Title: Security and Key Establishment in IEEE 802.15.4

Student: Eirik Klevstad

Problem description:

Internet of Things (IoT) is a network where devices, sensors, vehicles, buildings, and humans communicate and collaborate, along with collecting and exchanging information. IEEE 802.15.4 specifies the lower layers for low-rate wireless networks, which are widely seen as the foundation for current IoT communications. One of the potential weaknesses of the IEEE 802.15.4 standard is the lack of specification for key establishment and management.

This thesis will focus on key management for device-to-device security in IoT. It will review and compare the proposed protocols, and include both formal and informal security analysis, as well as analysis of both key management requirements and key agreement protocol design for IoT security. Another goal of the thesis will be to suggest improvements and alternatives to the proposed protocols.

Responsible professor: Colin Boyd, ITEM

Supervisor: Britta Hale, ITEM

Contents

List of Figures

List of Tables

Listings

List of Acronyms

6LoWPAN IPv6 over Low power Wireless Personal Area Networks.

AES Advanced Encryption Standard.

AKES Adaptable Key Establishment Scheme.

APKES Adaptable Pairwise Key Establishment Scheme.

AS Authentication Server.

BAN Burrows-Abadi-Needham.

CA Certificate Authority.

CCM Counter with CBC-MAC.

CTR Counter.

DoS Denial of Service.

EAP Extensible Authentication Protocol.

ECC Elliptic Curve Cryptography.

ECDH Elliptic Curve Diffie-Hellman.

ECDSA Elliptic Curve Digital Signature Algorithm.

GPS Global Positioning System.

GUI Graphical User Interface.

IEEE Institute of Electrical and Electronics Engineers.

IETF Internet Engineering Task Force.

IoT Internet of Things.

IP Internet Protocol.

KCI Key Compromise Impersonation.

KDC Key Distribution Center.

LEAP Localized Encryption and Authentication Protocol.

LKR Long-term Key Reveal.

MAC Message Authentication Code.

MIC Message Integrity Code.

MSC Message Sequence Chart.

OSI Open Systems Interconnection.

PFS Perfect Forward Secrecy.

RAM Random Access Memory.

RFID Radio Frequency Identification.

RSA Rivest-Shamir-Adleman.

SKR Session-Key Reveal.

SNMP Simple Network Management Protocol.

SS Service Server.

TCP Transmission Control Protocol.

TGS Ticket Granting Service.

TGT Ticket Granting Ticket.

TLS Transport Layer Security.

UDP User Datagram Protocol.

WLAN Wireless Local Area Network.

WPA Wi-Fi Protected Access.

WPA2 Wi-Fi Protected Access II.

WPAN Wireless Personal Area Network.

wPFS Weak Perfect Forward Secrecy.

WSN Wireless Sensor Network.

Chapter 1
Introduction

This should be the introduction to the thesis.

Chapter 2
Background and Related Work

2.1 Internet of Things

Over the last decade, a concept called the *Internet of Things* has gained increased attention, both from the research community and commercial actors, as well as from consumers. The term IoT was, accordingly to most sources, coined in 1999 by the British visionary Kevin Ashton in a presentation about Radio Frequency Identification (RFID) [3] [60]. Ashton's definition of the concept was a world where computers do not depend on human beings to provide them with information. Out of all the petabytes of information available on the Internet, the majority has been created and captured by humans performing some sort of action. In his opinion, IoT is about providing computers with the ability to gather information on their own.

A computational device containing some sort of sensor is attached to your everyday physical device, creating a bridge between our physical world and the cyber world [38]. The connection to the Internet allows us to monitor and control these devices and sensors from a remote distance. Another vital part of IoT is device-to-device communications, essentially enabling devices to communicate with each other without human aid, and exchange and retrieve information. Such devices could be sensors monitoring some operation, a physical area, or even attached to a physical body. The possibilities are more or less unlimited. Imagine a home automation and surveillance system for your cabin, where lights, heaters, smoke detectors, underfloor heating, motion detectors, security cameras, garage and so on, are all interconnected with each other through small wireless devices. As it is called the *internet* of things for a reason, your system and devices would be accessible over the Internet, allowing you to monitor the current status of your cabin remotely from your couch at home, as well as looking at historical data of the different sensors and devices. When the weekend arrives and you head for the mountains, the IoT provides you with an opportunity to preheat different (or all) sections of the cabin, deactivate the alarm, and perhaps instruct the sauna to start getting cosy.

Another approach is to avoid using a monitor to remotely control the system, and instead allowing the system to observe and act on your behaviour. We want the devices to know us and figure out the correct thing to do without us telling them. For example, when pulling your car into the driveway, you want the garage door that is connected with your car to open up. The garage notifies your front door that you are home, which conveniently unlocks and notifies your house to turn on the lights in your hallway and perhaps the heater in your living room.

The possibilities that are revealed as the IoT grows larger and the services expand are infinite. The concept is highly applicable for different scenarios involving home automation, standalone consumer products, industrial and environmental facilities, as well as medical surveillance. While larger automation systems for homes and facilities have been the target for the research community and early adopters, the consumer market has been focused on so-called *wearables*. Wearables are fundamentally devices that you wear, such as smart watches, fitness trackers, virtual reality glasses, headphones, and smart clothing. Such human-centric devices are less about automation, and more focused on personal improvement. Nevertheless, the increase in IoT devices possibly provides us with a more cost efficient future, both in our use of time, as well as energy and consumption of other resources.

As the IoT is built upon the Internet, it faces the same types of security issues as the Internet itself. The amount of "things" connected to the Internet is calculated to be 6.4 billions by the end of 2016, which is almost a 30% increase from 2015. By 2020, the expected number of these "things" is more than 20 billion [30], providing attackers with equally many possible devices to attack. Given the knowledge that some of these devices may be medical (or have other sensitive applications), we quickly recognize potential catastrophic scenarios.

The IoT architecture can resemble the neural system of the human body. The perception layer controls our sensors which we use to obtain information about our environment by observing, feeling, smelling, tasting or hearing. As previously described, IoT devices are often deployed with one or more sensors to perform these "human operations" for information collection. The perception layer is mainly focusing on sensing and allowing IoT devices to observe their environment and collect information. Examples of such technologies are RFID, Wireless Sensor Network (WSN), and the Global Positioning System (GPS) [36]. Information from our human sensors are carried to the brain through a neural network. Much alike in IoT, the collected information is transmitted using the transportation layer. The transportation layer is running over some wireless or wired medium such as 802.15.4, IPv6 over Low power Wireless Personal Area Networks (6LoWPAN), 3G, Bluetooth or Infrared. Finally the information is processed by an intelligent entity. In our human body example, this would be the brain. In the IoT, the brain would be

an intelligent processing unit in the application layer which is able to compute and evaluate actions based on the received information [37] [64]. The application layer is also responsible for controlling the sensors, and performing global system management, and present data for the end user of the system.

As these layers covers different characteristics of IoT, they consists of different types of hardware and provide different types of services, hence they are subject to different types of security threats and solutions. The most adjacent problems to the scope of this thesis are the problems related to key establishment and key management, which define how two devices safely can establish secure communication between each other. Or in other words, how collected information is safely transmitted between the sensors and the "brain".

In an IoT world, the protection of data and privacy is an essential part. As previously mentioned, IoT technology may be a solution for problems involving sensitive information. In a medical facility, a possible scenario could be a WSN, which is a dynamic and bi-directional network of nodes where each node has one or more sensors connected to it. A patient may have sensors implanted in their body, as well as different instruments attached for measuring different properties. All these devices communicate with each other wirelessly, and the network is therefore a possible target for an attacker. To prevent the attacker from eavesdropping, and possible forging content in the network, encryption and authentication at the different nodes is crucial.

2.2 The IEEE 802.15.4 Standard

Following the evolution of IoT, the need for cheap devices to communicate efficiently between each other has arisen. Existing architectures such as 802.11 (WiFi) and Bluetooth are too expensive in terms of processing and energy consumption, as the idea of IoT is to connect even the smallest devices to a network or Internet. As these devices are small, they have a limited battery life, and hence need to use energy in a highly efficient matter.

Protocols using the Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 standard are envisioned for applications supporting smart homes, medical surveillance, monitoring systems for environmental and industrial systems, as well as sensor systems for heating and ventilation. As we know from the IoT, it is really the imagination that puts an end to the possibilities for interconnected devices. The Open Systems Interconnection (OSI) stack defines the internal structure of communications systems, and is shown in Figure 2.1. As the 802.15.4 standard only defines the physical and data link layer of the OSI stack, which can be seen in Figure 2.1, specifications need to be developed to utilize the possibilities provided by

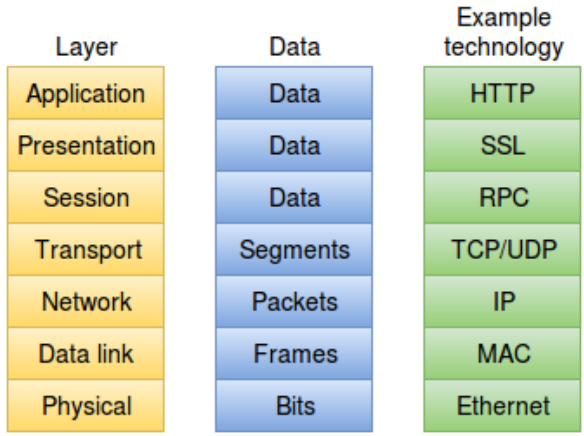

Figure 2.1: The OSI stack with layers, the data they carry, and an example of technology running on the different layers.

802.15.4 in the upper layers. ZigBee [1], maintained by the ZigBee Alliance, is the most notable example of specifications that uses 802.15.4 as its base. Others include WirelessHART [29], MiWi [62], and ISA100.11a [53].

The fundamental intention of the 802.15.4 standard is to provide low-rate, low-power communication between devices within a sensor network or Wireless Personal Area Network (WPAN). Its main use case is to let multiple devices within a short range communicate with each other over a low-rate radio, while maintaining a modest energy consumption. Figure 2.2 paints a picture of what 802.15.4 is, compared to more well-established concepts such as WiFi (802.11) and Bluetooth, focusing on energy consumption, complexity and date rate. While being smaller and more cost efficient than those found in more complex networks, devices running on 802.15.4 networks have a much more limited range (about 10 meters), and in most cases a throughput below 250 Kbps [32]. Not only is the 802.15.4 standard significantly lighter in terms of data rate and power consumption, it is also aimed at a different market than regular WPANs. WPANs are oriented around a person, creating a personal network for the user, which has higher demands to data rate, and can allow a higher energy consumption. 802.15.4, however, focuses on interconnecting devices that do not necessary have this constraint, such as industrial and medical applications.

Four basic security services are provided in the 802.15.4 link-layer security package, namely access control, message integrity, message confidentiality, and replay protection (sequential freshness) [59]. The IEEE 802.15.4 standard is delivered with a total of eight different security suites, providing none, some, or all of the described

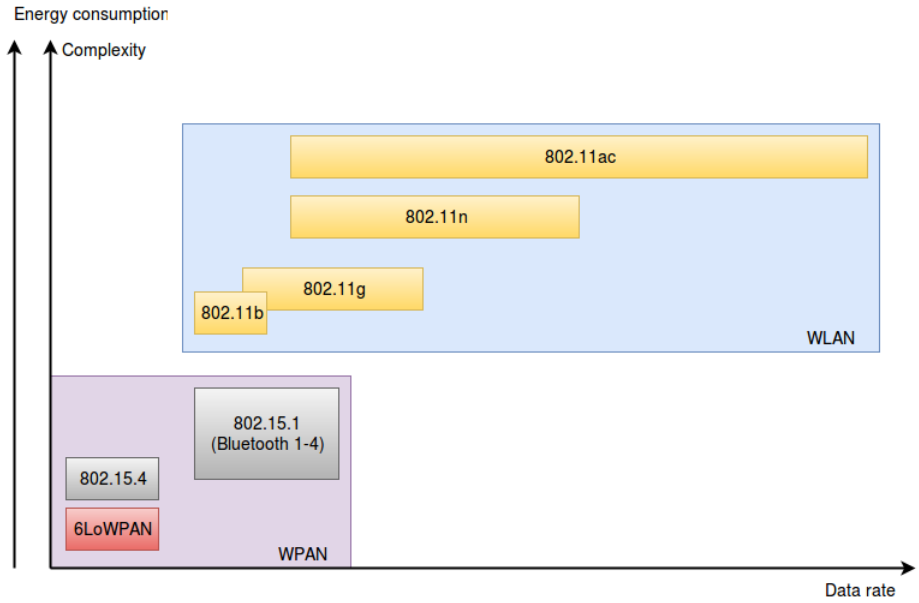

Figure 2.2: Figure of IEEE 802.15.4's operational space compared to other wireless standards [32].

security services, and it is up to the application designer to specify and enable the desired security properties. In the most secure end of the scale we find Advanced Encryption Standard (AES)-Counter with CBC-MAC (CCM), which is encryption using the block cipher AES with either 32, 64 or 128-bit Message Authentication Code (MAC). Such a suite provides both strong encryption and possibly unforgeable messages (a 64-bit MAC gives an adversary a 2^{-64} chance of successfully forging a message, and is used to enable legitimate nodes in the network to detect if a received message have been tampered with). On the other end of the scale we find a suite providing only confidentiality using AES in Counter (CTR) mode. This suite does not, however, provide any form of authentication – giving adversaries the possibility to forge messages, which can not be said to be especially secure. One of the things the 802.15.4 standard does not specify is how to deal with key establishment and key management, which therefore has to be dealt with in the higher layers.

2.3 6LoWPAN: Putting IP on Top of 802.15.4

Initially, the Internet Protocol (IP) was considered to be too "heavy" for low-power wireless networks such as the ones described by the 802.15.4 standard. The idea of implementing IP on top of 802.15.4 networks was born as early as 2001 under the question "Why invent a new protocol when we already have IP?"[49]. With IP, the

community already had a bundle of existing protocols for management, transport, configuring and debugging, such as Simple Network Management Protocol (SNMP), Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), as well as standardized services for higher layers such as caching, firewalls, load balancing, and mobility. Nevertheless, the initial idea of using IP in combination with sensor networks or WPANs was not accepted by various groups such as ZigBee [49]. The rejection did not, however, stop the initiative, and a group of engineers within Internet Engineering Task Force (IETF) started designing and developing what would later be known as 6LoWPAN.

