Securing the Internet of Things: decentralised security for wireless networks of embedded systems

Justin King-Lacroix

Oriel College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

January 2016

Abstract

The phrase 'Internet of Things' refers to the pervasive instrumentation of physical objects with sensors and actuators, and the connection of those sensors and actuators to the Internet. These sensors and actuators are generally based on similar hardware as, and have similar capabilities to, wireless sensor network nodes. However, they operate in a completely different network environment: wireless sensor network nodes all generally belong to a single entity, whereas Internet of Things endpoints can belong to different, even competing, ones. This difference has profound implications for the design of security mechanisms in these environments. Wireless sensor network security is generally focused on defence against attack by external parties. On the Internet of Things, such an insider/outsider distinction is impossible; every entity is both an endpoint for legitimate communications, and a possible source of attack. We argue that that under such conditions, the centralised models that underpin current networking standards and protocols for embedded systems are simply not appropriate, because they require such an insider/outsider distinction.

This thesis serves as an exposition in the design of decentralised security mechanisms, applied both to applications, which must perform access control, and networks, which must guarantee communications security. It contains three main contributions. The first is a threat model for Internet of Things networks. The second is BottleCap, a capability-based access control module, and an exemplar of decentralised security architecture at the application layer. The third is StarfishNet, a network-layer protocol for Internet of Things wireless networks, and a similar exemplar of decentralised security architecture at the network layer. Both are evaluated with microbenchmarks on prototype implementations; StarfishNet's association protocol is additionally validated using formal verification in the protocol verification tool Tamarin.

Acknowledgements

No doctorate is truly the product of a single person's work. That person always has a support network surrounding them, of family, of friends, and of colleagues. Those people deserve acknowledgement and thanks. Therefore, if you are reading this, and I forget to mention your name, please don't think that I have forgotten you; I promise, I have not.

The first person I wish to thank is my supervisor, Prof. Andrew Martin. For the meetings, for the revisions, for the guidance, the random dark alleys, the long discussions, and for putting up with my strong head for four years. I could not have asked for a better supervisor.

To my parents, Melanie King and Serge Lacroix, who have shown nothing but unwavering support, for always knowing I could do it, and making sure I knew it too. Whatever I say here, it wouldn't be enough. Thank you. Also, Mum, the fact that you and I were going through the process at roughly the same time was helpful for both of us! Congratulations, and a toast to the Two Doctors King!

To my brother, Jordan King-Lacroix. Without your browbeating, I would never have embarked on this crazy adventure. I truly wouldn't be where I am without you.

To David Grawrock, for feedback I desperately needed, exactly when I needed it, and for four years of teaching TCI.

To the Oxford Computer Science Extroverts Club: Andrew Paverd, Martin Dehnel-Wild, Chad Heitzenrater, Ranjbar Balisane, and John Lyle. We have learned from each other, torn apart each other's ideas, and rebuilt those ideas from the ground up. Something most of those ideas sorely needed. Thank you.

To my Oriel MCR co-Execs, Holly Elbert and Charles Masaki. We three villains did well indeed.

To Tim Spijkerman, thank you for listening.

To Paul Benson and Shane Cucow. You showed me love, and gave me a home away from home when I needed it. Above all, you believed in me. Thank you.

To Joelle Grogan, the Queen of Rituals, thank you for being my friend, and occasionally my rescuer. I'm still amazed that computer security and EU law and politics aren't as different as they might seem. Congratulations, Dr.

To Adam Thomas, my best friend. Thank you for being there, and for helping me make Oxford home. You've seen me through tears, frustrations, joys, and more jargon than you should ever have had to hear.

Finally, this doctorate was supported by a teaching assistanceship from the Oxford University Department of Computer Science. To the people there, thank you for letting me indulge my love of teaching.

Related publications

Several publications were produced in the course of this research. The table below lists them, along with a summary of my contribution to each, and the chapter(s) of this thesis which include their content.

Any entry for which my contribution is listed as 'full paper' indicates that the contents of the entire paper are substantively my own, with my supervisor contributing in an advisory capacity.

Content from these papers contributed by me will be used herein without further citation. Content contributed by others will be cited appropriately.

Citation	Contribution	Chapter
Justin King-Lacroix and Andrew P. Martin. BottleCap: a Credential Manager for Capability Systems. In *Proceedings of the seventh ACM workshop on Scalable Trusted Computing (STC'12)*, pages 45–54. ACM, 2012	Full paper	5 and 6
Justin King-Lacroix and Andrew P. Martin. KEDS: Decentralised Network Security for the Smart Home Environment. In *Second Internation Workshop on Smart Grid Security (SmartGridSec) 2014*, volume 8448 of *Lecture Notes in Computer Science*, pages 63–78. Springer International Publishing, 2014	Full paper	7
Justin King-Lacroix. Position Paper: Can the Web Really Use Secure Hardware? In *Proceedings of the Workshop on Web Applications and Secure Hardware (WASH'13)*, volume 1011 of *CEUR Workshop Proceedings*, pages 17–22, 2013	Full paper	
John Lyle, Andrew J. Paverd, Justin King-Lacroix, Andrea Atzeni, Habib Virji, Ivan Flechais, and Shamal Faily. Personal PKI for the smart device era. In *Proceedings of the 9th European PKI Workshop: Research and Applications (EuroPKI 2012)*, volume 7868 of *Lecture Notes in Computer Science*, pages 69–84. Springer Berlin / Heidelberg, 2013	Section 6.3	

Contents

List of Figures

List of Tables

Acronyms

ACL access control list.

AEAD authenticated encryption with associated data.

AES Advanced Encryption Standard.

AIK Attestation Identity Key.

AS Authentication Service.

CCM counter mode with CBC-MAC.

CFB cipher feedback.

DH Diffie-Hellman.

DoS denial of service.

DRTM dynamic root of trust for measurement.

DSA Digital Signature Algorithm.

ECC elliptic-curve cryptography.

ECDH elliptic-curve Diffie-Hellman.

ECDSA elliptic-curve Digital Signature Algorithm.

GCM Galois/counter mode.

HIP Host Identity Protocol.

HMAC hash-based message authentication code (MAC).

IoT Internet of Things.

IV initialisation vector.

LoC line of code.

MAC message authentication code.

MLE measured launched environment.

OS operating system.

PAL Piece of Application Logic.

PANA Protocol for Carrying Authentication for Network Access.

PCR platform configuration register.

PKI public key infrastructure.

PZH personal zone hub.

RBAC role-based access control.

RNG random number generator.

RSA Rivest Shamir and Adleman.

RTM root of trust for measurement.

SDCC Small Device C Compiler.

SGX Software Guard Extensions.

SHA1 Secure Hash Algorithm 1.

SHA2 Secure Hash Algorithm 2.

SoC system-on-a-chip.

SRK Storage Root Key.

SRTM static root of trust for measurement.

STS station-to-station.

TC Trusted Computing.

TCB Trusted Computing Base.

TEE trusted execution environment.

TGS Ticket-Granting Service.

TGT Ticket-Granting Ticket.

TPM Trusted Platform Module.

TXT Trusted Execution Technology.

WSN wireless sensor network.

Chapter 1

Introduction: the Internet of Things

The Internet of Things (IoT) has received a great deal of attention, both in academic literature, and popular media. Heer *et al* in [61] provide a generic definition for the phrase:

> The Internet of Things denotes the interconnection of highly heterogeneous networked entities and networks, following a number of communication patterns, including human-to-thing, thing-to-thing, and thing-to-things.

In this context, a 'thing' can refer to a variety of different device classes. However, primarily it revolves around the pervasive instrumentation of physical objects with sensors and actuators, and the connection of those sensors and actuators to the Internet [13,46,118]. Broadly, this comes in two forms: the Internet-connection of wireless sensor networks (WSNs), and the pervasive deployment of RFID tags [50]. These two are largely separable; in this thesis, we consider only the issues related to the former (WSN) class of IoT applications.

The deployment of the Internet of Things presents three main challenges. The first is the engineering of the systems in question: they are expected to have a long lifetime and be inexpensive to manufacture. The second is the design and implementation of network protocols to connect those systems: the aforementioned device engineering constraints lead to IoT nodes providing highly constrained software environments, with small power budgets, minimal processing capacity, and limited network connectivity. The final challenge is more global in scope: the re-engineering of the Internet itself to be able to support the kind of low-level, constant background traffic that such devices will generate, given that the original design of Internet protocols centred on client-server services with high-speed, but bursty, traffic patterns [118].

This thesis concerns itself with the second problem.

1.1 Wireless sensor networks – why the IoT is not the Internet

Wireless sensor networks are networks of sensor nodes, communicating over wireless networks. Sensor nodes are characterised by low power, low cost electronics, with concomitantly low performance, and the wireless network links they use are designed along similar lines. IEEE 802.15.4 [70] is a very widely deployed link/physical layer protocol, and exemplifies this trend. IoT nodes follow a similar pattern.

This hardware class imposes a variety of constraints on protocol design. First, since operating the radio has a high energy cost, protocols in this domain attempt to minimise the time the radio is powered, whether for reception or transmission of data. The node therefore spends much of its time unable to receive network packets, normally because it is in a low-power sleep state. This, to quote [13], "is absolutely anomalous for IP networks" – that is, for the Internet.

The amount of computational work things are capable of forms another major constraint. On the open Internet, secure channels are commonplace, set up using public-key cryptography. In a WSN – or IoT – environment, resource limitations curtail the use of public-key primitives. Only with rise of elliptic-curve cryptography (ECC) has public-key cryptography been available on low-power hardware platforms, and then only with recent advances in the performance of such platforms. They must still be used sparingly. (We discuss this point further in Section 2.1.)

This a recurring theme in WSN and IoT security. While the problems are the same as on the Internet at large (with a particularly good summation in [10]: "Wireless network security is different from wired security primarily because it gives attackers easy access to the transport medium."), the resource limitations of the nodes on the network drastically limit the sophistication of the defences that can be deployed [13].

Nevertheless, technologies and protocols for secure embedded wireless networking – particularly for WSNs – are not new. However, all share a fundamental assumption: that the 'network' in question forms a coherent administrative domain, within which all resources are controlled and administered by a single entity. (Throughout this thesis, we will refer to this entity as the network's controlling *stakeholder*.)

The Internet does not have this property. Its structure is inherently decentralised, both technically and organisationally. From a technical standpoint the Internet does not have a single 'core', although a small number of carriers possess high-throughput links and high-performance routers that carry a substantial volume of its traffic. From an organisational standpoint, no single entity can be said to absolutely control it. The Internet of Things is necessarily similar: IoT networks can easily contain nodes controlled by multiple different stakeholders, often with competing interests. On the greater Internet, however, a stakeholder can usually be identified who can be trusted to own and operate the network infrastructure on behalf of its users (the basic function of an Internet Service Provider). IoT networks are sufficiently small in scale that this is not always possible, requiring a finer-grained approach to security. We argue that such networks must, where possible, explicitly eschew centralised control: centralised controls necessitate the existence of a stakeholder to wield them.

1.2 Contributions from this project

Basic security principle suggests that security concerns in networked systems can largely be divided into two broad categories: *authentication* and *access control* [143]. For the former, this thesis instead uses *communications security*. Under this heading, we unify notions of authenticity, integrity, and confidentiality of both data and control messages, as well as authentication of communication partners. Notably, we consider a guarantee of authenticity to include both the source of a message, and the time at which it was sent; that is, our notion of authenticity also includes freshness. These are traditionally handled at the network and transport layers of the protocol stack.

Access control then involves security decisions made once those messages arrive; it includes whether control signals should be obeyed, or data messages should be considered authoritative. These issues are necessarily handled at the application layer [117].

The design principle which we pursue in this thesis is the decentralisation of security primitives providing these services. This thesis then serves as an exposition on how such primitives can be architected and implemented in a way that meets the requirements of the IoT domain.

We claim the following contributions:

- A threat model for the Internet of Things, in which we explore the IoT threat space more completely than the brief introduction above. This model is used to derive a set of requirements for security systems providing access control and communications security services on the IoT.
- An access-control system, BottleCap, providing an exemplar for a decentralised access control architecture, and an engineering proof of concept demonstrating its operation on desktop hardware. We also discuss what technological developments would make its implementation possible on IoT hardware.
- A network-layer protocol, StarfishNet, similarly providing an exemplar for a distributed and decentralised network architecture, informed by the BottleCap design. This includes an engineering proof of concept demonstrating the feasibility of such an architecture on current-generation IoT hardware.

1.3 Structure of this thesis

The remainder of this thesis is laid out as follows:

Chapter 2 provides background on the technical underpinnings of this thesis. Of particular note is Section 2.1, which collates previous results on the performance of elliptic-curve cryptography on IoT-class processors.

Chapter 3 provides an overview of related work. It primarily looks at previous approaches to WSN and IoT network security. It also summarises relevant theoretical work, including a light treatment of general access control models.

Chapter 4 introduces a threat model for IoT networked systems. We then use this threat model as a basis from which we derive a general set of requirements for IoT security.

We then revisit the related work in Chapter 3, applying our requirements to each system in turn.

Chapter 5 describes BottleCap, our exemplar for decentralised access control, with a proof of concept implemented on desktop hardware. We discuss what technological developments would be necessary for an implementation on IoT-class hardware to be possible.

Chapter 6 evaluates BottleCap. It presents benchmarking data that illustrate the bottlenecks in the implementation, including a brief discussion of how those bottlenecks could be circumvented on an embedded system. Finally, it evaluates BottleCap against the requirements presented in Chapter 4.

Chapter 7 describes StarfishNet, our network-layer protocol, and an exemplar for the design of secure network protocols with decentralised architectures. It was designed with lessons learned from BottleCap in mind, and the prototype was implemented on a commodity WSN platform.

Chapter 8 evaluates StarfishNet. It details a formal model of the protocol, and proofs of its security properties. It then presents benchmarking data on our prototype implementation, demonstrating that its performance characteristics are appropriate for Internet of Things applications, and in fact by some metrics improve on the state of the art. It also presents an assessment of StarfishNet against the requirements from Chapter 4.

Chapter 9 concludes.

Chapter 2

Background

This chapter presents technical background relevant both to the literature review presented in Chapter 3, and the implementations presented in Chapter 5 and Chapter 7.

Section 2.1 contains a brief discussion of the use of public-key cryptography on embedded systems, and the performance of such cryptography on a variety of relevant processor types. Section 2.2 describes the salient properties of IEEE 802.15.4, a common link-/physical-layer protocol for wireless networking in this domain, including StarfishNet. These two sections provide the technical underpinnings of StarfishNet, and thus are largely relevant for Chapter 7.

Section 2.3 describes Trusted Computing (TC) mechanisms; Section 2.3.1 presents the base functionality of the Trusted Platform Module (TPM), while Section 2.3.2 extends this to dynamic root of trust for measurement (DRTM), which underpins the implementation of BottleCap. These two sections provide the technical underpinnings of BottleCap, and thus are largely relevant for Chapter 5. These sections are strongly influenced by the background section of [21], which has an identical technological foundation.

(All are also relevant to Chapter 3.)

2.1 ECC: Embedded public-key cryptography

Public-key cryptography has, until recently, been beyond the computational capabilities of embedded platforms. The finite-field arithmetic necessarily to implement it was simply impossible to perform with realistic speed on the class of processor commonly found in embedded systems. Elliptic-curve cryptography (ECC) fundamentally changes this: for a given level of security, ECC primitives require far smaller operands – and require far less computation – than their finite-field counterparts [170].

The result is that public-key protocols with meaningful levels of security are now realistic even on low-performance embedded systems, with higher performance of ECC operations on such platforms are an active area of research. Results in this area are quoted in Table 2.1.

We note that the information in Table 2.1 is not complete, with the results quoted from each paper having been curated along several lines. First, in papers where several

CPU	Clock speed	Curve	Time	Source
ATmega256RFR2	16 MHz	secp160r1	470 ms	µECC
ATmega256RFR2	16 MHz	secp192r1	810 ms	µECC
ATmega256RFR2	16 MHz	secp256r1	2220 ms	µECC
ATmega256RFR2	16 MHz	secp256k1	1615 ms	µECC
ATmega328P	16 MHz	secp192r1	4034 ms	nano-ecc
ATmega128	8 MHz	sect163k1	390 ms	[9]
ATmega128	8 MHz	sect163r2	810 ms	[9]
ATmega128	8 MHz	sect233k1	800 ms	[9]
ATmega128	8 MHz	sect233r1	1960 ms	[9]
ATmega128	8 MHz	secp160r1	810 ms	[53]
ATmega128	8 MHz	secp192r1	1240 ms	[53]
ATmega128	8 MHz	secp224r1	2190 ms	[53]
CC1010	3.6864 MHz	secp160r1	4580 ms	[53]
CC1010	3.6864 MHz	secp192r1	7560 ms	[53]
CC1010	3.6864 MHz	secp224r1	11980 ms	[53]
ATmega2560	16 MHz	curve25519	1424 ms	[68]
ATmega128	8 MHz	secp160r1	2117 ms	[97]
ATmega128L	7.3828 MHz	sect163k1	2160 ms	[159]
ATmega128L	7.3828 MHz	secp160r1	760 ms *(estimated)*	[169]
ATmega128	8 MHz	secp160r1	1350 ms	[171]

Table 2.1: Published results on the performance of ECC operations on 8-bit microcontrollers. Results were converted to a common unit system.

ECC operations are presented, only the elliptic-curve Diffie-Hellman (ECDH) result is quoted. Second, in papers discussing multiple variants of the same code, such as high-speed and small-size variants, the highest-performing result is taken. Third, only results for common elliptic curves are quoted; those for custom curves are not. Finally, only results for 8-bit microcontrollers are presented. Most work in this area is done on processors implementing the AVR architecture, but one publication also includes work on a microcontroller implementing the Intel 8051 architecture, also used in the system-on-a-chip (SoC) on which the StarfishNet prototype was implemented (see Chapter 7 and Chapter 8).

Two of the sources are README files for open-source ECC libraries. One is µECC[1], a heavily-modified version of which is used in StarfishNet (see Chapter 7). The other is is nano-ecc[2], another µECC variant.

[1] https://github.com/kmackay/micro-ecc
[2] https://github.com/iSECPartners/nano-ecc

2.2 IEEE 802.15.4

The physical/link layer underlying many current Internet of Things networks is IEEE 802.15.4 [70]. It has seen very high uptake, largely due to its use in ZigBee [177] and 6LoWPAN [119]. Its key properties are:

- 2.4GHz (ISM) radio frequency band.
- Intended range of approx. 10m.
- Transfer rate of 250kb/s.
- Maximum frame size of 127 octets, *including* link-layer headers.
- Addresses used in link-layer headers can be either 64-bit hardware addresses or software-configured 16-bit 'short' addresses. In the latter case, the link-layer headers are correspondingly reduced in size.
- Reliable packet transmission at the link layer, via link-layer acknowledgements.
- Mandatory support for 'secured' frames, using AES-128-CCM-64 authenticated encryption. Other encryption modes, including with shorter authentication tags, or eliding entirely either encryption or authentication, are optional.
- Support link-layer access control lists (ACLs), packets to be received from (or sent to) only particular addresses.
- The initialisation vector (IV) generation scheme for IEEE 802.15.4 also provides built-in replay protection.

Unfortunately, several link-layer services are subject to a variety of security pitfalls. Sastry and Wagner in [147] provide an excellent explanation of these flaws, and how designers of protocols using IEEE 802.15.4 can avoid them. They first identify common keying models in wireless networks: pairwise keying indicates that keys are established for each pair of communicating nodes; group keying indicates that groups of nodes with common goals will share a key; network-shared keying indicates that a single key is shared by all nodes on the network; and hybrid keying involves some combination of these. The ensuing discussion then identifies the following main flaws in IEEE 802.15.4:

- Several flaws in management of link-layer encryption state can lead to IV reuse, which must not happen in counter mode with CBC-MAC (CCM) [36].
- If the link layer fails to decrypt a packet, it provides to higher layers the ciphertext that failed to decrypt, but *not the IV used in the attempt.*
- Use of link-layer encryption without integrity protection can lead to a spectacular denial of service (DoS) attack, as a result of the design of link-layer replay protection.

- Link-layer acknowledgement packets contain no addressing data of any kind, and are unauthenticated.
- Low-power modes of many radios cause the ACL state to be lost.
- The implementation of replay protection is incompatible with the network shared keying model.
- The entire standard is essentially incompatible with group keying, since ACL entries (which are also the mechanism by which link-layer encryption is configured) can only specify a single address, not a group thereof.

Based on this work, the authors then provide recommendations for standard writers, hardware builders, and software writers making use of the standard. Of the last, there were two major suggestions. The first is to avoid the use of unauthenticated encryption modes. This is an instantiation of a general best-practice in the design of systems using cryptography, which is to explicitly authenticate encrypted data, rather than trying to rely on the properties of the encryption (and, naturally, never to use unauthenticated data in a system-critical context). The second is to avoid entirely the use of link-layer acknowledgements, instead implementing reliable data transmission in a higher layer.

From the nature of the flaws they identify, several other best-practices can be deduced:

- Avoid using the same key in different ACL entries – that is, with multiple different communication partners. This is tantamount to avoiding group keying of any kind, at least when using link-layer encryption.
- Protocols implementing end-to-end confidentiality and/or integrity protection between nodes outside direct communication range – particularly, those that perform routing – cannot use link-layer encryption to provide it, and must instead implement this functionality at the network layer.
- If making extensive use of ACLs, software must keep a shadow copy of all of the state stored therein, in case it is lost by the radio upon entering low-power mode.

Notably, many of the same flaws were also identified in [174], though these authors focus on recommendations for standard writers rather than software authors.

2.3 Trusted Computing

Trusted Computing is a series of technologies and protocols designed to enable access control decisions to be made regarding the identity and state of a particular physical machine. It is largely based on the TPM, a hardware cryptographic coprocessor with a particular interface, and strict requirements on the mechanism by which it is

integrated into the machine to which it is bound. The constraints thereby imposed – that is, the nature of the TPM's interface and its binding to the platform – allow both local and remote verifiers to make guarantees on the conditions under which cryptographic transactions have taken place. Of particular interest are guarantees regarding the software running on the machine. In essence, it enables a form of context-sensitive identity-based access control for machines [106].

Note on specification versions This description is based on version 1.2 of the TPM specification [164–166]. While the basic principles apply to all versions of this specification, the specific cryptographic algorithms in use are subject to change. In particular, while TPM 1.2 specifies the use of SHA1 or RSA, other versions may specify otherwise.

2.3.1 The Trusted Platform Module

The TPM is a cryptographic coprocessor. It provides facilities for the secure generation of cryptographic keys and random numbers, sealed storage, and measurement and cryptographic certification of software state.

Software state and measurement

A core feature of the TPM is its array of platform configuration registers (PCRs). Each PCR is a volatile memory block containing a SHA1 hash, and is reset to zero at a platform reset. PCRs may not be written to directly, only using an operation called `extend`. This operation takes, as input, a PCR, and a 160-bit value (generally a SHA1 hash). It concatenates the two together, producing a 320-bit string, which is then itself hashed using SHA1. The result is written back to the PCR. As a result, each PCR is effectively a digest of every hash that has been extended into it since boot.

These PCRs are used during the boot process to establish an authoritative statement of the software state of the machine. Each element in the boot takes a hash (called a *measurement*, in this context) of the next element, extends that measurement into a PCR, prior to transferring control to it. This chain of measurements then uniquely and unambiguously identifies both its elements and the order in which they occur. The first code to run on the platform (part of the firmware) must therefore be trusted to anchor this chain of measurements; it is referred to as the root of trust for measurement (RTM). This entire process is referred to as a *measured boot*, and establishes the machine's static root of trust for measurement (SRTM).

Keys, attestation, and storage

The TPM's cryptographic features are twofold: it can generate RSA key pairs using an onboard source of randomness, and it can decrypt or sign data using the private portions of these key pairs without exposing them to the rest of the platform. Restrictions can be imposed on the potential uses of these key pairs, called the *key policy*:

keys can be restricted to a specific TPM or permitted to migrate to other TPMs; additionally, many cryptographic operations can be restricted to be available only at specific times, such as when the PCRs take on specific values.

Sealed storage allows for similar restrictions to be imposed on arbitrary blocks of data. Sealed data blobs also include a statement of the state of the TPM at encryption time.

Bound storage is similar to sealed storage, except that bound data blobs can be generated by a remote machine, using the public portion of an encryption key pair stored in the TPM. The remote party then has a guarantee that the data so stored can only be decrypted by that TPM, when its PCRs are in the specified state.

Finally, using a TPM signing key pair, the TPM can generate a *quote*: a digitally-signed statement of the TPM's state, including the values of the PCRs. Combined with the measured boot mechanism described in Section 2.3.1, a TPM can therefore provide unforgeable proof of the machine's state to a remote party, in a protocol known as *remote attestation*. A quote must be signed by an Attestation Identity Key (AIK), an RSA key which the TPM has generated for the sole purpose of signing quotes. (In practice, an AIK may also be used for one other purpose: to sign other keys in the TPM storage hierarchy, thus attesting to the use to which the TPM will put them.)

The TPM's storage hierarchy is rooted in the Storage Root Key (SRK), which is erased and regenerated when the TPM's ownership is taken. The effect is that when a machine changes owners, all data belonging to the previous owner is securely erased.

2.3.2 Dynamic root of trust for measurement

The SRTM from Section 2.3.1 is referred to as 'static' because it occurs exactly once, at platform reset. Many processors and chipsets by Intel [72] and AMD [2] also expose functionality known as the DRTM: a special processor instruction causes the CPU, chipset, and TPM to collaborate to provably execute a small block of code in a protected environment. The Intel documentation, whose terminology we will use in this thesis, refers to this code block as the measured launched environment (MLE).

This process, termed a late-launch, consists of five steps: the CPU is partially reinitialised, the chipset enables DMA protections, the TPM resets the (so-called) Dynamic PCRs, the CPU sends the MLE to the TPM to be measured into a dynamic PCR (PCR17 on AMD platforms, PCR18 on Intel), and control is transferred to the MLE. Critically, this entire sequence occurs as a single atomic transaction – that is, an interruption at any point in the sequence will cause the platform to restart. Equally critically, this is the only way in which the Dynamic PCRs can be reset.

After late-launch, the MLE is in complete control of the machine, with no dependency on any previously-running software. For this reason, late-launch is often used while bootstrapping a hypervisor (see, for example, Intel's tboot[3]), to protect against boot sector viruses and device firmware vulnerabilities.

[3] http://www.sourceforge.net/projects/tboot

Intel's implementation of the concept, Trusted Execution Technology (TXT), provide an additional protection mechanism. Once a late-launch has been performed, the platform is considered to be in *Safer Mode*, in which a platform reset will cause main memory to be scrubbed of all data before the machine is permitted to reboot. This memory scrubbing will additionally occur during any platform resets caused by failure to initiate a late-launch. This mechanism is designed to defend against cold-boot attacks.

Flicker

The Flicker framework uses the late-launch mechanism to execute a part of a program, termed a Piece of Application Logic (PAL), with full isolation from the rest of the running system, returning control to the host operating system (OS) upon PAL completion. Its major emphasis is on minimisation of the PAL's Trusted Computing Base (TCB) – the sum total of the code which the PAL must trust for its security. Using the TPM, the PAL can additionally attest its isolation to a remote party, despite not having access to OS services during its execution.

The PAL can use sealed storage (based on the Dynamic PCR values, which will be the same at every Flicker invocation, as long as the supporting Flicker, PAL and hardware-support binaries haven't changed) for data which should only be accessible within the isolated environment; Flicker will extend the hashes of the input and output data to the PAL into the Dynamic PCRs once its execution is complete, disabling access to that data upon return to the OS, and also permitting attestation of the PAL's inputs and outputs.

Moreover, the Flicker runtime environment is small – without TPM interface code, only 2.7kLoC, much of which implements C standard library functions, and only a few hundred LoC of which run in the privileged mode of the CPU – providing only the services necessary for a minimal runtime environment.

2.3.3 Aside: the chain of trust

Remote attestation requires more than the hardware primitives described above: there must also be a mechanism for the verifier to establish trust in that hardware – that is, that the relevant keys actually belong to a TPM which is correctly attached to a machine. The mechanism used in TC is a chain of trust, rooted in a pair of certificates:

endorsement credential A certificate, signed by the TPM's manufacturer, asserting that the subject of the certificate (the TPM's cryptographic identity) is a valid TPM – that is, it adheres to the TPM specification [164–166].

platform credential A certificate, signed by the machine's manufacturer, describing the characteristics of the machine to which that TPM is bound, and the properties of that binding. The subject is a hash of the endorsement credential. It asserts that the machine, and the TPM's binding thereto, adhere to a TPM interface specification, such as [162,163].

Together, these certificates assert that the platform in question is a valid trusted platform, and specify the meaning carried by attestations from its TPM. (In practice, they are used to generate an AIK credential, a certificate that endorses an AIK, so that quotes signed by that AIK can be trusted. The mechanism by which this occurs is not relevant for the purposes of this thesis.)

Chapter 3

Related Work

This chapter presents a review of related work relevant to the problems addressed in this thesis. It divides this work into rough categories, with one section for each such category. Section 3.1 covers theoretical work; this includes abstract protocols, access control models, and similar. Section 3.2 covers key distribution models and algorithms. Section 3.3 presents concrete network-layer protocols from both the Internet of Things (IoT) and wireless sensor network (WSN) domains, as well as protocols used for similar tasks in industry. Finally, Section 3.4 explores mechanisms for providing either communications security or access control implemented at layers higher than the network – that is, transport or application.

We wish to particularly highlight Section 3.3.10, which describes ZigBee, a popular network-layer protocol in this domain. Its security model is described in detail, in order to facilitate its use as a basis for comparison during the description of StarfishNet in Chapter 7. Much of its terminology is also reused throughout the thesis.

3.1 Theoretical work

3.1.1 The access matrix

The *access matrix* of Harrison, Ruzzo, and Ullman [58] is the most general model known for access control. It imagines the existence of a matrix (Figure 3.1) describing an entire system in the following way:

- The rows of the matrix represent *subjects* (also called *security principals*), which perform actions in the system.
- The columns represent *objects* (also called *resources*, or sometimes *services*), on which actions are performed. Subjects may also be objects, as is the access matrix itself.
- A cell contains the *rights* of the subject with its row index on the object with its column index.

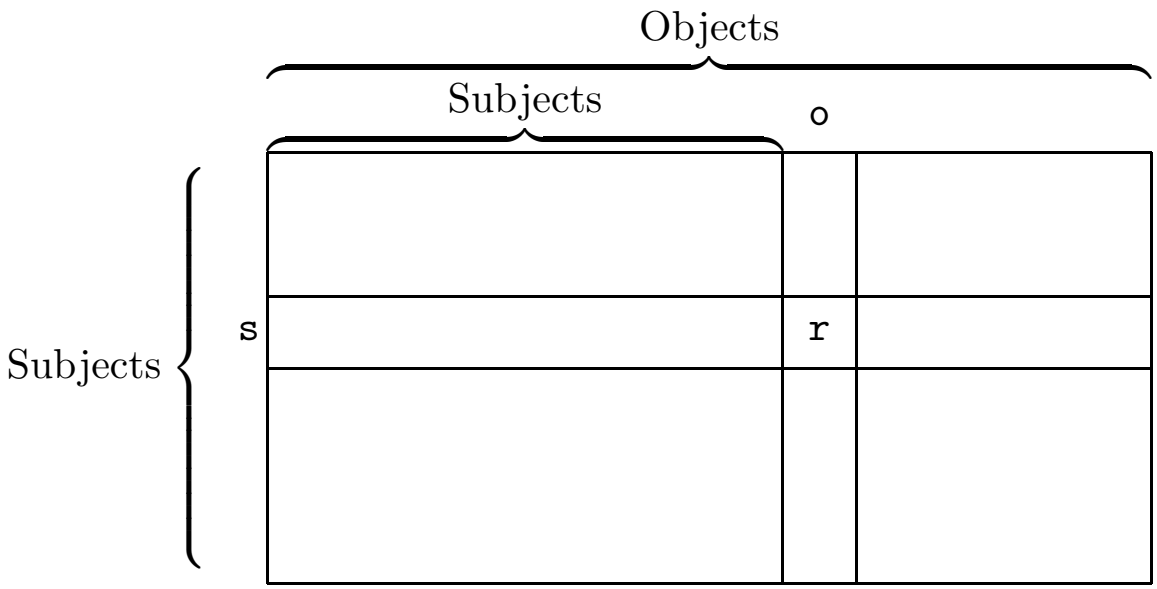


Figure 3.1: The HRU access matrix. `r` indicates the rights of subject `s` over object `o`. (Reproduction of diagram in [58].)

Systems can store this matrix in either a column-oriented fashion – that is, for each object, storing a list of subject and their rights over it – or a row-oriented fashion – storing, for each subject, a list of objects and the subject's rights over them. Saltzer and Schroeder in [142], one of the seminal papers on computer security, refer to these as *access control list (ACL)* and *capability-based* systems, respectively.

Modern security literature uses those terms slightly differently. The term "*ACL*" now refers to a specific construct: a list of 2-tuples associated with an object, of which the first element is the name of a subject, and the second is the set of rights that subject holds over that object. Conversely, a "*capability*" is specifically an opaque token which a subject can present to an object to invoke rights, with that token containing an indelible description of the rights is confers.

3.1.2 Role-based access control

Role-based access control (RBAC) [40] is a variation on the access control list concept, in which the ACL lists *roles* and their access permissions. Roles are then assigned to subjects as required, generally related to the structure of the organisation. Unlike most ACL-based systems, RBAC therefore requires that the creator (or owner) or an object *not* necessarily have the right to modify its ACL. This is instead determined by the system, based on the role(s) assigned to the object's creator, and the context in which it was created. RBAC is therefore a kind of *mandatory access control* [91] system, as opposed to the more common *discretionary access control* model [40]. Directory-based systems, including Microsoft Active Directory can often be configured to operate in either mode.

3.1.3 Attribute-based access control

Attribute-based access control [64] is an access control model that focuses on the attributes of a subject, rather than its identity. Rather than conforming to the HRU access matrix model above, it has been formalised as an augmentation to the access matrix [176], with fundamental operations on the access matrix being added or modified to support the creation and modification of attributes.

The most common implementation of this concept is the attribute certificate [25,26], a digital certificate binding a name to a set of access rights. (The binding of that name to a set of digital credentials is assumed to be done elsewhere, usually using traditional public key infrastructure (PKI) certificate chains.) Attribute certificates make designation of authority explicit: an identity, even once authenticated, must still present an attribute certificate to authorise any request. Moreover, they are flexible: they can be bound to any kind of identity, and can describe any relevant security property. The subtle disadvantage is that all access must be double-checked: every resource must be prepared to cryptographically verify the identity *and* authorisation certificates of every requester, potentially on every request.

SPKI (Simple PKI) [37,38] is a standard for digital certificates and the PKI to support them. While its primary goal is to be simpler X.509, it also notably provides explicit support for attribute certificates.

Trusted Computing Trusted Computing has strong elements of attribute-based security. The basic certificates used to build the chain of trust, the endorsement and platform credentials, are attribute certificates with the TPM's identity (the endorsement key) as subject. Its mechanisms also permit the use of properties other than identity when performing access control checks; remote attestation is particularly well-suited to this task. The attestation guarantee can serve as a basis on which to build further assertions, such as the state of operating system structures or applications. Logical attestation [156] is one such extension: the OS kernel can generate a cryptographic assertion of the execution environment of some binary. That assertion is then composed with an attestation of which kernel is running. Access control decisions vis-à-vis that machine can then be made solely based on its runtime attributes. (We observe a similarity between this principle and Final State Attestation, as described in [131].)

In a similar vein, the trusted execution environment (TEE) is a general-purpose mechanism that can be used to prove the execution of arbitrary computations; the canonical example is Flicker, presented in Section 2.3.2. This property – of having executed a specific program – could be a certifiable attribute.

3.1.4 Capability-based security

Capability-based security [30,94,142] is in many ways opposite to the identity-based paradigm observed by most access control systems. Whereas identity-based systems require requesters to authenticate an identity, with authorisation decisions made based thereon by the resource, a capability-based system eschews identity entirely:

requesters must directly prove their authority (and only their authority) to perform an action. This approach provides performance and scalability advantages over identity-based security [47]: access-control checks are made locally to the resource being accessed, eliminating both network round-trips to central directories, and (by extension) removing those central directories as bottlenecks. (It also eliminates from the critical path the need to verify multiple digital signatures, a particular issue when combining attribute certificates with PKI.) Finally, capability systems (including those discussed below) provide for great flexibility in system design and operation. They provide for simple delegation of access between security principals at a fine granularity, using capability derivation and transfer. Because authority can be delegated directly, complex identity delegation mechanisms (such as OAuth [57]) can also be avoided.

Of course, these advantages aren't without cost. In a capability-based system, security metadata must generally be allocated per capability issued, rather than per subject or object. In an environment where security metadata is expensive to allocate, such as inside an operating system kernel, this can be a particular problem.

