || '';
    if (typeof params.message == 'function') {
        params.message = params.message();
    }
    if (typeof params.message === 'object') {
        params.message = JSON.stringify(params.message);
    }

    params.hmac = bytesToHex(params.hmac);
    params.hmacBytes = params.hmac;

    var pairwiseResult = {
        requestId: params.requestId,
        hmac: params.hmac,
        hmacBytes: params.hmacBytes

    };
    if (pairwiseResult) {
        logger.debug("pairwise signature included, complete params", params);
        var messageToBeSigned = LBACSChainUtility.buildDataMessageToBeSigned(params);
        pairwiseResult.signature = _self.pkiContext.signMessage(messageToBeSigned);
    }
    LBACSChainUtility.createSignature(pairwiseResult.signature, function(err, signatureAddress) {
        if (err) {
            return callback(err, pairwiseResult);
        }
        pairwiseResult.signatureAddress = signatureAddress;
        return callback(null, pairwiseResult);
    });

};


/**
 * Sign Message using LBACS
 */
_self.signMessageForPairwiseAuth = function(params) {
    logger.debug("signMessageForPairwiseAuth params", params);
    params = params || {};
    params.message = params.message || '';
    var result = _self.pairwiseContext.generateToken({
        intention: params.intention || 0,
        message: params.message,
        peerIds: params.peerIds,
        fromPeerId: params.fromPeerId || 0
    });
    return result;
};


/**
 * Approve/Sign Authenticated Data and Publish on Blockchain
 */
_self.approveMessage = function(params, callback) {
    params = params || {};
    callback = noop(callback);

    _self.getMyAuthorEntity(null, function(err, authorEntity) {
```

```
            if (err) {
                logger.error("Unable to approve message. Did not retrieve author entity", err);
                return callback(err);
            }
            var signature = _self.pkiContext.signMessage(params.messageAddress);

            logger.debug("Retrieved Author Entity and Created Signature for Approved Message : " + signature);

            LBACSChainUtility.createSignature(signature, function(err, signatureAddress) {
                if (err) {
                    logger.error("Unable to publish approved message on chain, signature not created", err);
                    return callback(err);
                }
                LBACSAuthenticatedData.new(params.messageAddress, authorEntity.address, signatureAddress, {
                    from: myEthConfig.defaults.ownerAddress,
                    gas: myEthConfig.getLatestGasCost()
                }).then(function(_approvedMessage) {
                    logger.debug("Approved message published at : " + _approvedMessage.address);
                    callback(null, _approvedMessage);

                }).catch(function(err) {
                    logger.error("Unable to publish approved message on chain", err);
                    callback(err);
                });
            });

        });

    };



    _self._init = function(props) {
        props = props || {};
        //load and save private key if its first time
        _self.pkiContext.savePrivateKey();
        //create pairwise context from lbacsjs
        _self.pairwiseContext = new LBACSContext({
            nodeId: lbacsConfig.nodeId,
            nodePrivateKey: lbacsConfig.nodePrivateKey,
            initializeContext: true,
            peers: lbacsConfig.peers
        });

    }

    _self._init(props);


}

module.exports = LBACSAuthService;
```

Appendix K

IEEE1076 Very High Speed Integrated Circuit Hardware Definition Language implementation of LBACS integrated circuit components.

```
-- Mock implementation of a HMAC
entity hmacMock is
	port (
		intention, requestId : in BIT_VECTOR(7 downto 0);
	   nodeId, peerId : in BIT_VECTOR(7 downto 0);
	   pairwiseKey, message : in BIT_VECTOR(31 downto 0);
		hmac : out BIT_VECTOR(255 downto 0)
	);
end hmacMock;

architecture rtl of hmacMock is

begin
	process(pairwiseKey)
	begin
      for i in 0 to 8 loop
			for j in 0 to 31 loop
				hmac (i*j) <= pairwiseKey(j) xor message(j);
			end loop;
	   end loop;
	end process;
end rtl;

-- Mock implementation of a HMAC ends

-- LBACS truncation module - Series of Multiplexers
entity lbacsTruncation is
	port (
	   randomSeed : in BIT_VECTOR(7 downto 0);
	   hmacIn : in BIT_VECTOR(255 downto 0);
		hmacOut : out BIT_VECTOR(23 downto 0)
	);
end lbacsTruncation;

architecture rtl of lbacsTruncation is
begin
	process(hmacIn)
	   variable randomSeedInt : integer ;
	   variable tempInt : integer;
	begin
		randomSeedInt := 0;
		if (randomSeed(7)='1') then
		   randomSeedInt := randomSeedInt + 128;
		end if;
		if (randomSeed(6)='1') then
		   randomSeedInt := randomSeedInt + 64;
		end if;
		if (randomSeed(5)='1') then
		   randomSeedInt := randomSeedInt + 32;
		end if;
	   if (randomSeed(4)='1') then
```

```
                    randomSeedInt := randomSeedInt + 16;
                end if;
            if (randomSeed(3)='1') then
                    randomSeedInt := randomSeedInt + 8;
                end if;
            if (randomSeed(2)='1') then
                    randomSeedInt := randomSeedInt + 4;
                end if;
            if (randomSeed(1)='1') then
                    randomSeedInt := randomSeedInt + 2;
                end if;
            if (randomSeed(0)='1') then
                    randomSeedInt := randomSeedInt + 1;
                end if;

            tempInt := (41 * randomSeedInt) mod 32;
            hmacOut(7 downto 0) <= hmacIn(tempInt+7 downto tempInt);

            tempInt := (2*41 * randomSeedInt) mod 32;
            hmacOut(15 downto 8) <= hmacIn(tempInt+7 downto tempInt);

            tempInt := (4*41 * randomSeedInt) mod 32;
            hmacOut(23 downto 16) <= hmacIn(tempInt+7 downto tempInt);
   end process;
end rtl;
-- LBACS truncation module ends
-- LBACS Integrated Circuit begins
entity lbacs is
  port (
            intentionIn, requestIdIn : in BIT_VECTOR(7 downto 0);
            nodeId, peerIdA, peerIdB : in BIT_VECTOR(7 downto 0);
            pairwiseKeyA, pairwiseKeyB, message : in BIT_VECTOR(31 downto 0);
            randomSeedA, randomSeedB : in BIT_VECTOR(7 downto 0);
            intentionOut, requestIdOut : out BIT_VECTOR(7 downto 0);
            hmac1, hmac2 : out BIT_VECTOR(23 downto 0)
  );
end lbacs;
architecture rtl of lbacs is
        signal tempHMAC1 : BIT_VECTOR(255 downto 0);
        signal tempHMAC2 : BIT_VECTOR(255 downto 0);
begin
        intentionOut <= intentionIn;
        requestIdOut <= requestIdIn;

        ENTHMAC1 : entity work.hmacMock port map(
                intentionIn, requestIdIn , nodeId, peerIdA, pairwiseKeyA,message,tempHMAC1
                );
        HMAC1TRUN : entity work.lbacsTruncation port map(
                randomSeedA, tempHMAC1, hmac1
                );

        ENTHMAC2 : entity work.hmacMock port map(
                intentionIn, requestIdIn , nodeId, peerIdB, pairwiseKeyB,message,tempHMAC2
                );
        HMAC2TRUN : entity work.lbacsTruncation port map(
                randomSeedB, tempHMAC2, hmac2
                );
end rtl;
-- End LBACS module
```