A significant advantage with combining IP and 802.15.4 is the simplification of the connectivity model between various devices in the networks. As most 802.15.4-based specifications usually need custom hardware that tends to be complex, the possibilities to interconnect different networks with each other is somewhat limited. By turning to IP, the need for complex connectivity models is obsolete as it is possible to use well-understood technologies such as bridges and routers. Another advantage with using IP is that the routers between the 6LoWPAN devices and the outside networks (so-called edge routers) do not need to maintain any state as they are only forwarding datagrams.

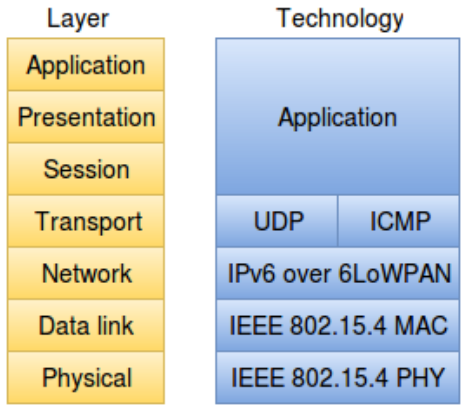

Figure 2.3: Figure of the 6LoWPAN stack, which uses the 802.15.4 physical and link layer, but adds an adoption layer in the network layer.

6LoWPAN enables wireless 802.15.4 sensor devices to connect directly to the Internet via IPv6 by providing an adoption layer in the network layer between it and the data link layer as shown in Figure 2.3. The adoption layer provides a unique functionality which both fragments and compresses incoming packets to enable the embedded devices in 802.15.4 networks to receive the packets while using the least amount of memory and energy [41]. Its fundamental idea is that you only "pay" for what you use. The dispatch header field identifies the type of header to follow,

and consists of 1 byte [49]. Such a header starts with either 00 or 01, respectively indicating whether the frame is a non-6LoWPAN frame or a regular 6LoWPAN-frame. Currently, only five different dispatch headers have been defined [34]. Therefore, there is a fair space for new headers as the standard and technology evolves. However, the special case of a header consisting solely of ones, adds an additional byte to the header, enabling a total of 320 different header types [49]. This greatly differs from IPv4 and Zigbee which define only one monotonic header, and can be used to greatly minimize the header size of a packet as some types of frames may consist of smaller payloads than others.

Compared to other alternatives such as ZigBee or Z-wave, 6LoWPAN's implementation did not prove to be any more expensive in terms of code size and Random Access Memory (RAM) requirements. 6LoWPAN seems to be a natural choice for the future IoT as a networking protocol. It is scalable thanks to IPv6, and its headers can be compressed to only a few bytes using its fragmentation and compression mechanism. Following the expected bloom in IoT devices over the next few years (20 billion by 2020), and the fact that the IPv6 address space is not going to be exhausted any time soon (roughly 2^{95} addresses for each and every one of us), 6LoWPAN may be the most reasonable approach.

2.4 Key Establishment and Key Management

As described, IoT devices communicate with each other over the network by utilizing some network protocol. There is, however, not always a guarantee that the network used for communication is secure. An attacker may be eavesdropping on the network, and may even be capable of intercepting and spoofing traffic sent between different nodes. From a security perspective, the described attacker is violating both the confidentiality and integrity of the exchanged information. To cope with this, devices should be encrypting and authenticating the data that they are exchanging.

Key establishment is a fundamental idea in cryptography where two (or more) communicating parties exchange information in order to generate cryptographic keys which enable them to perform some sort of cryptography on the messages that are sent between them. The problem is, however, how to safely agree upon the keys to use in the encryption-decryption process when the network itself cannot be trusted. For IoT devices and sensor networks, confidentiality and data integrity are important aspects. As previously described, IoT devices have limited resources in terms of battery life and processing abilities, making key establishment schemes that work well in other networks with access to more resources, such as WiFi, infeasible in an IoT scenario.

2.4.1 Long-Term and Session Keys

Long-Term keys

Long-term keys are keys that are deliberately stored somewhere on a device, as they are used multiple times for securing communication, and have no "expiration date". The shared secret key in symmetric key encryption, and the private key in public-key cryptography are examples of long-term keys.

Session Keys

Session keys are temporary keys that are used for a short period of time. By using session keys, the amount of ciphertexts encrypted with the same key is limited for an adversary to perform cryptanalysis on. Another advantage with using session keys is in the case of a node being compromised. If the protocol provides forward secrecy and known-key security (which will be described in Section 2.4.2), the data that is compromised is limited to that particular session. Session keys are not permanently stored at the client, and usually deleted after its expiration time, limiting the cost of memory, and distancing it from leaking previous session keys in the case of being compromised.

2.4.2 Security Attributes in Key Establishment Schemes

Authentication

Authentication is an important aspect of key establishment. More specifically, confirming the identity of the entity you are establishing keys with, as well as the confirming that the established keys are authentic. If authentication is skipped, the protocol will be weak for so-called man-in-the-middle attacks where an adversary intercepts and relays messages between two communicating parties to learn or modify its content. There are multiple ways for parties to provide authentication based on the chosen establishment scheme. For symmetric schemes, the inclusion of a MAC could provide authentication of the identity of the origin of the message. Schemes using public-key provides authentication through digital certificates, which are issued by a Certificate Authority (CA), and ensure the link between an identity and a public key.

For session keys, two properties are introduced, namely *implicit key authentication* and *explicit key authentication*. One of the most used ways of establishing session keys between two entities A and B is through A generating a random symmetric key, which is encrypted under B's public key, before it is transmitted. B is then able to extract the session key that should be used for encrypting data using his private key. *Implicit key authentication* assures that only the rightful owner of the public key, which the session key is encrypted under (in this case B), is able to

recover the session key. It does not, however, assure that B is in fact possessing the session key [33]. If the protocol also assures A that B has received and possesses the session key, the protocol provides a property called *key confirmation*, and if a protocol provides both implicit key authentication and key confirmation, we can say that the protocol overall provides *explicit key authentication*. As a side note, exactly how to define explicit key authentication is in some sense based on your view on life. Is the knowledge of that the other party possesses everything it needs to derive the shared key enough for confirmation, or is explicit key authentication to actually obtain something signed or encrypted using the pairwise key that you have derived. For this thesis, we will stick to the latter one as our definition of explicit key authentication.

Known-Key Security

Session keys are single-use symmetric keys that are used for a given period of the communication before being replaced and deleted from the system, and never to be used again. Known-key security is a property where the leak of information is minimized in the case of one (or multiple) session keys are compromised. For example in the case where session keys are derived from the private key, then the compromise should not lead to the compromise of the private key, nor any of the past or future session keys.

Perfect Forward Secrecy

Following in the lines of known-key security, Perfect Forward Secrecy (PFS) is a security attribute where in the case of the long-term private key of one (or both) of the communicating parties being compromised, it should not lead to the reveal of any of the past session keys that are used in the communication between the parties. The Heartbleed incident in 2014 was a painful example of the need for PFS, where a bug in the OpenSSL cryptographic software library leaked secret keys for certificates, as well as user names and passwords [27]. Attackers were able to retrieve 64 kilobytes of the memory of web servers for each attack (or "heartbeat"), which could be used to retrieve the private long-term keys of the web servers which did not support forward secrecy. The private keys could then be used to retroactively decrypt almost all traffic that had previously been recorded. One of the features with the Diffie-Hellman key exchange is that it can be used to provide forward secrecy for Transport Layer Security (TLS)'s ephemeral modes, making web servers implementing such versions of TLS immune to attacks that exploited the Heartbleed bug.

PFS is a desirable security property for key establishment protocols, but it is often difficult to achieve. Weak Perfect Forward Secrecy (wPFS) is a weaker type of PFS, where the adversary is assumed to be *passive* [39]. In the case of a long-term key compromise, previous sessions are guaranteed to be secure, but only if the adversary

was not actively interfering with the protocol during the session. As PFS is a property related to session keys, it is not an achievable property for symmetric key schemes.

Key-Compromise Impersonation

In this case, an adversary has obtained the long-term private key of an honest entity A. Key Compromise Impersonation (KCI) prevents the adversary both from impersonating A to other entities (establishing session keys with them), as well as preventing the adversary from impersonating other entities in communication with A (masquerading as a different entity in order to establish a session key with A). KCI is, however, a very difficult security property to achieve for symmetric key protocols.

Key Control

Key control is to prevent a party from computing a part of the session key without input from the other party. Essentially, one of the communicating parties should not be able to force the secret session key into something of its own choice. Key control is usually accomplished through both parties creating a random value, which is shared with the other party, and computed together into the shared key, for example in the Diffie-Hellman key exchange.

Unknown Key-Share

Unknown key-share resilience is an attribute in key agreement protocols where a key shared between two entities A and B can not be shared with any others without both consenting to it. When A and B are establishing a shared key, attacks targeting this process may want to convince A that it is sharing the key with B, while B in fact is under the impression that it is sharing the key with a third entity C. After the key establishment process is finished, A believes it has established a key with B (which is correct), but B is under the belief that it has established a key with C. This results in that when B thinks it is sending a message to C, A is the actual receiver of the message.

2.4.3 Key Establishment Architectures

Symmetric Key

Symmetric encryption is a technique for encrypting messages sent between two communicating parties, where the secret key used for encrypting the message is identical to the key used for decrypting it, as seen in Figure 2.4. Plaintext messages are processed through either a stream cipher, which encrypts the message byte by byte, or through a block cipher, which operates on a certain number of bits of the message for each round. The encryption process results in an encrypted message called a ciphertext. In schemes utilizing this form of encryption, it is essential that

both parties possesses the same shared secret (or key). One approach might be to load the shared secret into each of the parties in advance, which is inconvenient and difficult approach for a network where nodes may be joining and leaving after network deployment. The most reasonable approach would be for two nodes to agree upon the shared secret together in a possibly unsafe environment.

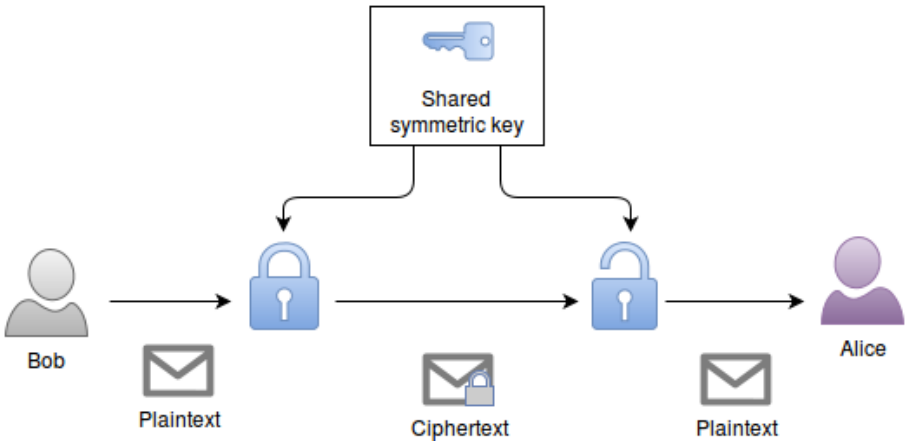

Figure 2.4: Figure of a symmetric encryption scheme, where both parties possess the same symmetric key used for encryption and decryption.

In the 1970s, Whitfield Diffie and Martin Hellman introduced the Diffie-Hellman key exchange, which allows two communicating parties to safely establish a shared secret without any prior knowledge of with each other[24]. The shared secret could then be used for encrypting and decrypting messages sent between the two parties. While being a straightforward and fast way of encrypting information, symmetric encryption has a major drawback in the case of one of the nodes is compromised. Node compromises would lead to an initially secure channel being insecure as the adversary could easily decrypt any message that it intercepts. Also, the sharing of the key is difficult to do in a secure manner. Another disadvantage with symmetric key schemes is the difficulty of authenticating the other party of the communication as they both encrypt data using the same key. For systems using symmetric encryption, authentication can be achieved through construction of MACs, which are cryptographic values generated from a symmetric key and the plaintext message. This enables the receiver of a message to compute the same MAC from the decrypted ciphertext and the shared symmetric key, and provides both authenticity of the sender and the integrity of the received message.

Online Servers and Trusted Third Parties

By using a client-server architecture, the idea of a symmetric key that is shared between two parties can be extended to also include mutual authentication and session keys. Alice and Bob wants to establish a shared key, but they do not necessary trust each other. However, they both trust Charlie, which vouches for them both and assist them in agreeing upon a shared key to use for communications. This analogy is also used by the Needham-Schroeder Symmetric Key Protocol, which introduces a trusted third party (often called a Key Distribution Center (KDC)) to generate and distribute a symmetric session key for Alice and Bob. When Alice wants to communicate with Bob, she notifies the server that she wants to establish a session with Bob. The server computes the session key and encrypts it as twice, one time using Alice's secret symmetric key, and one time using Bob's. The secret keys of the parties are stored in advance at the server, hence making it trusted.

The server then sends the encrypted session key to Alice, which decrypts the key encrypted with her symmetric key, and forward the other to Bob, which decrypts it using his key to obtain the session key. Bob sends Alice a nonce encrypted under the session key to prove to her that he has the session key, which Alice decrypts, performs a simple operation on, and re-encrypts it, before sending it back to Bob, proving to him that she possesses the session key as well. However, this version of the Needham-Schroeder protocol is vulnerable to replay attacks, but can be fixed by using timestamps or include random nonces [51].

The Needham-Schroeder Symmetric Key protocol is the basis for Kerberos, which is a trusted third-party authentication service [52]. Kerberos consists of an Authentication Server (AS) and a Ticket Granting Service (TGS), often hosted in the same KDC, and an Service Server (SS) for providing services to the clients. The authentication model consists of two different credentials: *tickets* and *authenticators*. Tickets are used to securely transmitting the identity of the client which the ticket was issued to between the AS and the SS, and contains information that is used to confirm that the client using the ticket is in fact the same client which it was issued to [61]. After generation, a ticket can be used multiple times until it expires. Authenticators are another type of credentials which is created by the client itself, encrypted, and passed along with the tickets sent from the client to ensure that the presented ticket is issued to it. Figure 2.5 shows the interaction between the different entities in Kerberos, and how the different tickets are passed through the authentication process.

When the client wants to contact another node in the network, it authenticates itself to the AS by providing the AS with its identity. The AS generates a Ticket Granting Ticket (TGT) and encrypts it under the client's secret key, and challenges the client: "If you can decrypt it, you are free to use the ticket to try to obtain a

server ticket from the TGS". When the TGS receives a TGT, it first verifies that it is valid, and that the client is authorized to access to requested service. It then responds with a new ticket for the client to provide when requesting a service from the SS. The protocol is finalized by the client sending the server ticket to the SS, which is verified, before a confirmation message is generated and passed back to the client. If the client is able to successfully verify the confirmation message, the client and server start a session where the server provides the requested service to the client.