3.1.5 The Resurrecting Duckling

The Resurrecting Duckling [157] is a security policy model for systems that expect to create semi-permanent associations. It assumes the existence of two devices: the *master* (or *mother duck*), which has a fixed owner, and the *slave* (or *duckling*), which does not. The goal is for the owner of the master to reversibly claim ownership of the slave.

The duckling can be in one of two states: *imprintable* or *imprinted*. In the former state, it recognises no owner, while in the latter, ownership has been claimed. *Imprinting* is the transaction by which ownership is taken, *death* that by which it is released in an orderly fashion, and *assassination* that by which it is forcibly revoked. It is expected that the duckling be constructed so that assassination is difficult or impossible by an attacker.

In detail, imprinting is the transfer of (or agreement on) a shared secret between mother duck and duckling. The model requires that this secret transfer take place over a secure link; as written, it assumes a physical cable. The duckling will then permit only its mother to perform management operations, including inducing death (that is, termination of the association by erasing the shared secret), and changing access control policy.

As the authors themselves remark, this model, in abstract, was already essentially in use throughout the industry by the time it was published, with the exception of the difficulty-of-assination criterion. We note a particular similarity with the SRK lifecycle model in Trusted Computing (see Section 2.3).

3.1.6 Multi-channel authentication

Multi-channel authentication [173] is a Diffie-Hellman-based protocol framework which models authentication properties derived from auxiliary channels – that is, channels

between the authenticating entities that exist by virtue of their context or environment. The paper enumerates a variety of different types of auxiliary channel, with different channel capacities, directional properties, and environmental factors affecting the mechism by which authentication can be achieved.

3.1.7 STS

The station-to-station (STS) protocol[89] is a theoretical protocol involving an authenticated Diffie-Hellman (DH) key exchange. In concept, it bears strong resemblance to the StarfishNet association sequence (described fully in Section 7.3), except that it delays authentication of the initiator until the third message in the protocol, so that all authentication and key confirmation metadata can be encrypted with the session key. The net result is that all privacy-sensitive information is exchanged within the secure channel, a useful property if either party requires identity privacy.

The protocol is vulnerable to an identity misbinding attack, wherein an active attacker can cause the participants of the protocol to disagree on the identities of the nodes purported to have authenticated[89]. Note that this does not result in the disclosure of confidential data directly; it is purely an authentication failure.

3.2 Key distribution protocols

This section covers literature on key distribution models and algorithms in wireless sensor networks and IoT-class networks. Nearly all literature follows a common set of terminology, which is also used throughout this thesis, and is elegantly presented in [92, Table 3]. We introduce the most common terms here:

Network(-wide) keying A symmetric key distribution model in which all nodes on a network use the same key for cryptographic operations, and thus all must possess this key. Network admission control often is either predicated on, or grants access to, this key.

Pairwise keying A symmetric key distribution model which is essentially the opposite of network-wide keying: each communicating pair of nodes establishes a different key, with that key only used to secure communications between those two nodes.

Group keying A hybrid of the above two models. A key may be shared among some number of nodes, but its distribution does not define the perimeter of the network, as it does in network-wide keying.

Network key The shared key in network-wide keying.

Link key In pairwise keying, the key shared by a given pair of nodes.

Group key The shared key in group keying.

3.2.1 Random key predistribution

Random key predistribution [39] is a key distribution model for WSNs. It uses symmetric cryptography only, and relies on the existence of a root key server. That server generates a large pool of keys, P. Each node, on deployment, is provisioned with a random subset (of size k) of P.

When attempting to communicate, nodes i and j exchange lists of the key identifiers with which they have been deployed. If those lists contain any common elements, one of those elements is selected as the link key. Pairs of nodes without a common key attempt to discover a path through the network, whose edges are links between those that do. While this does not result in a *unique* link key per connection, it does nevertheless partition the network such that the compromise of a single node only reveals an (ideally small) subset of the link keys on the network.

A later publication [27] describes several enhancements to this scheme:

q-composite The *q-composite* scheme is almost identical to the above, except that link keys are formed by hashing together *all* base keys shared between the relevant pair of nodes. This modification then allows a minimum number q of shared keys to be stipulated. The authors examine how this parameter affects the security of the links thereby established. They also discuss how manipulating q and the size of P (k is assumed to be set by the storage capacity of the nodes) affects the probability that any given pair of nodes will be able to form a link.

multipath reinforcement *Multipath reinforcement* is a variant key path discovery mechanism, and is essentially analogous to the q-composite scheme in a path discovery context. When establishing a shared key, two non-adjacent nodes perform the same path search as above. However, if more than one path is found, several (or even all) will contribute key material to the final link key. This adds resilience against the compromise of a single node, or of nodes along a single path.

random pairwise The *random pairwise* distribution scheme requires nodes to be provisioned with link keys, rather than elementary keys from which they will derive a link key. Each link key can then be unique, automatically granting both strong authentication of remote parties (since each link key is stored by exactly two nodes). It also reduces the impact of node compromise to only compromising the links in which that node is actively involved. The random element is then *which* links are provisioned, and which are not. As before, nodes which do not possess a shared key perform a path discovery through the network.

In all cases, the existence of a central key generation and distribution point necessitates both the treatment of the network as a single domain, and its control by a single entity. Nodes that are not explicitly compromised are assumed not to be malicious. The compromise of a node is only considered to disclose the keys it has stored.

In a later work[98], node compromise is considered in more detail. The authors point out that in the original scheme, as the number of compromised nodes increases, the number of compromised *keys* rises faster – potentially very quickly. They construct a model of pairwise key establishment schemes of this kind, show that all of the above schemes are instances of that model, and then use it to construct two more schemes:

random subset assignment Random subset assignment is very similar to the original random key predistribution. However, the pool P is of polynomials in two variables, with each variable representing a possible node identity (which identities are projected onto the positive integers), rather than keys. Nodes are then deployed with *polynomial shares*: for node i, some set of polynomials is selected. For each such polynomial $f(x, y)$, the node is then loaded with the *polynomial share* $f(i, y)$. Two nodes i and j that share a common polynomial can then evaluate $f(i, j) = f(j, i)$ to establish a link key. If they do not share a common polynomial, they must discover a path through nodes with which they do, again in a manner analogous to the original scheme.

grid-based key predistribution Grid-based key predistribution, unlike most of the above schemes, guarantees that any pair of nodes can agreen on a key. In this scheme, the nodes in the network are arranged in an m x m grid (with m^2 therefore bounded below by the number of nodes in the network). Each row and column is then assigned a polynomial (for $2m$ total polynomials). Any pair of nodes sharing a row or column can then establish a key using that row/column's polynomial. Other pairs of nodes perform path key discovery as usual; however, the topology of the grid permits them to calculate the most efficient path *a priori*. Moreover, the grid can be augmented with information on which nodes have been compromised, and any paths including compromised nodes recalculated accordingly.

3.2.2 Key infection

Key infection [6] is a security and key distribution model for wireless sensor networks. It uses symmetric cryptography only, and relies on the existence of a secure bootstrapping phase – that is, it assumes there exists a period of time, when the network is being set up, during which the nodes can broadcast at short range without an adversary present.

During bootstrapping, node i generates a key k_i, and broadcasts it to all of its neighbours. Node j, upon receiving this broadcast, generates its *link key* with node i, k_{ij}, and sends the message $\{j, k_{ij}\}_{k_i}$ – that is, the link key and its identity, encrypted with k_i – to i.

In order for non-adjacent nodes to communicate, a similar scheme is used. The path between those nodes must first be discovered. A link key is then established between the first pair of nodes in the path, which is augmented with key material generated by each other node in the path until it reaches the final node. If multiple paths exist between a pair of nodes, their path keys can be combined; this is a useful

defence mechanism in the event that one of the nodes along a path is compromised (similarly to the multipath reinforcement scheme from Section 3.2.1).

We note that the paper is written with the assumption that all sensor nodes belong to the same party, and are being deployed as a unit. Certainly, all nodes are trusted; a malicious node can trivially impersonate any of its neighbours.

3.2.3 'Blom' scheme

The Blom scheme [17] is another key predistribution scheme, also requiring only symmetric cryptography. This scheme is parameterised by λ, a security parameter, and N, the maximum number of nodes on the network. A 'master node' then generates two matrices over some finite field:

- G, a $(\lambda + 1) \times N$ matrix, with linearly independent columns, is public;
- D, a symmetric $(\lambda + 1) \times (\lambda + 1)$ matrix, is secret.

Then, letting

$$A = (D \cdot G)^T$$
$$K = A \cdot G$$

the link key between any two nodes i and j is then K_{ij} $(= K_{ji}$, since D is symmetric). Node i must then store the ith row of A (which it must keep secret), and the ith column of G (which is public). It can use these to derive a link key with any node on the network:

$$K_{ij} = A_i \cdot G_j$$
$$(= A_j \cdot G_i)$$

3.2.4 Pairwise key predistribution

Pairwise key predistribution [33] is based on the q-composite scheme (Section 3.2.1 and [27]), combined with the Blom scheme (Section 3.2.3 and [17]). Some number ω of Blom spaces (that is, D matrices; G can be reused) are constructed, and each each node with a q $(< \omega)$ keys from separate, randomly-selected spaces. For any pair of nodes i and j, the link key is established by determining their common Blom spaces, calculating the shared secrets from each of those spaces, and composing them in a manner analogous to the q-composite scheme.

3.2.5 Fast authenticated key establishment

Fast authenticated key establishment [65] is an authenticated key agreement protocol that uses asymmetric cryptography. Despite this fact, it still targets WSNs. It relies on the existence of two central authorities: a CA for issuing certificates, and a 'security manager' for performing admission and access control on the WSN.

The certificates issued are *implicit certificates*, which are a construct unique to ECC. Rather than being explicitly validated through a signature verification, an implicit certificate is instead used to derive the ECC key pair used. The public key and certificate are then used in an ECDH-like key agreement protocol. If (and only if) the protocol succeeds, the two communicating parties both possess the shared secret, and have correctly validated the implicit certificate.

In this particular scheme, the CA issues implicit certificates to sensor nodes and the security manager. Sensor nodes then contact the security manager to join the WSN. Each uses its implicit certificate in a key agreement protocol, with the end result of the protocol being a combination of a link key between them, and confirmation that both of their identities have been certified by the CA. Finally, the protocol is designed so that the majority of the ECC work is done by the security manager, which is assumed to be more powerful than the sensor node; the sensor node is only required to perform a single ECC operation.

3.3 Link- or network-layer protocols

3.3.1 Network admission control

A variety of common network- and link-layer protocols provide some means of *network admission control*: mechanisms for making, and enforcing, decisions on the admission or ejection of devices from the network. IEEE 802.1X [71] permits such decisions to be made at the link layer for Ethernet-class networks, and has been extended to wireless networks as Wi-Fi Protected Access. Other protocols can be tunnelled within the transaction, such as Trusted Network Connect [167] and Network Endpoint Assessment [144] (including its implementations, Microsoft's Network Access Protection and Cisco's Self-Defending Network). These protocols allow endpoints to provide metadata to support network access control decisions, including state and attribute information such as OS version and update state. They also permit more granular control than simply 'allow' or 'deny', generally at least allowing for a *remediation network* to which a node can be connected to force updates to take place.

3.3.2 Secure Lossless Aggregation

Secure Lossless Aggregation [16] is a network-layer protocol for WSNs where a large number of sensor nodes are communicating with a single gateway node, but not all can do so directly. Those that can must therefore act as routers.

It was designed for the "Machine-to-Machine" communications paradigm, where the gateway will generally have a cellular (or other long-range) communications interface, which acts as a direct link to a central service – this service not being on the public Internet.

The authors present a protocol with the following additional key insights:

1. Since all sensor readings have the same destination, permitting routers to aggregate readings from multiple sensors into a single onwards message is likely to

result in a substantial reduction in network transmissions. This is not possible if the integrity protection mechanism is exclusively end-to-end, since in this case only the gateway would be able to detect any message corruption; aggregator nodes must therefore be able to accomplish this.

2. At the same time, aggregator nodes should not be able to generate false readings, requiring some end-to-end protection.

The resulting protocol uses a pairwise keying model. Each sensor node requires two keys: one shared with the gateway, and one shared with the nearest aggregator. Two layers of cryptographic protection are then applied to each message. The inner layer is authenticated encryption, using the key shared with the gateway. The outer layer applies only integrity protection, using the key shared with the aggregator. Each message authentication code (MAC) is 32 bits long, for a total of 64 bits of integrity protection overhead (the same as the mandatory security mode in IEEE 802.15.4).

The aggregator is thus able to verify the integrity of sensor messages without learning their contents. It can also strip routing or addressing headers and the outer cryptographic layer, and send large batches of inner messages in bulk to the gateway. These messages will then only fail the second, inner layer of integrity-checking, if the aggregator has attempted to tamper with them.

We note that the key distribution model in this work strongly resembles that presented in [76].

A word on cryptography Unfortunately, this protocol's protection scheme has a large flaw. Because the two MACs are calculated separately, they can be attacked independently. The total work factor for such an attack is therefore the *sum* of the work factors of the individual attacks, rather than its *product*. In this case, since the two MACs are each 32 bits long, the total work factor for breaking this scheme is only $2 \times 2^{(32-1)} = 2^{32}$, or 32 bits of security, despite the 64 bits of payload space devoted to integrity protection. Security best-practice suggests that this is inadequate for real-world use, especially on the open Internet.

The security of this protocol could be greatly improved by increasing one (or both) of the MACs to 64 bits in length. Given that aggregator nodes are only semi-trusted, this suggests the end-to-end MAC would be the appropriate choice. Naturally, this would mean a sacrifice of 4 bytes of payload space available to sensors; the hop-wise MAC could also be shortened, reclaiming some of this space. However, such a change could cause problems for aggregators, since they may also buffer multiple sensor messages: those 4 bytes are added to *each* message in a batch, likely reducing the number of messages that can be aggregated in this way.

3.3.3 Sizzle

Sizzle [52] was an implementation of SSL for TinyOS [93], an OS designed for WSN nodes based on microcontroller-class processors. The TinyOS network stack uses the IEEE 802.15.4 PHY, although not its packet format. It providers only link-layer

transport, with no reliability guarantees, no cryptographic protections by default, and no routing services.

Sizzle was primarily published as a proof-of-concept that strong cryptography (ECC) was usable on WSN platforms. Its authors implemented a custom reliable transport above the (link-layer only) TinyOS network stack in order to demonstrate the performance of the prototype. While Sizzle is not a full network layer in any sense, it is included here as an 'honourable mention'.

3.3.4 TinySec

TinySec [77] is a link-layer security architecture for TinyOS, targetted at WSNs. The paper makes two observations. First, using strong cryptography is needed for communications over a wireless transport, due to the ease of access to that transport by third parties. Second, for many WSN applications, nodes on the same network are owned by the same entity, and therefore can trust each other. These applications generally seek to maintain the confidentiality and authenticity of WSN data in the presence of an external attacker. For these applications, end-to-end security of the kind needed on the open Internet often does not make sense. Therefore, although TinySec itself does not specify a keying model, the authors used network-wide keying on their testbed network. For similar reasons, the security model of TinySec does not consider node compromise.

TinySec permits two variants: authentication-only, and authenticated encryption. Although it predates the development of CCM, the second variant is strikingly similar thereto. We also note that the authors heavily optimised TinySec for its application domain: data messages are restricted to 30 bytes or less, and integrity-protection data is only 4 bytes (rather than the now-required 8).

The choice of cryptographic primitives also deserves mention: all cryptographic algorithms were implemented in software. The block ciphers used in TinySec, RC5 and Skipjack, were therefore selected for their low resource requirements. (On a modern sensor platform, we expect that AES would be used, since hardware implementations of AES are now commonplace.) The overhead imposed by TinySec, compared to the base TinyOS network stack, was found to be roughly 10% in energy, latency, and transmission overhead.

Finally, we note that the authors chose not to implement replay protection in TinySec. This appears to be a function of the implementation, rather than the protocol itself: the IV of encrypted packets includes a counter, which is incremented each time a packet is sent. However, the TinySec implementation does not record the counter values of received packets. This is in order to conserve memory: the authors considered the requisite 2 bytes of state per communication partner to be too high a resource cost for the devices then available.

3.3.5 MiniSec

MiniSec [102] is a network-layer protocol for TinyOS, and an evolutionary step from TinySec. The data messages it can carry are of the same length as TinySec. The

guarantees it provides are the same as TinySec, with the addition of replay protection. The cryptographic mechanism is similar: authenticated encryption with associated data (AEAD), using Skipjack as the block cipher, and a 4-byte authentication tag. The AEAD mode in use is not CCM, but another mode, OCB, which requires fewer block cipher operations (and is therefore faster), but is patent-encumbered, inhibiting its adoption.

MiniSec provides a pair of protocols: MiniSec-U implements unicast transmissions, and MiniSec-B broadcast and multicast transmissions.

Like TinySec, MiniSec does not specify a key distribution model. However, the MiniSec paper assumes a pairwise keying model, rather than the network-wide one assumed in TinySec. The conconmitant tracking and storage of pairwise keys implicitly assumes enough available resources on each network node that keeping track of per-partner packet counters is no longer a problem.

At the same time, MiniSec avoids allocating an explicit header field for the packet counter. Instead, nodes are expected to maintain rough synchronisation, and only the lowest bits of the counter are stored in the packet itself, overloading the 3 most significant bits of the frame length field – they are ordinarily unused, given the small maximum message size. The MiniSec-U variant also includes a counter resynchronisation protocol in case the nodes' state becomes desynchronised.

In the MiniSec-B variant, the packet counter is only 8 bits long, and is sent in its entirety; in this case, rather than adding a counter field, the remaining 5 bits are taken from the destination address field, on the assumption that they are not required for addressing.

3.3.6 TinySA

TinySA [48] is another evolutionary step along the path laid by TinySec and MiniSec. Each node in the network is provisioned with an ECC key pair, signed by a master signing key. These ECC keys are then used in a ECDH protocol to establish pairwise link keys. A node will only communicate with another node if both of their public keys have been signed by the master key.

Unfortunately, little more information is available.

3.3.7 SPINS

SPINS [134] is a WSN protocol suite for TinyOS. It provides similar security guarantees to MiniSec: data confidentiality, integrity, authenticity, and freshness. All cryptography in SPINS – much like TinySec and MiniSec – is symmetric.

Moreover, SPINS is designed to take node compromise into account: it attempts to limit the scope of such a compromise to only the resources held by that node. However, it does assume the existence of a base station with network administration responsibilities; compromise of the base station results in compromise of the network in its entirety.

SPINS consists of two subprotocols, divided in a similar manner to MiniSec: SNEP provides unicast packet transport, while µTESLA provides authenticated broadcast

and multicast transmission.

SNEP SNEP strongly resembles TinySec in both design and choice of cryptographic primitives. However, unlike TinySec (or, indeed, MiniSec), SNEP omits the value of the counter entirely from the packet. Nodes must instead maintain a count of packets received. Implicitly, only one packet can therefore be 'in flight' at a time, as opposed to the window provided by MiniSec's partial counter scheme. While this limits the throughput of the system, it also provides automatic replay protection, since packets automatically expire upon successful decryption and authentication. SNEP also includes a protocol, similar to that in MiniSec, for resynchronising counters, should this be necessary.

Communication between any pair of devices requires a *master secret*. Four keys are then derived from the master secret: for each direction of transport, separate authentication and encryption keys are needed. Nodes are provisioned with master secrets for communication with the base station. The base station then generates master secrets for other pairs of nodes when they establish a communication channel.

µTESLA µTESLA is a variant of the TESLA [133] protocol for authenticated broadcast, modified for the resource constraints of WSNs.

TESLA has two phases. During the setup phase, a potential sender generates a symmetric key. It hashes that key N times, with N the number of broadcasts it is likely to send, forming a hash chain of keys. It then digitally signs the *last* key in the chain, and broadcasts it to the entire network. This signature is the only public-key operation it will be required to perform. Other nodes on the network verify the signature, guaranteeing both the integrity and source of the initialisation message. This finishes the setup phase. During operation, for each broadcast transmission, the sender selects the next key *backwards* along the hash chain, encrypts and authenticates the message, and broadcasts the so-enciphered message. (If separate keys are used for encryption and integrity protection, they are both derived from the selected key in a predictable manner.) The sender then waits for a period *exceeding the time the message should take to reach the edge of the network*, and broadcasts the chosen key. Receiving nodes can then decrypt and verify the broadcast. Authenticity is checked by verifying that the key exists in the hash chain; this mechanism also detects missed broadcasts.

For µTESLA, the authors note that the constraints WSN hardware prohibit two operations on sensor nodes: digital signatures, and packet broadcasts to the entire network. They also remark that such nodes may not possess sufficient storage to hold hash chains of broadcast keys, and that transmitting a broadcast key for each broadcast message is energy-intensive, since it requires two packet transmissions. µTESLA therefore modifies TESLA in the following ways:

- The base station acts as trusted third party for authenticated broadcasts. A sensor node wishing to perform a µTESLA broadcast may request (via SNEP) that the base station perform the broadcast, or it may perform the broadcast itself, requesting the broadcast key from the base station.

- The base station will reveal the broadcast key in use at a fixed interval, called the *epoch*. The key so revealed decrypts and authenticates all broadcasts performed during that epoch.
- Authenticity of the broadcast key is bootstrapped using SNEP between the base station each sensor node individually, rather than using digital signatures.

3.3.8 Bluetooth

Bluetooth [18,55] is an industry-standard protocol for low-power communication between embedded devices. It has undergone a number of revisions, each improving either its speed, power consumption, or feature set. As with IEEE 802.15.4-based protocols, including StarfishNet (presented in Chapter 7) and ZigBee (see Section 3.3.10), it operates in the 2.4GHz (ISM) band. It uses a pairwise keying model, with a variety of available key establishment mechanisms.

Its stated purpose is point-to-point links between proximal devices, in a master-slave configuration. Many slaves may communicate with a single master, forming a *piconet*; only 7 of these may be *active* at any given time, with any others *parked* until the master unparks them.

A node may belong to several piconets at the same time. However, it may be the master of at most one piconet, and an active slave in at most one; it must be a parked slave in any others. In addition, the radio may only be tuned to the carrier frequency of one piconet at a time; a node must therefore context-switch the radio in order to participate in several piconets.

Bluetooth makes no provision for routing or forwarding; all slaves in a piconet must be within direct transmission range of the master.

Bluetrees [175] uses Bluetooth to construct a routing tree not unlike that in ZigBee (see Section 3.3.10), or indeed StarfishNet (see Chapter 7), permitting the formation of ad-hoc networks. However, the fact that a Bluetooth radio can only be tuned to a single piconet at a time limits its effectiveness: inter-piconet routing is achieved by switching piconets when necessary, with the predictable result that routers can only receive transmissions from a subset of their neighbours at any given time.

Bluetooth has also been subjected to a wide range of attacks, both on the software stacks in operating systems supporting it, and in the air protocol itself (see, for example, [56]).

3.3.9 6LoWPAN

6LoWPAN [66,119] is the standard implementing IPv6 over IEEE 802.15.4. However, it is not a simple standard: IPv6 requires a minimum datagram size of 1280 bytes from every link on which it is to operate, and has a minimum header size of 40 bytes. Given that the maximum frame size of IEEE 802.15.4 is 127 bytes, the standard therefore also describes a fragmentation and reassembly mechanism for IPv6 datagrams, and a header compression scheme.

Much of the work on applying IPv6 to the IoT uses UDP as the transport layer protocol. This is because TCP has inappropriate performance characteristics for IoT networks; as Atzori *et al* describe in [13]:

- TCP's connection-oriented nature, and the overhead involved in connection setup, are suboptimal for ephemeral interactions;
- TCP congestion control is designed for long-lived connections, with no provision for short, ephemeral interactions;
- TCP congestion control displays pathological behaviour on wireless networks; and
- TCP requires large amounts of memory on the receive side for reassembly of fragments and reordering of packets received out of order, which may not be possible on sensor nodes.

6LoWPAN, as a variant of IP, does not provide any security services. It must instead be combined with some other mechanism for end-to-end encrypted and integrity-protected data transport. Work on several such mechanisms is described below.

IPsec IKEv2/IPsec [79–83] is a widely-deployed, standards-based mechanism for end-to-end encryption and authentication between IP-based nodes. However, the protocols were not designed with the strict packet-size constraints of IEEE 802.15.4 in mind; their headers are larger than is practical on this medium. Recent work by Raza *et al.* [136] introduces compressed header formats for the IPsec protocols for exactly this reason. With this header compression, IPsec imposes a total per-datagram overhead of 26 bytes (including the compressed 6LoWPAN header). UDP imposes another 8 bytes, for a final per-datagram overhead of 34 bytes. Their experiments used pre-shared keys for authentication.

HIP The Host Identity Protocol (HIP) [75,120] is an alternative key exchange mechanism for IPsec. Unlike IKE, HIP headers to not contain any IP addresses, instead only exposing cryptographic identities, in order to improve compatibility with NAT and multihomed nodes. HIP is broadly similar to TLS session establishment: nodes perform a DH key agreement using ephemeral keys, authenticated using digital signatures.

A variant is currently under development for embedded systems, named HIP Diet Exchange [121,126]. To reduce the resource requirements of the protocol, this variant both requires the use of ECC, and eschews digital signatures entirely (thereby also eliminating the need for a hashing primitive. The DH keys used are also static, rather than ephemeral; as a result, the key confirmation can also be used to authenticate the remote party, since this verifies that it possessed the correct private DH key. Unusually, the DH shared secret is also *only* used to authenticate the HIP exchange; a separate key exchange, tunnelled within the HIP session, must be used to establish the key for IPsec communications.

PANA Protocol for Carrying Authentication for Network Access (PANA) [42], is a lightweight UDP-based protocol for carrying an EAP transaction over a routable network. Its primary usage in an IoT context is authenticating 6LoWPAN nodes to a 6LoWPAN border router [61,145], such as in the 6LoWPAN variant ZigBee IP [179]. No performance measurements in an IoT context appear to be available.

DTLS DTLS [138] is a variant of the common TLS/SSL security protocol, adapted for use on a datagram-only transport, such as UDP. Tschofenig *et al.* [168] examined the performance of DTLS on a WSN platform, using 6LoWPAN. They found that a DTLS handshake required 12 messages to complete a handshake, using authentication based on pre-shared keys. The results seem to imply that 6LoWPAN datagram fragmentation was not required; that is, each message could be transmittedinvoked-invoked entirely in a single IEEE 802.15.4 frame. They also indicate a total of 62 bytes per message in header overhead. This includes IEEE 802.15.4 link-layer headers, IPv6 headers (which could not use 6LoWPAN header compression), and UDP headers. Over all 12 messages, the DTLS handshake required the transmission of a total of 487 bytes.

A later development [67] presents a cryptographic delegation architecture for DTLS, permitting less powerful IoT nodes to delegate cryptographic operations for which they do not possess the resources to another, more powerful node on the network, the 'delegation server'. Those nodes must share a secret with the delegation server, which will generally be provisioned out-of-band. The delegation server can then use this key to relay the shared keys established during DTLS session setup to the sensor node. If the sensor node is not capable of public-key cryptography at all, the authors note that the delegation server can also use this architecture to perform coarse-grained access control, refusing to mediate DTLS handshakes with certain nodes should its access control policy forbid them. The delegation server's active involvement in DTLS sessions also greatly simplifies the revocation problem.

3.3.10 ZigBee

ZigBee [177] is the current industry standard network-layer protocol in the for Internet of Things networks. It is based on IEEE 802.15.4, and with security enabled, provides similar security to StarfishNet. However, all of its cryptography is symmetric – the key distribution is solved by provisioning each node with a *link key* to the Trust Centre, and having the Trust Centre in turn both manage the *network key*, and generate link keys between all other pairs of nodes. This eliminates the requirement for nodes to possess random number generators, but at dire cost if the Trust Centre is compromised.

The ZigBee Smart Energy Profile [178] introduces public-key authentication and key-establishment mechanisms based on ECC, in order to reduce the administrative burden of managing symmetric keys. However, the specification still recommends that its use be minimised; preferably only to establish the aforementioned Trust Centre link key, and once again delegating all further key generation to the Trust Centre.

Key type	Key function
Network	Packet security at network or application layer.
Link	Packet security at application layer only.
Master	Master secret for establishing link keys in high-security mode.

Table 3.1: ZigBee key types, and the layers at which they are used

Additionally, the ZigBee join protocol is highly complex, possessing multiple variations and execution traces. The authenticated variants can require between 8 and 36 packet transmissions. (The former in the case where only network keys are in use; the latter in the ZigBee Pro case, in high security mode – that is, with both link and master keys in use.) Both of these counts include exchanges between the edge router and Trust Centre, and acknowledgements thereof; these number 4 in the first case, and 14 in the second.

The large number of packet transmissions in the ZigBee join protocol is due in some part to the layering choices made: key establishment protocols are at the application layer, rather than the network layer. The network layer therefore treats them as ordinary messages, causing acknowledgement packets to be generated. Were this not the case, the network layer could treat key establishment packets as implicitly acknowledging each other, a technique we adopt in StarfishNet.

A recent study [3] conducts a detailed security analysis of ZigBee, along with WirelessHart [59] and ISA100.11a [73] (two other IEEE 802.15.4-based network protocols that bear strong similarity to it), thoroughly exploring a variety of attack vectors.

ZigBee Security

This section is based on information in the ZigBee [177] and ZigBee Smart Energy Profile [178] specifications, with some input from [3,15,31,69,95,154]. It assumes the use of a ZigBee security level which mandates both encryption and integrity protection of all packets.

In the ZigBee model, a packet may be encrypted with one of two broad key types: *link keys* are keys agreed pairwise between nodes, while the *network key* is known by all nodes on a ZigBee network. However, link keys are only available at the application layer; network-layer packet security can only use the network key. In high-security mode, an additional key type (the *master key*) is also available, which is used to periodically renegotiate link keys.

Packet encryption is done using AES-128 in the CCM* block cipher mode (a minor variation on the CCM [36] mode, described fully in the ZigBee specification). This mode is an authenticated encryption with associated data cipher; data can be encrypted, integrity-protected, or both, and additional plaintext can be included in the integrity check. In ZigBee, this *associated data* includes all headers at the network layer and above.

A summary of key types and their function is provided in Table 3.1.

The Trust Centre ZigBee security calls for a *Trust Centre*: a single node (usually the network coordinator), which is responsible for the generation and distribution of keys to all other nodes on the network. Devices are initially provisioned with a link or master key for the Trust Centre only; the resulting secure channel is then used to distribute all other keys.

The Trust Centre also serves as the network's cryptographic hub. It generates keys (master keys in high-security mode, link keys otherwise) for pairs of nodes that require secure end-to-end communications. It acts as a trusted intermediary for high-security link key establishment. It is also responsible for periodically refreshing the network key.

Finally, the Trust Centre controls admissions to the network. Routers are required to consult it upon receiving a request to join by a new node, and it can instruct routers to eject existing nodes at any time.

The way in which these responsibilities compose can lead to pathological behaviour, with the network key providing a prime example. Ordinarily, a refresh of the network key requires $O(1)$ (that is, constant) work: the Trust Centre generates the new key, encrypts it with the old key, and broadcasts this message to the entire network. However, if a node is ejected from the network, the network key must be changed in way that would prevent the exiting node from learning it. The only mechanism available to the Trust Centre to accomplish this feat is for it to transmit it to each other node on the network individually, encrypted with their Trust Centre link keys. This is an $O(N)$ operation – that is, linear in the size of the network.

ZigBee Smart Energy Profile The ZigBee Smart Energy Profile [178] introduces a key-establishment protocol based on ECC. In the specification, it is strongly recommended that this protocol only be used at network-admission time, to establish a link key between the Trust Centre and the device being admitted; the Trust Centre's cryptographic functions should be use to establish link keys with other devices on the network.

3.4 Higher-layer protocols with relevant features

3.4.1 UACAP

UACAP [109] is an authentication framework designed for spontaneous interactions. The framework largely implements the multi-channel authentication concepts from Section 3.1.6. The base protocol is a DH exchange, augmented with authentication data from an auxiliary channel, on the expectation that in the spontaneous interactions for which it is designed, such auxiliary channels are likely to exist.

3.4.2 Kerberos

Kerberos [158] is an early example of a distributed access control system, using only symmetric cryptography. It implements the Needham-Schroeder-Lowe protocol [101].

It requires separate directories of subjects and objects to be kept. However, in doing so it separates the act of authenticating as a user, to the act of invoking a service; in particular, services need never see the user's password. The directories are known as the *Authentication Service (AS)* and *Ticket-Granting Service (TGS)*, respectively. Subject authentication is handled by the AS, which issues a *Ticket-Granting Ticket (TGT)* to an authenticated user. Before a service request is made, the TGT is the presented to the TGS along with the details of the request; the TGS will then make an access control decision on the service to be requested, and issue a *client-to-server ticket*. This final ticket is proof to the service that the subject is both *authenticated* – that is, the request is coming from a valid subject – and *authorised* – it is permitted to access the service in question.

3.4.3 CapBAC

CapBAC [54] is a capability-based access control system designed for IoT applications. The capability data structure itself is written in a dialect of XACML [100], and digitally signed. Delegation of rights is supported: a capability representing delegated rights is signed by the holder of the rights being so delegated, and contains a reference to the capability containing the rights being delegated – that is, its parent capability – forming a tree of capabilities. The capabilities at the roots of these trees are signed by the owner of the object itself.

Each CapBAC capability names its holder, and is signed by the holder of its parent. In order to avoid depending on a PKI or similar trusted naming service, public key hashes may be used as names. This mechanism can also be used as a pseudonymity mechanism, should this be required..

Revocation is accomplished by two means. First, capabilities can be created with an expiry time, with a revocation of rights taking the form of a refusal to reissue the capability. Second, potentially to avoid a dependency on a global time source, the holder of a parent capability may explicitly revoke one of its children. Of course, this revocation must then be published.

3.4.4 Grid computing

Grid computing [44] describes the federation of disparate, usually heterogeneous, computational resources, sometimes in different ownership domains, towards the execution of large, distributed computing workloads.

Globus Globus [43] is a grid middleware stack that provides for the implementation of such a system. Core to its security model is the *proxy certificate*: a digital certificate temporarily delegating the identity of its issuer. The job submitter generates such a certificate, with a short expiry time – approximately the time the job is expected to take to complete, and signs it with his/her private key.

MyProxy MyProxy [88] is a credential storage service for Globus keypairs, so that users need not deal with the complexities of key storage. SHEMP [105] extends

MyProxy to use Trusted Computing techniques to anchor that key store to the machine serving it, in order to protect it against theft. Both of these simplify the use of proxy certificates to some extent, but they do nothing towards addressing their fundamental inadequacies.

Trusted Grid The Trusted Grid [29] specifically addresses the malicious grid host problem using Trusted Computing (covered in Section 2.3) and virtualisation. In a related vein, Daonity [104] uses the same technologies to assert the compliance of various grid stakeholders with each other's security policies. However, the granularity of these systems is coarse, going only as far as grid admission control – that is, deciding whether a node should be admitted to, or ejected from, the grid. They do not cover access controls applied once a node has been admitted.

3.4.5 Web of Things

The Web of Things [34] refers to the principle that machines on the Internet of Things should be able to expose services with Web-like interfaces. The idea was originally presented as a mechanism for easing the use of wireless sensor networks [20]: smart gateways bridging between WSN and Internet protocols would expose Web services which would only be a thin abstraction over the services exposed by sensor nodes, themselves written in a RESTful/SOA style. This idea was later extended to Web services being directly exposed by sensor nodes themselves [34,51], with smart gateways continuing to act as bridges to sensors which cannot.

With sensors exposed directly to the Internet, and exposing Web services to generic clients, later work introduced Web-like search primitives based on the network topologies of the sensor networks themselves. One proposal [108] amounted to partially flooding the network with a search query, allowing intervening routers and gateway nodes to restrict which areas of the sensor network are actually flooded according to the physical layout represented by the network topology. This could also be used as a coarse-grained access control system, by directing the flood based on the request's origin in addition to the query it contains.

3.4.6 Web-based authentication schemes

Federated Web identity Federated Web identity systems, such as OpenID [137] and Shibboleth [125], allow users to authenticate to multiple domains using a single set of credentials. Each domain is responsible for implementing and enforcing local access controls based on the user's identity, irrespective of the service providing that identity, while identity providers essentially act as large, open directories. Shibboleth is notable for providing more flexible mechanisms: the protocol permits more complex logic to be offloaded from the service to the directory. It can therefore be made to act in a somewhat authority-oriented manner – with the service simply supplying its access control rules, and the directory performing both authentication and authorisation on its behalf.

OAuth OAuth [57] is an identity delegation protocol for Web services. It does not perform cross-domain authentication itself (it is indended that OpenID be used for this purpose). It instead allows a service to, once authentication has been established, create an *authorisation token* which authorises another party to perform future actions on behalf of the authenticated identity. OAuth is not authority-oriented in the sense meant in this thesis; the authorisation token does not represent authority to perform an action, but rather the right to wield an identity in a limited way. As such, it bears a much closer resemblance to the proxy certificates discussed in Section 3.4.4 than a true authority-oriented system. Interestingly, OAuth tokens can be anonymous: the token holder does not necessarily know which identity it is using.