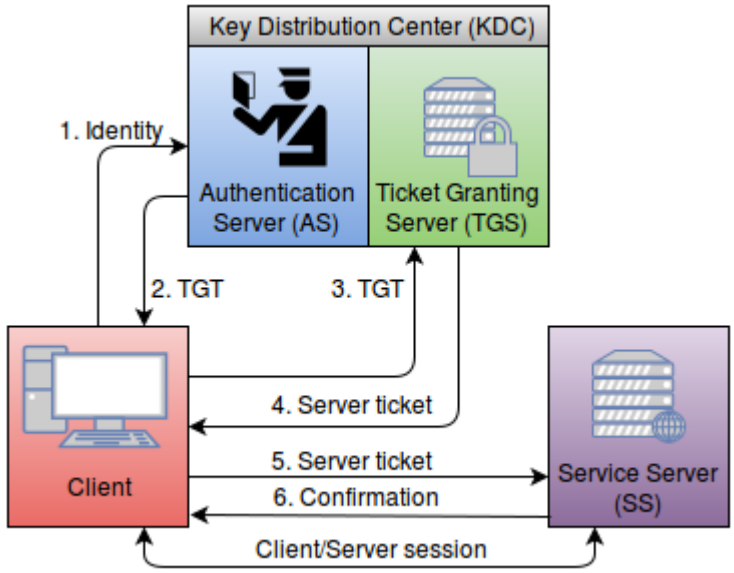

Figure 2.5: Figure of the interaction between the client, the KDC, and the SS in Kerberos.

Public-Key

As described in the section above, the Diffie-Hellman key exchange allowed two parties to agree upon a shared secret without any previous knowledge of each other. This laid the basis for public-key (or asymmetric) encryption, which consists of two keys which are generated mathematically: A private key for decryption, and a public key for encryption. In a public-key encryption system, users who want to send a message to Alice encrypt it using Alice's public key, which is published for the public. When Alice receives an encrypted message, she decrypts it using her private key, as seen in Figure 2.6.

Compared to symmetric encryption, public-key cryptography is significantly more computationally costly. However, the approach of using a public and private key

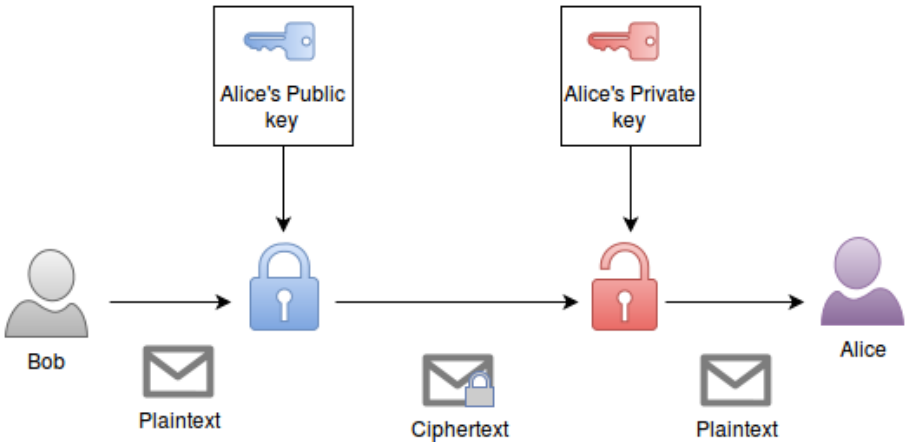

Figure 2.6: Figure of public-key encryption where a message to Alice is encrypted using her public key, and decrypted with her corresponding private key.

for communications is more convenient than using symmetric keys. In the case of a node compromise, only one part of the communication is compromised. The adversary has your private key, and decrypt messages sent to you. It can not, however, decrypt messages that you send to the other party as it does not possess the private key of the other party. Public-key cryptography also allows for authentication of the communicating parties by the use digital signatures, which are used to prove authenticity of a message, as well as proving that the message has not been modified in transmission.

As mentioned, public-key cryptography is significantly more computational expensive than symmetric encryption, which has led to a hybrid solution where a symmetric session key is established and encrypted under the public key of each recipient. Such an approach reduces the computation time of encryption and decryption, giving a more efficient encryption scheme.

Public-key cryptography often involves certificates, which are used to prove ownership of a public key, and contains information about the identity of the owner, and also the digital signature of the party that has issued that particular certificate (often called a CA). When using a certificate, the sender of a message is able to confirm the identity of the recipient, by validating the certificate and the public key. The recipient may have signed the certificate himself, but the most normal approach is to have it signed by a trusted third party, namely a CA, which often are companies specializing in signing certificates and acting as a trusted third party.

2.4.4 Key Establishment Schemes

Symmetric Key

The simplest possible scheme for symmetric key establishment is the network-shared key scheme, where every node in the network possess the same symmetric key which is used for encryption and decryption between all nodes in the network [54]. While being easy to set up, a network-shared symmetric key violates all of the security properties previously described. In addition, it leaves the network vulnerable to node compromises as wireless sensor nodes often are deployed in hostile and unattended areas, where the compromising of one node is equal to the compromising of the entire network [41]. Also, in 802.15.4, the network-shared key scheme is incompatible with replay protection, moving the responsibility of implementing such measures to the higher layers [59].

Pairwise keys is a better symmetric key scheme, where each node pair possesses their own symmetric key for communication between them. This, however, leads to higher memory requirements as the node has to store the symmetric key for possibly $N-1$ nodes (called fully pairwise key schemes), where the number of nodes in the network can be high [54]. Group keying is another approach where groups of nodes share the same symmetric key. This greatly reduces the memory consumption for the devices, and can provide a mild version of compromise resilience. Unfortunately, group keying is not supported in IEEE 802.15.4 [59]. When using pairwise keys, it is possible to provide a certain level of authentication, hence avoid unknown key-share attacks if implemented correctly, as well as key control, given that the two parties use a key exchange protocol where both are contributing to the key. The other properties, however, are not able to achieve using pairwise key establishment schemes.

Random pairwise keys is another scheme that can support the hunt for pairwise keys while still maintaining a modest memory consumption. Assume a "pool" of all the possible pairwise keys that can be created between the nodes in the network. Each node obtains a certain portion of these keys chosen at random (If A gets the pairwise key to B, B naturally also obtains the key for communicating with node A). By randomly delegating keys for different links between nodes, the idea is that there should be a possible path from A to C with high probability, even if they do not possess the key for direct communication, they are able to establish a multi-hop path between them by using the other nodes in the network [43]. This approach eliminates the need for storing $N-1$ keys in each node, and is also more compromise resilient than the (fully) pairwise key schemes as the adversary would only obtain a part of the pairwise keys used in the network if it compromises a node.

Online Servers and Trusted Third Parties

As mentioned, Kerberos is the most notable example of authentication systems utilizing a trusted third party, and it is implemented in most major operating systems such as Microsoft Windows, and systems running Unix such as OS X and Ubuntu. In IEEE 802.11, which is the standard for Wireless Local Area Network (WLAN) communication networks, protocols such as Wi-Fi Protected Access (WPA) and Wi-Fi Protected Access II (WPA2) may utilize Extensible Authentication Protocol (EAP) as their authentication framework, which provides methods for negotiating multiple different key establishment and authentication schemes.

Public-Key

Of the different public-key cryptosystems in use today, the Rivest-Shamir-Adleman (RSA) cryptosystem is the most commonly used, which provides key generation, key exchange, and authentication [63]. RSA is not that often used for actually encrypt data sent between two parties as it is a relatively slow algorithm, therefore, it is normal to encrypt a shared symmetric key under the parties' public keys to use for encryption of data. RSA provides authentication, while other properties involving session keys rely on the protocol used for establishing such keys. The ElGamal cryptosystem, which is based on the Diffie-Hellman key exchange, is another example of a system that is usually operated as a hybrid system utilizing both symmetric keys (for encryption) and public-key cryptography (for establishing symmetric keys).

Elliptic Curve Cryptography (ECC) are systems utilizing the algebraic structure of elliptic curves over finite fields, and can be used to both generate asymmetric key pairs and digital signatures, as well as providing key establishment [10]. Elliptic Curve Diffie-Hellman (ECDH), which is based upon the Diffie-Hellman key exchange is, perhaps, the most notable scheme. One of the benefits with ECC over more commonly used public-key algorithms such as RSA is the reduced key size, which leads to greater memory and energy savings, while providing approximate the same level of security (ECC-160, which has a key size of 160 bits, is equivalent to RSA-1024 in terms of cryptographic strength) [4]. When operating in a mode using *ephemeral* keys, which are temporary keys generated in a key establishment process, ECDH security properties such as key control, known-key security, and forward secrecy. ECDH does, however, not provide authentication, which has to be addressed separately.

2.4.5 Key Establishment Schemes in Wireless Sensor Networks and the Internet of Things

When it comes to WSN applications, symmetric encryption algorithms have historically been the most mature ones [36]. Sensor nodes running 6LoWPAN are powerful enough to implement cryptography standards such as AES-128, providing

such nodes with a satisfying level of encryption [57]. However, there exists several drawbacks with technology utilizing symmetric encryption. For starters, their key exchange protocols are often complex, which is a constraint for the scalability of the network. Also, as the IoT devices are placed in possible hostile environment, they may be physically tampered with by adversaries [41]. If they should successfully compromise one of the nodes, then the security of the entire network may be at stake. Finally, authentication is a rather complex and inconvenient procedure with symmetric encryption involving MACs, which leads to higher requirements for storage space, overhead in messages, and increased energy consumption.

Based on these issues, the research community looked to public-key cryptography, which had previously been considered an unsuitable solution for key establishment and key management in WSNs and other IoT related networks [31, 63]. While improving the security over symmetric key encryption, and also providing easier authentication and higher scalability, regular public-key protocols have issues related to energy consumption due to higher computational complexity, as well as being significantly more time consuming [28]. However, computer hardware specifications improves for every year, as more transistors are placed on data chips, and the processors are becoming more powerful and energy efficient. Public-key cryptography algorithms such as Rabin's Scheme, NtruEncrypt and ECC have proven promising results when implemented efficiently for wireless platforms [36], and especially ECC and its implementation of Elliptic Curve Digital Signature Algorithm (ECDSA) have reduced the time spent on constructing a digital signature from 34 seconds in 2005, to 0.5 seconds in 2009 [57]. There is not, however, any current scheme that provides a clear advantage over others, symmetric or asymmetric, as they all have different advantages and disadvantages.

Following the line of thought where hardware specifications continuously improve, devices are also getting smaller as new "doors" are opened based on the accessibility of better hardware. Currently, companies such as Samsung and Sony are filing patents for so-called "smart" contact lenses, which are allegedly capable of taking pictures of the user's current view, and transmitting the data wirelessly to another device [48, 65]. Processing units on something as small as a contact lens, which has to be transparent and not "bulky" to provide minimum distress on the eye, introduces a whole new level of demands to the energy efficiency of the components. As the data that is transferred from the lens obviously has to be secured in some way (having your "eyes" hacked does not sound especially tempting), we can only assume that the concept of symmetric key establishment is something that will be relevant in the distant future. Therefore, the rest of this thesis will have a special focus on symmetric key establishment.

2.5 Formal Security Analysis

As security protocols grow larger and more complex, they become more and more difficult for humans to analyse. One of the examples of the need for formal security analysis is the Needham-Schroeder Public-Key Protocol from 1978 [50]. The Needham-Schroeder Public-Key Protocol is based on public-key cryptography and was intended to allow two communicating parties to mutually authenticate each other. Throughout this section, the protocol (referred to as the Needham-Schroeder protocol) will be used as an illustrative example to underline the importance of formal security analysis.

One of the pioneering works on security analysis was conducted by Burrows, Abadi and Needham with their Burrows-Abadi-Needham (BAN) logic. BAN logic is a set of rules which can be used to determine whether received information is trustworthy or not, by formally describing the interaction between communicating parties [11]. It showed promising results in finding security flaws and drawbacks for several authentication protocols, but was later abandoned due to the fact that it verified insecure protocols as secure, and in some cases perfectly sound protocols to be insecure [46]. One of the protocols that was formally verified using BAN logic was the Needham-Schroeder protocol.

In fact, 17 years later after being deployed and widely used, Lowe discovered using the automatic tool Casper that the Needham-Schroeder protocol was insecure, and vulnerable to a man-in-the-middle attack [6, 44]. The discovery of that such a flaw had gone unnoticed for so many years puzzled the research community, leading to an increased interest in formal security analysis [20]. Researchers started developing tools for exhaustive search of the problem space of a protocol in order to detect possible abnormalities in protocol behaviour.

In order to conduct formal security analysis, we need a formal model to be able to study the protocol under precise assumptions. Formal security models are abstractions of descriptions of systems, aiming to improve the understanding of the security of the system by simplifying its interpretation. Models can be defined into two different groups: Computational and symbolic models. Computational models are detailed and cryptographic, while symbolic models are more abstract and simple.

By defining a formal security model, we aim to discover and correct errors, incompleteness and inconsistencies in protocol specifications, before they are exploited by adversaries. A protocol specification is a description of the behaviour of the different entities that are allowed to communicate with each other during an execution of the protocol [21]. More precise, a protocol description specifies the different roles in the protocol, each containing a sequential list of the messages that are sent and received from that particular role. It also contains the information of the initial knowledge of the protocol, which are the functions, constants and variables that the

protocol needs to execute correctly. Such a specification is expressed using a formal language, which has well-defined syntax and semantics, for example process algebra, predicate logic, and lambda calculus.

Computational models are another way of mathematically model security protocols, mainly used by cryptographers, hence it holds a more mathematical approach compared to the symbolic model. Computational models are said to be more realistic and detailed, compared to symbolic models. Messages are represented as bitstrings, which are sent into cryptographic primitives (can be seen as "functions") where they are computed on bit-by-bit, and come out as bitstrings [8]. Adversaries in computational models are modelled as powerful arbitrary and probabilistic Turing machines. They do not, however, account for physical attacks such as side-channel analysis and fault attacks, which may be more important as the device-to-device communication increases in the future. Security proofs offered by computational models are often acknowledged as powerful, but often difficult, long, and prone to errors [13]. For constructing proofs, symbolic models are much more efficient as they can more easier be automated to explore the entire problem space.