Webinos Webinos [45,103] is a Web-based authentication and access control system designed for the Internet of Things. It is based on the concept of the *personal zone*: the set of all devices owned by a particular user. This zone is centered on a *personal zone hub (PZH)*, which acts as the root of a small public-key infrastructure, issuing certificates to each device within the zone. Access control logic for the zone is encoded in an XACML [100] policy that is replicated throughout the zone, with the authoritative copy held at the PZH; the policy may be modified by any device in the personal zone, with devices trusted to implement access controls where necessary. Devices belonging to one user can identify themselves to another's PZH using OpenID; the users' OpenID identities can be used in their policies to allow requests by their devices to be authorised in a fine-grained manner. Each device is, however, still assumed to be owned by a single user, and actions by devices on resources outside the personal zone are performed using that user's delegated identity.

3.5 Language and operating systems security

3.5.1 Filesystems

The goal of filesystem security is to store access control metadata that would permit access control decisions to be made and enforced by the operating system. The design of security mechanisms to achieve this is is largely based on the OS security principles first explored in the Multics [151] time-sharing system. Its authors first explore the need for filesystem-level access control in general. They then describe the two fundamental ideas that would become the core of OS security: the association of user IDs with processes, and a hierarchical filesystem with an ACL associated with each file.

Modern filesystems have not deviated from this original conception; even Unix-style file permissions are a form of simplified ACL. ACLs remain the primary form of access control metadata stored in filesystems, including the default filesystems for commodity OSes: NTFS [114] for Microsoft Windows, HFS+ [8] for Apple Mac OS X, and Ext4 [107] and Btrfs [140], among others, for Linux.

The security provisions of network-based file access protocols, such as NFS [23,60], Microsoft CIFS [115], and Apple AFP [7], largely focus on the authentication of a

client to a file server, as a user ID on that file server. This permits the OS on the server to treat those requests as any other filesystem requests, enforcing access control policy accordingly. These protocols also provide mechanisms for editing ACLs on those files.

Finally, WebDAV [35] adds functionality to HTTP to permit it to be used as a file access protocol. Its security provisions are identical to the other protocols above.

3.5.2 Capability-based programming languages

Several programming language projects use capabilities for both encapsulation and access control of data objects. In general, they leverage the idea that a reference to an object is essentially a capability to that object, provided the language's security model prohibits references from being forged. Newspeak [19] is a language with no global namespace, resulting in Newspeak code possessing no *ambient authority* – that is, authority simply by virtue of executing. Instead, the resources to which an object has access are entirely described by the (transitive closure of the) set of object references it possesses. Repy [24] is a subset of Python, while DCC [41] and Joe-E [113] are subsets of Java, achieving similar goals. Inheriting from Joe-E is the E [116] distributed capability language, which eschews the Java language basis of Joe-E in favour of an entire new language, albeit one still based on the Java Virtual Machine. In all of these cases, language-level object orientation and memory-safety are responsible for the enforcement of access control policy.

Security Meta Objects [139] are a variation on this theme. They are functionally identical to capabilities; however, semantically, they provide for a separation between references and the access control policy they represent.

3.5.3 Capability-based operating systems

Several OS kernels, both in academia (including seL4 [87], Fiasco [90], Mach [1], and EROS/KeyKOS [152]) and industry (including OKL4 [62], CapROS[1], and Annex [49]), apply capability-based access control models to kernel resources, while protecting the capabilities themselves using virtual memory mechanisms. Such a model appears to be more representative of the security context of a variety of single-user systems – such as mobile phone baseband chipsets, or dedicated router devices – than Unix-style user accounts. In particular, it natively permits the flexible delegation of resources to the individual processes that need them, as those needs arise.

E (see Section 3.5.2) and Annex additionally support capability-based access to objects over a network. The mechanism used is the *password capability*, originally used in the Password Capability System [5]: each capability is associated with a 'password', a secret chosen randomly from a large space. Possession of a valid password is considered equivalent to possessing the capability itself, and grants access to the rights it describes.

A more sophisticated variation on this theme involves using an encrypted data structure, rather than a random secret, as the capability token. Implementations can

[1] http://www.capros.org/

be found for memory protection [99], high-performance distributed storage [47,127], rights delegation in identity-based systems [96], and managing cell handover in mobile telecommunications [22]. Kerberos tickets could also be considered a form of this.

Networked capability systems suffer from a substantial weakness: disclosure of the token – whether a simple secret, or an encrypted data structure – implies leakage of the authority it represents. The resultant access control failures are often difficult to detect; and double-spending [128]-like problems are also a risk. Additionally, since capability-based systems explicitly do not attempt to authenticate the identity of a requesting principal, the establishment of a secure channel must be handled explicitly. In a closed system, this can require trusted network infrastructure; for example, the Amoeba [122] and Accent [135] distributed OSes (the latter designed for sensor networks) require that hardware network addresses be unforgeable. In an open system, such a requirement is unrealistic; the authorisation data must instead be bound to the session in some other way. Social PaL [124] uses a DH key exchange, augmented in a manner reminiscent of multi-channel authentication (see Section 3.1.6), to achieve this.

Chapter 4

Motivation: threats and requirements

This chapter begins with Section 4.1, which presents a threat model for the Internet of Things (IoT). While some previous work has been done on this problem, to our knowledge, this is the most complete such model yet. This model concludes with an aside (Section 4.1.2) on one particular aspect of the IoT networking environment, the high likelihood of short, ephemeral transactions, and how that heavily influences the threat model in a non-obvious way.

Section 4.2 then distils this threat model into a set of broad requirements for IoT security systems, which Section 4.3 applies to the systems described in Chapter 3.

4.1 Threat model

A comprehensive threat model for the Internet of Things (IoT) remains absent from the literature. Some previous work has been done, identifying various classes of threats – in particular, [12,14] regarding IoT security, and [132] regarding 'personal networks', their conception of which bears some similarity to IoT, as well as work on threat models for WSNs[148]. This section expands and builds on that work, presenting a more extensive description of possible attacks against IoT devices.

The model is divided into three sections. First, the *threat taxonomy* describes our full IoT threat model. It is followed by a remark on ephemeral interactions, and the particular threats they present. Finally, the *adversary model* describes, based on the preceding two sections, the capabilities of the adversaries against which BottleCap and StarfishNet, the systems described in this thesis, will protect.

4.1.1 Threat taxonomy

This taxonomy consists of three parts: *attacker classes* describe classes of entities interacting with the system (and thus able to launch attacks), *attack vectors* describe mechanisms by which attacks can be launched, and *attack goals* describe the attack's intended effect.

Attacker classes

Device manufacturer The manufacturer of an IoT device.

In practice, treating the manufacturer as a single entity is an oversimplification: the notional manufacturer of a device is, in fact, merely at the head of a very deep supply chain. Every entity in this supply chain may will likely have similar, but not necessarily identical interests and incentives. All are capable of a wide variety of attacks.

This can be modelled as a device having more than one manufacturer. A motivating example is a mobile phone, in which the OS vendor and physical handset manufacturer (with its attendant supply chain) both retain some level of control over it.

Device user The entity in physical possession of the IoT device, with the potential to perform local or network-based attacks to learn the device's secrets, or use the device's functionality contrary to the manufacturer's intent.

We note that a device may have more than one user. A motivating example of this scenario is a dispenser of some consumable, which reports levels of that consumable to its supplier, so that refills are ordered automatically.

Device controller The entity with administrative control of the device. This may, or may not, be one of the device users. In particular, it is *not* if the device is remotely-administered (in which case, the remote administrator is the device's controller).

A device may have more than one controller. Once again, a motivating example is a mobile phone, in which the device user and the cellular network operator both exercise some level of administrative control.

Authorised insider An entity which has some authority in respect of the device, but which does not own it. That authority is often, but not always, delegated by one of the manufacturer, controller, or user.

Unauthorised insider An entity with the ability to communicate with the device, but no explicit authority to do so. This generally represents a second device on the same network as the first. These devices may, or may not, share a user, controller, or manufacturer.

Outsider An outside entity with no legitimate access to the device or network.

We note that the notion of ownership for IoT devices is complex. The motivating example of the mobile phone, which may have multiple manufacturers (even without taking its supply chain into account), multiple controllers (the cellular network and the end-user), and at least one user is an instructive one. Devices situated in one location, but maintained – both in terms of their hardware, such as refilling consumables or replacing batteries, and software, such as performing updates and configuration changes – by an entity located elsewhere complicate this issue further.

We have attempted to model this using the manufacturer/user/controller distinction, and allowing each of those roles to be taken by multiple entities concurrently.

Attack vectors

Hardware The interception of communication buses inside a device, or similar observation of and interference with its internal state.

Hardware attacks may result in the physical destruction of a device, such as the physical decapping of flash memory chips in order to copy their contents. This is of particular concern if those contents include cryptographic key material.

Software Software running on a device, irrespective of its control vector.

Local Applied to a software attack, indicates interaction with the device via built-in control interfaces, such as displays, buttons, and keypads. This can include recessed or concealed buttons, access to which requires partial disassembly of the device.

Remote Applied to a software attack, indicates interaction with the device via communication links to which the device is connected, such as an Internet connection, the connection to the control network, or a private management network. Also referred to as a *network attack*.

Impersonation Network attacks based on the misattribution of the source of the attack. This includes replay attacks, message forgery, or theft of credentials.

Routing Attacks based on manipulation of network infrastructure in order to cause messages to be lost or delivered to a node other than their intended destination, or to appear to come from somewhere other than their true source.

By-proxy Using information, credentials, or rights granted to some third party, either by controlling the behaviour of that third party, or by attacking it in some other way. (Analogous to a privilege-escalation attack on a desktop system.) Confused-deputy attacks are good examples of by-proxy attacks.

Signal injection Manipulation of environmental sensors by physical interference with their operation, such as affecting the reading given by an electronic compass through transmission of interfering electromagnetic signals.

Side channel An attack based on information which is below the level of abstraction of the interface being attacked. For example, attacks based on timing, analysis or manipulation of power consumption, traffic analysis, fault analysis, or electromagnetic analysis are all in this category.

Attack goals

Denial of service (DoS) Preventing the provision of some service entirely. This may mean blocking access to a physical resource controlling the device, or blocking network access to the device itself.

Theft of service The provision of some service without any knowledge on the part of its provider.

Auditing failure The provision of some service that is accounted for and/or audited incorrectly. This is distinct to *theft of service*, where service is provided entirely without record.

Economic The cause of long-term economic harm by one party to another, such as by damage to reputation or by causing a long-term increase in the use of some billable service. (Importantly, in this case all records kept are accurate.)

Reputation Seeking fame, or other effect on reputation, on the part of the attacker.

Breach of confidentiality The exfiltration of confidential information, or of information in a confidential form. This includes both the disclosure of confidential information by technical means, such as breaking a cryptographic primitive, and the violation of information disclosure policies, such as a privacy policy.

Breach of contract The exercise of legitimate authority in an illegitimate way or at an illegitimate time. The most obvious example is the inappropriate (but permitted) transmission of control signals to a device, but violation of a privacy policy may also notably fall into this category.

Information concealment A failure to disclose relevant information, at a time when that information should have been revealed. Information concealment attacks are often also breaches of contract, but not always. Network DoS attacks may also be done with this ultimate intention.

Collusion The perpetration of any of the above attacks by multiple entities acting in concert.

Infrastructure sabotage The perpetration of any attack on such a large scale that its effects may damage or disable supporting infrastructure. This could be, for example, a DoS attack against core routers, affecting a substantial fraction of the Internet, or a malicious control signal to a large number of residential IoT devices causing a failure in the electrical grid.

4.1.2 Ephemeral interactions, a further constraint

In the IoT literature – for example, [13,14,118,132] – a heavy emphasis is placed on the need for flexibility of network architecture, in order to support the rather challenging combination of highly heterogeneous devices, and the potential for large

numbers of short-lived interactions. We observe a third major contributing issue to such architectural concerns: the relevant actors cannot be determined ahead of time. If they could, a long-lived relationship could be formed, obviating the need for supporting ephemeral ones.

This presents a novel problem, and one essentially unaddressed by any of the work discussed in Chapter 3, or indeed the cross-section of the literature cited briefly at the start of this section: the entities interacting on an IoT network may be mutually distrusting. Despite this distrust, they must still exchange data and control signals (since otherwise they would likely not interact). *This includes any entity controlling the security infrastructure.* In fact, there is often a strong incentive to *be*, or become, that controlling entity, since it permits manipulation of that infrastructure in ways which can subtly (or overtly) attack other entities on the network. Ideally, therefore, security systems on IoT networks should not possess such a controlling entity; they should be constructed in a decentralised fashion, with no central control points. (Heer *et al* mention some of this in [61], but do not take the argument to its conclusion.)

4.1.3 Adversary model

Section 4.1.1 presents a wide variety of attacks. However, this thesis has a particular focus: the provision of mechanisms for guaranteeing the authentic and confidential transport of messages over a network, (that is, communications security), and for communicating policy regarding the actions that may be taken by their recipients (that is, access control). This focus necessitates a narrower set of threats to actually be considered during the rest of the thesis. This section describes the criteria by which the set of threats in Section 4.1.1 are narrowed for the purposes of this work.

The first narrowing criterion is one of scope. This thesis focuses on the architecture of networked security systems. Therefore, although an entity in physical possession of an IoT device may attack it a variety of ways, the mechanisms for protecting against such attacks are outside the scope of this work. We therefore ignore all attacks requiring physical possession of a device. This means the following vectors: *local software*, *hardware*, and *signal injection*. (That is, we accept the existence of these attacks, but we do not attempt to protect against them.)

Second, this thesis focuses on describing protocols and providing mechanisms. We therefore do not consider attacks on implementations. In particular, attacks due to bugs in the software code implementing security primitives, cryptographic algorithms, and applications are all out of scope. Also excluded from consideration are errors in access control policy, as are side channel attacks, as artefacts of implementation.

In a similar vein, the mechanistic focus of this thesis precludes distinguishing between attacks with different goals. We therefore do not attempt to, with one exception: denial of service. This class of attack is well understood to be possible at any layer (see, for example, [148]). While we do not ignore DoS attacks entirely, this reality indicates that they simply cannot be prevented. We will therefore mention them where relevant, but not actively attempt to defend against them.

Finally, due to the ephemeral nature of many IoT communications (as discussed in Section 4.1.2), there is no simple way to distinguish between insiders and outsiders;

furthermore, even authorised insiders are not necessarily benevolent. We therefore do not differentiate between different attacker classes. Instead, we provide strong identity guarantees to higher layer software, so that these distinctions, if relevant, can be drawn there. Our attacker model is therefore essentially Dolev-Yao: we consider a remote attacker with a high degree of control over the network medium. That attacker additionally has very powerful computational capabilities, and potentially some level of legitimate access to some subset of the network in aggregate. The only operations they cannot perform are those forbidden by cryptographic assumptions – that is, they may not perform any action requiring a key without first learning that key.

4.2 Requirements

The above discussions of threats and architectural needs can be reformulated into four broad requirements for security systems on the IoT – that is, systems that implement and enforce security policy:

R.1 Realistic deployments of IoT security systems will necessarily have to take into account the possibility of interactions between competing entities. The architecture of such security systems must therefore be decentralised, to ensure that no single entity controls the security infrastructure.

R.2 IoT security systems must remain secure in the fact of a strong adversary – the threat model in Section 4.1 suggests treating all nodes as potential Dolev-Yao [32] attackers.

R.3 IoT security systems must not require more resources (whether computational, performance, or networking) than are available on modern sensor platforms.

R.4 IoT security systems must nonetheless not restrict communications between nodes in arbitrary ways.

Each of these merits some explanation.

R.1 Decentralised communications is motivated by the argument in Section 4.1.2: networks may consist of mutually distrusting entities. Placing one such entity in a position of controlling the security infrastructure permits powerful and damaging attacks against others. A decentralised system architecture can also confer performance advantages: necessarily, the fewer parties must be involved in a transaction, the less communication is required to execute it, and therefore the larger the total number of transactions that can be supported by the system [28,112]. This is a particularly pertinent issue on the IoT: each packet transmission costs battery power, so there is a strong incentive to minimise the number of such transmissions each node must make, and maximise the efficiency with which network resources are used.

Decentralised control carries an adjunct responsibility: local policy enforcement. The policy underlying access control decisions may – and often will – be transmitted

to an IoT node from a remote policy engine. However, that node must be capable of enforcing the policy independently, since its network use must be minimised. More subtly, the policy engine in use must not be mandated by the *network* to which a node is connected, but by its owner(s) and/or controller(s).

Moreover, decentralisation does not necessarily exclude the possibility for a single entity to *observe* the current state of the system in aggregate. In many applications, there is merit in ensuring that this is possible. However, again, this should be mandated by the design and engineering of the application, not by that of the network infrastructure.

R.2 Section 4.1 presents a varied taxonomy of possible attacks against a networked system. The key insight underpinning them is that the IoT is part of, and connected to, the Internet. This is critical: the Internet itself is a hostile networking environment, on which all of the attack classes presented are not only possible, but realistic. The Dolev-Yao [32] adversary model applies perfectly to this environment, especially in the presence of mutually distrusting, interacting entities. This is particularly acute for wireless networks, on which this thesis is focused: to repeat the quote from [10], "Wireless network security is different from wired security primarily because it gives attackers easy access to the transport medium."

This requirement therefore carries a set of responsibilities: security systems (especially those designed for communications security) must make strong guarantees regarding the confidentiality, integrity, and authenticity of the messages they exchange.

R.3 At the same time, IoT nodes are highly constrained devices. Their processing capacity, in the worst case, is roughly equivalent to that of a WSN node; the same applies to the amount of available storage. This excludes from use many common cryptographic mechanisms available on the desktop, especially prime-field public-key cryptography. Fortunately, as Section 2.1 indicates, ECC (used judiciously) would appear to be practical.

This requirement also implies a need for *delegability*: certain cryptographic operations may take so long on very low-performance devices that they cannot be used in normal operation, but may be acceptable during a bootstrapping phase. Such devices should be able to delegate those operations to another (trusted) node with greater performance.

Finally, the strict constraints imposed by IEEE 802.15.4 (see Section 2.2) must be taken into account. Its very small frame size deserves particular mention: 127 byte PHY-layer frames imply a maximum data payload of roughly 100 bytes. This must be contain addressing and routing metadata, security overhead, and application data. Protocols must therefore be judicious in the overhead they incur, in order to maximise the space available for application data. This is another reason prime-field public-key cryptography is impossible: the key sizes required for a reasonable level of security are too large to fit into IEEE 802.15.4 packets.

Phrased another way, this requirement embodies the 'things' element of the phrase

"Internet of Things".

R.4 Ultimately, this thesis is about the Internet. Nodes must therefore be able to communicate in whatever manner is required to accomplish their tasks. Any security mechanisms introduced to satisfy the other requirements should not impede this.

4.3 Does the state of the art meet the requirements?

This section applies the requirements from Section 4.2 to the systems from Chapter 3. The main component of this work is Table 4.1. For each system, a requirement can be considered 'satisfied' (highlighted in green), or 'violated' (highlighted in red). We also include a third category, 'partial' (highlighted in yellow), where either it is not clear whether a system fulfils the requirement or not, or where a system provides a mechanism for fulfilling it, but that mechanism is not effective in an IoT environment. For systems which violate the decentralisation requirement (**R.1**), we assess the other requirements assuming that any centralised infrastructure remains uncompromised.

We have included clarifying sections below for systems whose assessment we believe requires further explanation.

4.3.1 6LoWPAN

The 'partial' assignment made to **R.3** for 6LoWPAN with no security (that is, IPsec, TLS, or similar), and the 'violated' for 6LoWPAN with security provision, is largely due to packet size.

6LoWPAN with no security, communicating with nodes on the same network, can take advantage of full header compression, imposing very small amounts of overhead per packet. However, when communicating with any other IPv6 node, full IPv6 headers must be used, imposing an overhead of 40 bytes per packet. If each packet is to take a single IEEE 802.15.4 frame, this indicates roughly 40% of available space consumed by the network layer; on the other hand, if the packet is fragmented, the overhead takes the form of extra packet transmissions (which also incur additional space overhead in the form of 6LoWPAN fragmentation headers on each fragment).

If security is introduced, this situation is necessarily worse. IPsec headers impose roughly an additional 20 bytes of overhead, consuming roughly 60% of available space on an unfragmented packet. With 6LoWPAN header compression, this drops to 26% of available space.

4.3.2 ZigBee

The 'satisfied' assignment made to **R.2** deserves some clarification, as does the 'partial' assignment made to **R.3**.

ZigBee uses strong cryptographic primitives, and uses them consistently. The ZigBee Smart Energy Profile introduced ECC to the specification, only adding to its

Key distribution protocols	**R.1**	**R.2**	**R.3**	**R.4**
Random key predistribution	Violated	Violated	Satisfied	Partial
q-composite	Violated	Partial	Satisfied	Partial
multipath reinforcement	Violated	Partial	Satisfied	Partial
random pairwise	Violated	Partial	Satisfied	Partial
random subset assignment	Violated	Partial	Satisfied	Partial
grid-based predistribution	Violated	Partial	Satisfied	Partial
Key infection	Satisfied	Violated	Satisfied	Satisfied
Blom scheme	Violated	Satisfied	Satisfied	Satisfied
Pairwise key predistribution	Violated	Satisfied	Satisfied	Violated
Fast authenticated key establishment	Violated	Satisfied	Satisfied	Satisfied
Network-layer protocols	**R.1**	**R.2**	**R.3**	**R.4**
Secure Lossless Aggregation	Violated	Violated	Satisfied	Violated
Sizzle	Satisfied	Satisfied	Partial	Violated
TinySec	Violated	Violated	Satisfied	Satisfied
MiniSec	Violated	Satisfied	Satisfied	Satisfied
TinySA	Violated	Satisfied	Satisfied	Satisfied
SPINS	Violated	Satisfied	Satisfied	Satisfied
Bluetooth	Violated	Violated	Satisfied	Violated
6LoWPAN	Satisfied	Violated	Partial	Satisfied
(with IPsec)		Satisfied	Violated	
ZigBee	Violated	Satisfied	Partial	Satisfied
Higher-layer protocols	**R.1**	**R.2**	**R.3**	**R.4**
UACAP	Satisfied	Satisfied	Satisfied	Satisfied
Kerberos	Violated	Satisfied	Satisfied	Satisfied
CapBAC	Partial	Satisfied	Violated	Satisfied
Federated Web authentication	Violated	Satisfied	Partial	Satisfied
Webinos	Partial	Partial	Partial	Satisfied
Capability-based OSes *(trust network)*	Satisfied	Violated	Violated	Satisfied
(do not trust network)		Satisfied		

Table 4.1: Application of requirements from Section 4.2 to systems from Chapter 3.

strength on this topic. However, ECC is only used for authentication to the Trust Centre. Once this link is established, all key exchanges with other nodes on the network use symmetric primitives, with the base key material provided by the Trust Centre. We therefore note that **R.2** is only satisfied as long as the Trust Centre is benign and operating correctly. (However, this does mean that ZigBee – forcibly – satisfies the delegability criterion of **R.3**.)

ZigBee also suffers from a more subtle problem: encryption and integrity protection of ZigBee packets are optional, and disabled by default. This leaves open the possibility for implementors or system integrators to simply forget to enable security features, yet still assume a benign network environment.

The assignment of 'partial' to **R.3** is largely due to the number of packet transmissions ZigBee requires. During normal operation, each message must be individually acknowledged, with no provision for acknowledgement windowing or similar techniques.

However, worse is the ZigBee join protocol: the number of packets required to join a ZigBee network (between 8 and 36, depending on the network's exact configuration) is simply excessive, and is exacerbated by the need for the Trust Centre to participate in a join transaction. This problem is particular acute where large numbers of ephemeral transactions – including large numbers of network joins and departures – are to take place.

Finally, the departure of a node also causes a large number of packet transmissions: the network key is normally updated via broadcast from the Trust Centre, with the new network key encrypted with the old. When a node is departing – whether by its own actions, or due to expulsion by the Trust Centre – this mechanism cannot be used, since this would reveal the new network key to the departing node. The network key must therefore be separately transmitted to (and encrypted for) to every node on the network individually, an $O(n)$ operation.

Chapter 5

BottleCap: distributed access control

This chapter describes BottleCap, the access-control system developed during this project. Its primary purpose is as an architectural exemplar for decentralised access control over a network: wielding rights granted by a relevant access control policy only requires the involvement of the provider of the relevant service, and the client wishing to invoke it. We chose a capability-based security model for this purpose, since this model naturally lends itself to satisfying this constraint.

BottleCap uses a trusted execution environment to store capability metadata on the client, alleviating the need for the service provider to allocate and store it. Part of the contribution of this work is a discussion on the requirements imposed on such a TEE.

The descriptions in this chapter cover the principles and protocols underlying BottleCap, as well as the implementation of the software that executes inside the TEE, since these are of research interest. The other requisite software, including client-side software outside the TEE, and server-side software altogether, is a much simpler prospect, relying only on standard cryptographic primitives and software engineering tools. Its discussion is therefore considered out of scope.

Unfortunately, the availability of TEEs for academic or prototyping work is limited. At the time this work was done, Flicker, based on the DRTM primitive from Trusted Computing (see Section 2.3), was essentially the only option. Its use limits the direct applicability of this work to the IoT domain, since an implementation only exists for desktop-class systems. This is only now changing, with Open-TEE[1] and OP-TEE[2], two open-source TEEs targeted at mobile platforms, approaching maturity. Even so, the hardware requirements of these two environments indicate that they cannot yet run on IoT-class devices. In addition, neither appears to possess a means of remotely establishing trust in the TEE, an absolute requirement for BottleCap to function. Nonetheless, these problems appear to be temporary: the relevant technologies are being introduced to an increasing variety of devices. We are of the

[1] https://github.com/Open-TEE
[2] https://github.com/OP-TEE/

opinion that, in the near future, the necessary functionality will become available on IoT-class hardware.

BottleCap's full API is reproduced in Appendix C, while source code for the prototype implementation can be found at `https://github.com/phoenix-frozen/bottlecap`.

5.1 Design

Access control systems all share a core design constraint: access control metadata must be shielded from unauthorised modification. In a capability-based system, this means the capability, a digital token that acts as both reference and authority (see Section 3.1.4 and Section 3.5.3), must be protected from modification by any party, including its holder. Password-capability systems achieve this by allocating capabilities alongside the resource, and associating with each a *password*. This password is given to the capability's holder, who can later use it to prove its rights, and thus invoke the capability. This system architecture is highly flexible, and permits a high level of dynamism, including arbitrary delegation of rights, with no central control points. However, it introduces two problems:

- Dissemination (or theft) of the password automatically results in delegation of its capability. Phrased another way, delegation of the access rights represented by the capability is *uncontrolled*.
- Access-control metadata must be allocated on a per-capability basis, and this metadata must be held *at the resource*.

BottleCap solves both problems. Its basic function is the instantiation of a *bottle*, an encrypted capability container held *at the client*. The contents of a bottle are opaque to the client, being managed exclusively through a BottleCap binary executing in a TEE. The TEE, in turn, must provide a means of storing the bottle's encryption key alongside the bottle, such that it is only usable by this binary.

BottleCap mitigates the uncontrolled delegation problem by never revealing the contents of the capability itself. In order to invoke a capability, its holder instead instructs BottleCap to generate a *ticket*, a time-limited proof of authority. The BottleCap ticket concept strongly resembles both the Kerberos [158] concept, for which it is named, and the *quote* structure underpinning Trusted Computing remote attestation. Comparison to the proxy certificates common in grid computing (see Section 3.4.4) is also instructive.

Notably, tickets *do* still suffer from the uncontrolled delegation problem. However, their validity periods can be made much shorter than that of the capability whose rights they assert, minimising the period of exposure should they be disclosed. Completely eliminating the problem is possible using the on-line version of the invocation protocol, in which tickets are valid only for a single transaction. Once a ticket expires, a new ticket must be issued by the holder of a capability on the machine to which that capability is bound. Therefore, any long-term capability-sharing must

Operation	Algorithm	Prototyping implementation
Public-key encryption	RSA	Software
Hashing	SHA1	Software
Integrity-protection	HMAC-SHA1	Software
Block cipher	AES-128	Software
Block cipher mode	CFB-128	Software

Table 5.1: Cryptographic algorithms in use in BottleCap.

involve the active participation of the holder, making BottleCap capabilities a form of Non-Delegable Authority [123].

BottleCap does allow capabilities to be delegated with the capability issuer's authorisation. The issuer may additionally specify restrictions on the delegated capability. This bears some similarity to the TPM-based single sign-on implementation described in [130], as well as the design of TPM key migration.

5.1.1 Structure of the prototype

As a result of the lack of availability of mobile or IoT-class trusted execution environments for prototyping, BottleCap is implemented over Flicker (see Section 2.3.2), which implements a TEE using the DRTM late-launch mechanism from Trusted Computing. (For a brief review of TC mechanisms, see Section 2.3.)

BottleCap then uses the TPM's storage hierarchy to protect its secrets. Each bottle is encrypted using a symmetric key, which is held in sealed storage, and accessible only to future invocations of BottleCap. Capability issuance, import and export similarly use TPM bound storage, to ensure that capabilities are only transmitted to platforms in which trust has been established.

5.1.2 Cryptographic tools in use

BottleCap uses well-known cryptographic primitives to fulfil its function, detailed in Table 5.1. The motivations for these choices were:

RSA This choice was dictated by the Flicker, which is uses TPM 1.2. TPM 2.0 permits the use of ECC. Whether or not an eventual BottleCap implementation for IoT-class hardware uses a TPM-based TEE, the use of ECC, with its small key sizes, would be a necessity.

AES-128-CFB AES-128 is used in cipher feedback (CFB) mode to encrypt the capability transfer and ticket data structures, as well as by the prototype to encrypt the bottle itself. AES is hardware-accelerated by on many platforms, including modern general-purpose processors, but also many IoT platforms. This made it an obvious choice. The CFB mode was chosen for its high performance and

```
struct cap {
  uint32_t magic_top;
  uint32_t urights;
  uint64_t oid;
  aeskey_t issuer;
  uint64_t expiry;
  uint32_t srights;
  uint32_t magic_bottom;
};
```

(a) Capability data structure

```
struct ticket {
  uint128_t  nonce;
  struct {
    uint64_t oid;
    uint64_t expiry;
    uint64_t amagic;
    uint32_t urights;
    uint32_t cmagic;
  }          authdata;
  sha1hash_t mac;
  sha1hash_t keyid;
};
```

(b) Ticket data structure

Figure 5.1: BottleCap core data structures. The capability, in (a), is 48 bytes long, while the ticket, in (b), is 88 bytes long. Neither structure contains any TPM keys.

resistance to ciphertext modification. Section 6.1 discusses this decision in more detail.

HMAC-SHA1 HMAC-SHA1 is used for integrity-protection of all encrypted data structures. SHA1 is designed for very high performance on general-purpose processors, and HMAC is a common construction for transforming hash functions into keyed MACs, providing proof of authenticity. Again, Section 6.1 discusses this decision in more detail.

5.2 BottleCap API

This section describes the BottleCap API. It begins by describing the basic concepts, data structures, and protocols around which BottleCap is designed. This is followed by a summary of key BottleCap operations – that is, the more important functionality exposed by the TEE module. The section continues with a description of the *service configurations*, archetypical system architectures, possible with BottleCap. Finally, a simple example ties all of these together.

5.2.1 Concepts, data structures, and protocols

This section will describe the fundamental concepts and data structures underlying BottleCap. Data structures are presented in the C language.

Data structure – the capability The capability is the central data structure of BottleCap, pictured in Figure 5.1(a). Its significant fields are: an issuer-defined 64-bit object identifier, `oid`; a 32-bit issuer-defined rights mask, `urights`, which

```
struct cap_transfer {
  tpm_aeskey_t key;
  uint128_t    iv;
  union {
    struct cap cap;
    uint8_t ciphertext[48];
  }
  sha1hash_t   mac;
};
```

Figure 5.2: BottleCap capability transfer structure. This is 84 bytes long, excluding the field containing a TPM-bound key, which can add up to 400 bytes.

represents the access rights granted by the capability; a 32-bit flags mask, `srights`, controlling which management operations may be performed on it, and when (such as permitting inter-bottle migration, or restricting such migration to be move-only); an expiry date, `expiry`, in 64-bit Unix time; and the capability's issuer key, `issuer`, a 128-bit AES key used in ticket generation.

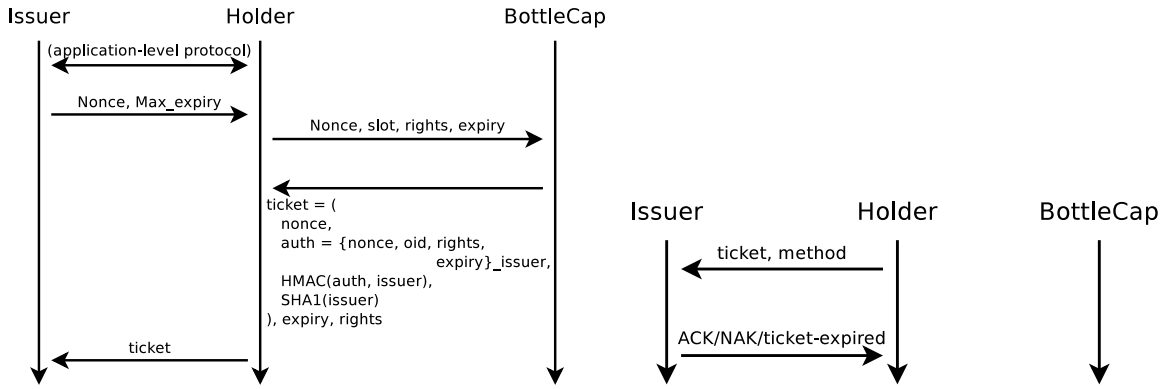
This information is sufficient to entirely relieve the issuer (or resource) from capability metadata.

API concept – issuing a capability Issuing a capability is essentially a key-transport operation. The issuer encrypts the capability with a session key, and binds that key to the recipient's machine (more accurately, that machine's TPM), the BottleCap binary, and the Flicker execution environment. Upon receiving this encrypted bundle, the recipient invokes BottleCap, which will unbind the session key, decrypt the capability, and find a free slot into which to insert it. The capability transfer data structure is shown in Figure 5.2.

The circumstances under which a capability is issued are up to the issuer. While the common case is expected to be as a result of other capability invocations, services will also need a bootstrapping protocol in order to grant initial access to a system. These bootstrapping protocols can take a multitude of forms; examples include well-known capabilities, capability escrow, and identity bridges (such as issuing an access capability in response to an OpenID authentication session).

Once the capability has been added to the bottle, the issuer and BottleCap share a secret, the capability's issuer key. This key is used in issuing tickets. Despite its name, this key need not be strictly per-issuer. The issuer can choose the granularity of issuer key usage (from, at one extreme, per-issuer, to at the other, per-capability-holder), which has a corresponding effect on the work required to revoke an issuer key should it be disclosed.

Notably, the issuer must determine that the encryption key presented by the client is genuinely TPM-based before binding a capability to it. It must also establish that that TPM is not under the physical control of an adversary [129]. This is an instance of the standard TPM enrolment problem.



(a) Ticket issuance protocol (b) Capability invocation protocol

Figure 5.3: BottleCap core protocols

API concept – capability attestation BottleCap's core guarantee is that a capability never leaves its purview. Instead, capability attestation embodies the principle of proof of possession. BottleCap provides the `bottle_cap_attest` call for this purpose: given a slot, an expiry date, and a mask of rights to invoke, BottleCap will issue a ticket representing the actual invocation of those rights. This ticket is sufficient to both authenticate and authorise capability-controlled resource requests.

Data structure – the ticket The ticket data structure is shown in Figure 5.1(b). The `authdata` block is encrypted using AES in CFB mode with the capability's issuer key, and the IV in the `nonce` field. Inside this block, the `oid` field reflects that of the capability, while the `urights` and `expiry` apply specifically to this ticket – BottleCap will not issue a ticket with a later expiry date or greater rights than the capability from which it is derived (a ticket with fewer rights or a shorter validity period is, of course, permitted). The ticket finally includes, in `hmac`, a SHA1 HMAC of `authdata`, keyed with the capability issuer key, and, in `keyid`, a SHA1 hash of the key itself.

API concept – time and expiry Enforcing expiry times in distributed systems usually requires globally synchronised clocks. In BottleCap, this is not the case. The expiry time of a ticket is checked at the time of resource access, and at the resource. Since BottleCap guarantees that a ticket will never be issued with an expiry time later than that of the capability it attests, the client's clock is not trusted or used, and need not be synchronised with the server.