The Dolev-Yao model is a symbolic and formal intruder model used to prove the security properties of cryptographic protocols. Symbolic analysis considers cryptographic primitives as "black boxes" based on the assumption of perfect cryptography. The black boxes are used to construct terms, which represents the computational operations that the adversary is allowed to perform [8]. While initially being a verification model built for public key protocols, the Dolev-Yao model is also the basis for most of the security analysis done by verification tools that focus on verifying secrecy and authentication properties [21]. The model is built upon three primary assumptions: Perfect cryptography, complete control of network, and abstract terms [25]. Firstly, the Dolev-Yao model assumes that the cryptography is perfect, essentially meaning that the cryptographic system can not be tampered with, and an encrypted message can only be decrypted by the party possessing the corresponding decryption key. The second assumption is that the adversary has complete control over the communication network, hence he is able to observe all messages that are sent between communicating parties, and can inject messages given that he is able to forge its content in a valid matter. Lastly, messages that are sent in the network are to be observed as abstract terms, where the attacker has two possible outcomes: Either he learns the complete content of the message, or he learns nothing at all.

Falsification, presented by Popper in 1934, is the theory of presenting an observation that would disprove the correctness of an alleged theory, or more informally; It is not possible to prove a theory from a single correct observation, but a single observation that contradicts the theory is enough to disprove it [55]. The falsification process in model checking is to formally assess the security properties of the protocol

in order to discover examples that disproves the claimed security by constructing counter-examples. Following in the same line of thoughts, we can perform verification by using formal models and languages to verify a statement (i.e. a security property). In formal security analysis, this is referred to as model checking, which uses the formal model to exhaustively verify whether it meets the alleged security properties [6]. Verification can also be done by constructing mathematical proofs for each of the security properties, proving that the alleged security property is fulfilled.

2.6 Related Work

Move to introduction?

Chapter 3
Symbolic Security Analysis Using Scyther

There exists multiple state-of-the-art tools for performing formal analysis of security protocols, for example Avispa [56], ProVerif [7], Tamarin Prover [47], and Scyther [15]. This thesis uses Scyther as its tool for conducting formal security analysis, and the following chapter will give an introduction to Scyther, how it works, and examples of usages.

3.1 The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols

Scyther is a tool for verification, falsification, and analysis of security protocols developed by Cas Cremers. The tool is based on a pattern refinement algorithm that enables unbounded verification, falsification, will also enabling the tool to perform characterization [17] on the protocol. Scyther allows its users to verify security protocols in two different ways. The first option is to execute Scyther scripts through the command-line interface, which provides an output file containing the results of the protocol verification. Option two is to use Scyther's own Graphical User Interface (GUI), which provides panels for both verification results, and in case of attacks being found, a visual graph of Scyther's proposed attack on the protocol. The most recent release of Scyther was published on April 4, 2014, and is currently available for Windows, OS X, and Linux.

Security protocol specifications are built up of messages that are sent between different entities and computation that is done at either side. Much like a blueprint, these specifications define what a protocol is allowed to do, and how it is allowed to communicate [22]. The blueprint can then be modelled using Scyther, where the entities are converted into roles, the messages are converted into send and receive events, and the security requirements into claim events. Scyther performs complete characterization of a protocol, where roles are broken down to a finite set of representative behaviours by analysing all the possible execution traces where the

events hold. The intuitive idea behind this algorithm is that the set of execution traces together represents all possible ways in which the protocol can execute. These traces are then grouped into patterns, which are partially ordered, symbolic sets of events [16]. From the patterns, Scyther is able to construct a complete set of attack traces for each security claim. When analysing protocols, the realizable traces are compared to the attack traces. If none of these realizable traces of the protocol exhibits an attack trace, then no attack exists, and the security property is verified.

Most protocols can be characterized into a finite set of traces, which enables Scyther to perform *unbounded* verification of the protocol. This greatly differs from the majority of other verification tools which perform *bounded* verification [17, 20]. When performing bounded verification, there exists a finite set of traces that the tool is able to verify, meaning that the entire space of possible states is not covered in the verification process [18]. At best, such a verification can guarantee that the security requirements hold under a finite subset of the actual state-space. Unbounded verification, however, is to verify all possible states, or behaviours, of the protocol which is a great enhancement compared to bounded verification algorithms. In addition to handling an infinite state-space, Scyther is also guaranteed to terminate, which gives it the ability to provide useful results even when it is not able to establish unbounded correctness, or in the scenario of where no attack is found.

As mentioned, a protocol specification contains a set of roles which serves as a blueprint that describes what the protocol is able to do. When executing the protocol, each of the different roles can be executed multiple times, and in parallel with each other by one or more agents [16]. The execution of a role is referred to as a *run*, and defines an unique instance of the protocol with respect to local constraints and the binding between the role and the actual agent acting out the role's behaviour. Scyther allows its users to state security claims which are evaluated as they appear in the protocol trace, either ending in a successful verification of the security property or in a failure. In the presence of a failure, Scyther will provide a concrete attack on the protocol by utilizing one of the attack traces from the pattern, and it will also present an attack graph to illustrate the threat. If the protocol developer is unsure of what types of claims should be stated for each role, Scyther has support for so-called verification of automatic claims, where Scyther will provide general claims such as secrecy for keys and values, and authentication of communicating parties.

Another of the major novelties in Scyther is the possibility for performing so-called multi-protocol analysis, which essentially means analysing multiple protocols that co-exist in the environment. Such an analysis has previously been infeasible because of an incredibly wide state-space, but thanks to Scyther's unique algorithm that operates on an unbounded state-space, multi-protocol analysis is possible.

Scyther is available in two versions. The first version is a plain implementation of Scyther, while the second version also contains options for creating a stronger adversary compromise model than the Dolev-Yao model. The compromise edition contains different Long-term Key Reveal (LKR) rules, which is used for modelling different adversary capabilities such as KCI, wPFS, and PFS, along with support for known-key security.

3.2 Scyther Syntax

The syntax used in Security Protocol Description Language (SPDL) files, which are protocol files that can be run and verified by Scyther, can resemble popular object-oriented languages such as C, C++, or Java. Listing 3.1 contains the structure of a minimum working example of a protocol we call Test, consisting of an outer class defining the protocol and multiple agents (or roles) inside the protocol. In this example, we define that our protocol consists of two communicating parties, U and V, without any specific behaviour.

```
protocol Test(U, V){
  role U { };
  role V { };
};
```

Listing 3.1: Example of the structure of a protocol modelled in Scyther, consisting of roles with different behaviours.

For each of the different roles in the protocol, behaviour can be added as a sequence of send and receive events, as well as variable declarations, constants and claims. For the U role, we can define a simple behaviour as shown in Listing 3.2, where U generates a random nonce `Ru` and sends it to V, before receiving a message from V containing the random nonces `Ru` and `Rv`. All events are labelled with either `send` or `recv` followed by a subscript and a number. The number indicates the message's position in a Message Sequence Chart (MSC), and must be incremented for each message sent.

```
role U{
  fresh Ru: Nonce; # Freshly generated nonce
  var Rv: Nonce; # Variable for receiving a nonce

  send_1(U, V, Ru); # Send message to V containing Ru
  recv_2(V, U, Ru, Rv); # Receive message from V containing
   Ru and Rv. The received Rv value is stored as the
   variable Rv.
};
```

Listing 3.2: Terms can be generated, and sent and received when communicating with other agents.

Typically, a `send`-event has a corresponding `recv`-event at the receiving role with the same number.

```
role V{
  [...]

  recv_1(U, V, Ru); # Receive message sent from U containing
    Ru. The received Ru nonce is stored as the variable Ru.
  send_2(V, U, Ru, Rv); # Send message to U containing the
   received nonce Ru and the freshly generated Rv.
};
```

Listing 3.3: Corresponding events in role V to events in role U.

Along with support for creating fresh nonces, variables, and terms, Scyther also provides a wide set of cryptographic elements such as hash functions, symmetric-key cryptography, and public-key cryptography, as well as declaring user specific types and macros, which are abbreviations of complex expressions. In Listing 3.4, a hash function is used to define a function that generates a Message Integrity Code (MIC) (which is essentially the same as a MAC). On the next line, we have created a macro representing the generation of a pairwise key between U and V. The key is represented as an encryption of the two values `Ru` and `Rv` using a symmetric key that is shared between U and V. Constants and functions defined outside of a role are considered to be global, and available to all of the defined roles in the protocol. When the protocol run reaches the `send_3` event, it looks up the macro for pairwise key and computes it by encrypting the `Ru` and `Rv` values using the symmetric key shared between U and V. `send_3` also contains an example of a MIC of the constant `msg` sent from U to V, which is created by hashing the message and the pairwise key together using

the predefined hashfunction `MIC`.

```
hashfunction MIC; # An hashfunction to represent a Message
    Integrity Code (MIC) generation.

macro PairwiseKey = {Ru, Rv}k(U, V);

role U {
  [...]
  const msg;
  send_3(U, V, {msg}PairwiseKey, MIC(msg, PairwiseKey)
}
```

Listing 3.4: Example on how to use hashfunctions, macros and encryption.

3.2.1 Security Claims

A sequence of events within a role is usually followed by a set of claim events. Claim events are used for describing security properties of a role, for example that some value should be considered secret, or that certain properties hold for authentication. Such claims can be formally verified using Scyther. If the protocol is not instructed with any security claims, Scyther is able to generate the general claims claiming secrecy for keys and values that are sent between roles, as well as authentication for communicating parties, by using the "Verify automatic claims" alternative provided by the GUI.

Secret

The first, most trivial security claim is secrecy. Secrecy expresses that the stated property is to be kept hidden from an adversary, even in the case of where the adversary controls the network used for communication. However, if one of the agents gets compromised by the adversary and the protocol is executed between an honest agent and the adversary, it would in the end learn what was meant to be kept hidden from it [21]. The secrecy claim does not hold for such cases (nor is it intended to), but for each case where the protocol is executed between two honest agents where the secret property is successfully kept hidden from the adversary. For our example protocol, we can claim that the two values `Ru` and `Rv` are supposed to be secret and thereby hidden from the adversary as shown in Listing 3.5. These claims will obviously fail as we have not specified that any encryption should be used on the messages that are passed between the two roles.

```
role U{
  [...]

  # Claims:
  claim_F1(U, Secret, Ru);
  claim_F2(U, Secret, Rv);
};
```

Listing 3.5: Example of how to claim secrecy for terms in Scyther.

Session-Key Reveal (SKR)

Session keys are created at the end of a key establishment process, and are usually used for a session of the communication, before being replaced. When they expire, they are deleted from the system and never used again, limiting the amount of ciphertexts available for the adversary to perform cryptanalysis. In Scyther, the claim SKR is used to identify the session keys in the protocol, and claim that they are secret. SKR can be used by Scyther to model unknown key share attacks (as described in Section 2.4.2), where Scyther will reveal any session key to the adversary, given that its session identifier (i.e. run identifier) differs from the current session's [14]. For the SKR claim to function correctly, Scyther's session-key reveal checkbox needs to be checked in the settings. If the SKR claim is used without enabling this setting, the claim is verified as a regular secrecy claim as defined above.

Aliveness

Aliveness is considered to be the weakest form of authentication, guaranteeing to the party stating the claim (U) that if the protocol is completed successfully, then the communicating party (V) has previously executed the protocol [45]. This does not necessarily mean that U knew he was interacting with V, nor does it mean that V has executed the protocol any time recently.

Weak Agreement

Weak agreement strengthens the authentication form introduced as aliveness. Such an authentication states that the responder in fact was executing the protocol with the initiator (U), and not just having run the protocol at some point in time [45]. By claiming that the protocol holds under the weak agreement, we state that if U successfully completes a run with the intended responder (V), then V also believes that it has previously run the protocol with U. Such a claim would prevent an adversary from acting as a responder by running another run of the protocol in parallel with a run with U, and conducting a man-in-the-middle-attack. The Needham-Schroeder

case presented in Chapter 2 failed on this claim, allowing Lowe to construct his attack.

Non-injective Agreement

Where the authentication provided by weak agreement does not specify which of the two communicating parties acted as initiator and responder, non-injective agreement does. It guarantees that if the initiator (U) successfully completes a run of the protocol, apparently with the responder (V), then V has completed a run with U, where he acted as a responder [45]. This does, however, not indicate that they both have executed exactly one run. There is still a possibility that U has executed multiple runs with a responder which he believed to be V, but may in fact have been communicating with the adversary. Another guarantee provided by non-injective agreement is that if U also sends a set of variables to V in the completed run, then they both agree that the data values correspond to all in the sent set of variables. In Listing 3.6, the example protocol claims that V is "alive", has run the protocol at some time with U, and that during this particular run, it was U and V that were communicating.

```
role U{
  [...]

  # Claims:
  claim_U1(U, Alive);
  claim_U2(U, Weakagree);
  claim_U3(U, Niagree);
};
```

Listing 3.6: Example of how to claim authentication by use of alive, weak-agreement, and non-injective agreement.

Non-injective Synchronization

Synchronization requires that all protocol messages occur in the expected order with their expected values, and that the behaviour is equivalent to as if the protocol was executed without the presence of any adversary [23]. The *injective synchronization* property states that the protocol executes as expected over *multiple* runs, claiming that it is not possible for an attacker to use information from previous runs to disrupt the current protocol execution [21]. Such an attack is known as a replay attack, and is used by an adversary to inject traffic into the protocol execution to induce undesirable or unexpected behaviour. Scyther, however, does not support this enhanced form of synchronization, hence it strongest type of synchronization is *non-injective synchronization*. Because of this, Scyther is not able to verify whether

or not a protocol is secure against replay attacks. Listing 3.7 contains an example on how to claim non-injective synchronization for the example protocol.

```
role U {
  [...]

  # Claims:
  [...]

  claim_U4(U, Nisynch);
}
```

Listing 3.7: Claim for declaring non-injective synchronization in Scyther.

Running, Commit

Running and commit signals can be used as a form of authentication for some variables from a set of terms sent in a message. By using these signals (in Scyther modelled as claims), we can verify that a variable sent from U to V, and then returned to U, has not been changed from its initial value during transmission. From a formal view, this can be seen as non-injective agreement over a set of terms [19].

The expression $claim(V, Running, U, Ru)$ denotes that V is currently executing the protocol with U, and with the nonce Ru. In U's case, $claim(U, Commit, V, Ru)$ indicates that the protocol as reached a point where authentication is claimed (U has completed the protocol run with V), where Ru is the variable that it claimed to be exchanged during this part of the run [58]. Usually, the *commit* claim is stated at the end of the protocol run. For the correctness of the *commit* claim to hold, it requires that the *running* signal is added in the communicating role, and preceding the *commit* claim in the trace.