Protocol – ticket issue Once a capability has been added to a bottle, BottleCap can generate a ticket for it. This can be done interactively or off-line. Figure 5.3(a) shows the interactive version. The assumption is that some other protocol is being carried out over a communication channel between the capability holder and the capability issuer (or service provider, if they are different – see Section 5.2.3). Upon determining that some capability-controlled rights must be invoked, the holder initi-

ates the ticket-issue protocol in an application-specific way. The issuer will provide a nonce and the latest permissible ticket expiry date. The holder will provide this nonce, an expiry date (capped by that stipulated by the issuer), and a mask of the rights it wishes to invoke to BottleCap, which issues a ticket (also returning the rights and expiry date in plaintext, in case they were altered). The holder then returns this ticket to the issuer.

The off-line version of this protocol requires prior knowledge of the maximum ticket validity time for the service being invoked. The holder can then request that BottleCap instead generate the nonce using the TPM.

The on-line version of the protocol is primarily designed to allow issuers to control the issuance of tickets as well as capabilities. The intention would be for the issuer to catalogue the ticket's HMAC during ticket-issue, so that it can later be checked during invocation.

Protocol – capability invocation To actually invoke the rights conferred by a capability, a ticket must be issued via the protocol above, and then the ticket must be presented to the service being authorised (see Figure 5.3(b)). During this step, the service can verify the ticket's integrity, validity, and adherence to various constraints (such as its maximum ticket validity time), as well as verifying that the ticket's HMAC is in the valid-ticket catalogue described above. Importantly, whereas the ticket issuance protocol suffers from any performance issues encountered by BottleCap, the invocation protocol does not: once a ticket has been issued, it may be freely reused until its expiry.

The invocation protocol can also use on-line ticket issuance. In this variant, when a request is made, the server supplies a nonce using which a single-use ticket is issued. The ticket is then supplied in a reply, after which the server responds with an authorisation decision. This eliminates the need to protect the ticket from disclosure, at the cost of a TEE invocation operation. For the TC-based TEE on which the prototype was implemented, this cost is very high.

5.2.2 Core API

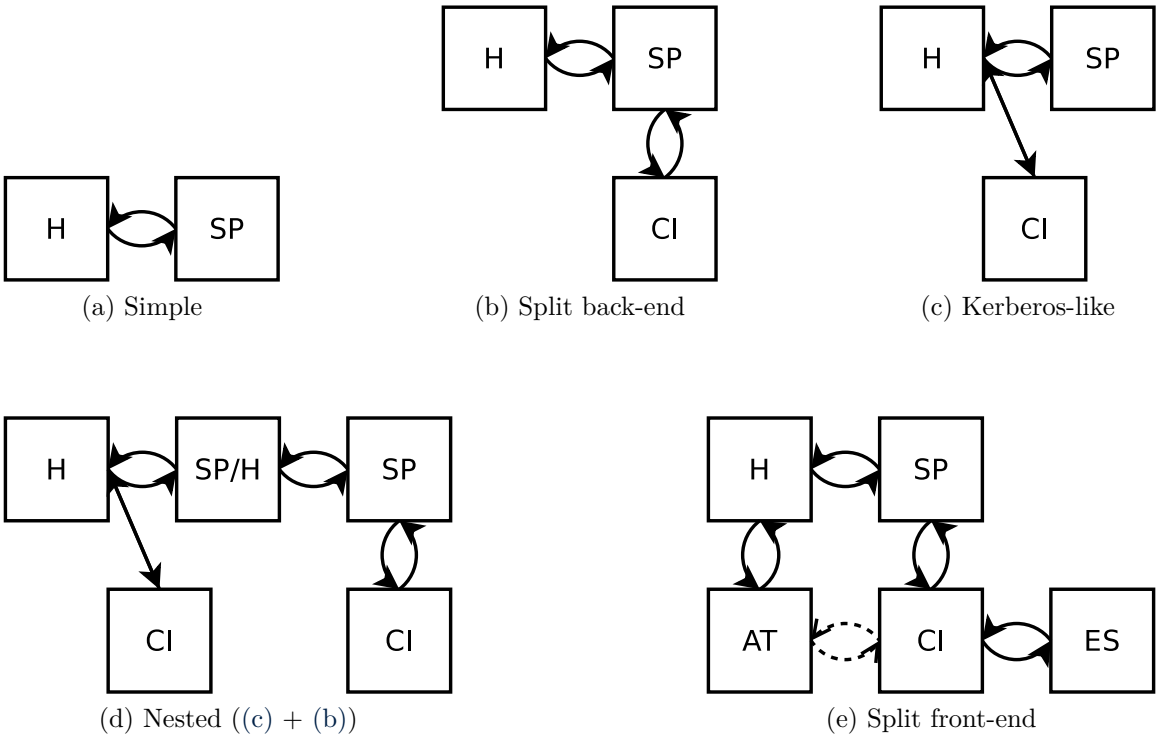
The BottleCap TEE module exposes a variety of functions for managing bottles and capabilities. Of those functions, four are fundamental to its operation:

`bottle_init` Initialise a new, empty bottle.

`bottle_cap_add` Takes as arguments a bottle and an exported capability. Adds the capability to a free slot in the bottle, and informs the caller which one.

`bottle_cap_delete` Takes as arguments a bottle and a slot index. Erases the contents of that slot, freeing it.

`bottle_cap_attest` Takes as arguments a bottle, a slot index, a nonce, an expiry time, and a rights mask. Generates a ticket for the capability in the indicated slot, using the provided nonce. Its expiry time is the earlier of the provided



(a) Simple (b) Split back-end (c) Kerberos-like

(d) Nested ((c) + (b)) (e) Split front-end

Figure 5.4: BottleCap service configurations. H stands for 'holder', SP for 'service provider', CI for 'capability issuer', AT for 'authentication token', and ES for 'escrow service'

expiry time and capability expiry time, and the rights it asserts are the bitwise-AND of the rights in the capability and the provided rights mask.

BottleCap may expose a variety of other functions designed to ease management of bottles and slots, and, indeed, our prototype does; see Appendix C for a complete listing of its API.

5.2.3 Service configurations

To illustrate the flexibility that BottleCap permits, this section describes five usage scenarios of varying degrees of complexity. These are illustrated diagrammatically in Figure 5.4.

Simple (Figure 5.4(a))

This is the simplest configuration. Each *service provider* (SP) issues capabilities to its own users (labelled H, for capability *holder*), though a user may store capabilities from multiple SPs in the same bottle.

Split back-end (Figure 5.4(b))

To simplify service implementation, access control can be handled by a separate *capability issuer* (CI) application, either on a remote server or co-located with the SP. One CI may serve many SPs.

Kerberos-like (Figure 5.4(c))
Once the SP and CI have been divided, the CI can even be exposed directly to clients. This configuration bears a striking resemblance to the Kerberos design, with the CI analogous to the TGS, and BottleCap to the AS. In a similar vein, the CI thus exposed also bears strong resemblance to a Web Security Token Service. As with (b), and again in analogy to Kerberos, a CI may serve any number of SPs.

Nested (Figure 5.4(d))
In a complex system, SPs may also need to invoke resources. BottleCap does not prevent this: SPs can freely act as holders (SP/H), resulting in arbitrary nestings of the above configurations.

Split front-end (Figure 5.4(e))
Fear of users losing their bottles, and thus the capabilities they contain, can be addressed using a simple remote storage service: the (encrypted) bottle can be safely stored anywhere. More sophisticated capability accounting or recovery requirements can be met by introducing an *escrow service* (ES), which the CI may contact upon issue of each capability. The ES might store, for example, a catalogue of capability-to-SRK mappings. The services requested (or the entire bottle) could even be bound to an *authentication token* (AT) possessed by the user.

5.2.4 Example scenario

Having described the essentials of the BottleCap API, this section concludes the chapter with an example scenario. It involves only two network nodes, a server and a client, in the 'Simple' service configuration. The client acts as the capability holder, and the server acting as service provider.

The scenario assumes that the client and server communicate over a secure channel. The channel need only be authenticated in one direction, from server to client; the client need not be authenticated to the server.

We will assume that the client is a trusted platform (in the TC sense), that it is capable of running Flicker, and that that our Flicker-based BottleCap prototype is in use. For a different TEE, while the terminology may differ, the essential mechanisms will not.

The example scenario involves a network filesystem. It stores a root policy which specifies that members of a specific set of organisations may use the file service. A user proves membership of an organisation using Shibboleth.

This scenario has essentially two phases: issuance and invocation. In the issuance phase, the client obtains a capability to the service. It uses Shibboleth to prove that it is a member of a permitted organisation. In addition, it must supply its

TPM's public SRK, a signature of that key by one of its TPM's AIKs, and the relevant AIK credential. (In short, a key with which the server can encrypt data so that it will only be accessible to the BottleCap TEE module, and evidence of this property of that key.) In response, the server creates a new directory for the user, and generates a capability transfer structure (see Figure 5.2) bound to the client's TPM, permitting administrative access to that directory. Upon receiving the capability transfer structure, the client invokes the `bottle_cap_add` BottleCap function to add the capability to a bottle, completing the issuance phase.

In the invocation phase, the client makes use of its capability to create a subdirectory. It first invokes the `bottle_cap_attest` function to generate a ticket, with a rights mask setting the bit corresponding to the create directory operation. It then contacts the server with its request, supplying the ticket.

The client may, at its option, set more than one bit in the rights mask, resulting in a ticket which asserts several rights. This decsion, as well as the selection of a ticket expiry time, is a tradeoff. Longer expiry times and richer rights masks benefit performance by minimising the number of TEE invocations required. However, the disclosure of such tickets is dangerous, since all of the authority in the ticket is thereby disseminated. By contrast, tickets with shorter expiry times and fewer rights benefit the security of the system, since their disclosure results in the dissemination of less authority, but at the cost of a greater number of (potentially expensive) TEE invocations. (The server can influence this decision by refusing to accept tickets asserting many rights, or with long validity periods.)

Chapter 6

Evaluating BottleCap

The limitations of BottleCap in no small part reflect the limitations of the technologies upon which it depends. Several features of the design (most notably the decision to impose time limits on tickets, rather than making them single-use) were conceived to minimise the performance impact of those technologies. This chapter evaluates BottleCap, beginning by presenting a discussion, in Section 6.1, of the engineering shortcomings in the BottleCap design and prototype, and the reasons behind them. We then present microbenchmarks detailing the precise performance impact of the mechanisms on which it is based in Section 6.2. Section 6.3 discusses of the size/functionality trade-off made in BottleCap's TCB. This is followed by revisiting the issue of the applicability of BottleCap to the Internet of Things in Section 6.4. Finally, in Section 6.5, BottleCap is evaluated against the requirements established in Chapter 4.

6.1 Engineering shortcomings in BottleCap

BottleCap is a research prototype with a variety of engineering shortcomings. Some of these are deliberate omissions, not affecting the research results; some demonstrate lessons learned during the project, suggesting changes required for real-world deployment to be practical; and some directly affect its possible applicability to the IoT environment.

First, there are elements of functionality that would be required for a real-world deployment, but are deliberately omitted by the BottleCap API. Such omissions are for one of two reasons: either the functionality would require a large amount of code and logic, which would be out of keeping with the TCB-minisation heuristic which pervades the design of TC-based primitives and applications (including BottleCap), or its implementation would require a solution to an open research problem.

Replay protection Replay protection for TC applications is an open problem, and is particularly acute for Flicker applications. The TPM provides non-volatile storage and monotonic counters for this purpose, but these are very limited; their use by Flicker applications for replay protection is known not to scale [21]. In the BottleCap case, this precludes the implementation of logical-move or similar operations.

Log-based virtual monotonic counters [146], a cryptographic scheme for multiplexing a single TPM physical counter into an unlimited number of virtual counters, solves the problem entirely. However, the generally poor performance of TPM-based cryptographic primitives makes this solution particularly ill-suited to the non-multitasking Flicker environment. The effectively unbounded execution time of the `ReadCounter` operation only adds to this problem, and is furthermore completely impractical on embedded systems.

A major difficulty on this topic is the client-managed nature of bottles – that is, while individual capabilities are controlled by their issuers, the API permits capability holders to entirely control their placement in bottles. Were bottles instead managed by capability issuers, replay protection would be possible using remote attestation.

For non-DRTM-based TEEs, this discussion depends on the properties of, and services provided by, the TEE in question.

Migration primitives Any primitive for data migration between bottles, whether on the same machine or different machines, is unimplemented. This primarily means capability export and inter-machine bottle migration. These functions were unnecessary for prototyping.

Secondarily, full client-driven migration of this kind would require BottleCap to be able to establish trust in a remote TEE. While this is not impossible, it requires a large amount of code, and potentially access to a network, neither of which is ideal in the constrained Flicker environment

For single-capability export, the BottleCap API is instead designed to delegate this decision to the capability issuer, although an actual implementation of this particular primitive was omitted from the prototype.

The client-managed nature of bottles precludes the existence of a full-bottle inter-machine migration primitive.

Software updates Software updates are an open problem in TC-based applications, and BottleCap is no exception. Since this problem is also outside the scope of what BottleCap was attempting to achieve, we elected to omit entirely a software update mechanism from the prototype. The capability export primitive, when implemented, would permit such upgrades; the difficulty is in ensuring that capability issuers trust the new binary, and cease trusting the old one.

For clarity, this only refers to upgrades of BottleCap itself, the Flicker runtime on which it runs, and the Intel SINIT module used to launch them – that is, all software in BottleCap's TCB. BottleCap is agnostic to any properties of the host OS from which it is launched.

Again, for TEEs not based on TXT, this discussion depends on the TEE in question.

Capability revocation While in a local operating system context, capability revocation is a fairly simple concept, revocation in a distributed system is a well

known, and exceedingly difficult, problem. Judicious use of the expiry mechanism and granularity of issuer keys can be used to approximate the effect. However, true revocation (or similar constructs, such as logical-move operations or count-limited capabilities) require replay protection.

The tools available to us also constrained our engineering in several ways:

Flicker Flicker suffers from numerous performance problems, some of which we discuss in this chapter. It is also specific to the x86 architecture. For IoT applications, BottleCap would therefore need to be ported to another, functionally similar environment. Examples include TrustVisor [110], for desktops and servers, or Trustonic[1], Open-TEE[2], or OP-TEE[3] on mobile platforms. However, of these, only TrustVisor possesses a means for establishing trust in a remote platform (or a TEE executing thereon). Such a primitive is an absolute requirement for BottleCap to function.

RSA Like the version of TC on which it is based, BottleCap depends heavily on the use of RSA, whose key and data structure size imposes a very high minimum on the amount of network traffic required to convey TC attestation and protected storage, as well as being beyond the computation capabilities of IoT-class devices. TPM 2.0 introduces support for ECC, making it much more amenable to implementation on such devices.

Finally, we found two substantial errors made in the design and implementation of BottleCap itself:

Lack of capability introspection BottleCap takes a wholesale approach to protecting capability metadata: the entire contents of a bottle are encrypted. While this simplifies design and implementation work, it has a usability cost: introspection of bottles, and the capabilities they contain, is difficult and slow.

However, only the issuer key need actually be kept secret from the capability holder; every other field need only be protected from *modification*. The other capability fields could be in plaintext, protected by the bottle's integrity-protection mechanism, allowing them to be analysed usefully by software outside BottleCap. In order to differentiate between capabilities by different issuers in the same bottle, a digest or hash of each capability's issuer key could also be stored in this unencrypted area.

Use of cryptographic primitives BottleCap uses AES-128 in CFB mode for encryption, and HMAC-SHA1 for integrity-checking. Magic numbers are also used in a variety of data structures, usually in spaces inserted in order to align these data structures on power-of-two boundaries in memory. The same key is used for both encryption and authentication, an unsafe design choice (despite the lack of known vulnerabilities in this particular case).

[1] https://www.trustonic.com/technology/trustzone-and-tee
[2] https://github.com/Open-TEE
[3] https://github.com/OP-TEE/

Critical operation	Time taken
DRTM late-launch	717.83 ms
TPM seal AES key	306.69 ms
TPM unseal AES key	940.86 ms
TPM RNG (16 bytes)	4.28 ms
Encrypt 4kiB bottle	37.01 ms
Decrypt 4kiB bottle	38.55 ms
HMAC 4kiB bottle	18.20 ms

Table 6.1: Benchmarking of critical operations on prototyping hardware

The data structure alignment was later determined to be unnecessary, and relying on the CFB mode's behaviour under modification to be unsafe. All of these measures should therefore be replaced with a single AEAD algorithm, such as AES-128 in CCM mode (used in StarfishNet), or Galois/counter mode (GCM) (which has very high performance on AES-NI-capable Intel processors). This includes tickets: both CCM and GCM can be used in an authentication-only manner, if required. This would both simplify BottleCap and shrink its TCB, since the number of cryptographic algorithms (and thus the code to implement them) would be reduced, and en/decryption and authentication would be done in a single, common operation, also slightly improving performance. Finally, the sizes of all data structures would be substantially reduced.

6.2 Microbenchmarks of common operations

BottleCap is a Flicker application. Making any BottleCap call thus requires a Flicker invocation, which imposes hardware overheads. In addition, most calls involve bottle encryption and decryption, the generation of bottle and ticket HMACs, the generation of random IVs, and/or the sealing or unsealing of keys to the TPM. Using a Dell OptiPlex 780 workstation with an Intel Core 2 Quad Q9550 processor (maximum clock frequency 2.83GHz), we used the CPU's in-built cycle counter to time each of these operations. The results are presented in Table 6.1, which shows the mean time over 100 invocations with identical parameters, after filtering for processor clock speed variations. Using the same method, several BottleCap calls were also benchmarked, with results shown in Table 6.2. Rather than selecting the four core functions, we selected BottleCap functions which used varying sets of cryptographic operations. The calls we chose, and the cryptographic operations they perform, were:

`bottle_init` Initialises an empty bottle, generates and seals a new AES encryption key using the TPM random number generator (RNG), generates a new IV using the TPM RNG, encrypts the bottle, and generates its HMAC.

`bottle_cap_add` Unseals the bottle's encryption key, verifies its HMAC, decrypts the bottle, unbinds the AES key in the transfer structure, verifies the transfer

BottleCap call	Time taken
`bottle_init`	1091.92 ms
`bottle_cap_add`	2710.19 ms
`bottle_query_free_slots`	1755.14 ms
`bottle_expire`	1779.51 ms
`bottle_cap_attest`	1757.28 ms

Table 6.2: Time to execute core BottleCap calls. Timing began just before late-launch initiation, and ended upon return from Flicker.

structure HMAC, decrypts the transfer structure, then generates a new bottle IV, encrypts the bottle, and generates its new HMAC.

`bottle_query_free_slots` Unseals the bottle's encryption key, verifies its HMAC, decrypts the bottle, then re-encrypts the bottle with the same IV. This is safe because the bottle doesn't actually change; the re-encryption could and should be optimised out. For the same reason, the HMAC is not regenerated.

`bottle_expire` Unseals the bottle's encryption key, verifies its HMAC, decrypts the bottle, generates a new IV, re-encrypts the bottle, and generates its new HMAC.

`bottle_cap_attest` Unseals the bottle's encryption key, verifies its HMAC, decrypts the bottle, encrypts the ticket, generates the ticket HMAC, and then performs the same spurious re-encryption operation as for `bottle_query_free_slots` (once again, the bottle does not change).

Those two tables, taken together, clearly show that two greatest overheads are, by far, imposed by hardware operations: a late-launch operation, invoking the TEE, takes over 700ms, while a TPM unseal operation, to retrieve the bottle's encryption key, takes over 900ms. All other contributions taken together, including HMAC calculations, AES encryption and decryption operations, and program logic, are dwarfed by them. Moreover, the implementations of the cryptograhic primitives in use were *software-only*. For hardware-accelerated implementations, this comment would be even more true.

6.3 TCB

Flicker provides two related, but distinct, guarantees for security-sensitive applications. The first is isolation: the PAL (that is, in this case, BottleCap) is not subject to interference from any software or hardware during execution, as would any TEE. The second is TCB minimisation: by virtue of the late launch event, the only code in the PAL's TCB is what is linked into its binary. Of course, as a corollary, the only services available to it are those it itself provides. The TCB size of BottleCap

Component	Source size
Flicker runtime environment	2688 LoC
of which runs in kernel mode	*516 LoC*
Flicker TPM interface module	1825 LoC
PolarSSL (AES and SHA1 modules)	1610 LoC
BottleCap implementation	1037 LoC

Table 6.3: BottleCap codebase size

(measured by `sloccount`[4]) is shown in Table 6.3. Of the four contributions, only the Flicker runtime environment could be pared down (by removing unused functionality from the C standard library implementation); every other part of the codebase provides precisely the functionality required.

Notably, the Flicker loader ensures that most of this code runs in virtual memory, in an unprivileged processor mode, so as to protect the suspended operating system from bugs in PAL code. Therefore, the only code running in kernel mode, and thus with the ability to damage the host OS, is the italicised line in Table 6.3.

In addition, the TCB of BottleCap includes the TCB of the underlying hardware platform. While much of the platform's firmware is eliminated from consideration by the design of the DRTM primitive, the SINIT module and any code running in System Management Mode are *not* so eliminated. The latter, in particular, is of great concern to users of TXT [172].

6.4 Applicability of BottleCap to the Internet of Things

The BottleCap prototype is implemented over Flicker, which in turn depends on TPM and DRTM hardware primitives to bootstrap the TEE it provides. In the BottleCap prototype, this was Intel TXT, although AMD's DRTM implementation is similar in all relevant ways.

First, for Flicker-type use, TXT operations have a variety of shortcomings. The most substantial of these, as our results clearly demonstrate, is performance: with an invocation time of over 700ms, TXT is simply inappropriate for frequent execution of small binaries in the Flicker style. Worse, it is by design impossible to hide this latency using multiprogramming techniques: TXT shields the execution of the TEE by stopping all other activity on the machine.

Second, BottleCap is heavily limited by the poor performance of TPM-based cryptographic operations: the TPM takes over 900ms to unseal a 128-bit AES key. Similar latencies occur for other public-key cryptographic operations.

TXT also has a trust problem: while the binary it launches is shielded from the actions of the environment from which it is launched, the same does not work in reverse. A late-launched binary has total control over the system, meaning it can

[4] `http://www.dwheeler.com/sloccount/`

corrupt the host OS in arbitrary ways. Flicker does install a runtime environment for its PALs which prevents this, but a far safer design would be to be able to context-switch between the running OS and the TEE in a way which protects both. Such primitives do not really exist on current hardware. ARM's TrustZone, a common basis for TEEs on mobile platforms, does provide a context-switching interface between the 'normal world' and the 'secure world', but the latter has a similar ability to arbitrarily control the former. The closest existing equivalent would be TrustVisor [110], the successor project to Flicker. Its implementation, also based on TXT, involves a specialised hypervisor which provides peer containers for both the PAL and 'host' OS.

The implementation of BottleCap as a TrustZone software module – or, indeed, one for Intel's Software Guard Extensions (SGX) [4,63,111] – alleviates both issues: the invocation overheads of both are far lower than TXT, and all cryptographic operations would take place on high-speed application processors. The RSA requirement imposed by TC would also be lifted, permitting the use of ECC, with its smaller keys and better performance. However, a mechanism for remotely establishing trust in the TEE is necessary for BottleCap. Commercial TEEs, such as that offered by Trustonic[5], reportedly provide such services, but their exact parameters are not public. TPM 2.0, which permits the use of ECC, is also a possibility, as is SGX.

Clearly, the BottleCap prototype is inappropriate for IoT applications. However, the reasons for this are largely engineering constraints due to the TEE used. The architecture and protocols, including the cryptographic primitives in use – apart from RSA – are entirely applicable, given the prevalence of AES acceleration on IoT-class hardware. The only other requirement is a TEE which provides attestation-like services for remote trust establishment.

6.5 Evaluation against IoT requirements

While BottleCap cannot be used in an IoT context for the reasons outlined in the rest of this chapter, continuing rapid development in this area suggests that this is likely to change in the near future. This section will assess BottleCap against the requirements presented in Section 4.2, keeping this possibility in mind.

6.5.1 R.1: decentralised architecture

BottleCap itself is designed to be decentralised: it requires no central authentication service of any kind, with all access control decisions made by the service provider, and enforced by a combination of BottleCap itself, and the TEE in which it executes. In particular, the service provider need not contact an authentication service at the time a service is invoked: the ticket itself provides *prima facie* evidence of the rights being invoked. This requirement is therefore satisfied by design.

We note, however, that the capability issuer must establish trust in the platform in use by the holder. Ultimately, this implies that either the issuer or its trusted

[5] https://www.trustonic.com/technology/trustzone-and-tee

delegate must inspect the base certificates asserting the properties of that platform. This is a form of centralisation: the entities qualified to issue such certificates are few, and named.

6.5.2 R.2: resistance to strong adversaries

The cryptographic primitives used by BottleCap are well-studied, and BottleCap uses them in a fashion consistent with their design (apart from the key-reuse flaw mentioned in Section 6.1). As a result, the ticket – which is essentially a message from BottleCap to the service provider – is protected from modification by or issuer-key disclosure to even a Dolev-Yao attacker. Even the holder in unable to mount such an attack. BottleCap therefore does satisfy this requirement.

However, unless it is single-use, a BottleCap ticket is still sensitive information that must be transmitted over a secure channel. Such a requirement is not unusual for a network-based access control system, and secure channels are not a problem BottleCap is intended to solve. In this case, it nonetheless represents a limitation of the BottleCap architecture: against an adversary who can breach the channel in use, BottleCap does permit limited theft of authority.

6.5.3 R.3: performance

BottleCap is designed to that access control checks are *purely local*. They do not necessitate contacting another service, and only require symmetric cryptography. This is a substantial performance advantage, especially when verifying large numbers of tickets. On the service provider side, BottleCap therefore meets this requirement. (The situation would be further improved by the changes to data structure and cryptographic primitives mentioned in Section 6.1.)

On the capability holder side, the same argument largely applies: the mechanisms used by BottleCap are almost entirely based on symmetric cryptography. However, the invocation overhead of the TEE in which it runs also has a great effect. The Flicker TEE used for the prototype performs very poorly, and is only available on desktop-class hardware, making the prototype essentially unable to meet this requirement. However, in a TEE with lower invocation overhead, on an IoT-class processor – such as the TrustZone-based options mentioned earlier – this requirement can easily be met.

6.5.4 R.4: unrestricted communication

BottleCap imposes no restrictions on communication patterns by design. Any machine can be a capability holder or service provider, with no restriction.

However, we highlight one proviso: capability issuers *must* be able to establish trust in platforms acting as capability holders. This places a practical restriction on what machines are able to hold BottleCap containers. Thus, in practical application, BottleCap only partially fulfils this requirement.

System	R.1	R.2	R.3	R.4
BottleCap	Satisfied	Satisfied	Satisfied	Partial

Table 6.4: Application of requirements from Section 4.2 to BottleCap.

We also note that, unlike most capability systems, the holders of BottleCap capabilities *cannot* arbitrarily delegate their capabilities; they require the authorisation of the capability issuer to do so. Authority *can* be delegated by sharing tickets, but this runs counter to the design, as does offering tickets as a service. This could validly be considered a communications restriction, especially compared to other capability systems, although we do not consider it so.

Chapter 7

StarfishNet: establishing communications security

This chapter describes StarfishNet, the network-layer protocol developed during the course of this project. Its primary purpose is as an architectural exemplar for IoT networks, in which security responsibilities are decentralised, with communications between any pair of nodes only requiring those two nodes to be trusted.

StarfishNet provides communications and routing services above the industry-standard IEEE 802.15.4 link layer. It can coexist with other network-layer protocols operating on the same 802.15.4 PAN, although it cannot communicate with them directly. (Section 7.8 describes one method by which bidirectional communications can be established between a StarfishNet node and an endpoint on an IP network.) StarfishNet only provides unicast data transport. The reason why broadcast and multicast functionality was omitted is discussed in Section 8.1.

StarfishNet follows the recommendations in [147] for designers of protocols based on IEEE 802.15.4 (see Section 2.2), and disables link-layer acknowledgements, packet security, and ACLs, with corresponding functionality instead implemented at the network layer. The target hardware for the StarfishNet prototype is the Texas Instruments CC2530 ZigBee Development Kit, an evaluation kit designed to showcase the capabilities of the CC2530 [160] SoC. This SoC is marketed by Texas Instruments for IoT applications, and exemplifies the IoT hardware class. Its processing core is an Intel-8051-class microcontroller. It has 8KiB of RAM and 256KiB of programmable ROM, an onboard IEEE 802.15.4 PHY, and an onboard AES128 cryptographic co-processor. All of the evaluation boards in this kit can be powered by standard AA batteries.

Much of the discussion in this chapter will include comparisons with ZigBee. This is no accident: ZigBee is well known as a high-security, low-power network-layer protocol, is aimed at the same class of device as StarfishNet, and is currently used in many of the IoT networks for which StarfishNet is designed.

StarfishNet's full API is reproduced in Appendix B, while source code for the prototype implementation, a networking module for the Contiki[1] IoT OS, can be

[1] http://www.contiki-os.org/

found at https://github.com/phoenix-frozen/StarfishNet.

7.1 Fundamental principle: the perimeterless network

StarfishNet is designed around the need for a decentralised wireless network (requirement **R.1**). This requirement, *in extremis*, has a surprising conclusion: the network perimeter should not be a security boundary. The reason is that any admission control must necessarily be performed by some node (or set thereof) on the network, which would by definition be a single point of control, as well as a single point of failure. Even if this central node does not act as trusted third party for all cryptographic exchanges (as with the ZigBee Trust Centre), it still fundamentally violates **R.1**, privileging one of the network's participating entities over the others.

StarfishNet was therefore designed as a network without a perimeter. Joining a StarfishNet network is always possible, to any node. Access control decisions are made at the endpoint of a given communication transaction, based on the information available to that endpoint. The act of joining the network does not grant any rights, nor access to any network-wide keys – in fact, no such keys exist in StarfishNet. Joining instead merely provides a routable network address.

This architectural principle is essentially absent from the literature, making this network design a novel contribution of this thesis[2]. Most other protocols in this domain perform some level of admission control, and essentially all – in fact, essentially all network protocols altogether – contain some built-in assumption that the network has a well-defined owner[3].

7.2 Design

StarfishNet's purpose is to provide communications security for communicating IoT nodes. In practice, this means three guarantees for data in transit: [61,118]

Confidentiality Data in transit can be read only by the sender and recipient.

Integrity Data in transit must be protected from modification.

Authenticity A data message has a guaranteed origin, destination, and context.

Requirement **R.2** indicates the use of strong cryptography for providing these guarantees. Authenticated Diffie-Hellman protocols are an example of just such strong

[2]The practice of deperimeterisation of networks in business IT does bear some similarity to this, chiefly in mechanism rather than principle. Publications by the Jericho Forum (https://www.opengroup.org/jericho) are excellent resources on this topic.

[3]Even Wi-Fi with security disabled – that is, open access, with no authentication or security – makes this assumption: the access point is still the single point of administration (and single point of failure) for the entire network, and can dissociate stations as desired. At the same time, Wi-Fi in this configuration provides no confidentiality or integrity guarantees for any traffic on the network.

Operation	Algorithm	Prototyping implementation
Key agreement	ECDH (secp160r1)	Software
Digital signatures	ECDSA (secp160r1)	Software
Hashing	SHA1	Software
Block cipher	AES-128	Hardware
Block cipher mode	CCM-64	Hardware

Table 7.1: Cryptographic algorithms in use in StarfishNet.

cryptography, and require no third-party participants. Additionally, they can be completed in three messages and three public-key cryptographic operations – an advantage from a performance and resource-consumption standpoint. StarfishNet was therefore based on such a protocol. Each communicating pair of nodes performs an authenticated DH exchange to establish a shared secret. This shared secret is then used as a key in an AEAD algorithm to provide confidentiality, integrity, and authenticity protection for all further communications. (Comparison with TLS, in the ephemeral DH mode, is instructive.)

For the prototype, we permitted a slight relaxation of requirement **R.1**. The initial node (the *network coordinator*) on a network establishes the parameters for, and becomes the root node of, a routing tree, analogous to that of ZigBee in its tree routing mode, and functioning very similarly. This was a compromise for the sake of simplicity: it leaves the power to revoke address allocations in the hands of the coordinator, permitting some fairly spectacular DoS attacks. Mechanisms akin to the mesh routing and stochastic addressing features of ZigBee Pro would remedy this vulnerability. Their implementation, however, was of little relevance to the research at hand, and thus did not justify the very high additional complexity. (Importantly, the coordinator is not involved in the cryptographic exchanges between nodes, and is not able to violate the fundamental security guarantees of StarfishNet.)

Finally, StarfishNet provides reliable data transport between nodes, using a custom acknowledgement/retransmission scheme.

7.2.1 Cryptographic tools in use

StarfishNet uses cryptographic algorithms that are commonplace in WSNs, and frequently hardware-accelerated, detailed in Table 7.1. The motivations for these choices were:

ECC (secp160r1) This was motivated by the addition of 163-bit ECC operations to ZigBee, as part of its Smart Energy Profile. The curve used in StarfishNet is of near identical size to that used in ZigBee, with the only reason for not using the same curve being lack of library support. A curve of this size provides 80 bits of security, considered the minimum viable security level for the open Internet. At the same time, 160-bit keys are small enough for authenticated DH

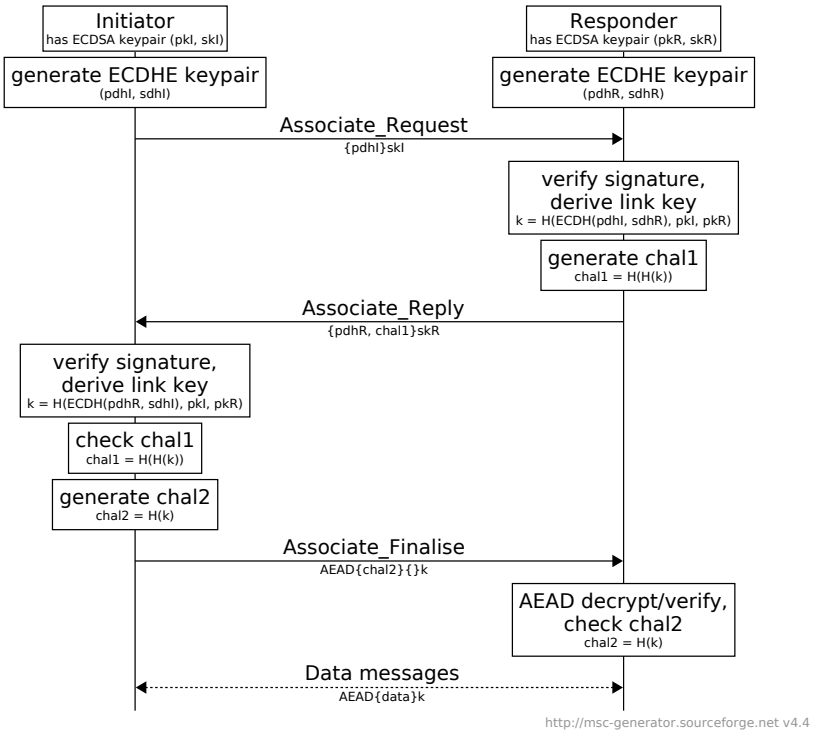


Figure 7.1: StarfishNet association protocol. In this diagram, each party is assumed to know each other's public signing key. However, this is not a requirement of StarfishNet.

protocols to be performed over 802.15.4, in the minimum number of messages, without message fragmentation.

AES-128-CCM-64 This algorithm is mandatory for the provision of IEEE 802.15.4 link-layer security, and is thus frequently hardware accelerated. (Note, however, that StarfishNet does not use IEEE 802.15.4 link-layer security directly.)

SHA1 SHA1 is recommended for deprecation by its publisher, NIST. However, its performance on IoT-class hardware is still higher than its recommended replacement, SHA2; the fact that the output of the hash function is the same size as the output of the ECC operations in use was additionally convenient for prototyping. For any real-world deployment, StarfishNet should switch to using SHA2 instead. Additionally, many recent SoCs, including the immediate successor to our prototyping platform, provide SHA2 hardware acceleration.

Ideally, we would eliminate hashing entirely. However, the use of elliptic-curve Digital Signature Algorithm (ECDSA) digital signatures – and the native support for certificate transport and verification – precludes this option.

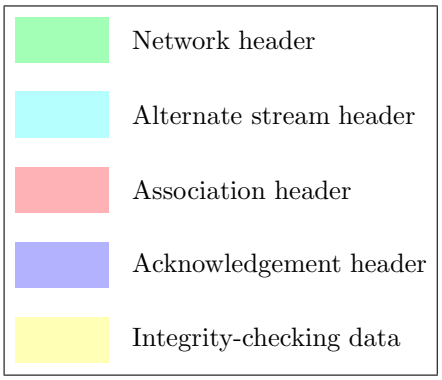


Figure 7.2: Packet diagram legend

7.3 Association protocol description

The setup phase of communication between two StarfishNet nodes is called the *association protocol*, in which the two nodes form a *security association*. This primarily involves the establishment of a shared *link key*, which will be used for encryption and authentication of data packets.

StarfishNet assumes that each node possesses an ECDSA key pair, which serves as its root identity. This ECDSA key pair is, in turn, used to authenticate an elliptic-curve Diffie-Hellman exchange, performed with ephemeral ECDH key pairs. Key confirmation data is exchanged to ensure that the two nodes have indeed derived the same shared key. The protocol is shown diagrammatically in Figure 7.1.

StarfishNet's association protocol requires exactly three messages, the theoretical minimum for an authenticated DH protocol. Detailed descriptions of the contents of and assumptions for each message follow:

1: `Associate_Request` (Figure 7.3(a))

The initiator (`I`) first generates an ephemeral ECDH key pair for this association, and then sends an `Associate_Request` packet to the responder (`R`). This contains, at minimum, the newly-generated ECDH public key. `I` may also choose to include its public signing key in this packet; if so, it sets the `D` bit in the network header. Finally, `I` may also request that `R` supply its public key in the `Associate_Reply`, using the `Q` bit.

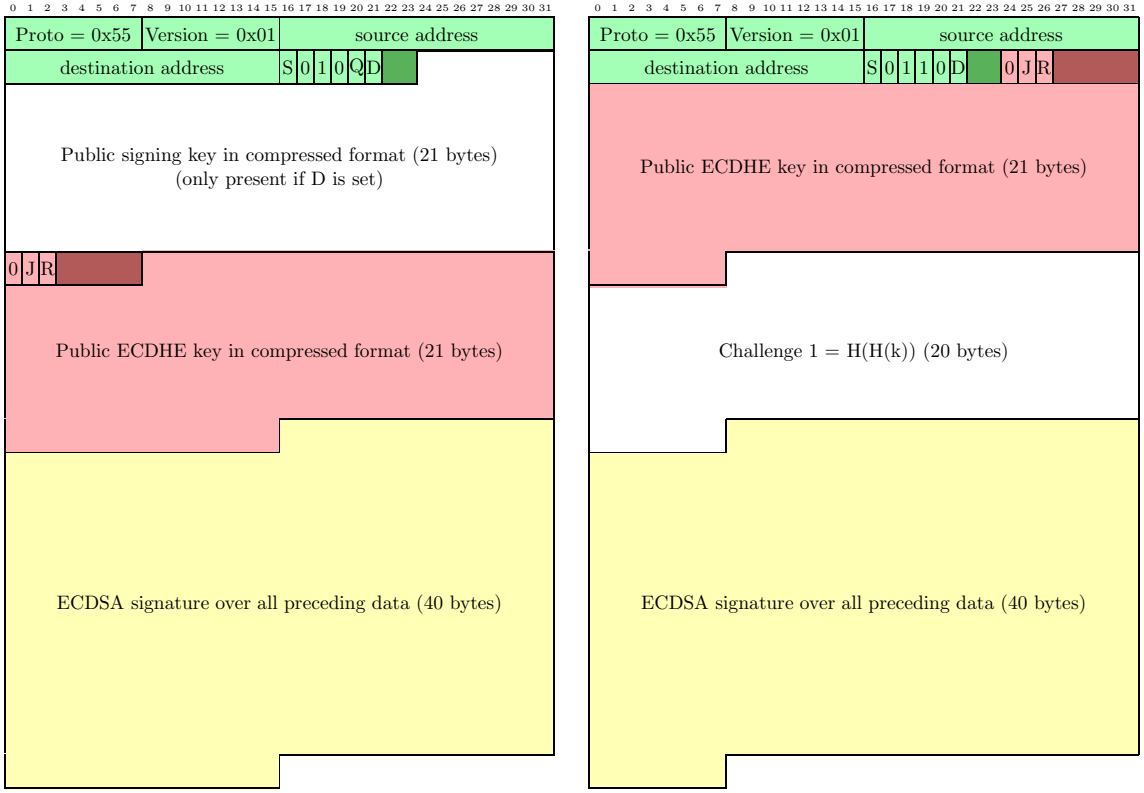
This packet must be signed.

The `Associate_Request` message is, excluding link-layer headers, either 90 bytes long, if `I`'s public signing key is included, or 69 bytes long if it is not.

2: `Associate_Reply` (Figure 7.3(b))

`R` receives the request, and verifies its signature. It then generates its ephemeral ECDH key pair, and uses `I`'s public ECDH key to derive the link key. It then transmits an `Associate_Reply` packet to `I`, containing its own ECDH public key, and the key-confirmation challenge, `chal1` – a double-hash of the link key.

This packet must again be signed.



(a) `Associate_Request` (b) `Associate_Reply`

Figure 7.3: StarfishNet association packets. `J` and `R` are relevant during a network join. The optional signing key field – indicated by `D` – is omitted from (b) for brevity, as is the optional alternate stream header indicated by `S`.

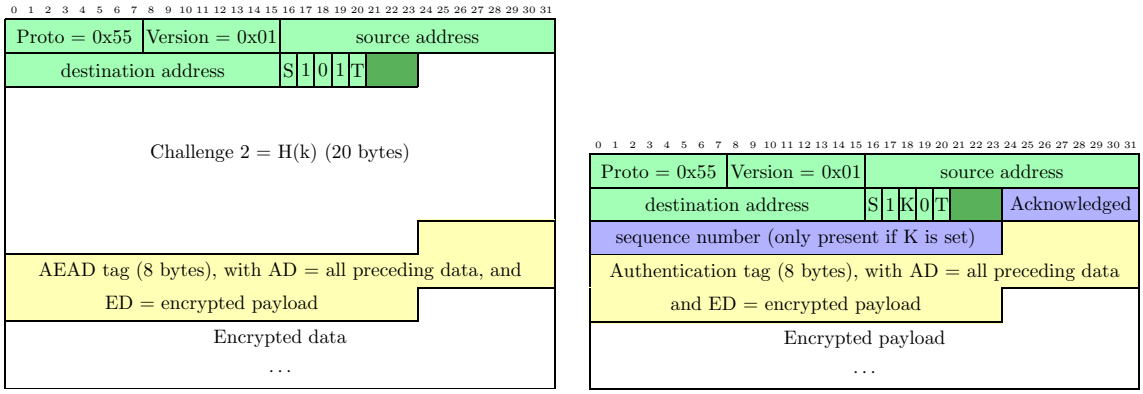
As with `Associate_Request`, `R` may include its public signing key, if it is not known to `I`, by setting the `D` bit. However, the `Q` bit must not be set: to generate an `Associate_Reply`, `R` must already possess `I`'s public signing key, making a request for it redundant.

Excluding link-layer headers, the `Associate_Reply` message is 110 bytes long if `R`'s public signing key is included, or 89 bytes long if not.

3: `Associate_Finalise` (Figure 7.4(a))

`I` receives the reply, and verifies its signature. It then uses `R`'s public ECDH key to derive the link key, and verifies `chal1`. If this succeeds, it generates and sends an `Associate_Finalise` packet, containing a second key-confirmation challenge, `chal2` – this time, a single hash of the link key. This packet is not signed, only encrypted, with all StarfishNet headers (including the key-confirmation header) in the associated-data section – this means `chal2`, like `chal1`, is transmitted in the clear.

The `Associate_Finalise` message can be seen as either 35 bytes long,



(a) `Associate_Finalise` (b) Plain data

Figure 7.4: StarfishNet data packet format, first (`Associate_Finalise`) and subsequent. `T` indicates the payload type: `T = 1` means a certificate, while `T = 0` means plain data. The optional alternate stream header, indicated by `S`, is omitted for brevity.

excluding any data payload and link-layer headers, or imposing 20 bytes of overhead on the first data message. (These perspectives are equivalent.)

This protocol is very similar to STS from Section 3.1.7. However, STS chooses to transmit the authentication data from the initiator, encrypted, in its third message. The same information is transmitted in StarfishNet's first message, the `Associate_Request`, in order to minimise the amount of data that must be provided in `Associate_Finalise`, and maximise the space available for applicaiton data in that packet. As a result, the identities of both participants in the StarfishNet version are public.

Note on link-layer header sizes The size of 802.15.4 headers is variable, because both the source and destination address for a packet can be specified in three different ways: "long address" (64 bits long), "short address" (16 bits long), and no address (zero length). Generally, long addresses are unique to a device and set by its manufacturer, short addresses are set by software, and no address implies the link-layer coordinator. Since StarfishNet does not specify a link-layer coordinator, all frames are addressed using the short address mode, with short addresses allocated by StarfishNet via the scheme described in Section 7.5. This results in a frame header size of 11 bytes. Joining a network is the exception to this rule. Since the joining node does not yet possess a short address, it uses its hardware address until the network join has completed, increasing the frame header size by 6 bytes (to 17). This penalty is only incurred for the first two messages of a network join.

7.3.1 Joining a StarfishNet network

The act of joining a StarfishNet network has little meaning from a security standpoint. Its primary outcome is the allocation of a network address, and with it, the ability to transmit routed packets. This procedure is as follows:

1. The node intending to join, called the *child*, performs a *discovery transaction* by broadcasting a link-layer beacon request. Nearby routing nodes respond with link-layer beacon replies. The child thus generates a list of nearby StarfishNet routers, and their signing keys (which are included in the beacon reply).

2. The child selects a routing node based on PAN ID, signal strength, and advertised capacity. This routing node will be called the *parent*.

3. The child sends an `Associate_Request` to the parent, using its hardware address as source address in the link-layer header, and the special value `0xFFFE` in the network-layer header. The `J` bit in the association header must be set, to indicate this is a network join. The child may additionally set the `R` bit if it is capable of routing. (The destination address at both layers is the parent's short address.)

4. The parent allocates either an address, if `R` was clear, or address block, if `R` was set, from its pool. If `R` was set, but it no longer has any blocks available, it may allocate a single address. (Address allocation is explained in detail in Section 7.5.)

5. The parent sends an `Associate_Reply` to the child. The child's hardware address must be used as the destination in the link-layer header; the network-layer header contains the allocated address (or the base address of the allocated block). The `J` bit in the association header indicates whether an address was allocated, while the `R` bit indicates what type (block or single) of allocation was performed.

6. The child configures the received address as its network address, and uses it for all future communication. (If it received a block, it uses the first address in that block.) Its hardware address will not be used again. If it was allocated an address block, and is capable of routing, it must begin advertising that capability, by responding to discoveries from potential future children.

7. The association protocol completes as normal.

Upon joining a StarfishNet network, or at any other time, the child may elect to perform a further discovery transaction in order to establish which routers are its neighbours, if it did not compile this information during the initial discovery transaction.

7.4 Data transport

Once a security association is formed, data packets (see Figure 7.4(b)) are exchanged between the associated notes. Each packet is encrypted using AEAD, with network-layer header information (including any stapled acknowledgement) forming the associated data to be integrity-checked. In line with the cryptographic requirements

```
struct SN_Statement {
    SN_Public_key_t subject;
    uint8_t         type;
    SN_Hash_t       assertion;
};
```

(a) Implicit certificate

```
struct SN_Certificate {
    SN_Statement_t  statement;
    SN_Signature_t  signature;
    SN_Public_key_t endorser;
};
```

(b) Explicit certificate

Figure 7.5: StarfishNet certificate types

on CCM mode, a new IV is generated for each packet, consisting of a hash of the sender's ECDH public key for this security association, and the sequence number of the packet. The ECDH key is included in order to separate the IV sequences for the inbound and outbound data streams.

Each node maintains two packet counters for every security association in which it participates: the *transmit counter*, which counts packets encrypted and enqueued for transmission, and the *receive counter*, which counts packets received and successfully decrypted and authenticated. (This is similar to SNEP, from Section 3.3.7.)

7.4.1 Access control: a lesson learned from BottleCap

BottleCap provides a decentralised, capability-based access control system. However, its security is predicated on a fundamental guarantee: that symmetric encryption keys can be stored on a remote machine *without any risk that they will be exposed to the software on that machine* (or rather, any software other than the BottleCap binary). This is the purpose of the TC mechanisms it uses: to provide a TEE that enforces that guarantee, in a fashion that is *remotely verifiable*.

The remote verification property uses TC remote attestation, for which trust is ultimately rooted in the two certificates described in Section 2.3.3: the endorsement credential and the platform credential. These certificates bear a strong resemblance to the attribute certificates described in Section 3.1.3, with the TPM and platform manufacturers acting as the relevant attribute authorities.

In a similar vein, attribute certificates are increasingly being investigated as an access control method for IoT applications [155]. This is for good reason: for communications between things to occur without human intervention, especially for ephemeral interactions, there must exist some means by which one node can present evidence of its physical properties – such as the sensors and actuators it possesses – to others.

In StarfishNet, therefore, we introduced primitives for the transport and transparent integrity-checking of two kinds of access control data structure:

explicit certificates A digital certificate, albeit not one conforming to X.509. Instead, a StarfishNet certificate is a custom data structure designed for absolutely minimal size, shown in Figure 7.5(b). The structure contains five fields: `subject` contains the public key of the certificate's subject; `type` defines how to interpret the `assertion` field; `assertion` describes the attributes being

asserted by the certificate, with its interpretation controlled by `type` (it could be a bitfield, a small data structure, or a hash of a large data structure, for example); `signature` contains an ECDSA signature over the preceding three fields; and `endorser` contains the public key needed to verify `signature`.

Since ECDSA signature verification is a demanding operation on IoT platforms, explicit certificates may be sent as simple data messages in order to avoid StarfishNet's automatic signature verification.

implicit certificate The implicit certificate – or statement – structure, shown in Figure 7.5(a) contains a strict subset of the information in an explicit certificate. In particular, it elides the `signature` and `endorser` fields. Without a signature (or, indeed, a signer), this is not a digital certificate in the traditional sense. Instead, an implicit certificate's endorser is considered to be the sender of the message containing it, with its authenticity guaranteed by StarfishNet's transport mechanism.

Implicit certificates therefore strongly resemble BottleCap tickets. (The StarfishNet implicit certificate mechanism can be seen as an instantiation of BottleCap, in which a different issuer key is *always* used for each capability holder. That issuer key is the StarfishNet link key.)

Finally, StarfishNet certificates can be used to implement *cryptographic delegation*, in which a node delegates the responsibility for performing association transactions to another, usually higher-power, node; we will refer to these as the *slave node* and *master node*, respectively. The slave node is then only required to complete a single association, with the master. It then issues a StarfishNet explicit certificate, with the master as subject, and the assertion stating that it is a delegation certificate. The slave can then transmit this certificate to nodes attempting to perform an association, instructing them to contact the master instead. (Analogously to the ZigBee recommended use of ECC; see Section 3.3.10.)

7.4.2 Aside: implementing the Resurrecting Duckling

When using attribute certificates on a StarfishNet network, the most natural keying model is the following: each device has a single ECDSA key pair serving as its identity, used in all StarfishNet association transactions, and serving as the subject of attribute certificates issued by the device manufacturer for establishing its properties. However, this keying model suffers a major drawback: it becomes impossible to change the device's identity without losing those attribute certificates. Implementing the Resurrecting Duckling device lifecycle model (see Section 3.1.5), which requires a change of long-term key material upon a change of owner, is therefore impossible.

In order to implement the Resurrecting Duckling, we suggest that devices use two key pairs, which we shall name the *device root key* and *device identity key*. The former would function as the manufacturer-set identity of the device, and thus the subject of manufacturer-supplied attribute certificates, and would be unchangeable by software. The second would be created on the device at the time its ownership is taken, to

be used as the identity in all StarfishNet association transactions, as well as the subject of attribute certificates whose scope is limited by the device's ownership. The device root key would then only ever be used to sign two kinds of certificates: those designating a device identity key – necessary so that the manufacturer's assertions can be used in operation – and those later revoking it. It can therefore be protected by strong means – such as inside a TPM or similar hardware – with restrictions on the signatures it is permitted to issue. (An alternative model would have the device identity key issue its own revocation certificate. The device root key then only ever issues a single kind of certificate. This also permits privacy-preserving revocation of identity keys.)

7.5 Addressing and routing

StarfishNet uses a 16-bit network address, in order to take advantage of the short address functionality of 802.15.4. The join procedure outlined above clearly establishes a tree structure on the network, with the first node (or *coordinator*) at its root. This is referred to as the *routing tree*, and is analogous to the ZigBee construct of the same name.

When a coordinator starts up and establishes a StarfishNet network, it selects three global parameters which are constant over the lifetime of the network:

PAN ID This is a 2-byte link-layer parameter, and allows multiple networks to coexist on the same channel.

branching factor, b The branching factor of the routing tree: the number of sub-blocks into which a block may be divided. This also determines the maximum number of routing children a router may have. It must be a power of 2.

leaf blocks, L the number of sub-blocks in each block reserved for allocation to non-routing children, which are leaf nodes in the routing tree. These sub-blocks are always at the beginning of the address block. It must be at least 1.

These parameters are published by the coordinator, and every other router in the network, in their router advertisement messages. Every router also publishes its depth in the tree in bits, d, with the coordinator having depth 0. As a result, each new router can, upon joining, easily determine the size of the block, s_b, that it has been allocated:

$$\begin{aligned} B &:= \log_2(b) \\ s_b &= 2^{16-Bd} \end{aligned}$$

These calculations only require simple binary arithmetic, so are fast even on the simplest processors.

The coordinator is necessarily 'allocated' a block 16 bits in size, covering the entire address space. That block is further split into b sub-blocks, of which the first L are

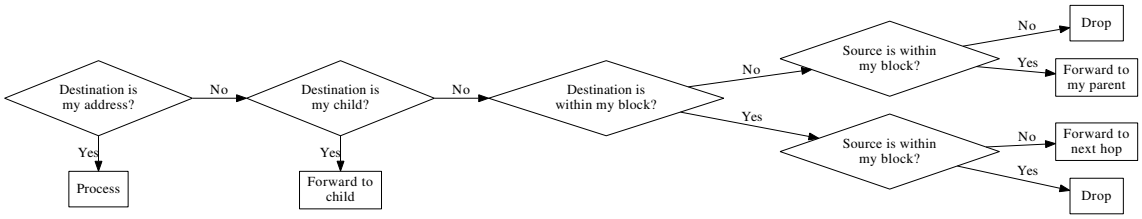


Figure 7.6: Routing decision tree for StarfishNet tree routing algorithm.

reserved for leaf nodes. The coordinator takes the first address from that range, which is always `0x0000`.

Any other router, upon being allocated a block, performs an analogous operation: it divides that block into b further sub-blocks, of which the first L will be reserved for leaf nodes. It then allocates to itself the first address from those L sub-blocks, which is always the first address.

The addresses `0xFFFE` and `0xFFFF` are reserved, since both have a special meaning in IEEE 802.15.4. `0xFFFE` is the 'invalid address', used to indicate that a node does not possess a short address; it is used in StarfishNet for the same purpose. `0xFFFF` is the link-layer broadcast address.

7.5.1 Routing algorithm

The routing algorithm in use in StarfishNet is very similar to the 'tree routing' algorithm used in ZigBee. Each router executes the decision sequence shown in Figure 7.6 every time a packet is received by the network layer.

The algorithm for testing whether an address is within a given block uses the power-of-2 property above. We observe that, since s_b is a power of 2, bitwise-AND with $s_b - 1$ is equivalent to reduction modulo s_b. Similarly, with an overbar denoting bitwise-NOT, we note that bitwise-AND with $\overline{s_b - 1}$ is equivalent to subtracting the remainder of division by s_b.

Therefore, with s_b again the block size, a the address, a_b the base (first) address of the block, and $\wedge$ the bitwise AND operation, we can ask:

$$a \wedge \overline{(s_b - 1)} = a_b?$$

This question (that is, whether $a \wedge s_b$ is equal to a_b) is equivalent to asking whether a is in the block. Again, these calculations require only binary arithmetic, meaning they perform well on any CPU.

Non-routing nodes Non-routing nodes do not follow this decision sequence. When receiving packets from the network, they simply ignore any are not destined for them. When transmitting, they always transmit to their parent.

Neighbour optimisation Any node may transmit a packet directly to its destination if the destination node is a neighbour (that is, within transmission range).

Likewise, a node is free to 'shortcut' across the tree – that is, if any node on the routing path is its neighbour, it may transmit directly to that neighbour, instead of through the entire tree. This shortcutting may be done at any point along the packet's route.

7.6 Alternate streams

StarfishNet provides a single, bidirectional message sequence between a pair of nodes. However, some applications may require the presence of several independent message streams. This may be for separating disparate concerns, or to avoid *head of line blocking*, also a known issue with TCP [149]. StarfishNet therefore permits a pair of nodes to establish an *alternate stream*: a secondary message stream between the same pair of nodes.

In order to use alternate streams, a pair of StarfishNet nodes must already have a security association; we will refer to its message stream as the *master stream*. Establishing an alternate stream then consists of performing a separate, new association transaction, with new ECDH keys. The signing keys used during this association may not be included in the alternate stream's `Associate_Request` and `Associate_Reply` packets, as they can during the master association; they must be exchanged by some other means, such as inside the master stream. They need not, however, be the same as those used to establish the master stream.

Alternate streams maintain separate encryption keys, as well as all other state, to each other, and to the master stream. A diagram of a packet containing alternate stream data is shown in Figure 7.7(a); it differs only from a packet containing master stream data (see Figure 7.4(b)) in the addition of the alternate stream header. Other packets in an alternate stream are likewise identical to their master stream counterparts, except for the addition of this header.

Naturally, this feature is not without overhead. Each alternate stream is a separate StarfishNet association, including keys, sequence numbers, and association state. Additionally, each packet in an alternate stream contains an alternate stream header, imposing an overhead equal to one plus the length of the alternate stream index.

7.7 Overhead-reduction techniques

In addition to the design features hitherto discussed, StarfishNet applies three techniques specifically designed for reducing potential overhead. One, *acknowledgement stapling*, is widely used in protocol design, including TCP. Our implementation differs slightly, and contributes to the goal of minimising radio transmissions.

StarfishNet also introduces a technique from the WSN domain (and used successfully in several of the protocols in Section 3.3), which we term *counter elision*. It is designed to minimise the size of StarfishNet headers, maximising the available application data per packet.

Finally StarfishNet introduces a novel technique, *acknowledgement tagging*, for removing the potential overhead in network traffic introduced by these optimisations.

Aside: header compression in Internet protocols Several Internet protocols specify header compression formats for low-capacity links. During the period when Internet connections commonly took place over serial lines, Van Jacobsen TCP/IP header compression [74] was designed to maximise the usage of those links by omitting header fields unless they were changing from message to message. Likewise, 6LoWPAN (see Section 3.3.9) specifies a variety of header compression formats, both at the network layer (for IP) and transport layers (for UDP). CoAP [153] response piggybacking has a similar effect.

7.7.1 Acknowledgement stapling

Protocols offering reliable packet delivery do so by means of acknowledgements. These are often, as in the case of ZigBee, separate packets, sent alongside the data stream. The alternative, used notably in TCP, is for acknowledgement information to be included within the headers of data packets – that is, 'stapled' to the data packets.

StarfishNet takes the latter approach, in order to avoid incurring a radio transmission for every received packet. However, in TCP, an acknowledgement number field is always present, and ignored in situations where it is not relevant. StarfishNet instead omits it entirely in such cases.

7.7.2 Counter elision

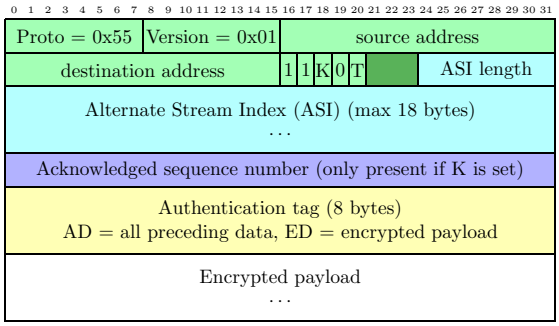
Each side of a security association keeps a receive counter, which records the sequence number of the last successful decryption, and thus implicitly the next valid sequence number. Since only one sequence number is valid for reception at any given time, with invalid packets dropped, data is therefore received by the application precisely the order it was sent.

The initialisation vector for each packet is a hash of the sender's public ECDH key and the packet's sequence number. Since the ECDH public keys are necessarily already known to both sides, and only one sequence number is valid for reception at any given time, each packet's IV in its entirety can therefore be derived by both nodes, and thus the IV need not be explicitly included in the packet's header.

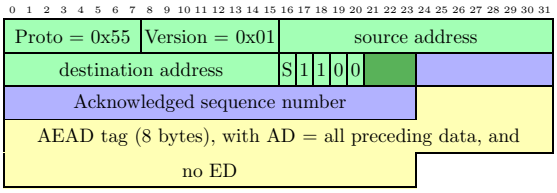
An identical technique is notably used in SNEP (see Section 3.3.7).

7.7.3 Acknowledgement tagging

In situations where one peer in an association transmits much more data than the other, there may not be a conveniently-timed data transmission to which to staple an acknowledgement. StarfishNet therefore needs a mechanism to transmit acknowledgement-only packets, similar to those used by ZigBee; these are termed *pure acknowledgements* in StarfishNet parlance. However, counter elision causes both parties to update shared state on each packet transmission, requiring this state change to be acknowledged. Pure acknowledgements thus cannot be encrypted in the normal way. If they were, constant background transmissions of acknowledgements would be required during



(a) Alternate stream (note that `S = 1` in the network header)



(b) Pure acknowledgement packet

Figure 7.7: Variant data packet formats. In (a), `T` indicates the payload type, in the same way as Figure 7.4; note that `S = 1`. In (b), the optional alternate data stream header has been omitted for brevity, and `T = 0`.

periods of low activity. For low-power devices, where radio transmissions must be minimised, this is clearly inappropriate.

StarfishNet solves this problem by generating the IV for pure acknowledgements differently, appending a 3-byte *acknowledgement tag* to the string to be hashed: the constant value "ACK". (The sequence number used is that being acknowledged.) Upon transmission of such a packet, the transmission counter therefore need not be updated: this IV is only ever be used for packets of this form, which are necessarily all identical, and so the uniqueness property of the IV is preserved.

A pure acknowledgement is depicted in Figure 7.7(b).

7.8 Interoperability with IP networks

To quote [61]:

> A direct interpretation of the term *Internet of Things* refers to the use of standard Internet protocols for human-to-thing or thing-to-thing communication in embedded networks.

Roman *et al* in [141] also take this view, presenting a taxonomy of methods by which WSNs are generally integrated into the IoT in this way. This section will describe a method for doing the same for a StarfishNet network, using the mechanisms described in this chapter to permit StarfishNet nodes to communicate with almost any other machine on the Internet. The only exception is machines on other StarfishNet networks, for which this method is ill-suited. This method falls under the *gateway* classification in the aforementioned taxonomy.

Whether or not an IoT network is actually using IP to communicate, there is clearly a requirement that it be able to communicate with machines that do. In particular, machines on the Internet should be able to communicate transparently with nodes on a StarfishNet network that is Internet-connected. This section presents an illustrative example for how this could be achieved, using alternate streams.

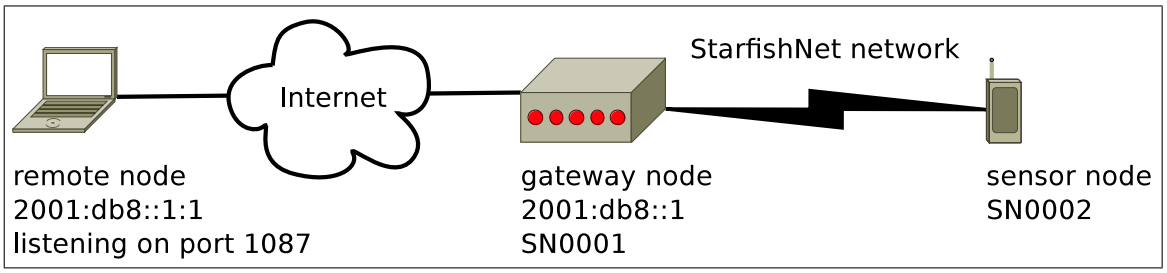


Figure 7.8: Example StarfishNet/IP interoperability scenario. StarfishNet addresses are shown in the form `SNxxxx`. The StarfishNet network has been assigned the IPv6 prefix `2001:db8::/112`.

The example scenario involves three nodes. One node is connected only to a StarfishNet network, and will be referred to as the *sensor node*. The second is a gateway connecting the StarfishNet network to the Internet, and will be referred to as the *gateway node*. The final node is the machine with which communication is to be established via the Internet, referred to as the *remote node*. The scenario is presented diagrammatically in Figure 7.8.

The scenario rests on the following core assumptions:

- Communication is to be established between the sensor node and the remote node. Naturally, this communication must be via the gateway node; however, the gateway should not be able to decrypt or modify data in transit.
- The StarfishNet network has been assigned an IP address range. The range should be of a size that permits the host ID section of the IP address to be at least 16 bits long, the length of a StarfishNet address. The gateway node must be the router 'responsible' for this address range. (Note that the sensor node need not know this address range.)
- The sensor node has established a security association with the gateway node.

We will also make several minor assumptions for the sake of simplicity:

- All Internet communications take place exclusively using IPv6.
- All nodes possess the public signing keys of all other nodes with which they will communicate.
- The sensor node is initiating communications.
- The sensor node knows the IP address of the remote node, and the UDP port on which it is listening.
- The remote node knows the IP address range of the StarfishNet network.

In this example, the remote node will communicate with the gateway using UDP. The gateway will then extract the relevant data from the UDP packet, and forward it across the StarfishNet network to the sensor node.

To initiate communications with the remote node, the sensor node attempts to establish an alternate stream with the gateway. In the `Associate_Request`, it sets the alternate stream index to the IP address and UDP port number with which it is to communicate (in the example in Figure 7.8, `2001 0db8 0000 0000 0000 0000 0001 0001 043F`, an 18-byte index).

The gateway then extracts the IP address and port number from the alternate stream index, and encapsulates the entire `Associate_Request` in a UDP datagram with that destination. The source IP address is set to the StarfishNet network's prefix, followed by the sensor node's StarfishNet address (in this case, `2001:db8::2`). The source UDP port is set as needed, or may be ignored. Finally, the datagram is sent to the remote node.

The remote node, upon receiving the datagram, generates an `Associate_Reply` for the same alternate stream index, and encapsulates it in a UDP datagram with the inverse addressing information. It transmits it back to the gateway, which strips the UDP encapsulation and forwards it to its destination over the StarfishNet network. Note that although the remote node must know both the sender and gateway nodes' StarfishNet addresses to do this, both are included in the headers of the `Associate_Request`. The signatures on both messages, combined with both nodes' possession of each others' public keys, guarantee both the source and integrity of the messages.

Communication continues in this manner, with StarfishNet packets being encapsulated in UDP datagrams for transport over the Internet. The alternate stream index provides the information required for the gateway to route packets to the remote node, while both the IP address and encapsulated StarfishNet header provide the information needed to route in the reverse direction.

If the remote node initiates communication rather than the sensor node, it is still the *remote node*'s IP address and UDP port that are used in the alternate stream index. The remote node must additionally discover (or otherwise possess) the StarfishNet addresses of the gateway and sensor nodes.

Variations on this scheme can be used in situations with slightly different assumptions. For example, rather than discarding the UDP port number on the StarfishNet side, a longer alternate stream index can instead be used to provide a range of channels between the remote and sensor nodes. If the network does not have a range of IP addresses (that is, the only available IP address is that of the gateway), the gateway node's IP address can be used in all IP addressing, with StarfishNet routing information extracted solely from the encapsulated StarfishNet packet – this variant is therefore in the *front-end* classification in Roman's taxonomy. UDP port numbers on the gateway node could also be used in this way.

We note that, unfortunately, this mechanism does not appear to be usable for connecting two StarfishNet networks over the Internet.

Finally, there is no requirement for the gateway node to permit all bridging requests; connection to the IP network is a resource which it is exposing, and it may apply an access control policy to that resource. In fact, this is very likely desirable. Filtering incoming requests can help avoid the StarfishNet network being flooded with spurios packets. Filtering outgoing requests can help inhibit unauthorised or

Packet type	StarfishNet		802.15.4		**Total**
`Associate_Request`* *(with signing key)*	90	+	11	=	**101**
`Associate_Request`* *(join, with signing key)*	90	+	17	=	**107**
`Associate_Request`* *(no signing key)*	69	+	11	=	**90**
`Associate_Reply`* *(no signing key)*	89	+	11	=	**100**
`Associate_Reply`* *(join, no signing key)*	89	+	17	=	**106**
`Associate_Finalise`	35	+	11	=	**46**
Data packet	15	+	11	=	**26**
Data packet *(alternate stream; excludes ASI)*	>16	+	11	=	**>27**
Data packet *(acknowledgement stapling)*	19	+	11	=	**30**
Pure acknowledgement*	19	+	11	=	**30**

Table 7.2: Space overhead for key StarfishNet packet types, in bytes. Packet types marked with a * carry only StarfishNet metadata, not data.

malicious sensors from connecting to the Internet.

7.9 Summary

This chapter describes StarfishNet, the network-layer protocol developed during the course of this project. It was designed to provide an exemplar fulfilling the requirements from Section 4.2, particularly **R.1**, decentralised security, and **R.2**, the ability to maintain its security guarantees in the face of a strong adversary.

Three fundamental design decisions contribute to its ability to meet these requirements: the use of an authenticated DH protocol between each communicating pair of nodes to establish a shared secret, the invariant that *all* communications between those two nodes must be secured with that shared secret, and the perimeterless network architecture.

We close this chapter with a summary, in Table 7.2, of the space overhead imposed by StarfishNet – that is, the amount of space consumed by StarfishNet headers – for a set of core packet types. The next chapter, Chapter 8, evaluates StarfishNet against our requirements, analysing in greater detail whether it can be considered to have met them.

Chapter 8

Evaluating StarfishNet

This chapter evaluates StarfishNet. We begin with a short description of known engineering shortcomings in the prototype (Section 8.1); these do not substantially affect its research value, but do inhibit its real-world deployment. Section 8.2 then presents a brief description of the Tamarin protocol verification tool, followed by our work in using it to verify the security guarantees provided by StarfishNet. Experimental validation using the prototype is then presented in Section 8.3. Finally, Section 8.4 uses all of this information to assess StarfishNet against the requirements from Section 4.2.

8.1 Engineering shortcomings in StarfishNet

StarfishNet is a research prototype with a variety of engineering shortcomings. Some of these are deliberate omissions, not affecting the research results; some demonstrate lessons learned during the project, indicating needed changes to the protocol; and some reflect the availability of resources in a research context.

First, functionality that would be necessary for a network-layer protocol of this kind, but not necessary for the research in this thesis, was omitted from StarfishNet, as were some features of StarfishNet from the prototype implementation. Similarly, some such functionality was implemented in a reduced or simplified form:

Routing and addressing The ZigBee-inspired tree-based routing and address allocation used in StarfishNet is very simple to develop and understand, and very power-efficient. However, it results in a network that still possesses a degree of centralisation, enabling some spectacular DoS attacks by the coordinator or other routers over large sections of the network. Replacing tree routing with ZigBee-Pro-like stochastic addressing and full mesh routing would avoid these attacks, at the expense of both energy and time: during a network join, the joining device is must ensure that its generated address is not in use; likewise, during operation, nodes must perform route discovery in order to communicate.

Retransmission algorithm The algorithm for determining when to retransmit a packet is simple, but wasteful. It also expresses highly pathological behaviour when the retransmission buffer is full, or if packet drops are frequent.

Dissociation Dissociation is completely unimplemented, since it was not necessary for the research done. As a result, StarfishNet treats associations as permanent; if one node loses the association's state, there is no way to inform its partner.

Rekeying Rekeying is also completely unimplemented, for the same reason. It is, however, a necessary feature: CCM imposes a hard limit on both the amount of data and number of messages that can be exchanged under a given key.

Mobility In principle, StarfishNet permits nodes to easily adapt to network topology changes. This helps mitigate the aforementioned DoS attacks by routers. However, this functionality is not implemented in the prototype codebase. Handling mobility-related issues, which requires similar kinds of adaptation, is also unimplemented.

Broadcast and multicast The problem of encrypted and authenticated broadcast and multicast transmission is essentially a solved one, using (μ)TESLA or a similar scheme (see Section 3.3.7). TESLA could be integrated into StarfishNet with only minor modifications and modest engineering effort; however, we did not see any research value in doing so. StarfishNet does not, therefore, implement any broadcast or multicast functionality.

Address resolution StarfishNet provides no mechanism for discovering the address of a node given its public key. This functionality would be important in general, but particularly so for any mobility provisions.

Service discovery Service discovery for StarfishNet nodes is also completely unimplemented, for two reasons. First, this functionality was not needed for the experiments in this thesis. Second, secure service discovery on IoT networks is itself an open area of research.

Message transport Data transport was implemented for data messages and explicit certificates only; implicit certificates were left unimplemented.

Delegation One of the intended uses of StarfishNet explicit certificate transport is *cryptographic delegation*: the ability for a StarfishNet node to delegate association (particularly ECC) processing to another, more powerful node. Implementation of this feature would only require modest engineering effort, but was not needed for the prototype.