This pattern is a scheme for authentication properties, but it also allows for expressing authentication for additional information specific to a certain part of the protocol run, for example some variable inside the message. Occurrence of a *commit* signal in U's protocol run guarantees that a corresponding *running* signal has previously occurred in V's protocol run, which guarantees that the received message containing Ru must have been transmitted by V [58]. Listing 3.8 contains an example of how we can claim non-injective agreement over a variable, in this case the nonce Ru.

```
role U{
  [...]

  send_1(U, V, Ru);
  recv_2(V, U, Ru, Rv);
  claim_U5(U, Commit, V, Ru); # Authentication over the term
    Ru is claimed
};

role V{
  [...]

  recv_1(U, V, Ru);
  claim_V6(V, Running, U, Ru); # Claim that V is currently
   running the protocol with U with the value Ru
  send_2(V, U, Ru, Rv);
}
```

Listing 3.8: Example of running and commit claims in Scyther to provide authentication for a set of terms.

3.3 Defining an Adversary Compromise Model

Formal adversary models are described in Section 2.5. Compromising of long-term keys can, for example, allow an adversary to recover previous session keys (and future) and decrypt the traffic if the protocol does not provide forward secrecy. Another option is for the adversary to perform KCI where it impersonates the victim towards other agents, or impersonate other entities in communication with the victim. Scyther allows for customizing different adversary models through its settings for a adversary compromise model, which enables a strong Dolev-Yao style adversary with support for verifying security properties such as PFS, wPFS, KCI and known-key security. These security properties are decomposed in Table 3.1 into their basic property, type of security property, and what adversary model (which will be elaborated in the next section) provides them.

Security property	Basic property	Adversary model
KCI	Authentication	{LKR, Actor}
PFS	Secrecy	{LKR, After}
wPFS	Secrecy	{LKR, Aftercorrect}
Known-Key Security	Session key secrecy	{SKR}

Table 3.1: Relationship between security properties and the adversary models in Scyther [5].

The initial adversary in the Dolev-Yao intruder model has access to the long-term keys of the communicating parties that do not participate in the current run of the protocol. In other words, if A and B are communicating with each other, then the adversary has access to C's private long-term key during the execution of the protocol. Scyther's initial intruder model, however, does not have access to these keys without directly specifying it in its LKR settings. When enabling {LKR, Others}, the adversary is identical to the Dolev-Yao intruder model, and has access to the long-term private keys for other agents than the those currently involved in the protocol execution [5].

Section 2.4.2 mentioned KCI, where an adversary in possession of A's long-term private key is able to impersonate A when communicating other agents, or impersonate other entities when communicating with A. Such an attack can be performed by the adversary after enabling {LKR, Actor} in Scyther's adversary compromise model. Forward secrecy is the security property where previous communication is protected in the case of compromising of the long-term key, and is enabled in the adversary model by specifying {LKR, Aftercorrect} and {LKR, After}. These properties restricts the compromise of long-term keys to only occur after the protocol execution [5]. {LKR, Aftercorrect} is used to model wPFS, and is the weaker case of forward secrecy, where the adversary is considered to be *passive*. For the adversary model, this would restrict it from both injecting messages and obtaining the private keys of the communicating parties after the protocol run. {LKR, After} models an *active* adversary capable of actively interfering with the protocol during it run while obtaining the long-term private keys, hence protocols able to provide secrecy in the present this adversary is said to provide PFS.

Figure 3.1 illustrates the different LKR rules in two dimensions; *when* a compromise occurs, and *whose* long-term keys are compromised. The rows indicate when the compromise occurs, and can either be before the run, during, or after. Columns describe whose keys are compromised, where actors are agents that execute the protocol, peers are communicating partners during the execution, and others are agents not participating in the protocol run. The different capabilities are captured

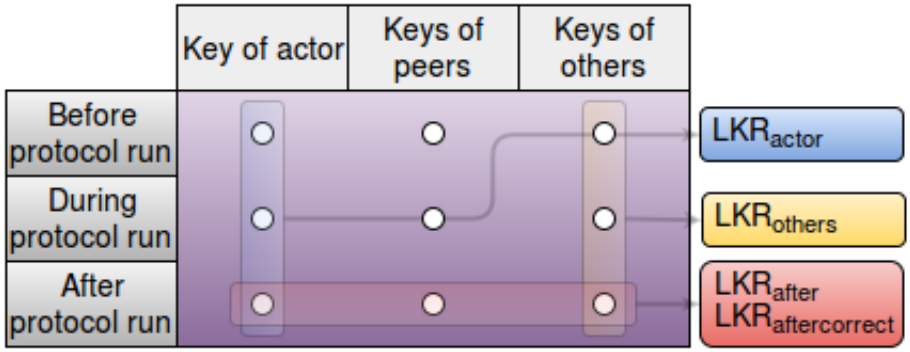

Figure 3.1: Mapping of LKR rules. Rows display when the compromise occurs, columns display whose data gets compromised, and the boxes captures the capabilities of the different adversaries, which are labelled to the right [5].

and labelled as different LKR rules, as shown on the right hand side of the figure.

In addition to compromising long-term keys, Scyther is also able to model the security property for known-key security. By enabling the SKR rule, the adversary is allowed to obtain all session keys whose session identifier (the identifier of that particular run of the protocol) differs from the current run's identifier.

3.4 Scyther's Graphical User Interface

As mentioned, Scyther provides a GUI for easily understanding and assessing the security of a protocol. If we continue our example of the protocol Test from the section on Scyther's syntax, we now want to verify all the stated security claims. By using the GUI, we can configure the verification process by stating a maximum number of runs, the adversary compromise model, as well as more advanced options for how to prune the search space. Scyther provides three options in its GUI: verification of claims, verification of automatic claims, and characterization of the protocol [17]. Figure 3.2 contains the results of running a verification of the claims previously described for a secure protocol.

When no attacks are found, Scyther provides one of two comments: *No attacks within bounds* or *No attacks*. In the first case, Scyther was not able to find any attacks within the bounded state-space, meaning that it may or may not be an attack in the unbounded state-space. The latter, however, states that there was not found any attacks within both the bounded and the unbounded state-space. In this case, Scyther can construct a formal proof of the absence of any attacks, hence the security property is successfully verified. Scyther returns an `Ok` status code and a *Verified* message for each claim that is successfully verified. As we see in Figure 3.2, Scyther

Claim				Status		Comments
TEST	U	TEST,U1	Alive	Ok	Verified	No attacks.
		TEST,U2	Weakagree	Ok	Verified	No attacks.
		TEST,U3	Niagree	Ok	Verified	No attacks.
		TEST,U4	Nisynch	Ok	Verified	No attacks.
		TEST,U5	Commit V,Ru	Ok	Verified	No attacks.
	V	TEST,V1	Alive	Ok	Verified	No attacks.
		TEST,V2	Weakagree	Ok	Verified	No attacks.
		TEST,V3	Niagree	Ok	Verified	No attacks.
		TEST,V4	Nisynch	Ok	Verified	No attacks.

Figure 3.2: Results of a verification process using Scyther where all claims are successfully verified.

is not able to find any attacks on the protocol. To illustrate the case of Scyther actually finding an attack, we try to verify the claims introduced in the paragraph on secrecy in Section 3.2.1, claiming that `Ru` and `Rv` are secret. In our example protocol, both nonces are sent in plaintext between U and V, hence this claim will naturally fail, as seen in Figure 3.3.

Claim				Status		Comments	Patterns
TEST	U	TEST,F1	Secret Ru	Fail	Falsified	Exactly 1 attack.	1 attack
		TEST,F2	Secret Rv	Fail	Falsified	Exactly 1 attack.	1 attack

Figure 3.3: Results of a verification process using Scyther when a claim fails.

The status *Falsified* states that the claim is provable false. When a claim is proved to be false, Scyther will also provide a comment; either *At least X attack(x)*

or *Exactly X attack(s)*. In the first case, X attacks where found by Scyther, but the search is not able to detect whether or not there may be other attacks as well. In the other case, Scyther can prove that within the given state-space, there are exactly X attacks.

Whenever Scyther finds an attack on a protocol, it will also provide a concrete illustration of the attack as a graph. Figure 3.4 shows an example of such a graph. The top box for each vertical alignment of boxes describes the run, which is confined inside the grey boxes. It contains a description with the identifier of the run, instance type (i.e. what type of role it is running as), which agents it assumes it is communicating with, and also what fresh values that are generated and instantiated in the run [19]. Boxes symbolises events in the different runs, connected by arrows which symbolises ordered constraints. Incoming arrows do not indicate that the messages is sent directly in this step, but is merely an ordering stating that this message can only be received *after* something else has happened. For example, in Figure 3.4, the `recv_2` event in `Run 1` can only happen after Bob has sent his message in the `send_1` event.

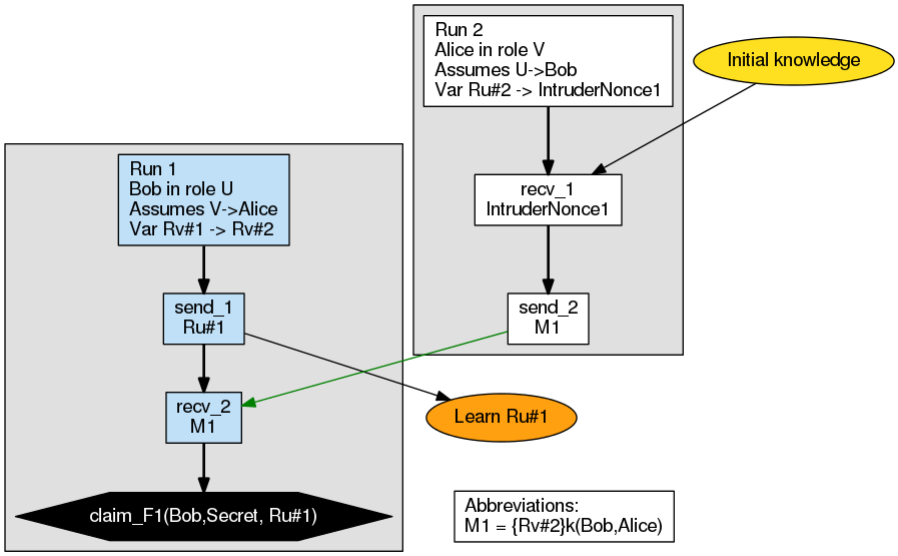

Figure 3.4: When Scyther finds an attack on a protocol, if will also provide a graph of the attack.

Arrows in Scyther graphs can be coloured differently. Red arrows indicate that the sent message does not correspond to the received message, which means that the adversary used some information from the sent message in order to construct the one that is received [19]. Green arrows indicates that the sent message is identical to

the received message. The last possible color (other than black, which does not carry any specific information) is yellow. Yellow arrows indicate that the two parties agree upon the message that is exchanged between the two, but do not agree upon who was the sender and receiver during the exchange.

When a message is sent, it is instantly obtained by the adversary. Initial knowledge (or intruder knowledge) corresponds to the intuition that the intruder is able to generate fresh values of any type, which it in Figure 3.4 uses to generate the nonce that is sent in the `send_2` event. The green oval shape indicates where the adversary obtains the information which falsifies the claim, which in this example is the `Ru` value that is sent in plaintext. The two last boxes in the graph are the black box at the bottom of `Run 1` which contains the claim that is falsified by Scyther, and the white box to its right which contains abbreviations of the messages that are passed between roles to increase readability of the graph.

Chapter 4
Three Protocols for Key Establishment in 6LoWPAN

4.1 Adaptable Pairwise Key Establishment Scheme (APKES)

Adaptable Pairwise Key Establishment Scheme (APKES) is a proposed protocol by Krentz et al. for handling key establishment and key management in 6LoWPAN [41]. It is currently implemented in the operating system Contiki, which is targeted at the sensor network community. As previously described, 6LoWPAN is a protocol stack for integrating WSNs running on 802.15.4 with IPv6 networks, and enables the nodes in the network to communicate with each other or remote hosts over IP. APKES provides a framework for establishing pairwise keys for nodes in 6LoWPAN networks. The advantage with pairwise keys over other key schemes such as a network-shared key is related to node compromises. In 6LoWPAN networks, devices are often placed in potential hostile and unattended areas, greatly increasing the possibility of being tampered with by attackers.

In the case of a network shared key, the whole network would be compromised in the event of a node compromise. Also, the attacker would be able to add new nodes to the network, as the upper-layer protocols rely on the 802.15.4 security sub-layer which is able to filter out replayed packets and prevent injection, but not discover node compromises [41]. A solution to the tampering problem could be to construct tampering-proof nodes, but this is expensive and difficult, hence not a preferable solution [2]. Pairwise keys, however, would only compromise the communication going to or from that particular node, and the establishment of such keys is the main focus of APKES.

Figure 4.1 illustrates how APKES is implemented at the link layer along with the 820.15.4 security sublayer. In its implementation, APKES introduces three special messages which are used in the key establishment process, namely `HELLO`, `HELLOACK`, and `ACK` [41]. These are defined as 802.15.4 command messages, which are only processed by the data link layer (i.e they are not passed to upper layers),

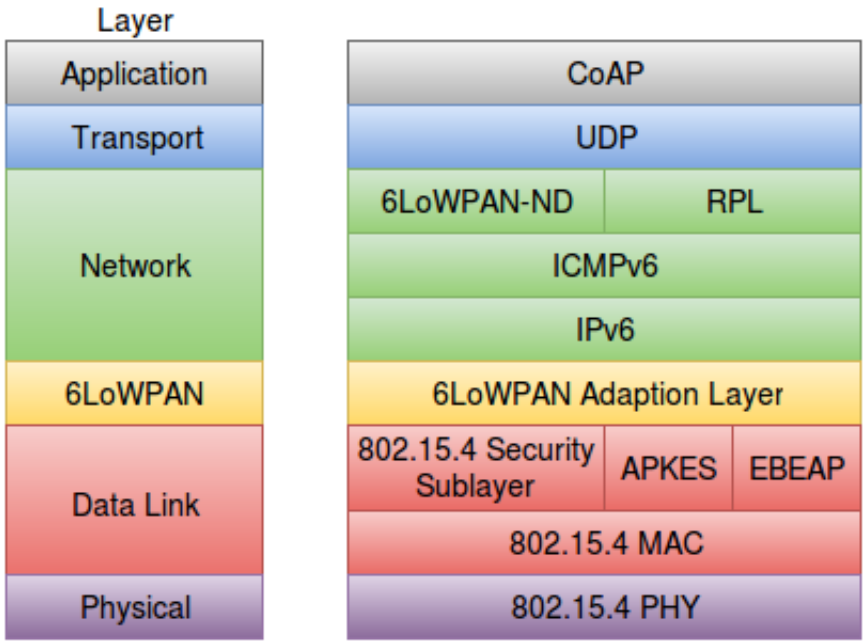

Figure 4.1: APKES is positioned in the data link layer in the 6LoWPAN stack expanding the 802.15.4 security sublayer [41].

hence APKES is able to establish pairwise keys for networks building on 802.15.4 independently from the protocols running in the upper layers.