In a similar vein, the engineering of the StarfishNet prototype is appropriate for a research prototype, but not necessarily for any real-world deployment, for the following reasons:

Memory optimisation needed The node table and retransmission buffer consume a large amount of memory, relative to the total available on the platform. Some optimisation in this area is therefore needed. However, we note that the size of the retransmission buffer is a tradeoff: since StarfishNet data messages require acknowledgement, and a packet must be kept in the retransmission buffer until

this occurs, the number of slots in the retransmission buffer therefore limits the maximum transmission rate of the system.

Error recovery Packet transmission is assumed to succeed – that is, errors in the radio layer are not detected or handled, except by the normal retransmission mechanism.

Certificates Rather than use a custom certificate data structure, StarfishNet could easily be modified to support SPKI certificates [37,38], which were designed with similar goals in mind. We note, however, that this could impose another dependency on a hashing algorithm, which may not be desirable (see below).

The tools available to us also constrained our engineering in several ways:

SDCC The compiler used, Small Device C Compiler (SDCC)[1], is the only freely-available compiler for the Intel 8051 CPU core on the CC2530. The compiler recommended by the SoC manufacturer, Texas Instruments, is a commercial compiler sold by IAR Systems[2]; full-featured debugging tools from Texas Instruments are only available as plugins for this compiler. (Conversely, Contiki only compiles with SDCC, not with the IAR compiler.) We do not know if the code generated by these compilers differs greatly in its performance or memory consumption, but have seen anecdotal evidence that SDCC tends to generate inferior code.

µECC The information in Section 2.1 indicates that ECC operations should be possible with high performence on our hardware, on our hardware, which is comparable to (and, in fact, should be 10 times faster than – see Section 8.3.2) the CC1010 mentioned in Table 2.1. However, as later sections will demonstrate, we experienced extremely poor performance on these operations. We expect that this is due to µECC. However, to the best of our knowledge, µECC is the only open-source ECC library available that could be modified for compilation with SDCC for our testbed platform.

System-on-a-chip Mere months after the purchase of our prototyping hardware, Texas Instruments released a successor to its SoC, the CC2538[161]. This SoC contains a more advanced cryptographic processor, which implements SHA2 and ECC in addition to various modes of AES. Its CPU core, in addition to being far higher-performance, is also based on the ARM architecture[11] for which high-quality tools and libraries are much more readily available. Using this hardware would have alleviated the preceding two issues.

Finally, there were also errors made in the design of StarfishNet itself, which were either discovered after the work was complete, or during its course:

[1] `http://sdcc.sf.net`
[2] `https://www.iar.com/iar-embedded-workbench/8051`

Key confirmation The explicit key confirmation performed in the `Associate_Reply` and `Associate_Finalise` messages is useful for modelling and verification purposes. However, it is wasteful: strictly, the `challenge2` field in `Associate_Finalise` is unnecessary, since key confirmation is also implicitly verified by successful decryption of an encrypted data packet. Elimination of this field would allow `Associate_Finalise` to carry as much data as any other data packet.

The `challenge1` field in `Associate_Reply` can also be eliminated. Instead, a MAC of the contents of the packet could be performed – keyed with the shared secret – and *this* signed using ECDSA. This would entirely eliminate the need for a hash function from the association protocol, and unify key confirmation and integrity verification into a single operation.

IV generation The IV generation scheme currently uses the sender's ephemeral DH public key. This results in storage overheads, since each node must store both the ephemeral and signing keys of all of its partners. Instead, IVs should be generated using signing keys (which must always be stored, since they are the subjects of digital certificates).

Additionally, as an upcoming section will clearly demonstrate, the use of SHA1 for generating data message IVs was an error. Its performance is very low in our tests, due to the software implementation in use. Since IV generation for CCM does not, in any case, require a cryptographically secure hash function of this kind, a simpler operation would have been a better choice. With a maximum length of 13 bytes for a CCM IV, reducing the relevant public key (21 bytes long) to a length of 9 bytes or less using a simple, XOR-based mechanism, and appending the packet sequence number, would have been entirely adequate, and performed far better. (Some provision would have to be made for acknowledgement tagging, but this is not a difficult problem.)

8.2 Verification

To verify the basic security properties for which the protocol was designed, we modelled it in the Tamarin Prover [150]. Tamarin assumes a Dolev-Yao attacker, wherein the adversary can intercept and modify packets arbitrarily, with the restriction that they may not break any cryptographic assumptions. While details regarding the operation of Tamarin are considered beyond the scope of this paper, we include below a short description of Tamarin syntax and usage, to aid in reading the following sections. We then present the slightly simplified protocol model used, and the security properties proven. Relevant excerpts of the model are included in figures; the full model can be found in Appendix A.

8.2.1 Tamarin

Tamarin's three key concepts are rules, facts, and lemmas. It also permits the definition of arbitrary functions, or the declaration of a function without a definition (in which case it remains abstract).

A *fact* is simply a logical statement, which takes any number of parameters. Facts are used to represent states or stages in a protocol, and their arguments to move information between these states. A fact may be prefixed with an exclamation point (`!`) to denote that it is persistent (see below). Tamarin also provides a number of built-in facts, of which four were used in our model:

`Fr(x)` means that x was freshly generated.

`Out(x)` denotes the transmission of x onto the network (at which point it becomes known to the adversary).

`In(x)` denotes the reception of a network message containing x (which may have been generated or altered by the adversary).

`K(x)` means the adversary knows x.

Rules are triples, written as `p --[ a ]-> c`, where `p`, `a`, and `c` are all lists of zero or more facts. (The shorthand `p --> c` may be used if a is empty.) We call `p` the *premises*, `a` the *actions*, and `c` the *conclusions*. A rule represents the transition between one state (represented in turn by a fact) and another, the conditions required for that transition, and the operations performed during its course. Thus, the rule above can be read as "if `p` is true, then `c` is true, and `a` was done". A fact is established by its presence in `a` or `c`, and consumed by its presence in `p`. The exception is persistent facts, which may be appear in the premises of multiple rules.

Finally, facts may contain variables. A variable marked with a `~` denotes a newly-generated quantity, such as a nonce, while `$` indicates a global name.

Lemmas are logical statements regarding actions, which Tamarin attempts to prove. The facts appearing in a lemma must be actions in a rule; they may not merely be conclusions. Lemmas may be of either the all-traces or exists-trace types; the former must be true for all possible execution traces of the protocol, while the latter only require that a trace exist for which the lemma holds. The symbols `Ex` and `All` mean 'there exists' and 'for all', respectively, while a `#` denotes a timing variable – that is, a variable containing the abstract timestamp of an event, for reasoning about event ordering; a timing variable is associated with an event by way of the `@` symbol. Finally, `==>` denotes implication.

Tamarin provides predefined sets of rules and functions for common cryptographic primitives, as *builtins*. We used the following builtins in our model:

hashing provides the `h(x)` abstract function, which represents a cryptographic hash;

```
RULE Sk_Reveal:
  [ !Sk($A, ltkA) ]
  --[ SkReveal($A) ]->
  [ Out(ltkA) ]

RULE Link_Key_Reveal:
  [ !LinkKey($A, $B, k) ]
  --[ LinkKeyReveal($A, $B, k) ]->
  [ Out(k) ]
```

Figure 8.1: Tamarin rules used to model key compromise

signing provides `pk(sk)`, a function relating a public key to its private counterpart, and `sign{x}sk` and `verify(pk, x)`, which model the generation and verification of a digital signature on a message; and

diffie-hellman provides the '^' operator, and algebraic rules for modelling Diffie-Hellman key agreement. It was written for finite-field DH, but is generic enough to represent ECDH with no alterations.

8.2.2 Protocol model and security properties

An excerpt from our Tamarin model can be found in Figure 8.2. We have omitted rules generating the persistent facts `!Sk(A, skA)` and `!Pk(A, pk(skA))`, which simply represent nodes' private and public signing keys, respectively.

Each rule is named for the type of message generated in the course of its execution, with the actual transmission and reception of those messages modelled using the `Out` and `In` facts. Actions are generated in all but the `Associate_Request` rule; these are used in the lemmas establishing key agreement. Note that message 3 is modelled as containing a `MAC` of the key confirmation data; this is to represent that this message is AEAD-encrypted, with the key confirmation data sent in plaintext.

Finally, the `Sk_Reveal` rule models the disclosure of a node's private signing key, while the `Link_Key_Reveal` rule models the disclosure of the link key itself. (See Figure 8.1.)

We use this model to prove the following properties (see Figure 8.3):

key agreement both nodes agree on the key established, and on the identity of the partner with whom it was established;

key secrecy the established key is only known to the two nodes performing the protocol; and

perfect forward secrecy compromise of a node's private signing key (its only long-term key material) does not reveal the link keys established by any previous run of the protocol.

```
RULE Associate_Request:
  LET peI = 'g'^~seI
  IN
  [ !Sk($I, skI), !Pk($R, pk(skR))
  , FR(~seI) ]
  -->
  [ Associate_Request($I, $R, ~seI)
  , OUT(<peI, SIGN{peI}skI>) ]

RULE Associate_Reply:
  LET
    peR = 'g'^~seR
    k = H(<peI^~seR, pk(skI), pk(skR)>)
    chal1 = H(H(k))
  IN
  [ !Sk($R, skR) , !Pk($I, pk(skI))
  , FR(~seR)
  , IN(<peI, SIGN{peI}skI>) ]
  --[ LinkKeyRProvisional($I, $R, k) ]->
  [ Associate_Reply($I, $R, k)
  , OUT(<peR, chal1, SIGN{<peR, chal1>}skR>)
  ]

RULE Associate_Finalise:
  LET
    k = H(<peR^seI, pk(skI), pk(skR)>)
    chal1 = H(H(k))
    chal2 = H(k)
  IN
  [ !Pk($R, pk(skR)), !Pk($I, pk(skI))
  , Associate_Request($I, $R, seI)
  , IN(<peR, chal1, SIGN{<peR, chal1>}skR>)
  ]
  --[ LinkKeyI($I, $R, k) ]->
  [ OUT(<chal2, MAC(chal2, k)>) ]

RULE Associate_Finalise_Recv:
  LET chal2 = H(k)
  IN
  [ Associate_Reply($I, $R, k)
  , IN(<chal2, MAC(chal2, k)>) ]
  --[ LinkKeyR($I, $R, k) ]->
  []
```

Figure 8.2: Tamarin model of StarfishNet association protocol

```
LEMMA Key_Agreement_I: "ALL I R lk #i .
  LinkKeyI(I, R, lk) @ i &
  NOT(Ex #e . SkReveal(I) @ e) &
  NOT(Ex #e . SkReveal(R) @ e)
  ==>
  (Ex #r . LinkKeyRProvisional(I, R, lk) @ r)"

LEMMA Key_Agreement_R: "ALL I R lk #i .
  LinkKeyR(I, R, lk) @ i &
  NOT(Ex #e . SkReveal(I) @ e) &
  NOT(Ex #e . SkReveal(R) @ e)
  ==>
  (Ex #r . LinkKeyI(I, R, lk) @ r)"

LEMMA Key_Secrecy: "ALL I R lk #i .
  LinkKeyR(I, R, lk) @ i &
  NOT(Ex #r . LinkKeyReveal(I, R, lk) @ r) &
  NOT(Ex #r . SkReveal(I) @ r) &
  NOT(Ex #r . SkReveal(R) @ r)
  ==>
  NOT(Ex #k . K(lk) @ k)"

LEMMA Perfect_Forward_Secrecy: "ALL I R lk #i #k .
  LinkKeyR(I, R, lk) @ i &
  K(lk) @ k
  ==>
  (Ex #r . LinkKeyReveal(I, R, lk) @ r) |
  (Ex #r . SkReveal(I) @ r & r < i) |
  (Ex #r . SkReveal(R) @ r & r < i)"

LEMMA Protocol_Works: EXISTS-TRACE "
  Ex I R lk #p #i #r .
    LinkKeyRProvisional(I, R, lk) @ p &
    LinkKeyI(I, R, lk) @ i & p < i &
    LinkKeyR(I, R, lk) @ r & i < r &
    NOT(Ex #k . K(lk) @ k) &
    NOT(Ex #r . LinkKeyReveal(I, R, lk) @ r) &
    NOT(Ex #l . SkReveal(I) @ l) &
    NOT(Ex #l . SkReveal(R) @ l)"
```

Figure 8.3: Lemmas proven by Tamarin on the protocol in Figure 8.2

Algorithm	Prototype implementation	Real-world implementation
ECC (secp160r1)	Software[3]	Hardware[4]
SHA1[5]	Software[6]	Hardware[4]
AES-128	Hardware	Hardware
CCM-64	Hardware	Hardware

Table 8.1: Implementations of StarfishNet cryptographic algorithms on prototype hardware, and expectations for real-world devices.

The lemma used for perfect forward secrecy is slightly stronger, stating that if a link key was established, and the attacker knows it, then the attacker must have gained knowledge of one of the two participants' private keys *before* the protocol took place. This means that a link key is not compromised by the disclosure of either node's private key, or of any previous link key.

We include an extra lemma, (called `Protocol_Works` in Figure 8.3, or `Honest_I_And_R_Still_Work` in the full model), to assert that, for a pair of honest nodes performing the protocol, it can also successfully terminate.

The full Tamarin model can be found in Appendix A.

Aside: a vulnerability is found In an earlier version of StarfishNet, the link key was derived by hashing only the raw ECDH result, not the public signing keys of the participants. This resulted in an identity-misbinding vulnerability identical to the one in STS (see Section 3.1.7 and [89]).

Fixing this vulnerability is simple: the final hash operation was changed to include the public signing keys of the associating nodes (binding the link key to their identities).

The vulnerability was found – and the fix verified – by Tamarin, using the model we have just finished describing.

8.3 Experiment

Our prototype was implemented on an industry-standard SoC for IoT devices, the Texas Instruments CC2530 [160]. The SoC provides hardware implementations of the IEEE 802.15.4 PHY layer, as well as hardware acceleration of AES-128 and AES-CCM. Table 8.1 lists the cryptographic algorithms in use in StarfishNet, and the implementations available on our prototyping hardware; it also indicates our expectations for their availability in real-world deployments.

[3]Heavily-modified version of `https://github.com/kmackay/micro-ecc`

[4]The successor to our test SoC, the Texas Instruments CC2538, provides hardware implementations of of SHA2 and ECC, in addition to as AES and CCM[161].

[5]Used in the prototype for its higher performance than SHA2, when implemented in software.

[6]Modified version of `http://bradconte.com/sha1_c`

Operation	Time	Energy
SHA1 hash (25 bytes)	7.30 ms	197 µJ
SHA1 hash (49 bytes)	7.45 ms	201 µJ
Hardware PRNG (128 bytes)	0.55 ms	14.8 µJ
AES-128 encryption (single block)	0.09 ms	2.43 µJ
AES-CCM encryption/decryption (96 bytes)	0.81 ms	21.9 µJ
AES-CCM MAC generation (96 bytes data + 11 bytes AD)	0.61 ms	16.5 µJ
ECDSA signature generation	47.64 s	1.28 J
ECDSA signature verification	59.23 s	1.60 J
ECDH key agreement	57.61 s	1.55 J
802.15.4 frame transmission (115-byte payload + 11-byte header)	4.93 ms	424 µJ

Table 8.2: Benchmarking of critical operations on prototyping hardware.

The experiments performed on the prototype were divided into two stages. The first consists of microbenchmarks of the key software and hardware operations underlying StarfishNet. The second then consisted of measuring packet round-trip latencies across a StarfishNet network.

Following the discussions of those two series of experiments is an assessment of the memory requirements of the prototype.

8.3.1 Microbenchmarks of key operations

Using the 15.625kHz timer on the CC2530, we measured the time taken to perform the fundamental operations underlying StarfishNet – that is, those primitives upon which it is built. These operations are chiefly cryptographic in nature, but also include hardware primitives, such as the use of the hardware pseudorandom number generator, or the transmission of a packet by the built-in IEEE 802.15.4 radio. Table 8.2 shows our results, using the cryptographic coprocessor to perform AES and CCM operations in hardware. To demonstrate the effect of hardware acceleration, we also performed similar measurements using the software implementations of these primitives in the Contiki source tree. These results are shown in Table 8.3. The difference is clear: hardware acceleration results in a performance gain of roughly two orders of magnitude.

Finally, Table 8.2 and Table 8.3 both also show energy consumption figures, in addition to time taken for each operation. These were calculated from the time taken.

Operation	Time	Energy
AES-128 encryption (single block)	4.89 ms	132 µJ
AES-CCM encryption/decryption (96 bytes)	37.24 ms	1.01 mJ
AES-CCM MAC generation (96 bytes data + 11 bytes AD)	54.77 ms	1.48 mJ

Table 8.3: Cryptographic algorithm performance without hardware acceleration, for comparison with Table 8.2.

First, from standard electrical theory, we recall the following:

$$\begin{aligned} P &= IV \\ E &= Pt \\ \therefore E &= IVt (\text{for constant } I \text{ and } V) \end{aligned}$$

Five of the seven nodes on the testbed network were powered by a pair of standard AA batteries connected in series (the other two were powered via USB from a desktop computer), which have a combined energy capacity of approx. 15kJ. This power supply has a nominal voltage of 3V.

To determine the current consumption of the SoC, we consulted its datasheet [160]. During a radio transmission, at the transmit power used in these experiments (the default in Contiki), the nominal current drawn by the CC2530 is 28.7mA; we used this value for calculating the '802.15.4 frame transmission' entry in Table 8.2. All other operations in both tables consist solely of computation. We therefore used the current value quoted in the datasheet for 'medium CPU activity, no peripherals'; the datasheet quotes a range, from which the maximum was selected, 8.9mA. Finally, the current drain values for the two timers used by Contiki (90µA and 0.6µA) were added to both.

Errors and significant figures

All of the preceding measurements (except those on ECC operations; see below) were performed using the 15.625kHz timer on the CC2530. For operations that only require computation, the only source of error is therefore the precision of that timer – 64µs. Moreover, all of the measurements were taken over a large number of runs – 4096 for hardware-accelerated AES and CCM operations, 48 for software AES and CCM operations, and 256 for SHA1 operations. The error margins on these measurements are therefore less than the numerical precision with which they are presented in Table 8.2 and Table 8.3.

The 802.15.4 frame transmission entry is taken from the data presented in the next section. The measurement and error-handling methodology there therefore apply to it.

The poor performance of ECC operations in the prototype necessitated a different approach, since the timer used for the preceding measurements has a maximum range of approx. 4s. We used the 128Hz system clock instead, which has an effectively infinite range, and timed 200 repetitions. Once again, this reduces the error margins on these measurements below the numerical precision with which they are presented in Table 8.2.

8.3.2 Benchmarking of StarfishNet

The next series of experiments used the same 15.625kHz timer to measure the round-trip latency of a StarfishNet packet – that is, the total amount of time required from the time `SN_Send()` is called with a message, to the time the callback registered with `SN_Receive()` is called with its reply. The reply contained the same data as the initial message. All packets exchanged during the course of these measurements were StarfishNet data packets with stapled acknowledgements. Each packet therefore had 30 bytes of header information: 11 bytes of link-layer headers, 11 bytes of StarfishNet headers, and 8 bytes of integrity-protection data.

We consider latency to be the most informative performance measurement for a network of this kind: IEEE 802.15.4 is designed for low-rate, low-power networking, chiefly for conveying measurements from sensor nodes or signals to actuators. It is not designed for long continuous streams of data, and neither is StarfishNet, suggesting that little would be learned from a throughput measurement.

In order to separate the latency contributions from various components of the protocol stack, we performed a test of the full protocol, followed by tests of successively more reduced versions of the protocol: we first disabled packet encryption, then authentication, then IV generation. (This last test therefore involves no cryptography at all.) We varied two parameters during each test: the length of the data message, from 1 to 96 bytes, and the 'distance', from 1 to 6 hops (with 1 denoting direct transmission, and 6 denoting 5 intervening routers).

Each message/response cycle was performed 1000 times, with latencies measured individually. Points where retransmissions or spurious acknowledgements occurred were discarded, since these indicate the presence of data corruption, implying interference on the channel. We also discarded outliers, which we defined as points for which the latency was more than 10ms higher than the minimum, since they imply that link-layer collision avoidance was invoked. This was the only mechanism we had for detecting delays due to the IEEE 802.15.4 collision-avoidance algorithm; we could not detect them explicitly due to a bug in the network stack.

We also used the same method to measure the time required by the radio driver to perform the actual transmission. This was only done once for each packet size, for the direct transmission case. The results were then multiplied by the appropriate number of packet transmission events to derive equivalent figures for greater distances.

The data was then plotted, with each point representing an arithmetic mean over identical repetitions, filtered as described above. The standard deviation was also calculated, and included as error bars – which, for most points, are sufficiently small as to be invisible. The plots are shown in the following figures:

- Figure 8.4: Latency vs message length, direct transmission.
- Figure 8.5: Latency vs message length, 1 intervening router.
- Figure 8.6: Latency vs message length, 2 intervening routers.
- Figure 8.7: Latency vs message length, 3 intervening routers.
- Figure 8.8: Latency vs message length, 4 intervening routers.
- Figure 8.9: Latency vs message length, 5 intervening routers.
- Figure 8.10: Latency vs distance, for several packet sizes.

Each figure is composed of two panels. The upper panel shows the full set of results, while the lower panel elides the radio transmission and no cryptography data sets, and uses a false origin to emphasise the effect of packet security.

The gaps between trendlines in the first six figures allow the various contributions to the latency to be divided into the following categories:

Radio transmission Time to perform the physical radio transmission.

No cryptography - radio transmission Time spent in network stack and OS logic.

IV generation - no cryptography Time spent generating packet IVs.

Integrity protection - IV generation Time spent generating packet MACs.

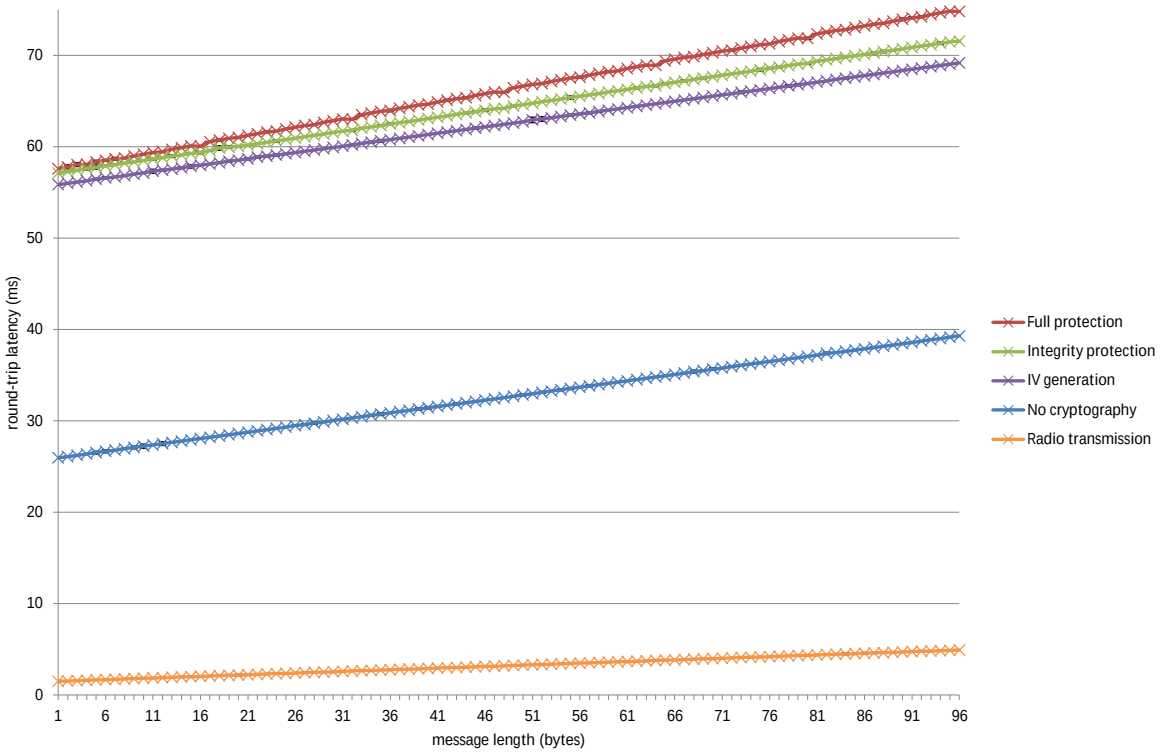
Full protection - integrity protection Time spent encrypting packet data.

The largest contribution to round-trip latency is clearly network stack and OS logic. This reflects the unoptimised, prototype nature of our implementation. Optimisation work would be very likely to reduce this cost, as would a more capable optimising compiler.

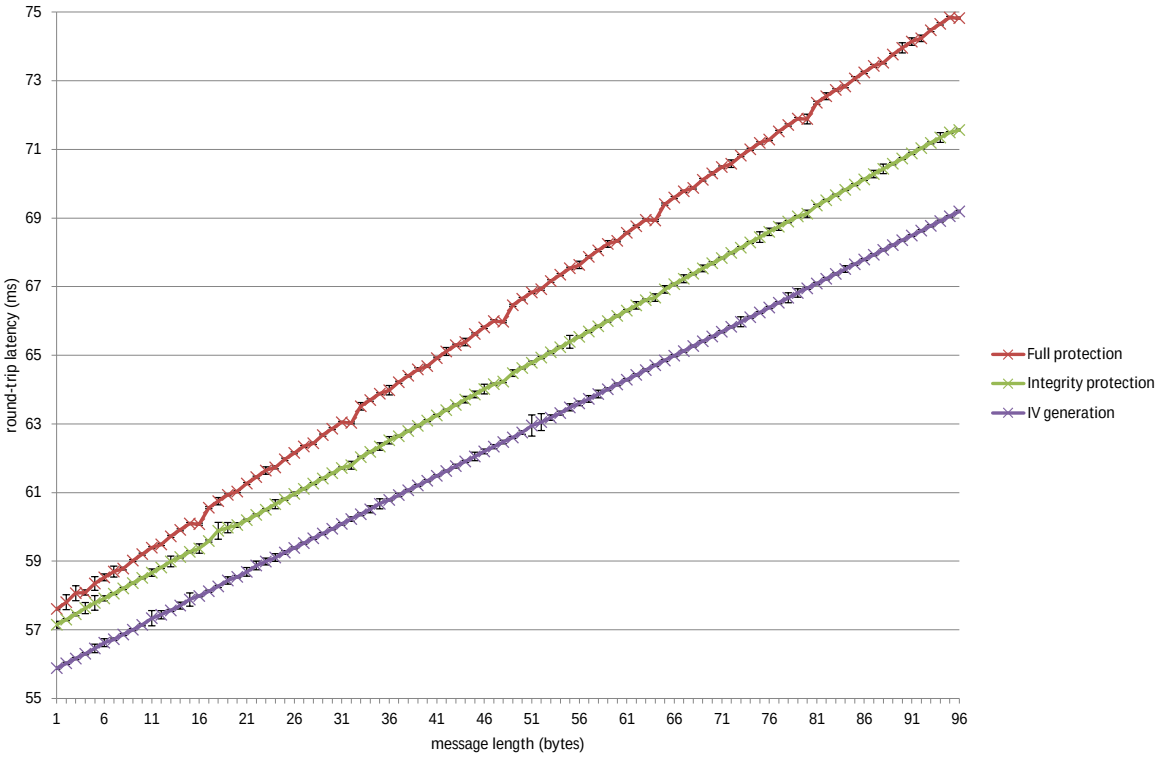
The second-largest contribution is also clear: packet IV generation. This clearly reveals a performance bug in StarfishNet: the use of SHA1 in the generation of packet IVs, exacerbated by SHA1 being implemented in software in the prototype. (Recall that a packet's IV is a hash of its sequence number and the sender's ephemeral public key.) This is simply an oversight; the use of a cryptographically secure hash or random number generator is not necessary for CCM IVs [36]. IV generation can therefore be done using a much simpler digest algorithm.

Figure 8.10 shows that the overhead introduced by packet encryption is *constant* for a given packet size. In principle, this should not be a surprise; StarfishNet's cryptographic protections are end-to-end. However, experimental measurements provide show that the reality does, indeed reflect the principle. More subtly, it also demonstrates that in general, StarfishNet behaves as expected under routing: at least within the measurements we could perform, latency grows linearly with distance.

Finally, we note that the latency introduced by packet encryption and authentication is low; between 1.7ms per round-trip (for a 1-byte message), and 5.6ms (for a

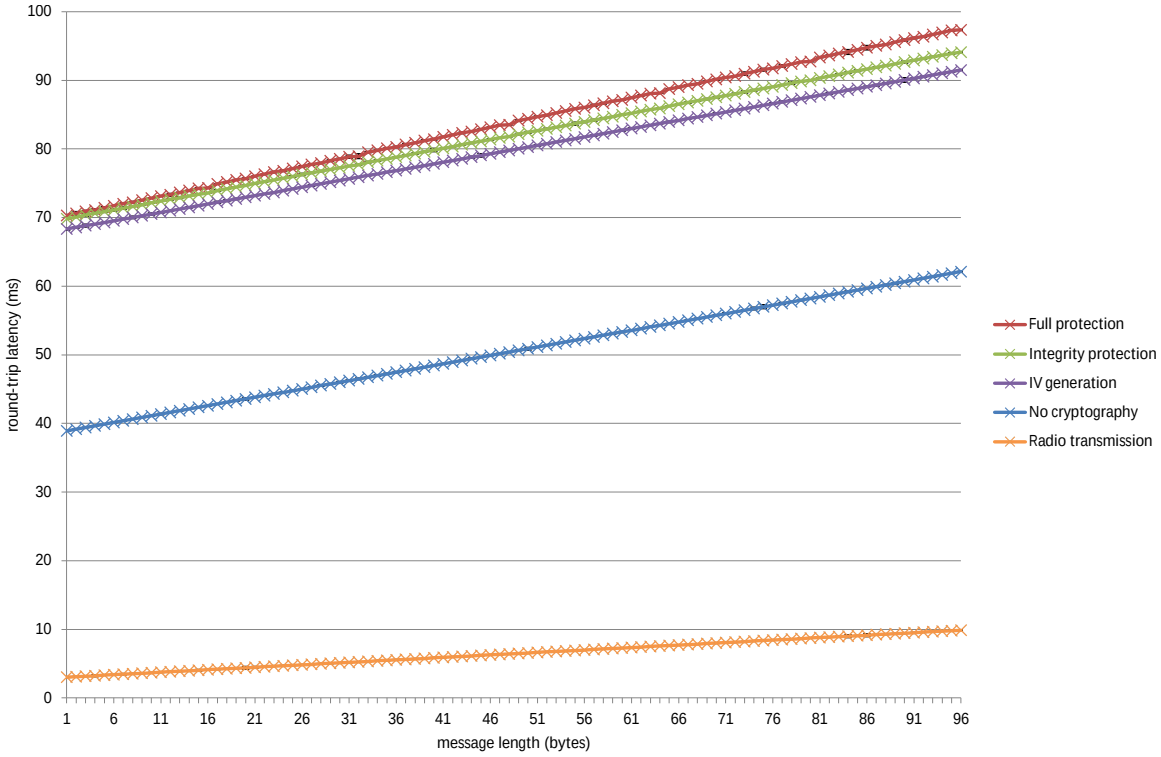


(a) Full results

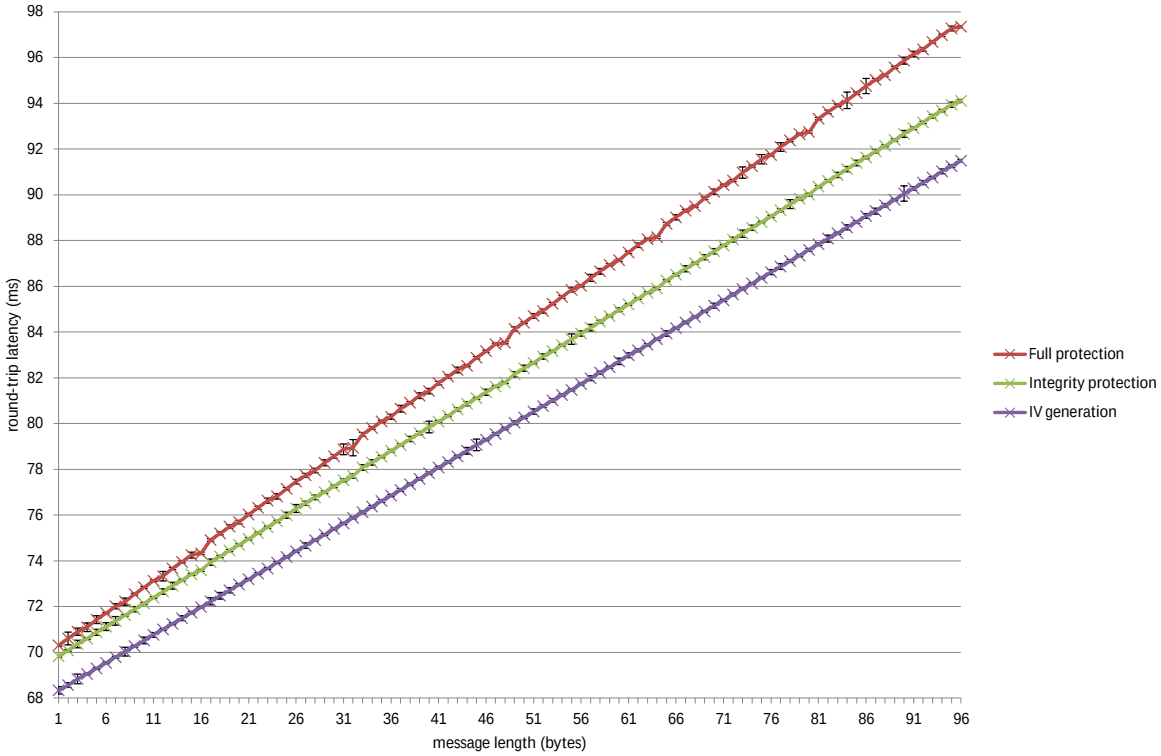


(b) False origin, with 'no cryptography' results omitted

Figure 8.4: StarfishNet round-trip latency vs message length, between adjacent nodes. Colour indicates level of cryptographic protection.