APKES provides a "pluggable" key establishment scheme for 6LoWPAN networks using pairwise keys, where the developer of a 6LoWPAN network picks an appropriate key establishment scheme and delegates APKES into handling the key establishment with other nodes [41]. As there is really no superior scheme for 6LoWPAN networks, the use of pluggable schemes enhances the overall usability of the protocol, as the developer can use the most appropriate scheme based on the challenges he faces. The only function of the plugged-in scheme is to feed APKES with the shared secret for the communicating nodes, and APKES will handle both key establishment and key management. Examples of pluggable schemes that have been suggested for APKES are Localized Encryption and Authentication Protocol (LEAP) [66], Blom's Scheme [9], and random pairwise keys [12]. In the case of random pairwise keys, path key establishment has to be implemented in addition to APKES.

During the key establishment process, a responding node goes from not being a neighbour, to a tentative neighbour, before ending up as a permanent neighbour, given that the key establishment was successfully executed. The change of neighbour

status is implemented to prevent Denial of Service (DoS) attacks on nodes by flooding them with messages for starting key establishments (`HELLO` messages), which would force them to reply to each message (denoted as `HELLOACK`), potentially drain their battery. Also, injecting and replaying these replies could also aid an attacker in draining the network's nodes for battery. Upon receiving a `HELLO` message, the responder (B) checks if the initiator (A) is already a neighbour, and that it has available space in its list of tentative neighbours, which is limited to M_t neighbours.

APKES modifies the security sub-layer of 802.15.4 to instantly discard data frames that arrives from non-permanent neighbours, only accepting `HELLO`s, `HELLOACK`s, or `ACK`s. By limiting the number of tentative neighbours, B is protected against consecutive `HELLO` messages, which are discarded without being processed when the number of tentative neighbours exceeds M_t. The list of tentative neighbours is processed for each `HELLO`, where neighbours whose expiration time has expired are deleted. The status change from tentative to permanent neighbour is potentially done upon receiving a valid, non-replayed `HELLOACK` or `ACK` from a non-permanent neighbour.

4.1.1 Protocol specification

Key establishment in APKES consists of a three-way handshake, as described in Figure 4.2. When a node A in a 6LoWPAN network running APKES wants to establish contact with other nodes, it broadcasts an unauthenticated `HELLO` message containing a random nonce Ra. Upon receiving a `HELLO`, B computes a random nonce, Ra, as well, and stores the concatenation of the two. B then waits for a random time T_w. The waiting period is introduced to avoid flooding A with responses, as there may be an unknown number of nodes that received the broadcasted `HELLO` message. After T_w, B loads its key $K_{B,A}$ from the pluggable key scheme, and uses this key to authenticate a `HELLOACK` message containing the computed R_B nonce and the received R_A. MICs are generated by the 802.15.4 security sublayer, through the use of CCM*, which is a modified version of the regular CCM allowing for the payload of the frame to be encrypted using AES with a 128-bit key [41]. APKES uses $K_{B,A}$ to authenticate the `HELLOACK`, and sends it to A. Afterwards, B obtains the pairwise key $K'_{B,A}$ for future communication with A, by plugging $K_{B,A}$ it into the AES algorithm along with the two nonces.

When A receives a `HELLOACK` message, it verifies the attached MIC by extracting its key $K_{A,B}$ from the pluggable scheme and computing the MIC for the concatenation of R_A and R_B. A then computes the pairwise key for communicating with B by plugging it into the AES algorithm. A also checks that the R_A value has not been tampered with, and is equal to the value it initially sent in its `HELLO` broadcast. The three-way handshake ends with A sending an `ACK` to B that is authenticated using

the pairwise key $K'_{A,B}$. When B receives the `ACK`, it verifies the MIC by using its derived pairwise key $K'_{B,A}$. After this process, A and B have successfully agreed upon a shared pairwise key where $K'_{A,B} = K'_{B,A}$, which is to be used for encrypting all future communication between the two nodes.

Three-way handshake in APKES

$$
\begin{aligned}
& A : \textit{Generate } R_A \textit{ randomly} \\
A \rightarrow * : & \texttt{HELLO}\langle R_A \rangle \\
& B : \textit{Generate } R_B \textit{ randomly. Wait for } T_w \leq M_w \\
& B : K_{B,A} \textit{ from pluggable scheme} \\
B \rightarrow A : & \texttt{HELLOACK}\langle R_A, R_B \rangle K_{B,A} \\
& B : K'_{B,A} \; = \; AES(K_{B,A}, R_A || R_B) \\
& A : K_{A,B} \textit{ from pluggable scheme} \\
& A : K'_{A,B} \; = \; AES(K_{A,B}, R_A || R_B) \\
A \rightarrow B : & \texttt{ACK}\langle\rangle K'_{A,B}
\end{aligned}
$$

Figure 4.2: Figure of the messages sent between communicating parties during APKES' three-way handshake.

4.1.2 Assumptions of Security Properties

One of the focuses of APKES is to provide authentication of parties during the key establishment process. By inspecting the messages that are exchanged between the two parties in Figure 4.2, we observe that no encryption is involved in the handshake, but messages are authenticated by the use of MICs. These MICs are either computed using $K_{A,B}$ (the pre-shared secret) or $K'_{A,B}$ (the established pairwise key). Therefore, we can assume that entity authentication has to hold for the two communicating parties. Also, as mentioned in Section 2.4.2, *implicit* and *explicit* key authentication are two of the other attributes within authentication. For a three-way handshake such as the one used by APKES, the initiator achieves *implicit* key authentication, while the responder (B) achieves *explicit* key authentication. As the pairwise key is computed from the two nonces that are shared between A and B, and the secret from the pluggable scheme (which we assume is secure), both know that the only parties that can compute the pairwise key is those possessing the pre-shared secret, giving them both implicit key authentication.

B also receives an `ACK` which is authenticated using the pairwise key $K'_{A,B}$, effectively meaning that A has computed the pairwise key, and which B can confirm by

checking the attached MIC, hence it can be said to achieve explicit key authentication. From A's point of view, however, it has no confirmation of that B has in fact computed the pairwise key, other than it knows it is supposed to (and has to in order to verify the authenticity of the `ACK`. Also, as APKES is a key establishment protocol, the established key is of course claimed to be secret from the adversary.

4.1.3 Weaknesses and Challenges with APKES

APKES establishes a shared symmetric key between nodes, which is used to encrypt and decrypt data that is sent between them. One issue that the protocol does not address is the case where, for some reason, the node is forced to do a reboot. To avoid replay attacks, a node needs to keep control over the frame counters of the nodes it communicates with. These frame counters need to be swapped from the RAM memory of the device to a non-volatile storage over time. Such storages are for most 802.15.4 devices flash memory, making the swapping process both energy and time consuming [40]. In the Contiki operating system (where APKES is implemented), reboot commands are issued whenever processes get stuck or when replacing the battery of the device [26]. In the case of a reboot without storing the frame counter, neighbouring nodes would simply discard all messages from the node, as the frame counter would start at zero, and the frames would be considered replayed. Another issue with storing anti-replay data is that APKES does not remove information of disappeared neighbours (nor does it discover that a node has left the neighbourhood), which may unnecessarily seize a large part of the node's memory over time.

In addition to the weaknesses related to frame counters and storing anti-replay protection data, APKES has issued related to its usage of temporary and permanent neighbours. As mentioned, the life cycle of a neighbour node ranges from not being associated at all, to becoming a temporary, and finally a permanent neighbour during the key establishment process. However, APKES discards `HELLO` messages from permanent neighbours to prevent DoS attacks. This means that if a neighbour reboots, it goes into a deadlock with previous neighbours, where it is not able to establish any new keys with these nodes as its `HELLO`s would silently be discarded. The broadcasting of `HELLO`s only occurs immediately after the node is booted up, which means that after the node is up and running, it will not attempt to connect to any new neighbours that may have been deployed afterwards. One can argue that it is the responsibility of the post-deployed nodes to establish contact with "early birds", but deployed nodes should nevertheless be able to discover new nodes during runtime.

These issues can be applied into a real life scenario to better understand the limitations of the protocol. Assume a WSN for medical applications with multiple devices attached to certain medical equipment. Networks around patient are not

necessary stationary, as they may be moved to different facilities in the hospital. Also, certain medical equipment are very expensive, which would encourage reuse and reassignment of the devices (i.e. supporting mobility for these devices may be an important factor for certain cases).

4.2 Adaptable Key Establishment Scheme (AKES)

The Adaptable Key Establishment Scheme (AKES) aims to improve and fix the weaknesses that was introduced in APKES, and is currently implemented in the Contiki operating system [40]. Its main goal is to establish session keys between devices in a 6LoWPAN network, while being able to withstand reboots and movement from one network to another. As described in Section 4.1.3, APKES suffered from issues when rebooting the device, and it was not able to provide any mobility. Most of these issues can be solved by one "simple" adjustment: Establishing session keys between nodes instead of long-term keys. By establishing session keys, MICs from previous sessions would be invalidated, which enables node to delete data used for providing replay protection (such as frame counters), and will also filter out old frames. Also, this removes the problem related to frame counters being reset after a reboot, as mentioned in Section 4.1.3.

AKES builds on the approach from APKES, where the underlying scheme is pluggable provides AKES with the pre-shared secret between nodes. Before an 802.15.4 node is able to run AKES, addressing information (which uniquely identifies a node within an 802.15.4 network and is used by the pluggable scheme when establishing the shared secret) and keying material has to be preloaded into it. AKES also has access to the same command frames `HELLO`, `HELLOACK`, and `ACK`, which are used to establish session keys, and only processed by the data link layer. Figure 4.1 describes where APKES is implemented in the 6LoWPAN stack, and as AKES is merely an improvement over APKES, it is implemented on the same layer as the 802.15.4 Security Sublayer.

As in APKES, AKES also utilizes a differentiation between non-neighbours, temporary neighbours, and permanent neighbours. When a node sends a `HELLO` it will obtain a status as a temporary node at the receiver. This status will be changed to permanent upon receipt of an authentic `ACK` message as part of the final step in the session key establishment. Keep in mind that one of the issues with APKES was the deadlock state rebooted nodes would start in with previously permanent neighbours. In AKES, permanent neighbours who transmits a `HELLO` message will obtain a status as a temporary neighbour in addition to its old permanent neighbour status, until the `ACK` is received. After receiving the `ACK`, the permanent neighbour status is deleted and the temporary is turned into a permanent one, which effectively renews the session between the two nodes. When a permanent neighbour(i.e. a

session key) is established, the neighbour is assigned an expiration time where the key becomes invalid. The lime time of a session is, however, prolonged for each received, authentic frame from the particular session, and can also be prolonged by issuing certain commands.

AKES introduces two tasks for preventing deadlocks and increasing mobility for devices, while still keeping DoS attacks in mind: Periodically pinging its permanent neighbours to delete disappeared nodes, and discover new neighbours by routinely broadcasting `HELLO`s. When a session with a neighbour expires, the node issues an authenticated `UPDATE` command and sends it to the node, which potentially responds with an `UPDATEACK`. A received `UPDATEACK` leads to both parties of the session extending the lifetime of their key, while the absence of such an acknowledgement, it will try a few times before eventually giving up and deleting the neighbour from its view of the network.

Trickle, which is an algorithm for distributing information in WSNs [42], is adopted by AKES for discovering new neighbours in a routinely matter. The challenge is to define *how* often the node should broadcast `HELLO`s in order to discover new nodes and changes to the network topology, which Trickle aim to solve by applying different network statistics into its algorithm.

Three-way handshake in AKES

$$
\begin{aligned}
A &: \textit{Generate } R_A \textit{ randomly} \\
A \rightarrow * &: \texttt{HELLO}\langle PAN_A, ID_A, R_A, C_A\rangle \\
B &: K_{B,A} \textit{ from pluggable scheme} \\
B &: \textit{Generate } R_B \textit{ randomly. Wait for } T_w \leq M_w \\
B &: K'_{B,A} \;=\; AES(K_{B,A}, R_A||R_B) \\
B \rightarrow A &: \texttt{HELLOACK}\langle PAN_A, ID_A, PAN_B, ID_B, R_B, I_{A,B}, C_B, P_A\rangle K_{B,A} \\
A &: K_{A,B} \textit{ from pluggable scheme} \\
A &: K'_{A,B} \;=\; AES(K_{A,B}, R_A||R_B) \\
A \rightarrow B &: \texttt{ACK}\langle PAN_B, ID_B, PAN_A, ID_A, I_{B,A}, C_A\rangle K'_{A,B}
\end{aligned}
$$

Figure 4.3: Figure of the messages sent between communicating parties during AKES' three-way handshake.

4.2.1 Protocol Specification

In AKES, the key establishment process consists of a three-way handshake where the two nodes establish a session key, as described in Figure 4.3. Initially, the node

A broadcasts a `HELLO` message to its neighbours containing a randomly generated nonce value R_A along with the identity of the node, its Personal Area Network (PAN) address, and the frame counter C_A. The `HELLO` broadcast is authenticated using Easy Broadcast Encryption and Authentication Protocol (EBEAP) [41], which is a protocol for authenticating broadcast frames in 6LoWPAN networks, or a pre-distributed group session key. When B receives a `HELLO` broadcast, generates a random nonce as well, denoted as R_B. It then proceeds to request the shared secret $K_{B,A}$ from its pluggable scheme, and uses this key to derive the pairwise session key $K'_{B,A}$ as $AES-128(K_{B,A}, R_A||R_B)$. B then crafts a `HELLOACK` response which is sent to A containing R_B. The `HELLOACK` is authenticated by adding a MIC generated with $K_{B,A}$, in addition to B's PAN address, identity, and other values related to frame counters and EBEAP authentication. In the response, B attaches a field P_A as well to indicate whether or not A is currently registered as a permanent neighbour of B, and is also capable of piggybacking group session keys. If the P_A field is set, A can choose to disconnect from the establishment, which would be normal if the `HELLO` was just a routine broadcast. Upon receiving the `HELLOACK`, A validates the attached MIC by computing the pairwise session key $K'_{A,B}$ in the same manner as B. After this, future communication is encrypted using the shared pairwise session key until it expires.