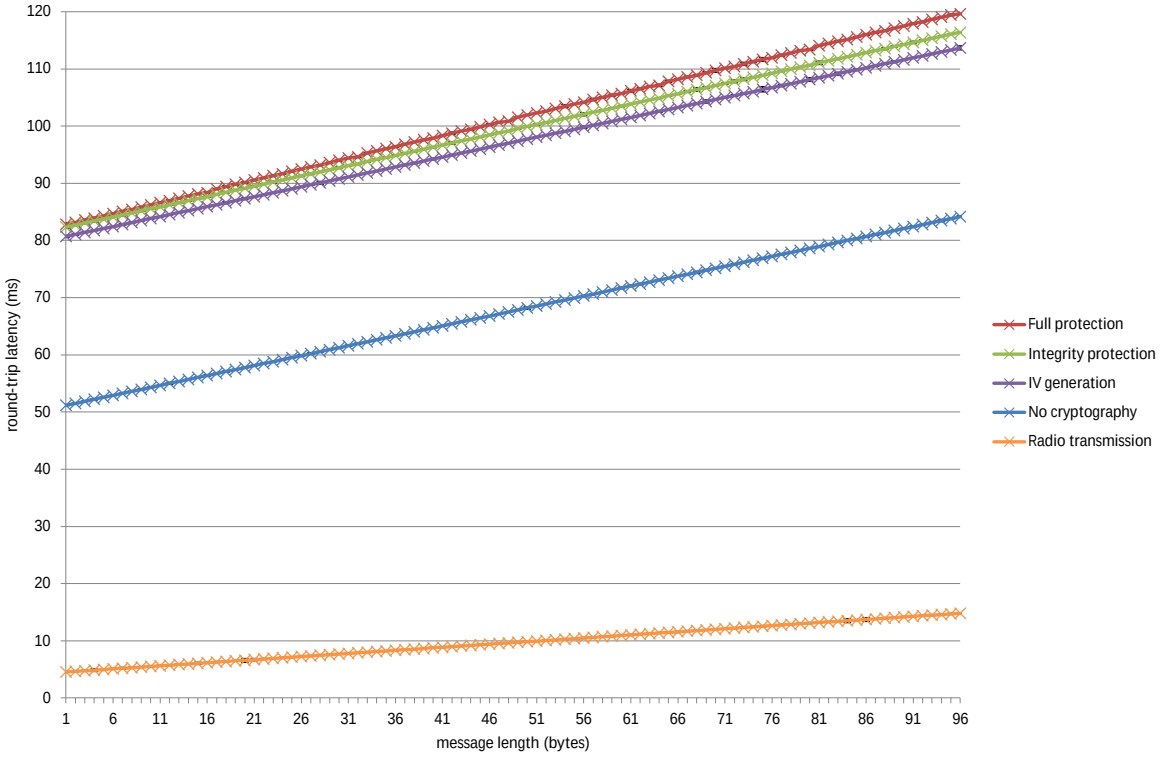


(a) Full results

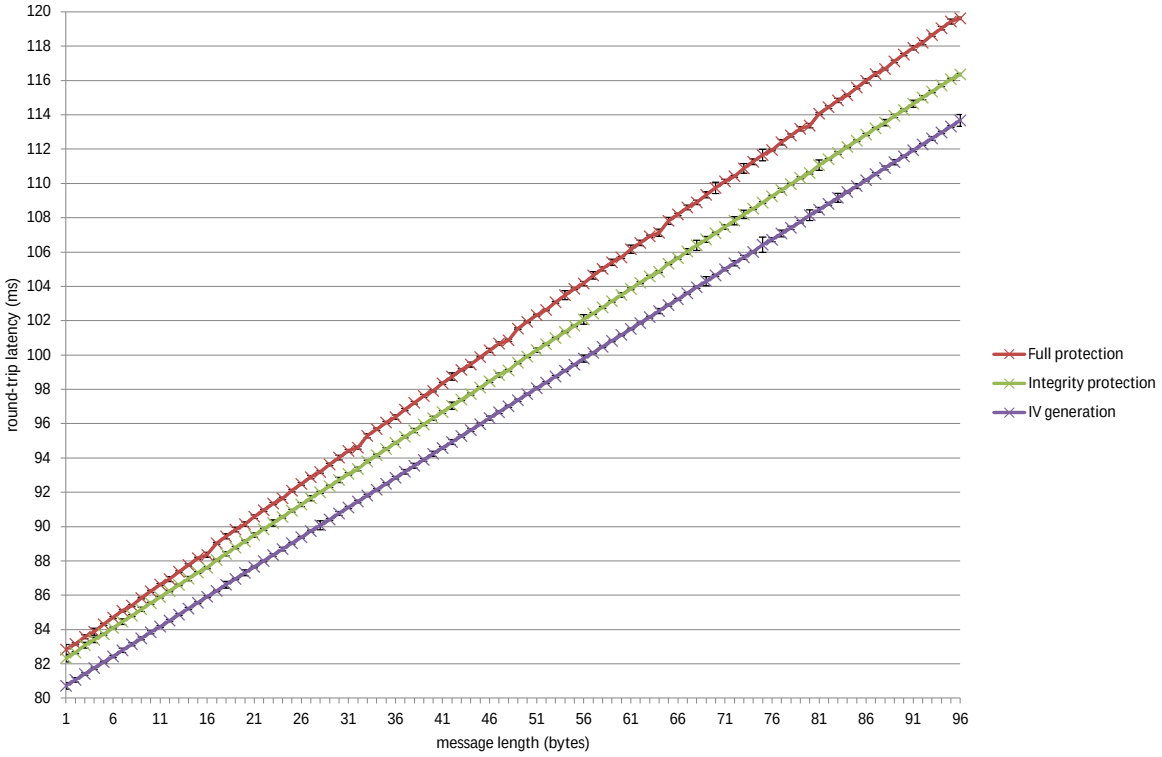


(b) False origin, with 'no cryptography' results omitted

Figure 8.5: StarfishNet round-trip latency vs message length, between nodes with 1 intervening router. Colour indicates level of cryptographic protection.

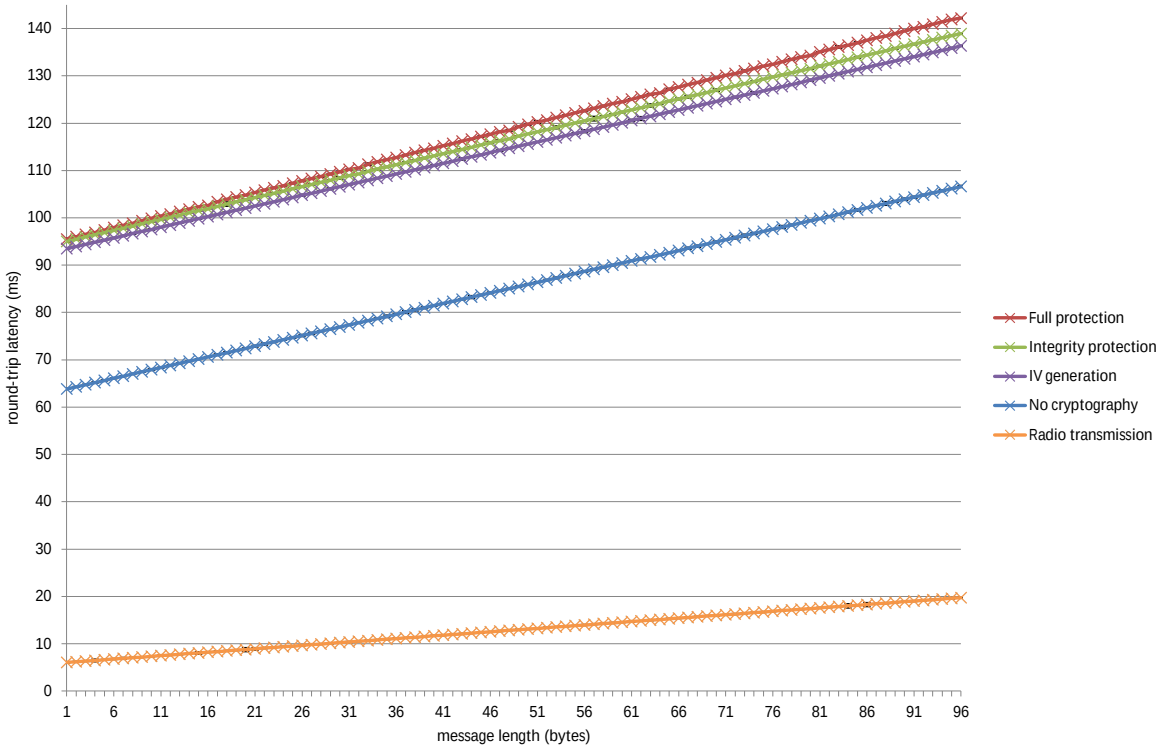


(a) Full results

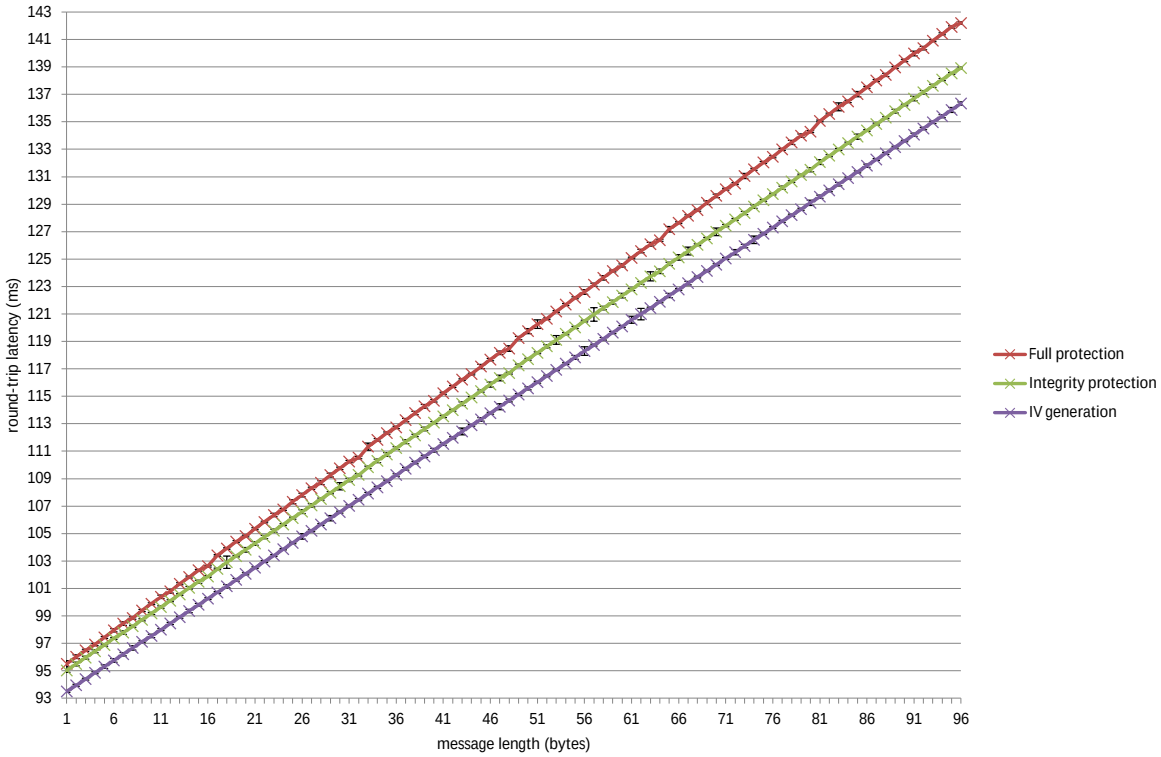


(b) False origin, with 'no cryptography' results omitted

Figure 8.6: StarfishNet round-trip latency vs message length, between nodes with 2 intervening routers. Colour indicates level of cryptographic protection.

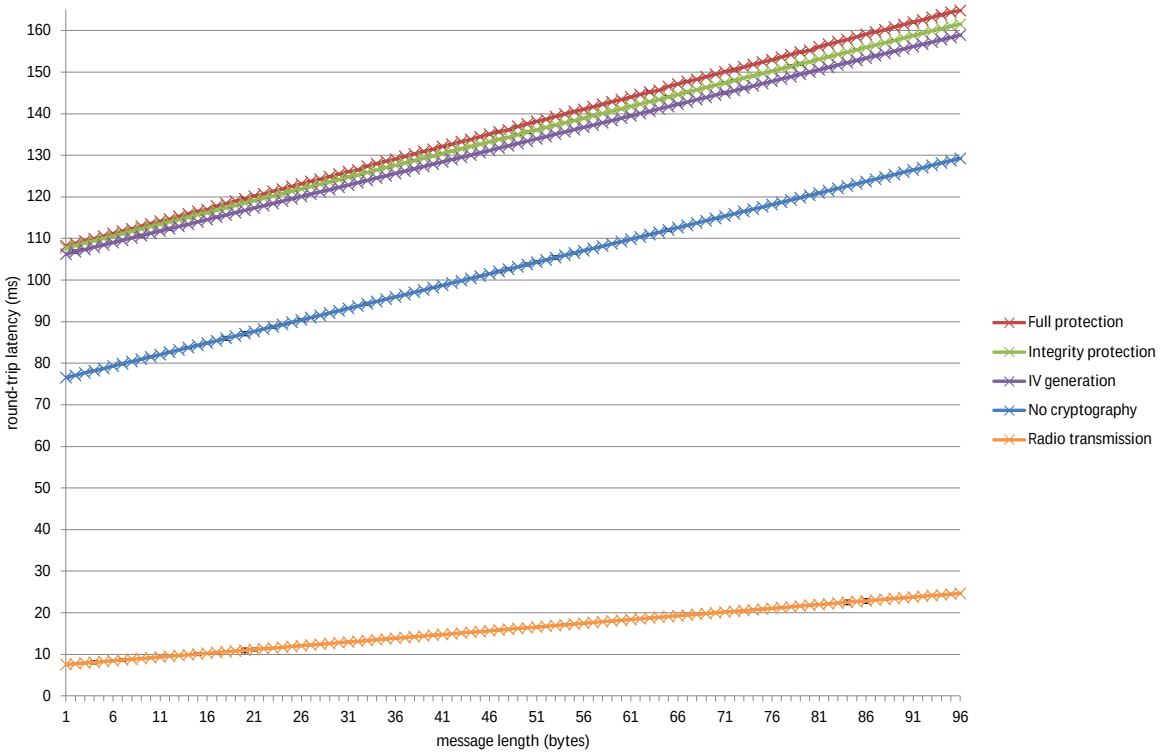


(a) Full results

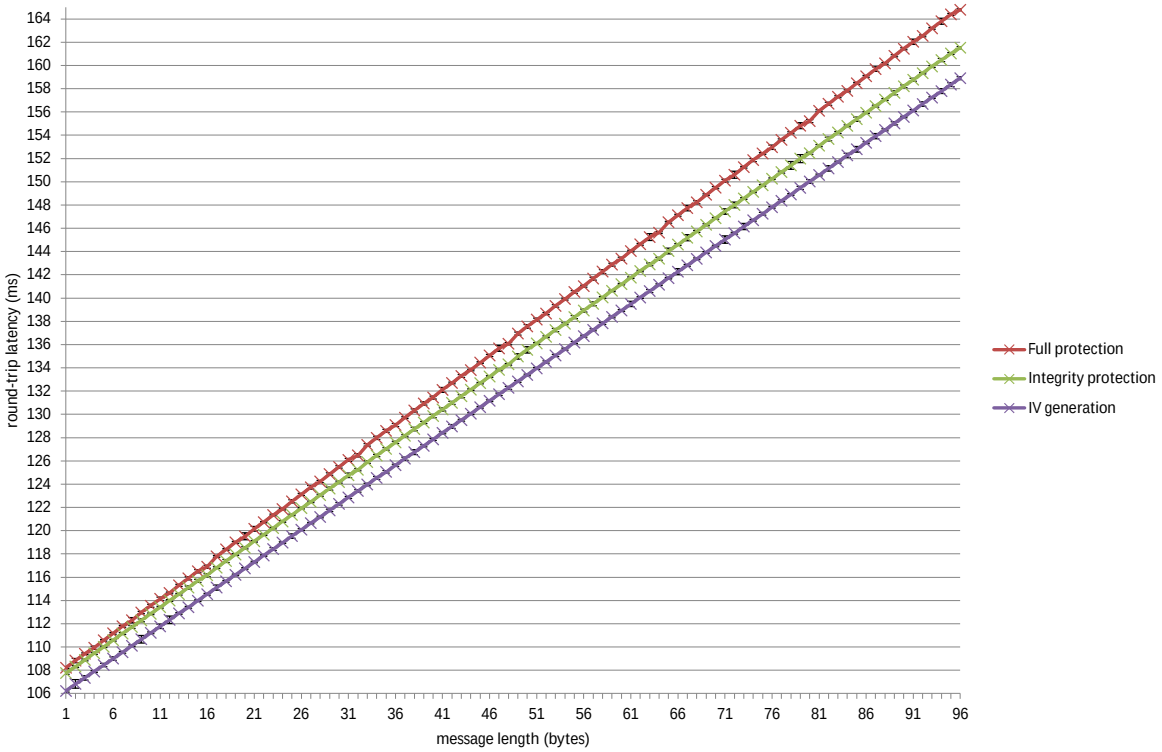


(b) False origin, with 'no cryptography' results omitted

Figure 8.7: StarfishNet round-trip latency vs message length, between nodes with 3 intervening routers. Colour indicates level of cryptographic protection.

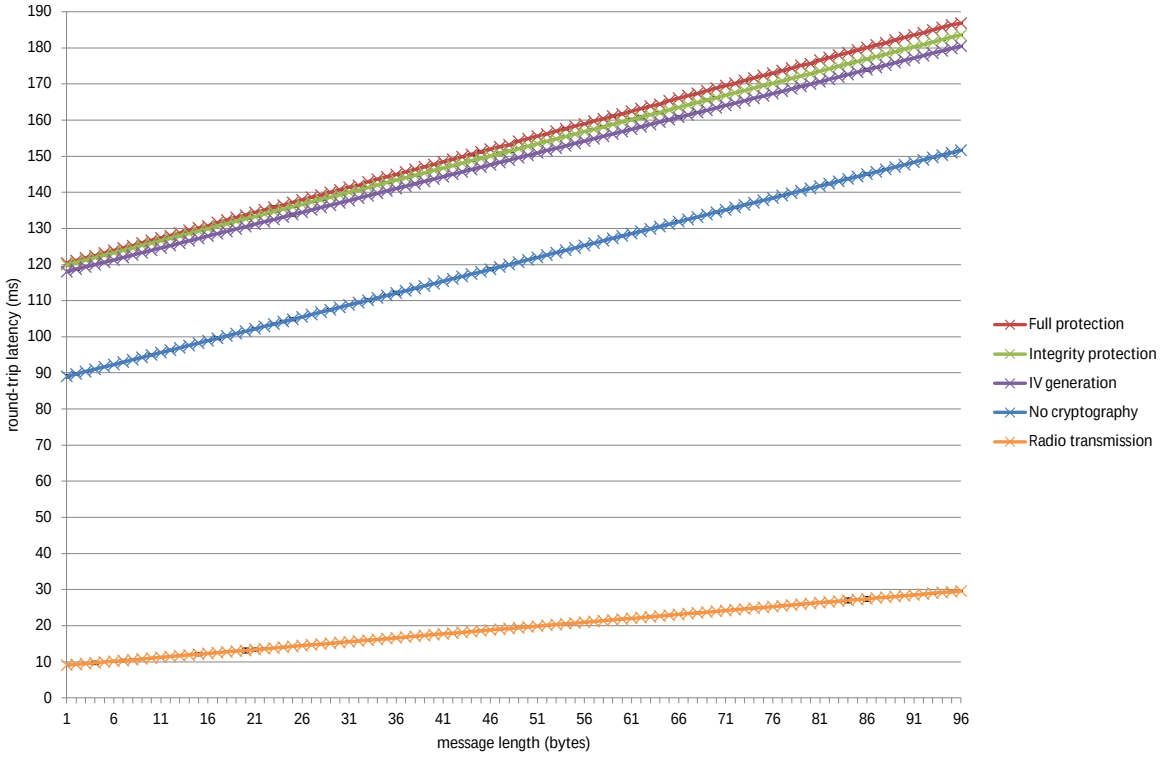


(a) Full results

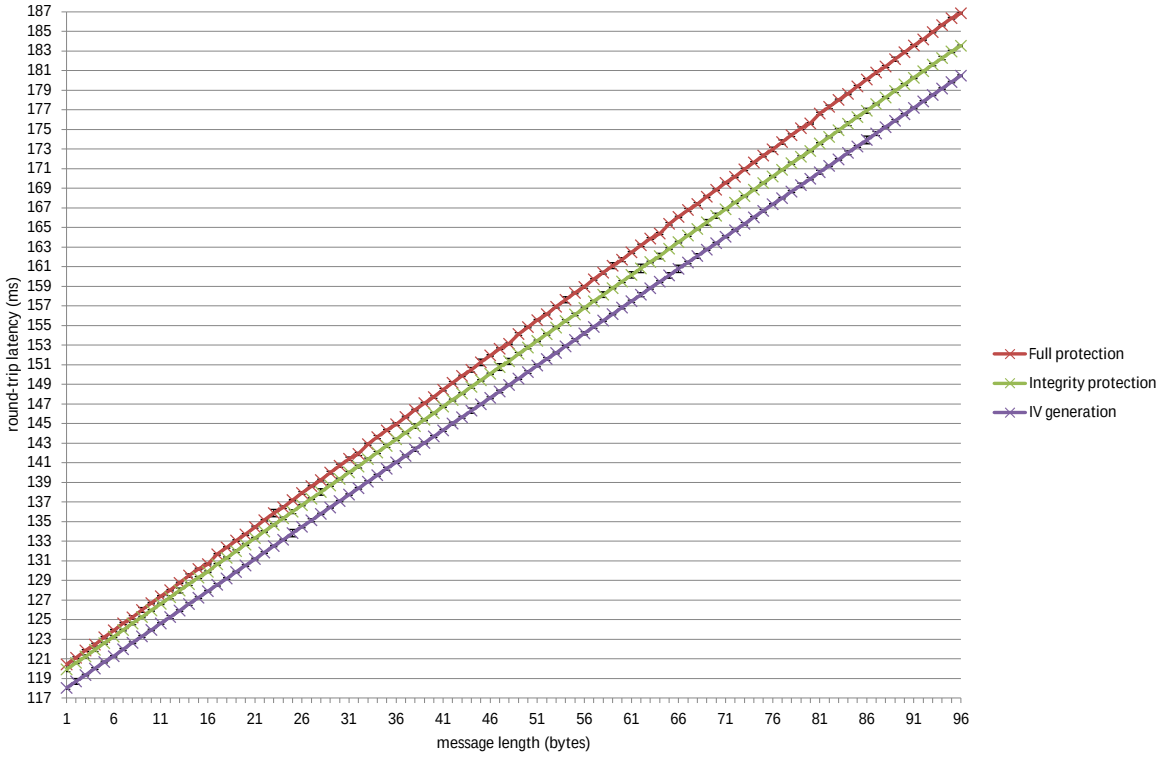


(b) False origin, with 'no cryptography' results omitted

Figure 8.8: StarfishNet round-trip latency vs message length, between nodes with 4 intervening routers. Colour indicates level of cryptographic protection.

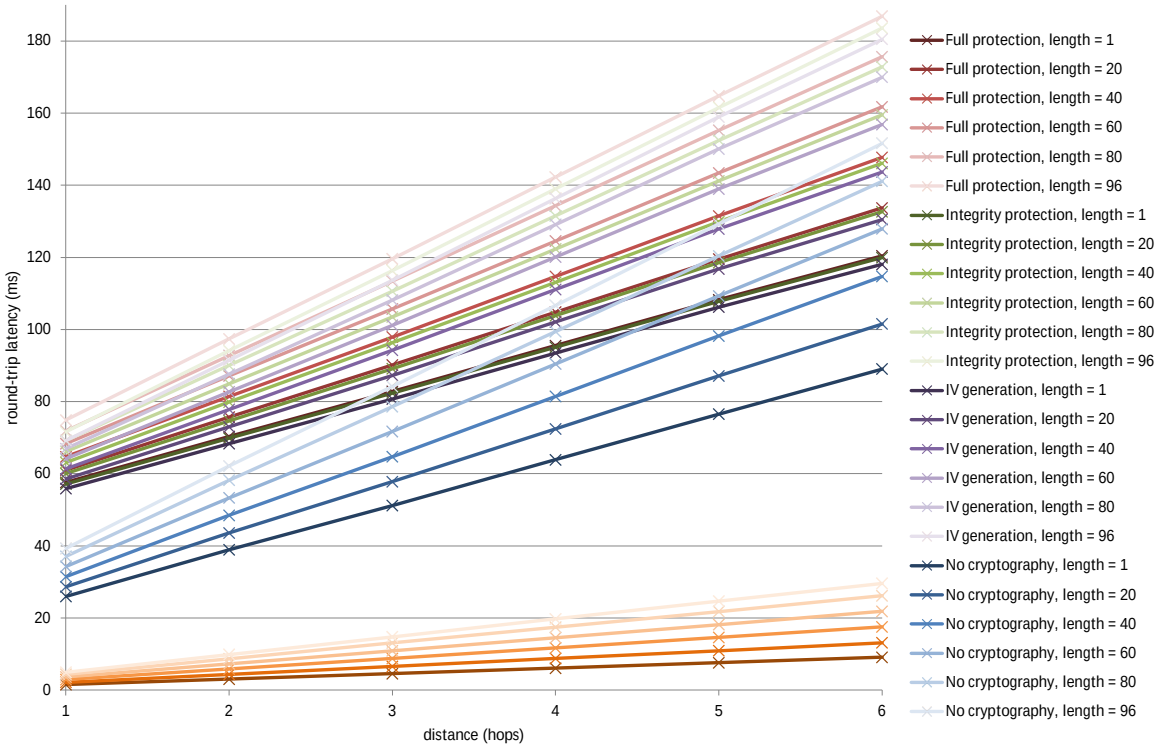


(a) Full results

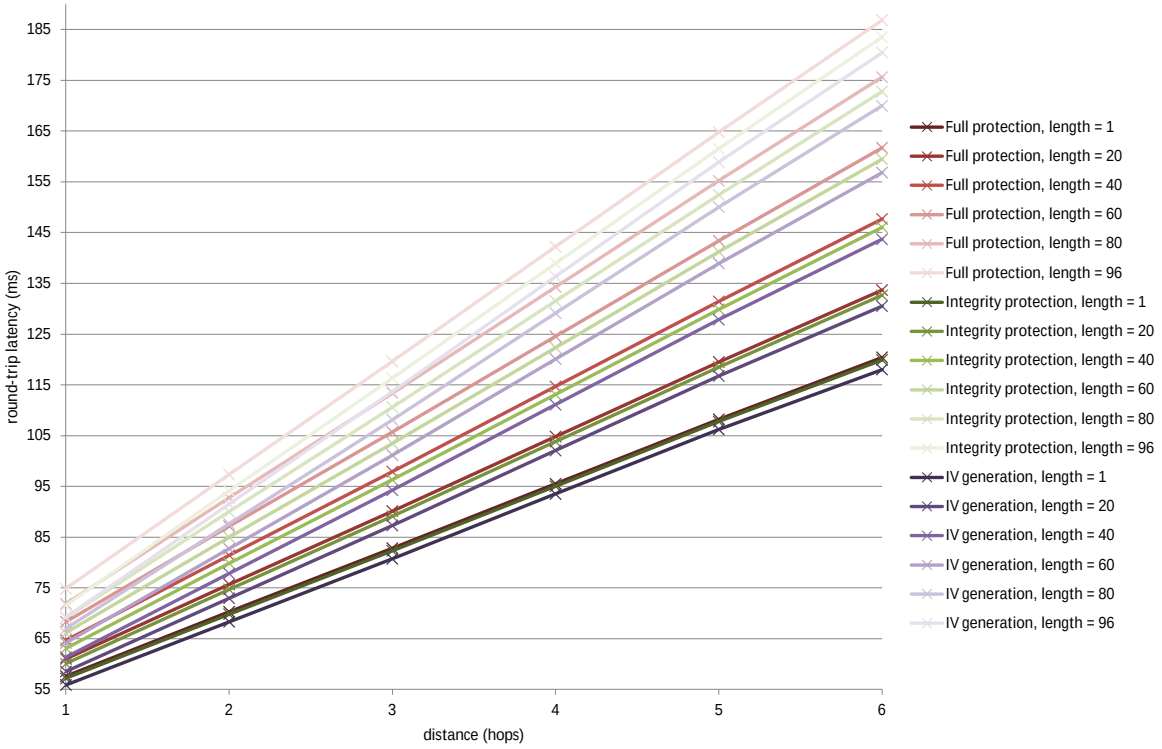


(b) False origin, with 'no cryptography' results omitted

Figure 8.9: StarfishNet round-trip latency vs message length, between nodes with 5 intervening routers. Colour indicates level of cryptographic protection.



(a) Full results



(b) False origin, with 'no cryptography' results omitted

Figure 8.10: StarfishNet round-trip latency vs routing distance. Colour indicates level of cryptographic protection, brightness indicates message length.

96-byte message). As a fraction of the total, if SHA1 were eliminated from IV generation, this represents between 6.6% (1-byte message) and 14.3% (96-byte message) of round-trip latency between adjacent nodes (and less between non-adjacent nodes).

Association, and the cost of ECC

Most of this section has been devoted to the performance of data transport in StarfishNet. However, one of the limiting factors on the utility of StarfishNet is, in fact, the computational load imposed by *association* transactions. As the ECC benchmarks in Table 8.2 indicate, this is substantial in the current prototype. Each association transaction requires six ECC operations: two signatures, two signature verifications, and two ECDH operations. (It also requires two key generations, for the ephemeral ECDH key pairs, but these can be performed outside the association 'critical path'.) Worse, they must occur sequentially. In our prototype, an association transaction requires nearly 8 minutes of computational work; starting up a network of seven nodes, such as that used in our experiments, can take up to one hour. Apart from its energy consumption, this has serious usability implications.

However, Section 2.1 shows that our results on this point are *not* indicative of what is possible on this platform: on a very similar SoC, the CC1010, ECC operations with the same parameters took 4.5 seconds, an order of magnitude faster than our results. Furthermore, we note that the core clock speed of the CC1010 is 14.7456MHz, with each instruction taking four clock cycles; that is, an instruction rate of 3.6864MHz. By contrast, our hardware, the CC2530, is a single-cycle core – that is, each instruction takes one clock cycle – with a core clock speed of 32.768MHz. This corresponds to an increase in performance of roughly an order of magnitude from the CC1010.

In other words, based on the information in Section 2.1, it is definitely possible to perform ECC operations in 4.5 seconds (assuming parity with the CC1010), and very likely in 0.5 seconds (based on the performance differences between the SoCs). Under these conditions, ECC is still far more costly than AES operations, both in time and energy (roughly 4.5s/120mJ in the former case and 0.5s/15mJ in the latter; compare to 92ms/2.5mJ for software AES, and 1.4ms/38µJ for hardware AES).

8.3.3 Memory consumption

Table 8.4 shows the memory consumption of the StarfishNet prototype on the CC2530, which provides 8KiB of RAM and 256KiB of ROM. These reports are broken down by subsystem. Table 8.4 also lists the size, in lines of code, of the subsystem in question.

All resource allocations are done statically on this platform, except in two subsystems: the neighbour table requires an extra 8 bytes during a network join, and µECC allocates a substantial amount of intermediate data during ECC operations. Both of these are allocated from a 1024-byte heap that forms part of SDCC's C runtime environment; this heap is itself statically allocated in RAM. (Neither this heap, nor the stack, are included in Table 8.4.)

Table 8.4 makes several effects abundantly clear. First, there is a visible issue with either the way in which the StarfishNet prototype is written, or with SDCC: the

Subsystem	Code size	Data size	Source size
µECC	25804 B	159 B	1995 LoC
SHA1	5889 B	344 B	117 LoC
AES	2344 B	0 B	225 LoC
Other crypto	4467 B	342 B	280 LoC
Neighbour table	4387 B	1096 B	264 LoC
Retransmission	784 B	2023 B	392 LoC
Other StarfishNet	52546 B	611 B	2148 LoC
Other Contiki	39333 B	1932 B	7442 LoC

Table 8.4: StarfishNet ROM and RAM consumption, and source code size, broken down by subsystem. RAM figures do not include stack or heap space.

StarfishNet codebase is only 2kLoC long, but when compiled requires over 52kiB of ROM. Again, efforts towards optimisation and/or use of a higher-quality optimising compiler would be likely to reduce this resource requirement. In a similar vein, hardware implementations of ECC and SHA1/SHA2 would largely reduce the ROM consumption of those two modules to roughly the level of the AES module (which itself could be further optimised).

RAM consumption is primarily due to the neighbour table, the table containing the state of all security associations established by the node, and the retransmission buffer, which stores all packets sent that have not yet been acknowledged; each was 8 entries long in these tests. For the former, the change to the IV generation scheme suggested in Section 8.1 would indeed help: using long-term public keys to generate packet IVs would permit the memory containing ephemeral ECDH keys to be reclaimed once association is complete. This represents 62 bytes per entry, nearly half of its total memory consumption. Additionally, some of the information in this table (such as public signing keys) could be stored in ROM, yielding further savings.

Reducing the memory consumption of the retransmission buffer is more challenging. This is another optimisation task for future engineering.

Nonetheless, these results are relatively promising: while the resource demands of StarfishNet are by no means minimal, the size of the codebase suggests that they can be reduced with standard embedded systems engineering work.

8.4 Evaluation against IoT requirements

8.4.1 R.1: decentralised architecture

StarfishNet is based on a three-message authenticated Diffie-Hellman protocol, to establish a link key between any pair of communicating nodes. That link key is stored only at the nodes communicating, and all data messages are required to be secured with it. There are therefore no trusted third parties, and in fact no parties at all, able to discern the key or the data it protects, provided the key is not actively

disclosed, neither machine is compromised, and the relevant cryptographic primitives remain unbroken.

StarfishNet does have a coordinator. However, that coordinator only acts as the root as the routing tree. It is capable of denying service, by revoking the addresses it has allocated to its children, but is otherwise incapable of meaningfully affecting the functioning of the network. Moreover, we would expect a real deployment to use an addressing scheme akin to ZigBee stochastic addressing, and a routing algorithm akin to ZigBee mesh routing, both of which eliminate entirely the central point of control.

This requirement is therefore satisfied.

8.4.2 R.2: resistance to strong adversaries

The StarfishNet association protocol is resists attack by a Dolev-Yao attacker; our results with Tamarin demonstrate this. At the end of a successful association transaction, each node is guaranteed that the node with which it has associated possesses the private counterpart to the public key it presented.

Separately, information must be established about the properties of that public key, potentially including some external name or identity with which it is associated. StarfishNet's certificate transport functionality is designed to assist with this second issue. The intention behind their design is the use of attribute certificates, but this is not required; the transport primitives are generic enough to support a variety of cryptographic assertions. Determining which authorities to trust to make such assertions, and what rights are granted by the facts thereby asserted, is necessarily the responsibility of applications built atop StarfishNet, and the developers thereof.

This requirement is therefore satisfied.

8.4.3 R.3: performance

This requirement is a complex and multifaceted one on embedded systems. It includes a requirement that security operations fit within the power and storage envelopes available, but also limits the amount of permissible overheads even within those envelopes. It also implies the presence of particular features. We will consider all of these.

First, the cost of StarfishNet's packet security is low; as a fraction of total latency, making a few unfavourable assumptions, and for the longest possible message, the worst case is 14.3%. One of those assumptions is the removal of SHA1 from the packet IV generation algorithm, a requirement for any real-world deployment. Finally, we note that the DMA capabilities of the AES coprocessor are not used in Contiki, inhibiting the CPU from performing other work while an encryption operation is being performed. Using these capabilities may lead to further energy efficiency gains.

The cost of associations, however, is a different story. Associations involve substantial numbers of ECC operations. The experiments presented in this chapter make it clear that ECC performance on our testbed is poor, severely limiting the number of associations that can be performed by any given node. However, previous results

involving software implementations of ECC indicate that our results are very much an outlier. Performance gains of at least an order of magnitude should be possible; gains of two orders of magnitude are likely. Even if this proves not to be the case, the Texas Instruments CC2538 – the successor to our testbed hardware, the CC2530 – provides a hardware implementation of ECC, which is likely to perform better still.

StarfishNet also provides the ability for computationally constrained nodes to delegate the responsibility for performing ECC operations to another, more powerful, node (in fact, using the attribute certificate mechanism). In th event that the ECC performance gains mentioned above are not realised, this represents a possible alternative.

StarfishNet packet headers were carefully designed to consume as little space as possible. Two particular examples are the length of StarfishNet addresses, and the design of the association protocol. StarfishNet addresses were chosen to be 16 bits wide, in order to take advantage of IEEE 802.15.4 short addresses, reducing the size of link-layer headers. Carrying the identities of communicating parties in plaintext allowed the third message in the authenticated DH exchange to be very small, increasing the amount of application data that it can transport.

Finally, StarfishNet was deliberately engineered to conserve power where possible. The network join and association protocols – both the same protocol, in fact – are simple, requiring only three packet transmissions each (in sharp contrast to ZigBee, which requires at least 8). Acknowledgement stapling also contributes to this, by eliminating the transmission of separate acknowledgement packets under most circumstances, also in sharp contrast to ZigBee, which requires a separate acknowledgement for every data packet. Our results show that the energy cost of acknowledgement stapling – that is, adding four bytes to a packet – is *far* lower than transmitting a separate acknowledgement packet.

We note one major caveat: StarfishNet is very much a research prototype. Engineering, and a small number of protocol changes, are required to reduce the RAM and ROM requirements of the network stack. The retransmission policy is also unoptimised, and represents a potential source of spurious packet transmissions, and thus a waste of energy. The DMA capabilities of both the radio and AES coprocessors are entirely unused, and represent a great source of both performance gains and power savings. However, these represent challenges of engineering, and all are surmountable.

In summary, we consider this requirement to have been satisfied.

8.4.4 R.4: unrestricted communication

StarfishNet implements a ZigBee-like tree routing scheme. Although this scheme is not ideal, with a non-optimal routing algorithm with no capacity to adapt to router failures, and some central management, we note that this scheme is a simple one designed for testing and development purposes. We would expect a real deployment to use an addressing scheme akin to ZigBee stochastic addressing, and a routing algorithm akin to ZigBee mesh routing, both of which eliminate entirely both the central point of control and the inability to adapt to router failures.

System	R.1	R.2	R.3	R.4
StarfishNet	Satisfied	Satisfied	Satisfied	Satisfied

Table 8.5: Application of requirements from Section 4.2 to BottleCap.

Otherwise, StarfishNet imposes no restrictions on communication patterns by design. Any machine can communicate with any other, at any time. StarfishNet only requires that they both have valid short addresses (that is, they have both joined the network), and perform an association before exchanging data.

In particular, Figure 8.10 demonstrates the behaviour of StarfishNet as the number of intervening routers increases, for small numbers of routers. It demonstrates another consequence of the end-to-end design of StarfishNet cryptographic protections: the trendlines themselves indicate that the amount of work performed by each router for packets they route is small, with no cryptographic work required at all.

StarfishNet even permits interoperability with IP networks, provided there is a StarfishNet node that can act as an Internet gateway. The exception is linking two StarfishNet networks, which cannot be done easily using the mechanisms described in this thesis.