4.2.2 Assumptions of Security Properties

4.3 Secure Authentication and Key Establishment Scheme

The third, and last protocol which will be discussed in this thesis, is the Secure Authentication and Key Establishment Scheme (SAKES). SAKES is claimed to provide secure authentication and key establishment for nodes in a device-to-device network running on 6LoWPAN [35]. Previous described protocols such as APKES and AKES enables devices to directly communicate with each other without any previous authentication have taken place. The architecture in SAKES as seen in Figure 4.4 consists of end devices, 6LoWPAN routers, 6LoWPAN border routers, and remote servers providing services to the devices. End devices are typically sensors, with very limited computational power. Routers are more powerful,

Authentication -works

Key establishment - assumptions - what is wrong - correct to something that is modellable

4.3.1 Protocol Specification

4.3.2 Assumptions of Security Properties

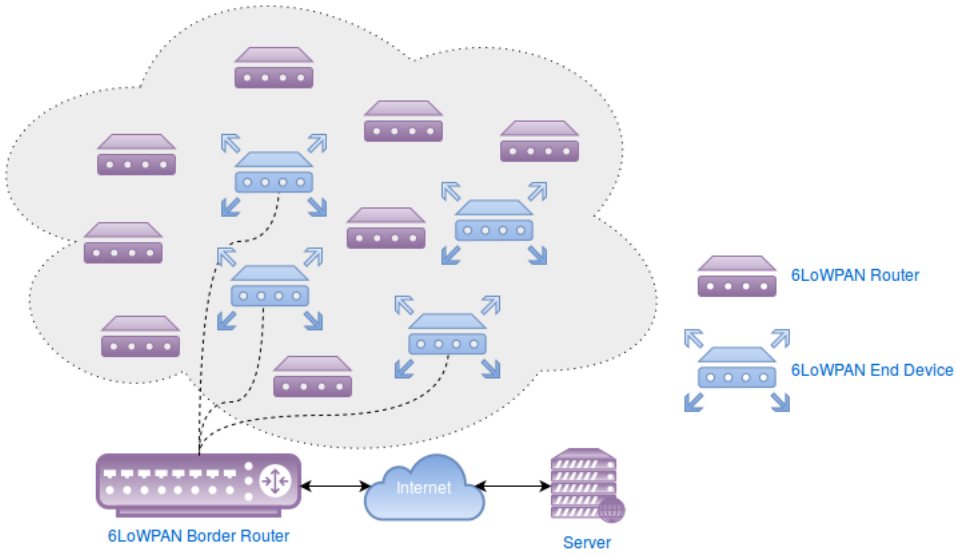

Figure 4.4: Caption goes here

Authentication in SAKES

$$
\begin{aligned}
&A: \textit{Generate } N_A \textit{ randomly}\\
A \rightarrow B &: N_A\\
&B: \textit{Generate } N_B \textit{ randomly}\\
B \rightarrow A &: \{N_A\}_{K_{AB}}\\
&A: \textit{Construct } C_A: \{ID_A, ID_B, ID_D\}_{K_{AC}}\\
A \rightarrow B &: \{C_A, ID_A, N_A\}_{K_{AB}}\\
B \rightarrow C &: \{C_A, ID_B, N_B\}_{K_{BC}}\\
&C: \textit{Verify the identity of } A, B, \textit{ and } D\\
&C: \textit{Construct } C_C: \{ID_A, ID_B, ID_D\}_{Kpriv_C}\\
&C: \textit{Generate } N_C \textit{ randomly and a public key pair } (Pk_B, Sk_B)\\
C \rightarrow B &: \{N_C, C_C, Pk_B, Sk_B\}_{K_{BC}}\\
C \rightarrow A &: \{ID_B, N_C\}_{K_{AC}}
\end{aligned}
$$

Figure 4.5: Figure of the messages sent between the end device (A), router (B), and authentication module (C) during SAKES' authentication phase.

Key Establishment in SAKES

$$
\begin{aligned}
&B \rightarrow D : \{C_C, N_B, P_K, B\}_{Sk_B} \\
&\quad D : Calculate\ Session\ Key\ (SK_D) :\ SK_D = g^{Pk_C * Sk_D} \mod P \\
&\quad D : Generate\ N_D\ randomly \\
&D \rightarrow B : \{Pk_D, P, g, N_D, \}_{Sk_D} \\
&\quad B : Calculate\ Session\ Key\ for\ A\ (SK_A) :\ SK_A = g^{Pk_D * Sk_C} \mod P \\
&B \rightarrow A : \{SK_A, N_B\}_{K_{AB}} \\
&Claim : SK_A = SK_D
\end{aligned}
$$

Figure 4.6: Figure of the messages sent between communicating parties during SAKES key establishment between the end device (A), the 6LoWPAN router (B), and the remote server (D).

Chapter 5
Verification of Three Key Establishment Protocols

5.1 Modelling Security Claims

As mentioned in Section 2.4.2, key establishment schemes desire certain security properties. In the verification of the security protocols of this thesis, the following properties are verified: *Entity authentication*, *Implicit key authentication*, *Explicit key authentication*, *Known-key secrecy*, *Key control*, and *Secrecy of key*. As mentioned in Section 2.4.2, symmetric key establishment schemes are not resilient against KCI attacks, and do not provide forward secrecy.

Entity authentication: Entity authentication between nodes corresponds to the security claim `Alive`, and can also be verified through stronger claims such as `Weakagree`. MORE

Implicit key authentication Implicit key authentication is modelled through the settings of the adversary compromise model described in Section 3.3. This property is modelled by allowing the adversary to obtain the long-term keys and impersonate anyone except for the nodes that are supposedly establishing keys.

Explicit key authentication Is achieved when the protocol satisfied both implicit key authentication and key confirmation. This is modelled through the security claim for *non-injective agreement* denoted as `ni-agree`, but can also be modelled by using `running` and `commit` claims.

Known-key security By revealing session keys to the adversary after usage (i.e. the session key is expired, and will never be used again) known-key security can be modelled. This is done by setting the *Session-key reveal* rule in the adversary compromise model.

Key control Scyther has no support for verifying key control. Therefore, this security property has to verified by hand.

Secrecy of key To model a key (or any other property) as secret, the `secrecy` claim is used in Scyther.

Forward secrecy Both PFS and wPFS are related to active adversaries, and is modelled through the adversary compromise model, where rules for leaking the long-term private key which the session keys are derived from can be enabled.

Key compromise impersonation KCI is also a property related to an active adversary, and is therefore available through the adversary model where the adversary can be allowed to obtain the long-term private key of the actors.

5.2 Formal Security Analysis of APKES

The APKES protocol is modelled using Scyther, and can be viewed in its entirety in Appendix A.1. The protocol is modelled as two roles, the initiator A and the responder B, agreeing upon a pairwise key through the message exchange that is presented in Figure 4.2. There is not specified any concrete type of pluggable scheme (i.e. the scheme where APKES obtains the shared secret between two nodes), hence we assume that whatever scheme is used is secure. In the model, the shared secret obtained from the pluggable scheme has modelled by using Scyther's built-in support for shared symmetric keys, where the two nodes A and B initially share a symmetric key.

5.2.1 Security Claims

By taking starting point in the protocol specification from Section 4.1.1 and the alleged security properties from Section 4.1.2, the protocol is modelled as a SPDL-script, which can be verified by Scyther. Listing 5.1 describes the various security claims that is chosen for A.

```
claim_A1(A, Alive);
claim_A2(A, Weakagree);
claim_A3(A, Niagree);
claim_A4(A, Nisynch);
claim_A5(A, Commit, V, Ru);
claim_A6(A, Secret, PairwiseKey);
```

Listing 5.1: Security claims for role A in APKES.

In Listing 5.2 below, the security claims for the role B in APKES are stated. Compared to the claims for A, B does not contain the `Commit` claim, as the `Running`, `Commit` approach is used in role A to provide agreement over the nonce Ru. APKES states that the R_A value has to be checked whether or not it has been tampered with, before the pairwise key can be derived at the initiating side. This can be verified by modelling the protocol to agree upon the R_A value during the protocol execution, and committing to this.

```
claim_B1(B, Alive);
claim_B2(B, Weakagree);
claim_B3(B, Niagree);
claim_B4(B, Nisynch);
claim_B5(B, Secret, PairwiseKey);
```

Listing 5.2: Security claims for role B in APKES.

5.2.2 Adversary

In the description of APKES, no specific adversary is mentioned. If we assume that such a protocol would be used for key establishment in 6LoWPAN networks, which are potentially deployed in hostile areas, we can assume that the adversary would be able to monitor all messages sent over the network. As APKES does not utilize any session keys, but rather agreeing upon a fixed long-term key, we model the adversary in a Dolev-Yao way without giving it any active capabilities other than being able to obtain the long-term keys of nodes not participating in the current key establishment process.

5.2.3 Results

Figure 5.1 shows the verification result from running the model of APKES through Scyther in the presence of the adversary described above. Scyther was able to perform an unbounded verification of the model, with only one possible trace from the characterization process, effectively meaning that there is only one possible way to execute the protocol. APKES provides verifiable entity authentication, explicit key authentication, and holds the non-injective synchronization property, which means that every message in protocol was executed as expected, even in the presence of an adversary.

Claim				Status		Comments
APKES	U	APKES,U1	Alive	Ok	Verified	No attacks.
		APKES,U2	Weakagree	Ok	Verified	No attacks.
		APKES,U3	Niagree	Ok	Verified	No attacks.
		APKES,U4	Nisynch	Ok	Verified	No attacks.
		APKES,U5	Commit V,Ru	Ok	Verified	No attacks.
		APKES,U6	Secret {Ru,Rv}k(U,V)	Ok	Verified	No attacks.
	V	APKES,V1	Alive	Ok	Verified	No attacks.
		APKES,V2	Weakagree	Ok	Verified	No attacks.
		APKES,V3	Niagree	Ok	Verified	No attacks.
		APKES,V4	Nisynch	Ok	Verified	No attacks.
		APKES,V5	Secret {Ru,Rv}k(U,V)	Ok	Verified	No attacks.

Figure 5.1: Result of verifying APKES' security claims using Scyther.

5.3 Formal Security Analysis of AKES

5.3.1 Security Claims

5.3.2 Adversary

5.4 Formal Security Analysis of SAKES

5.4.1 Security Claims

5.4.2 Adversary

Chapter 6
Discussion and Evaluation

Protocol	Entity authentication		Implicit key authentication	Explicit key authentication		Key compromise impersonation	Forward Secrecy	Known-key security	Key control	Secrecy of key
	Of U	Of V		Of U	Of V					
APKES	✓	✓	✓	✓	x	x	x	✓	✓	✓
AKES	x	x	x	x	x	x	x	x	x	x
SAKES	x	x	x	x	x	x	x	x	x	x

Table 6.1: Table of the security properties that are satisfied in the different protocols.

6.0.3 Limitations / Not covered by Scyther

APKES has support for avoiding replay attacks through the use of frame counters. Replay attacks, or *injectivity* is impossible to model in Scyther, hence the formal security analysis has not covered this class of possible attacks on the protocol.

When using pluggable scheme. Have to store key. Apkes + leap: Delete masterkey after establishing pairwise keys with all nodes. What happens if a node goes down, replaced, and rebooted? No masterkey to use. Can't establish keys with that node. hmm. Fuck this.

Chapter 7
Conclusion

References

[1] Alliance, T. Z. Zigbee. http://www.zigbee.org/. Accessed: 2016-04-15.

[2] Anderson, R. and M. Kuhn (1996). Tamper Resistance - A Cautionary Note. In *Proceedings of the second Usenix workshop on electronic commerce*, Volume 2, pp. 1–11.

[3] Ashton, K. That "Internet of Things" Thing. http://www.rfidjournal.com/articles/view?4986. Accessed: 2016-03-31.

[4] Barker, E. (2016). Recommendation for Key Management Part 1: General. *NIST Special Publication 800*(4), 57.

[5] Basin, D. and C. Cremers (2010). Modeling and Analyzing Security in the Presence of Compromising Adversaries. In *Computer Security–ESORICS 2010*, pp. 340–356. Springer.

[6] Basin, D., C. J. Cremers, and C. Meadows (2012). Model Checking Security Protocols.

[7] Blanchet, B. ProVerif. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/. Accessed: 2016-02-23.

[8] Blanchet, B. (2012). Security Protocol Verification: Symbolic and Computational Models. In *Proceedings of the First international conference on Principles of Security and Trust*, pp. 3–29. Springer-Verlag.

[9] Blom, R. (1984). An optimal class of symmetric key generation systems. In *Advances in cryptology*, pp. 335–338. Springer.

[10] Bos, J. W., J. A. Halderman, N. Heninger, J. Moore, M. Naehrig, and E. Wustrow (2014). Elliptic curve cryptography in practice. In *Financial Cryptography and Data Security*, pp. 157–175. Springer.

[11] Burrows, M., M. Abadi, and R. M. Needham (1989). A Logic of Authentication. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, Volume 426, pp. 233–271. The Royal Society.

[12] Chan, H., A. Perrig, and D. Song (2003). Random key predistribution schemes for sensor networks. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pp. 197–213. IEEE.

[13] Cortier, V., S. Kremer, and B. Warinschi (2011). A Survey of Symbolic Methods in Computational Analysis of Cryptographic Systems. *Journal of Automated Reasoning 46*(3-4), 225–259.

[14] Cremers, C. and M. Horvat (2014). Improving the ISO/IEC 11770 Standard for Key Management Techniques. In *Security Standardisation Research*, pp. 215–235. Springer.

[15] Cremers, C. J. Scyther. https://www.cs.ox.ac.uk/people/cas.cremers/scyther/index.html. Accessed: 2016-02-23.

[16] Cremers, C. J. (2006). *Scyther: Semantics and Verification of Security Protocols*. Ph. D. thesis, Eindhoven University of Technology.

[17] Cremers, C. J. (2008a). The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. In *Computer aided verification*, pp. 414–418. Springer.

[18] Cremers, C. J. (2008b). Unbounded Verification, Falsification, and Characterization of Security Protocols by Pattern Refinement. In *Proceedings of the 15th ACM conference on Computer and communications security*, pp. 119–128. ACM.

[19] Cremers, C. J. (2014). *Scyther User Manual*. Oxford University.

[20] Cremers, C. J., P. Lafourcade, and P. Nadeau. Comparing State Spaces in Automatic Security Protocol Analysis. *Formal to Practical Security 5458*, 70–94.

[21] Cremers, C. J. and S. Mauw (2005). Operational Semantics of Security Protocols. In *Scenarios: Models, Transformations and Tools*, pp. 66–89. Springer.

[22] Cremers, C. J., S. Mauw, and E. De Vink (2003). Defining Authentication in a Trace Model.

[23] Cremers, C. J., S. Mauw, and E. P. de Vink (2006). Injective Synchronisation: An Extension of the Authentication Hierarchy. *Theoretical Computer Science 367*(1), 139–161.

[24] Diffie, W. and M. E. Hellman (1976). New Directions in Cryptography. *Information Theory, IEEE Transactions on 22*(6), 644–654.

[25] Dolev, D. and A. C. Yao (1983). On the Security of Public Key Protocols. *Information Theory, IEEE Transactions on 29*(2), 198–208.

[26] Dunkels, A., B. Grönvall, and T. Voigt (2004). Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pp. 455–462. IEEE.

[27] Durumeric, Z., J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, et al. (2014). The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pp. 475–488. ACM.

[28] Eschenauer, L. and V. D. Gligor (2002). A Key-Management Scheme for Distributed Sensor Networks. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pp. 41–47. ACM.

[29] Foundation, H. C. WirelessHART. http://en.hartcomm.org/main_article/wirelesshart.html. Accessed: 2016-04-15.

[30] Gartner. Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015. http://www.gartner.com/newsroom/id/3165317. Accessed: 2016-03-31.

[31] Gaubatz, G., J.-P. Kaps, and B. Sunar (2004). Public Key Cryptography in Sensor Networks — Revisited. In *Security in Ad-hoc and Sensor Networks*, pp. 2–18. Springer.

[32] Gutierrez, J. A., M. Naeve, E. Callaway, M. Bourgeois, V. Mitter, and B. Heile (2001). IEEE 802.15.4: A Developing Standard for Low-power Low-cost Wireless Personal Area Networks. *network, IEEE 15*(5), 12–19.

[33] Hankerson, D., A. J. Menezes, and S. Vanstone (2006). *Guide to Elliptic Curve Cryptography*. Springer Science & Business Media.

[34] Hui, J. and P. Thubert (2011). Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. *RFC6282*.

[35] Hussen, H. R., G. A. Tizazu, M. Ting, T. Lee, Y. Choi, and K.-H. Kim (2013). SAKES: Secure Authentication and Key Establishment Scheme for M2M Communication in the IP-Based Wireless Sensor Network (6LoWPAN). In *Ubiquitous and Future Networks (ICUFN), 2013 Fifth International Conference on*, pp. 246–251. IEEE.

[36] Jing, Q., A. V. Vasilakos, J. Wan, J. Lu, and D. Qiu (2014). Security of the Internet of Things: Perspectives and Challenges. *Wireless Networks 20*(8), 2481–2501.

[37] Khan, R., S. U. Khan, R. Zaheer, and S. Khan (2012). Future Internet: The Internet of Things Architecture, Possible Applications and Key Challenges. In *Frontiers of Information Technology (FIT), 2012 10th International Conference on*, pp. 257–260.

[38] Kopetz, H. (2011). *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Chapter Internet of Things, pp. 307–323. Springer.

[39] Krawczyk, H. (2005). HMQV: A High-Performance Secure Diffie-Hellman Protocol. In *Advances in Cryptology–CRYPTO 2005*, pp. 546–566. Springer.

[40] Krentz, K.-F. and C. Meinel (2015). Handling Reboots and Mobility in 802.15.4 Security. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pp. 121–130. ACM.

[41] Krentz, K.-F., H. Rafiee, and C. Meinel (2013). 6LoWPAN Security: Adding Compromise Resilience to the 802.15.4 Security Sublayer. In *Proceedings of the International Workshop on Adaptive Security*. ACM.

[42] Levis, P., T. Clausen, J. Hui, O. Gnawali, and J. Ko (2011). The trickle algorithm. *Internet Engineering Task Force, RFC6206*.

[43] Liu, D., P. Ning, and R. Li (2005). Establishing pairwise keys in distributed sensor networks. *ACM Transactions on Information and System Security (TISSEC) 8*(1), 41–77.

[44] Lowe, G. (1996). Breaking and Fixing the Needham-Schroeder Public-key Protocol Using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 147–166. Springer.

[45] Lowe, G. (1997). A Hierarchy of Authentication Specifications. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, pp. 31–43. IEEE.

[46] Mao, W. and C. Boyd (1993). Towards Formal Analysis of Security Protocols. In *Computer Security Foundations Workshop VI, 1993. Proceedings*, pp. 147–158. IEEE.

[47] Meier, S., B. Schmidt, C. Cremers, and D. Basin (2013). The Tamarin Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification*, pp. 696–701. Springer.

[48] Michallon, C. Sony files to patent new contact lenses that can record video, store it, play it back - and adjust zoom, focus and aperture automatically. http://www.dailymail.co.uk/sciencetech/article-3567402/Sony-patent-application-reveals-new-contact-lenses-record-video-store-play-adjust-zoom-focus-ape html. Accessed: 2016-05-04.

[49] Mulligan, G. (2007). The 6LoWPAN Architecture. In *Proceedings of the 4th Workshop on Embedded Networked Sensors*, EmNets '07, pp. 78–82. ACM.

[50] Needham, R. M. and M. D. Schroeder (1978). Using Encryption for Authentication in Large Networks of Computers. *Commun. ACM 21*(12).

[51] Needham, R. M. and M. D. Schroeder (1987). Authentication revisited. *ACM SIGOPS Operating Systems Review 21*(1), 7–7.

[52] Neuman, B. C. and T. Ts' O (1994). Kerberos: An Authentication Service for Computer Networks. *Communications Magazine, IEEE 32*(9), 33–38.

[53] of Automation, I. S. ISA100.11a. http://www.nivis.com/technology/ISA100.11a.php. Accessed: 2016-04-15.

[54] Perrig, A., J. Stankovic, and D. Wagner (2004). Security in Wireless Sensor Networks. *Communications of the ACM 47*(6), 53–57.

[55] Popper, K. (2005). *The Logic of Scientific Discovery*. Routledge.

[56] Project, A. AVISPA. http://www.avispa-project.org/. Accessed: 2016-02-23.

[57] Roman, R., C. Alcaraz, J. Lopez, and N. Sklavos (2011). Key Management Systems for Sensor Networks in the Context of the Internet of Things. *Computers & Electrical Engineering 37*(2), 147 – 159. Modern Trends in Applied Security: Architectures, Implementations and Applications.

[58] Ryan, P. and S. A. Schneider (2001). *The modelling and analysis of security protocols: the csp approach*. Addison-Wesley Professional.

[59] Sastry, N. and D. Wagner (2004). Security Considerations for IEEE 802.15.4 Networks. In *Proceedings of the 3rd ACM workshop on Wireless security*, pp. 32–42. ACM.

[60] Simmonds, C. "The Internet of Things": What the Man Who Coined the Phrase Has to Say. www.theguardian.com/sustainable-business/2015/feb/27/the-internet-of-things-what-the-man-who-coined-the-phrase-has-to-say. Accessed: 2016-03-31.

[61] Steiner, J. G., B. C. Neuman, and J. I. Schiller (1988). Kerberos: An Authentication Service for Open Network Systems. In *USENIX Winter*, pp. 191–202.

[62] Technology, M. MiWi. http://www.microchip.com/wwwAppNotes/AppNotes.aspx?appnote=en536181. Accessed: 2016-04-15.

[63] Wander, A. S., N. Gura, H. Eberle, V. Gupta, and S. C. Shantz (2005). Energy Analysis of Public-Key Cryptography for Wireless Sensor Networks. In *Pervasive Computing and Communications, 2005. PerCom 2005. Third IEEE International Conference on*, pp. 324–328. IEEE.

[64] Wu, M., T.-J. Lu, F.-Y. Ling, J. Sun, and H.-Y. Du (2010). Research on the Architecture of Internet of Things. In *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*, Volume 5, pp. V5–484–V5–487.

[65] Yadron, D. Samsung patent reveals "smart" contact lens with built-in camera. https://www.theguardian.com/technology/2016/apr/06/samsung-smart-contact-lens-camera-patent. Accessed: 2016-05-04.

[66] Zhu, S., S. Setia, and S. Jajodia (2006). LEAP+: Efficient Security Mechanisms for Large-Scale Distributed Sensor Networks. *ACM Transactions on Sensor Networks (TOSN) 2*(4), 500–528.

Appendix A
Scyther Scripts

A.1 Scyther - APKES

```
/*
  Adaptive Pairwise Key Establishment Scheme (APKES)
*/

usertype Index;

hashfunction MIC;

macro PairwiseKey = {Ru, Rv}k(U, V);
macro Hello = (Ru, SAu);
macro HelloAck = (Ru, Rv, Iuv, SAv);
macro HelloAckMIC = MIC(HelloAck, k(U,V)); # Use
    PairwiseKey to achieve explicit key authentication?
macro Ack = (AckMsg, Ivu);
macro AckMIC = MIC(Ack, PairwiseKey);

const SAu; # U's Short Address
const SAv; # V's Short Address

const Iuv: Index; # U's index in V's list of neighbours
const Ivu: Index; # V's index in U's list of neighbours

const AckMsg;

protocol APKES(U, V)
{
  role U
  {
```

```
    fresh Ru: Nonce;
    var Rv: Nonce;

    #send_!0(U, U, k(U, V)); #To model if the adversary has
    obtained the preshared secret in LEAP.
    # HELLO
    send_1(U, V, Ru, SAu);

    # HELLOACK
    recv_2(V, U, HelloAck, HelloAckMIC);

    #ACK
    send_3(U, V, (Ack, AckMIC));

    # Claims:
    claim_U1(U, Alive); # V was "alive" as U was able to
    execute the protocol correctly
    claim_U2(U, Weakagree);
    claim_U3(U, Niagree); # Non-injective agreement
    claim_U4(U, Nisynch); # Non-injective synchronization
    claim_U5(U, Commit, V, Ru); # claim that the recv2 value
     of Ru has not been changed from the send_1 Ru-value
    claim_U6(U, Secret, PairwiseKey); # The pairwise key is
    kept secret from adversary


}

role V
{
    fresh Rv: Nonce;
    var Ru: Nonce;

    # HELLO
    recv_1(U, V, Ru, SAu);
    claim_V6(V, Running, U, Ru); # To make sure that the Ru
    is not tampered with.

    # HELLOACK
    send_2(V, U, HelloAck, HelloAckMIC);
```

```
    # ACK
    recv_3(U,V, Ack, AckMIC);

    # Claims
    claim_V1(V, Alive);
    claim_V2(V, Weakagree);
    claim_V3(V, Niagree);
    claim_V4(V, Nisynch);
    claim_V5(V, Secret, PairwiseKey);

  }
}
```

scripts/apkes.spdl

A.2 Scyther - AKES

```
/*
  Adaptive Key Establishment Scheme (AKES)
*/

hashfunction MIC;

macro PairwiseSessionKey = {Ru, Rv}k(U,V); # Where k(U,V) is
    the key from the plugged-in scheme
macro HelloAckMIC = MIC(PANu, IDu, PANv, IDv, Rv, Iuv, Cv,
   Pu, PairwiseSessionKey);
macro AckMIC = MIC(PANv, IDv, PANu, IDu, Ivu, Cu,
   PairwiseSessionKey);

const PANu; # U's Personal Area Network (PAN) Id
const PANv; # V's Personal Area Network (PAN) Id

const IDu: Agent; # U's extended, short or simple address
const IDv: Agent; # V's extended, short or simple address

const Cu; # V's frame counter of the last accepted frame
   from U
const Cv; # U's frame counter of the last accepted frame
   from V
```

```
const Pu; # Flag indicating whether or not U is currently
    one of V's permanent neighbours

const SAu; # U's Short Address
const SAv; # V's Short Address

const Iuv; # U's index in V's list of neighbours (EBEAP)
const Ivu; # V's index in U's list of neighbours (EBEAP)

const ack-msg;


protocol AKES(U, V)
{
  role U
  {
  fresh Ru: Nonce;
  var Rv: Nonce;

  # HELLO
  send_1(U, V, PANu, IDu, Ru);

  # HELLOACK
  recv_2(V, U, (PANu, IDu, PANv, IDv, Rv, Iuv, Cv, Pu,
    HelloAckMIC));

  # ACK
  send_3(U, V, (ack-msg, PANv, IDv, PANu, IDu, Ivu, Cu,
    AckMIC));

  # Claims
  claim_U1(U, SKR, PairwiseSessionKey); # The pairwise
    session key is kept secret from adversary
  claim_U2(U, Alive); # V was "alive" as U was able to
    execute the protocol correctly
  claim_U3(U, Weakagree);
  claim_U4(U, Niagree); # Non-injective agreement
  claim_U5(U, Nisynch); # Non-injective synchronization

  }
```

```
  role V
  {
  var Ru: Nonce;
  fresh Rv: Nonce;

  # HELLO
  recv_1(U, V, PANu, IDu, Ru);

  # HELLOACK
  send_2(V, U, (PANu, IDu, PANv, IDv, Rv, Iuv, Cv, Pu,
    HelloAckMIC));

  # ACK
  recv_3(U, V, (ack-msg, PANv, IDv, PANu, IDu, Ivu, Cu,
    AckMIC));

  # Claims
  claim_V1(V, SKR, PairwiseSessionKey);
  claim_V2(V, Alive); # V was "alive" as U was able to
    execute the protocol correctly
  claim_V3(V, Weakagree);
  claim_V4(V, Niagree); # Non-injective agreement
  claim_V5(V, Nisynch); # Non-injective synchronization

  }

}
```

scripts/akes.spdl

Appendix B
Notations

B.1 Notations

Notations used in protocol specifications

Symbol	Meaning
A, B, C, D	Nodes A, B, C, D
$\langle$ ... $\rangle$	Unauthenticated message
$\langle$ ... $\rangle k$	Authenticated message with key k
$\{$ ... $\}_k$	Message encrypted with key k
$A \rightarrow B$	Message sent from U to V
$A \rightarrow *$	Message broadcasted from U
ID_{node}	Identity of a node
$AES(k, m)$	AES encryption of message m with key k
$X \mid\mid Y$	Concatenation of two terms X and Y