Despite this exception, we consider this requirement to have been satisfied.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

The Internet of Things presents a novel threat environment. Connecting embedded systems to the Internet provides the flexibility of the Internet, but exposes also them to the threats of the Internet. Additionally, the applications envisaged by the IoT place heavy emphasis on ephemeral interactions, which present their own security challenges. Chapter 4 examined these issues, presenting a more extensive IoT threat model than currently exists in the literature (a contribution in its own right), which motivated for the rest of the thesis. It also synthesised a set of broad requirements for IoT security systems.

Of those requirements, that for a decentralised architecture is key. BottleCap is a capability-based access control system, designed to provide an exemplar for such an architecture, and the second contribution in this thesis. It requires a trusted execution environment with remotely verifiable properties (the example used in the prototype is that provided by TXT and Flicker). This TEE exports an encrypted capability container, or *bottle* with particular properties: each slot contains a single capability, whose *contents* are managed by the capability's issuer. Managing the space in a bottle, however, is the responsibility of the *client*; it may have capabilities from any number of issuers. BottleCap transactions take place entirely between the client and service provider; no trusted third parties need be involved, other than those required to establish trust in the TEE. While the TC basis of the BottleCap prototype places it firmly outside the IoT domain, there is no reason it could not be ported to a more suitable environment should one become available. The announcement by ARM that TrustZone will be supported on future microcontroller-class processors suggests that TrustZone based TEEs are particularly attractive for this use. Such a CPU forms the processing core of the CC2538, the successor to the CC2530 we used to prototype StarfishNet.

BottleCap does not attempt to provide or establish a secure channel; it assumes one already exists. StarfishNet, our third contribution, is a network-layer protocol for IoT networks, and addresses this need. StarfishNet implements a highly decentralised network architecture, the *perimeterless network*: the network perimeter is

uncontrolled, with no node providing admission control services, but equally, the act of crossing that perimeter – that is, joining the network – convers no rights. Nodes establish secure channels using elliptic-curve cryptography in a pairwise manner, ensuring that the only two nodes possessing their shared key are the nodes using it to communicate. StarfishNet is also carefully designed to consume as few resources – whether storage, computation time, or network bandwidth – as possible. (This is another reason for the end-to-end design of StarfishNet's protections: no re-encryption is ever necessary.) While our prototype does not necessarily meet that goal in its current form for a number of reasons – its codebase requires a large amount of code storage, ECC operations are extremely slow, and hashing primitives are inappropriately used, among others – all can be resolved with some engineering effort. Certainly, as an architectural and design exemplar, it is a success.

In short, this thesis presented two exemplars for decentralised architecture of network security systems. While each had its shortcomings, these were of an engineering, rather than a principled nature. It also indicates the value of empirical data: both BottleCap and StarfishNet demonstrated critical performance issues that were easily visible in experiment, but were not necessarily obvious during design. For BottleCap, this was chiefly the time required to invoke the Flicker TEE; for StarfishNet, the fact that the IV generation scheme uses SHA1 – which is both unnecessary, and performs poorly when implemented in software – and the ephemeral, rather than long-term, key – resulting in a large amount of unnecessary storage overhead. Potential performance issues resulting from the interaction between the coding style used and SDCC may also have contributed substantially.

9.2 Future work

Future work on these issues can take a variety of directions. Most obvious is the engineering work required to develop the BottleCap and StarfishNet prototypes into viable real-world systems. Much of what needs to be done towards this end has already been discussed in Chapter 6 and Chapter 8.

Availability of tools is another line of engineering work worth pursuing. Particularly for StarfishNet, development and experimentation were impeded by the lack of good tools for the Intel 8051 architecture. The SDCC compiler performs only basic optimisation compared to, for example, the freely available Intel or GNU C compilers (neither of which targets this architecture); the only high-quality compilers for the 8051 are commercial products. In addition, while the CC2530 supports advanced debugging capabilities, the manufacturer only supplies tools for using them as an extension to one such commercial compiler.

Future research based on this work could take a variety of directions. Hardware attacks are often ignored in protocol work of the kind we have done. However, in reality, they cannot be. Extending these, or similar, architectures to deal with this issue is an open problem (although BottleCap does benefit from recent work in Trusted Computing in this regard).

In a related vein, although TC has a sophisticated model for establishing trust

in a remote platform, mobile TEEs as yet do not. Trustonic[1] claims to have such a mechanism, but its details are not public, making it difficult to assess this claim, or the constraints thereon. Furthermore, since mobile TEEs are largely based on ARM TrustZone, a CPU-based isolation mechanism with no underlying cryptography, it is unclear even how the hardware primitives available on mobile platforms could be wielded to accomplish such a goal. This is definitely an avenue for further research.

StarfishNet also does not attempt to address denial of service attacks; in fact, its cryptographic protections have the effect of reducing other forms of attack to a denial of service. In addition, the IoT domain presents unique types examples of DoS attack, such as the saturation of slow IEEE 802.15.4 links, battery-draining attacks, or routing-based attacks. Defending against all of these are open problems; defence against routing attacks, such as presented in [78], is of particular relevance.

StarfishNet's delegation feature, in which one node can delgate ECC operations to another, more powerful node is useful on hardware on which the capacity for performing ECC is extremely limited. This technique is not unique to StarfishNet; for example, a similar one for DTLS [67] is described in Section 3.3.9. However, in both the DTLS and StarfishNet instantiations, the target – that is, the powerful node – is entirely trusted by the delegator. Delegation protocols or techniques which impose a lower trust burden than this appear to be an unexplored area of research.

Secure service discovery on an untrusted network, like an IoT network, is also an open problem. Establishing a secure channel is a necessary component, as is strong, flexible access control. However, the ability to establish trust in a remote party – that is, ensure that the service being offered is available and being provided honestly – is equally critical to many of the domains to which IoT technologies are being applied. An aggressively decentralised architecture, such as that of StarfishNet, may exacerbate this problem; this is definitely an area where more work is needed.

Finally, many of the design choices made in both StarfishNet and BottleCap have privacy implications. StarfishNet does not attempt to conceal the identities of associating parties. This decision was made for engineering reasons: to minimise the amount of association overhead in packets capable of carrying application data. As a result, any entity that can observe an association transaction will know the identities of the parties to that association. This may have deleterious consequences in privacy-sensitive applications. On the other hand, a BottleCap ticket only identifies the *issuer* of the capability backing it; the holder is private. The privacy achieved with respect to that issuer is variable; it depends on the granularity with which issuer keys are used by the capability issuer. The privacy achieved with respect to a third party, however, is much greater: unless that third party has some knowledge of the meaning of issuer key hashes in this context, they learn nothing. In general, the privacy implications of this class of engineering decisions is another open area of research.

[1] https://www.trustonic.com/technology/trustzone-and-tee

9.3 Final remarks

The design of network-based security systems is a challenging problem, and one that is highly sensitive to the context in which those systems are being applied, and the constraints imposed by that context. Open systems, such as the Internet, pose a particular challenge: they essentially represent a hostile networking environment, in which even the infrastructure cannot be trusted. This is equally true on the Internet of Things, whose focus on constrained devices, wireless networks, and ephemeral interactions only magnify the challenge.

This thesis explores one principle for addressing that challenge: building systems according to an aggressively decentralised architecture. Centralised designs can offer advantages in performance, in that they often obviate any need for computationally demanding cryptographic operations, or the computational and storage resources to enforce access control policy *in situ*. However, they expose the system to attack from the party controlling the centralised infrastructure, and are limited in their ability to meet the scalability challenges of the Internet. Decentralised designs clearly address the first issue, on which this thesis focuses; they may help address the second as well, a promising avenue for future research.

Finally, BottleCap and StarfishNet are also designed with a view towards interoperability: both can easily coexist, communicate with, and integrate into existing infrastructure. Rather than requiring changes to core Internet infrastructure, both only require support at the relevant endpoints or subnetworks. In a sense, this ability for different solutions to coexist is the hallmark of Internet design, one which carries over to the Internet of Things.

Bibliography

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the USENIX Conference*, pages 93–112. USENIX Association, 1986.

[2] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming Guide*.

[3] Cristina Alcaraz and Javier Lopez. A Security Analysis for Wireless Sensor Mesh Networks in Highly Critical Systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 40(4):419–428, jul 2010.

[4] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[5] M. Anderson, Ronald D. Pose, and Chris S. Wallace. A Password-Capability System. *The Computer Journal*, 29(1):1–8, 1986.

[6] Ross Anderson, Haowen Chan, and Adrian Perrig. Key infection: smart trust for smart dust. In *Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP 2004)*, pages 206–215. IEEE, 2004.

[7] Apple, Inc. Apple Filing Protocol Programming Guide. https://developer.apple.com/library/mac/documentation/Networking/Conceptual/AFP/Introduction/Introduction.html.

[8] Apple Inc. HFS Plus Volume Format. Technical Note TT1150.

[9] Diego F. Aranha, Ricardo Dahab, Julio López, and Leonardo B. Oliveira. Efficient implementation of elliptic curve cryptography in wireless sensors. *Advances in Mathematics of Communications*, 4(2):169–187, 2010.

[10] William Arbaugh. Wireless security is different. *Computer*, 36(8):99–101, 2003.

[11] ARM Holdings. *ARM Architecture Reference Manual*.

[12] Ahmad Atamli and Andrew P. Martin. Threat-Based Security Analysis for the Internet of Things. In *2014 International Workshop on Secure Internet of Things (SIoT)*, pages 35–43. IEEE, 2014.

[13] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[14] Sachin Babar, Parikshit Mahalle, Antonietta Stango, Neeli Prasad, and Ramjee Prasad. Proposed Security Model and Threat Taxonomy for the Internet of Things (IoT). In *Recent Trends in Network Security and Applications*, volume 89 of *Communications in Computer and Information Science*, pages 420–429. Springer Berlin / Heidelberg, 2010.

[15] Paolo Baronti, Prashant Pillai, Vince W. C. Chook, Stefano Chessa, Alberto Gotta, and Y. Fun Hu. Wireless sensor networks: A survey on the state of the art and the 802.15.4 and ZigBee standards. *Computer Communications*, 30(7):1655–1695, 2007.

[16] A. Bartoli, J. Hernandez-Serrano, M. Soriano, M. Dohler, A. Kountouris, and D. Barthel. Secure Lossless Aggregation for Smart Grid M2M Networks. In *Proceedings of the IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 333–338. IEEE, 2010.

[17] Rolf Blom. An Optimal Class of Symmetric Key Generation Systems. In *Advances in Cryptology*, volume 209 of *Lecture Notes in Computer Science*, pages 335–338. Springer Berlin / Heidelberg, 1985.

[18] Bluetooth Special Interest Group. The Bluetooth Specification.

[19] Gilad Bracha, Peter von der Ahe, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as Objects in Newspeak. In *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 405–428. Springer-Verlag, 2010.

[20] C. Buckl, S. Sommer, A. Scholz, A. Knoll, A. Kemper, J. Heuer, and A. Schmitt. Services to the Field: An Approach for Resource Constrained Sensor/Actor Networks. In *International Conference on Advanced Information Networking and Applications Workshops, 2009 (WAINA '09)*, pages 476–481, 2009.

[21] Sven Bugiel and Jan-Erik Ekberg. Implementing an application-specific credential platform using late-launched mobile trusted module. In *Proceedings of the Fifth ACM Workshop on Scalable Trusted Computing (STC'10)*, pages 21–30. ACM, 2010.

[22] Liang Cai, Sridhar Machiraju, and Hao Chen. CapAuth: A Capability-based Handover Scheme. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM) 2010*, pages 1–5. IEEE, 2010.

[23] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. RFC 1813 (Informational), June 1995.

[24] Justin Cappos, Armon Dadgar, Jeff Rasley, Justin Samuel, Ivan Beschastnikh, Cosmin Barsan, Arvind Krishnamurthy, and Thomas Anderson. Retaining sandbox containment despite bugs in privileged memory-safe code. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS'10)*, CCS'10, pages 212–223. ACM, 2010.

[25] David Chadwick. Authorisation in Grid computing. *Information Security Technical Report*, 10(1):33–40, 2005.

[26] David Chadwick, Alexander Otenko, and Edward Ball. Role-based access control with X.509 attribute certificates. *IEEE Internet Computing*, 7(2):62–69, 2003.

[27] Haowen Chan, Adrian Perrig, and Dawn Song. Random key predistribution schemes for sensor networks. In *Proceedings of the IEEE 2003 Symposium on Security and Privacy*, pages 197–213. IEEE Computer Society, 2003.

[28] Thomas M. Chen and Stephen S. Liu. A model and evaluation of distributed network management approaches. *IEEE Journal on Selected Areas in Communications*, 20(4):850–857, 2002.

[29] Andrew Cooper. *Towards a trusted grid architecture*. PhD thesis, University of Oxford, 2010.

[30] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[31] Gianluca Dini and Marco Tiloca. Considerations on Security in ZigBee Networks. In *Proceedings of the 2010 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'10)*, pages 58–65. IEEE, 2010.

[32] D. Dolev and A. C. Yao. On the security of public key protocols. In *22nd Annual Symposium on Foundations of Computer Science, 1981. (SFCS '81)*, pages 350–357, 1981.

[33] Wenliang Du, Jing Deng, Yunghsiang S. Han, Pramod K. Varshney, Jonathan Katz, and Aram Khalili. A Pairwise Key Predistribution Scheme for Wireless Sensor Networks. *ACM Transactions on Information and System Security*, 8(2):228–258, 2005.

[34] Simon Duquennoy, Gilles Grimaud, and Jean-Jacques Vandewalle. The Web of Things: Interconnecting Devices with High Usability and Performance. In *International Conference on Embedded Software and Systems, 2009. ICESS '09*, pages 323–330, 2009.

[35] L. Dusseault. HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV). RFC 4918 (Proposed Standard), June 2007. Updated by RFC 5689.

[36] Morris Dworkin. SP 800-38C Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality. NIST Special Publication 800-38C, NIST, 2004.

[37] C. Ellison. SPKI Requirements. RFC 2692 (Experimental), September 1999.

[38] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. RFC 2693 (Experimental), September 1999.

[39] Laurent Eschenauer and Virgil D. Gligor. A Key-management Scheme for Distributed Sensor Networks. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 41–47. ACM, 2002.

[40] David F. Ferraiolo, Janet A. Cugini, and D. Richard Kuhn. Role-based access control (RBAC): Features and motivations. In *Proceedings of 11th annual computer security application conference*, pages 241–248, 1995.

[41] Philip Fong. Discretionary capability confinement. *International Journal of Information Security*, 7(2):7–154, 2008.

[42] D. Forsberg, Y. Ohba, B. Patil, H. Tschofenig, and A. Yegin. Protocol for Carrying Authentication for Network Access (PANA). RFC 5191 (Proposed Standard), May 2008. Updated by RFC 5872.

[43] Ian Foster. The Globus Toolkit for Grid Computing. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, CCGRID '01, page 2. IEEE Computer Society, 2001.

[44] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[45] Christian Fuhrhop, John Lyle, and Shamal Faily. The webinos project. In *Proceedings of the 21st international conference companion on World Wide Web (WWW'12 Companion)*, pages 259–262. ACM, 2012.

[46] Neil Gershenfeld, Raffi Krikorian, and Danny Cohen. The Internet of Things. *Scientific American*, 291(4):76–81, 2004.

[47] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the eighth ACM international conference on Architectural support for programming languages and operating systems (ASPLOS VIII)*, pages 92–103. ACM, 1998.

[48] Johann Großschädl. TinySA: A Security Architecture for Wireless Sensor Networks. In *Proceedings of the 2006 ACM CoNEXT Conference*, CoNEXT '06, pages 55:1–55:2. ACM, 2006.

[49] D. A. Grove, T. C. Murray, C. A. Owen, C. J. North, J. A. Jones, M. R. Beaumont, and B. D. Hopkins. An Overview of the Annex System. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, pages 341–352. IEEE, 2007.

[50] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[51] Dominique Guinard, Vlad Trifa, and Erik Wilde. A Resource Oriented Architecture for the Web of Things. In *Proceedings of the 2010 IEEE Internet of Things Conference (IoT 2010)*, pages 1–8, 2010.

[52] Vipul Gupta, Matthew Millard, Stephen Fung, Yu Zhu, Nils Gura, Hans Eberle, and Sheueling Chang Shantz. Sizzle: A Standards-Based End-to-End Security Architecture for the Embedded Internet. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications*, pages 247–256. IEEE, 2005.

[53] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-Bit CPUs. In *Cryptographic Hardware and Embedded Systems (CHES) 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer Berlin / Heidelberg, 2004.

[54] Sergio Gusmeroli, Salvatore Piccione, and Domenico Rotondi. A capability-based security approach to manage access control in the Internet of Things. *Mathematical and Computer Modelling*, 58(5):1189–1205, 2013.

[55] J. C. Haartsen. The Bluetooth radio system. *IEEE Personal Communications*, 7(1):28–36, 2000.

[56] C. T. Hager and S. F. Midkiff. An analysis of Bluetooth security vulnerabilities. In *IEEE Wireless Communications and Networking*, volume 3, pages 1825–1831, 2003.

[57] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard), October 2012.

[58] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471, 1976.

[59] HART Communication Foundation. WirelessHART Device Specification. Document HCF_SPEC-290, 2008.

[60] T. Haynes and D. Noveck. Network File System (NFS) Version 4 Protocol. RFC 7530 (Proposed Standard), March 2015.

[61] Tobias Heer, Oscar Garcia-Morchon, René Hummen, Sye Loong Keoh, Sandeep S. Kumar, and Klaus Wehrle. Security Challenges in the IP-based Internet of Things. *Wireless Personal Communications*, 61(3):527–542, 2011.

[62] Gernot Heiser and Ben Leslie. The OKL4 microvisor: convergence point of microkernels and hypervisors. In *Proceedings of the first ACM Asia-Pacific Workshop on Systems (APSys'10)*, pages 19–24. ACM, 2010.

[63] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 11:1–11:1. ACM, 2013.

[64] Vincent C. Hu, D. Richard Kuhn, David F. Ferraiolo, and Jeffrey Voas. Attribute-Based Access Control. *Computer*, 48(2):85–88, 2015.

[65] Qiang Huang, Johnas Cukier, Hisashi Kobayashi, Bede Liu, and Jinyun Zhang. Fast authenticated key establishment protocols for self-organizing sensor networks. In *Proceedings of the ACM international conference on Wireless sensor networks and applications (WSNA)*, page 141. ACM Press, 2003.

[66] J. Hui and P. Thubert. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. RFC 6282 (Proposed Standard), September 2011.

[67] René Hummen, Hossein Shafagh, Shahid Raza, Thiemo Voigt, and Klaus Wehrle. Delegation-based Authentication and Authorization for the IP-based Internet of Things. In *Proceedings of the 11th IEEE International Conference on Sensing, Communication, and Networking (SECON '14)*, pages 284–292, 2014.

[68] Michael Hutter and Peter Schwabe. NaCl on 8-Bit AVR Microcontrollers. In *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 156–172. Springer Berlin / Heidelberg, 2013.

[69] Ondrej Hyncica, Peter Kacz, Petr Fiedler, Zdenek Bradac, Pavel Kucera, and Radimir Vrba. On Security of PAN Wireless Systems. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 4017 of *Lecture Notes in Computer Science*, pages 178–185. Springer Berlin / Heidelberg, 2006.

[70] IEEE. IEEE Standard for Local and metropolitan area networks, Part 15.4: Low-Rate Wireless Personal Area Networks. *IEEE Std 802.15.4-2003*, 2003.

[71] IEEE. IEEE Standard for Local and metropolitan area networks – Port-Based Network Access Control. *IEEE Std 802.1X-2010*, 2010.

[72] Intel Corporation. *Trusted eXecution Technology (TXT) – Measured Launched Environment Developer's Guide.*

[73] International Society for Automation. Wireless systems for industrial automation: Process control and related applications. *ISA Std ANSI/ISA-100.11a-2011*, 2011.

[74] V. Jacobson. Compressing TCP/IP Headers for Low-Speed Serial Links. RFC 1144 (Proposed Standard), February 1990.

[75] P. Jokela, R. Moskowitz, and J. Melen. Using the Encapsulating Security Payload (ESP) Transport Format with the Host Identity Protocol (HIP). RFC 7402 (Proposed Standard), April 2015.

[76] Gaurav Jolly, Mustafa C. Kuşçu, Pallavi Kokate, and Mohamed Younis. A low-energy key management protocol for wireless sensor networks. In *Proceedings of the Eighth IEEE International Symposium on Computers and Communication (ISCC 2003)*, volume 1, pages 335–340, 2003.

[77] Chris Karlof, Naveen Sastry, and David Wagner. TinySec: A Link Layer Security Architecture for Wireless Sensor Networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 162–175. ACM, 2004.

[78] Chris Karlof and David Wagner. Secure routing in wireless sensor networks: attacks and countermeasures. *Ad Hoc Networks*, 1(2–3):293–315, 2003.

[79] C. Kaufman. Internet Key Exchange (IKEv2) Protocol. RFC 4306 (Proposed Standard), December 2005. Obsoleted by RFC 5996, updated by RFC 5282.

[80] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296 (INTERNET STANDARD), October 2014. Updated by RFCs 7427, 7670.

[81] S. Kent. IP Authentication Header. RFC 4302 (Proposed Standard), December 2005.

[82] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303 (Proposed Standard), December 2005.

[83] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005. Updated by RFCs 6040, 7619.

[84] Justin King-Lacroix. Position Paper: Can the Web Really Use Secure Hardware? In *Proceedings of the Workshop on Web Applications and Secure Hardware (WASH'13)*, volume 1011 of *CEUR Workshop Proceedings*, pages 17–22, 2013.

[85] Justin King-Lacroix and Andrew P. Martin. BottleCap: a Credential Manager for Capability Systems. In *Proceedings of the seventh ACM workshop on Scalable Trusted Computing (STC'12)*, pages 45–54. ACM, 2012.

[86] Justin King-Lacroix and Andrew P. Martin. KEDS: Decentralised Network Security for the Smart Home Environment. In *Second Internation Workshop on Smart Grid Security (SmartGridSec) 2014*, volume 8448 of *Lecture Notes in Computer Science*, pages 63–78. Springer International Publishing, 2014.

[87] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*, pages 207–220. ACM, 2009.

[88] Daniel Kouřil and Jim Basney. A Credential Renewal Service for Long-Running Jobs. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 63–68. IEEE Computer Society, 2005.

[89] Hugo Krawczyk. SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE Protocols. In *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 400–425. Springer Berlin / Heidelberg, 2003.

[90] Adam Lackorzynski and Alexander Warg. Taming subsystems: capabilities as universal resource access control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems (IIES'09)*, pages 25–30. ACM, 2009.

[91] Donald C. Latham. Department of Defense trusted computer system evaluation criteria. Department of Defense Standard DoD 5200.28-STD, 1985.

[92] Johnson C. Lee, Victor C. M. Leung, Kirk H. Wong, and Henry C. B. Chan. Key management issues in wireless sensor networks: current proposals and future developments. *IEEE Wireless Communications Magazine*, 14(5):76–84, 2007.

[93] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An Operating System for Sensor Networks. In *Ambient Intelligence*, pages 115–148. Springer Berlin / Heidelberg, 2005.

[94] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[95] Hongwei Li, Zhongning Jia, and Xiaofeng Xue. Application and Analysis of ZigBee Security Services Specification. In *Second International Conference on Networks Security Wireless Communications and Trusted Computing (NSWCTC)*, volume 2, pages 494–497. IEEE, 2010.

[96] Jun Li and Bruce Christianson. A Domain-Oriented Approach for Access Control in Pervasive Environments. In *Proceedings of the 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC'08)*, volume 2, pages 278–284, 2008.

[97] An Liu and Peng Ning. TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. In *Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on*, pages 245–256, 2008.

[98] Donggang Liu and Peng Ning. Establishing Pairwise Keys in Distributed Sensor Networks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 52–61, 2003.

[99] Lanfranco Lopriore. Encrypted Pointers in Protection System Design. *The Computer Journal*, 55(4):497–507, 2012.

[100] Markus Lorch, Seth Proctor, Rebekah Lepro, Dennis Kafura, and Sumit Shah. First experiences using XACML for access control in distributed systems. In *Proceedings of the 2003 ACM workshop on XML security (XMLSEC'03)*, pages 25–37. ACM, 2003.

[101] Gavin Lowe. Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer Berlin / Heidelberg, 1996.

[102] Mark Luk, Ghita Mezzour, Adrian Perrig, and Virgil D. Gligor. MiniSec: A Secure Sensor Network Communication Architecture. In *Proceedings of Sixth International Conference on Information Processing in Sensor Networks (IPSN'07)*, pages 479–488. IEEE, 2007.

[103] John Lyle, Andrew J. Paverd, Justin King-Lacroix, Andrea Atzeni, Habib Virji, Ivan Flechais, and Shamal Faily. Personal PKI for the smart device era. In *Proceedings of the 9th European PKI Workshop: Research and Applications (EuroPKI 2012)*, volume 7868 of *Lecture Notes in Computer Science*, pages 69–84. Springer Berlin / Heidelberg, 2013.

[104] Wenbo Mao, Fei Yan, and Chunrun Chen. Daonity: grid security with behaviour conformity from trusted computing. In *Proceedings of the first ACM workshop on Scalable Trusted Computing (STC'06)*, pages 43–46. ACM, 2006.

[105] John Marchesini and Sean Smith. SHEMP: Secure Hardware Enhanced MyProxy. In *Proceedings of the Third Annual Conference on Privacy, Security, and Trust (PST)*, 2005.

[106] Andrew P. Martin. The Ten Page Introduction to Trusted Computing. Research Report CS-RR-08-11, 2008.

[107] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Ottawa Linux Symposium*, volume 2, pages 21–33, 2007.

[108] Simon Mayer, Dominique Guinard, and Vlad Trifa. Searching in a Web-based Infrastructure for Smart Things. In *Proceedings of the 3rd International Conference on the Internet of Things (IoT 2012)*, 2012.

[109] Rene Mayrhofer, Jürgen Fuß, and Iulia Ion. UACAP: A Unified Auxiliary Channel Authentication Protocol. *IEEE Transactions on Mobile Computing*, 12(4):710–721, 2013.

[110] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, SP'10, pages 143–158, 2010.

[111] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[112] Anthony R. Metke and Randy L. Ekl. Smart Grid security technology. In *Proceedings of the IEEE Conference on Innovative Smart Grid Technologies (ISGT)*, pages 1–7. IEEE, 2010.

[113] Adrian Mettler, David Wagner, and Tyler Close. Joe-E: A Security-Oriented Subset of Java. In *Proceedings of the Network and Distributed Systems Symposium*. Internet Society, 2010.

[114] Microsoft Corp. NTFS Technical Reference. `https://technet.microsoft.com/en-us/library/cc758691%28WS.10%29.aspx`.

[115] Microsoft Corp. Server Message Block (SMB) Protocol Versions 2 and 3. `https://msdn.microsoft.com/en-us/library/cc246482.aspx`.

[116] Mark S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

[117] Mark S. Miller, Bill Tulloh, and Jonathan S. Shapiro. The Structure of Authority: Why Security Is Not a Separable Concern. In *Multiparadigm Programming in Mozart/Oz*, volume 3389 of *Lecture Notes in Computer Science*, pages 2–20. Springer Berlin / Heidelberg, 2005.

[118] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012.

[119] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (Proposed Standard), September 2007. Updated by RFCs 6282, 6775.

[120] R. Moskowitz, T. Heer, P. Jokela, and T. Henderson. Host Identity Protocol Version 2 (HIPv2). RFC 7401 (Proposed Standard), April 2015.

[121] R. Moskowitz and R. Hummen. HIP Diet EXchange (DEX). Internet-Draft draft-moskowitz-hip-dex, July 2015.

[122] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: a distributed operating system for the 1990s. *Computer*, 23:4–53, 1990.

[123] Toby Murray and Duncan Grove. Non-delegatable authorities in capability systems. In *Journal of Computer Security*, volume 16, pages 743–759, 2008.

[124] Marcin Nagy, Thanh Bui, Emiliano De Cristofaro, N. Asokan, Joerg Ott, and Ahmad-Reza Sadeghi. How Far Removed Are You? Scalable Privacy-Preserving Estimation of Social Path Length with Social PaL. *arXiv preprint arXhiv:1412.2433*, 2014.

[125] Mark Needleman. The Shibboleth Authentication/Authorization System. *Serials Review*, 30(3):252–253, 2004.

[126] Pin Nie, Juho Vähä-Herttua, Tuomas Aura, and Andrei Gurtov. Performance Analysis of HIP Diet Exchange for WSN Security Establishment. In *Proceedings of the 7th ACM Symposium on QoS and Security for Wireless and Mobile Networks*, Q2SWinet '11, pages 51–56. ACM, 2011.

[127] Zhongying Niu, Hong Jiang, Ke Zhou, Dan Feng, Shuping Zhang, Tianming Yang, Dongliang Lei, and Anli Chen. DSFS: Decentralized security for large parallel file systems. In *Proceedings of the 11th IEEE/ACM International Workshop on Grid Computing (GRID'10)*, pages 209–216, 2010.

[128] Ivan Osipkov, Eugene Y. Vasserman, Nicholas Hopper, and Yongdae Kim. Combating Double-Spending Using Cooperative P2P Systems. In *Proceedings of the 27th IEEE International Conference on Distributed Computing Systems (ICDCS'07)*, pages 41–50, 2007.

[129] Bryan J. Parno. Bootstrapping trust in a "trusted" platform. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Security (HOTSEC'08)*, pages 9:1–9:6. USENIX Association, 2008.

[130] Andreas Pashalidis and Chris Mitchell. Single Sign-On Using Trusted Platforms. In Colin Boyd and Wenbo Mao, editors, *Information Security*, volume 2851 of *Lecture Notes in Computer Science*, pages 54–68. Springer Berlin / Heidelberg, 2003.

[131] Andrew J. Paverd. *Enhancing Communication Privacy Using Trustworthy Remote Entities*. Phd thesis (under review), University of Oxford, 2016.

[132] Andrew J. Paverd, Fadi El-Moussa, and Ian Brown. Characteristic-based Security Analysis of Personal Networks. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication (UbiComp '14 Adjunct)*, pages 979–986. ACM, 2014.

[133] Adrian Perrig, Ran Canetti, J. D. Tygar, and Dawn Song. Efficient authentication and signing of multicast streams over lossy channels. In *Security and Privacy, 2000. S P 2000. Proceedings. 2000 IEEE Symposium on*, pages 56–73, 2000.

[134] Adrian Perrig, Robert Szewczyk, J. D. Tygar, Victor Wen, and David E. Culler. SPINS: Security Protocols for Sensor Networks. *Wirel. Netw.*, 8(5):521–534, 2002.

[135] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the ACM symposium on Operating systems principles (SOSP)*, SOSP'81, pages 64–75. ACM, 1981.

[136] Shahid Raza, Simon Duquennoy, Joel Höglund, Utz Roedig, and Thiemo Voigt. Secure communication for the Internet of Things – a comparison of link-layer security and IPsec for 6LoWPAN. *Security and Communication Networks*, 7(12):2654–2668, 2014.

[137] David Recordon and Drummond Reed. OpenID 2.0: a platform for user-centric identity management. In *Proceedings of the second ACM workshop on Digital identity management (DIM'06)*, pages 11–16. ACM, 2006.

[138] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012. Updated by RFC 7507.

[139] Thomas Riechmann and Franz J. Hauck. Meta objects for access control: extending capability-based security. In *Proceedings of the workshop on New security paradigms (NSPW)*, pages 17–22. ACM, 1997.

[140] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage*, 9(3):9:1–9:32, 2013.

[141] Rodrigo Roman, Javier Lopez, and Cristina Alcaraz. Do Wireless Sensor Networks Need to be Completely Integrated into the Internet? *Future Internet of People, Things and Services (IoPTS) eco-Systems Workshop*, 2009.

[142] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, 1975.

[143] Ravi S Sandhu and Pierangela Samarati. Access control: principle and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

[144] P. Sangster, H. Khosravi, M. Mani, K. Narayan, and J. Tardo. Network Endpoint Assessment (NEA): Overview and Requirements. RFC 5209 (Informational), June 2008.

[145] B. Sarikaya, Y. Ohba, R. Moskowitz, Z. Cao, and R. Cragie. Security Bootstrapping Solution for Resource-Constrained Devices. Internet-Draft draft-sarikaya-core-sbootstrapping, July 2012.

[146] Luis F. G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas. Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In *Proceedings of the ACM workshop on Scalable Trusted Computing (STC)*, STC'06, pages 27–42. ACM, 2006.

[147] Naveen Sastry and David Wagner. Security considerations for IEEE 802.15.4 networks. In *Proceedings of the ACM workshop on Wireless security (WiSe)*, pages 32–42. ACM Press, 2004.

[148] Mohit Saxena. Security in Wireless Sensor Networks A Layer-based Classification. Technical report, Purdue University, 2007.

[149] M. Scharf and S. Kiesel. NXG03-5: Head-of-line Blocking in TCP and SCTP: Analysis and Measurements. In *Proceedings of the 2006 Global Telecommunications Conference (GLOBECOM '06)*, pages 1–5, 2006.

[150] Benedikt Schmidt, Ralf Sasse, Cas Cremers, and David Basin. Automated Verification of Group Key Agreement Protocols. In *2014 IEEE Symposium on Security and Privacy (S&P)*, pages 179–194. IEEE, 2014.

[151] Michael D. Schroeder and Jerome H. Saltzer. A Hardware Architecture for Implementing Protection Rings. *Commun. ACM*, 15(3):157–170, 1972.

[152] Jonathan S. Shapiro and Norm Hardy. EROS: a principle-driven operating system from the ground up. *IEEE Software*, 19(1):6–33, 2002.

[153] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252 (Proposed Standard), June 2014.

[154] Taeshik Shon, Bonhyun Koo, Hyohyun Choi, and Yongsuk Park. Security Architecture for IEEE 802.15.4-based Wireless Sensor Network. In *Proceedings of the 4th International Symposium on Wireless Pervasive Computing (ISPWC'09)*, pages 1–5. IEEE, 2009.

[155] S. Sicari, A. Rizzardi, L. A. Grieco, and A. Coen-Porisini. Security, privacy and trust in Internet of Things: The road ahead. *Computer Networks*, 76:146–164, 2014.

[156] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 249–264. ACM, 2011.

[157] Frank Stajano and Ross Anderson. The Resurrecting Duckling: security issues for ubiquitous computing. *Computer*, 35(4):22–26, 2002.

[158] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Proceedings of the Usenix Conference (Usenix Winter 1988)*, pages 191–202, 1988.

[159] Piotr Szczechowiak, Leonardo B. Oliveira, Michael Scott, Martin Collier, and Ricardo Dahab. NanoECC: Testing the Limits of Elliptic Curve Cryptography in Sensor Networks. In *Wireless Sensor Networks*, volume 4913 of *Lecture Notes in Computer Science*, pages 305–320. Springer Berlin / Heidelberg, 2008.

[160] Texas Instruments. CC2530 Datasheet. Document SWRS081B, 2009.

[161] Texas Instruments. CC2538 Datasheet. Document SWRS096D, 2012.

[162] Trusted Computing Group. TCG PC Client Specific Implementation Specification For Conventional BIOS 1.20, 2005.

[163] Trusted Computing Group. TCG PC Client Specific TPM Interface Specification 1.21, 2011.

[164] Trusted Computing Group. Trusted Platform Module Main Specification 1.2 Part 1: Design Principles, 2011.

[165] Trusted Computing Group. Trusted Platform Module Main Specification 1.2 Part 2: Structures, 2011.

[166] Trusted Computing Group. Trusted Platform Module Main Specification 1.2 Part 3: Commands, 2011.

[167] Trusted Computing Group. TCG Trusted Network Connect TNC Architecture for Interoperability 1.5, 2012.

[168] Hannes Tschofenig, Sandeep S. Kumar, and Sye Loong Keoh. A Hitchhiker's Guide to the (Datagram) Transport Layer Security Protocol for Smart Objects and Constrained Node Networks. Internet-Draft draft-tschofenig-lwig-tls-minimal, July 2013.

[169] Leif Uhsadel, Axel Poschmann, and Christof Paar. Enabling Full-Size Public-Key Algorithms on 8-Bit Sensor Nodes. In *Security and Privacy in Ad-hoc and Sensor Networks*, volume 4572 of *Lecture Notes in Computer Science*, pages 73–86. Springer Berlin / Heidelberg, 2007.

[170] S. A. Vanstone. Next generation security for wireless: elliptic curve cryptography. *Computers & Security*, 22(5):412–415, 2003.

[171] Haodong Wang and Qun Li. Efficient Implementation of Public Key Cryptosystems on Mote Sensors (Short Paper). In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security*, volume 4307 of *Lecture Notes in Computer Science*, pages 519–528. Springer Berlin / Heidelberg, 2006.

[172] Rafal Wojtczuk and Joanna Rutkowska. Attacking Intel Trusted Execution Technology. In *Black Hat DC*, 2009.

[173] Ford-Long Wong and Frank Stajano. Multi-channel Protocols. In Bruce Christianson, Bruno Crispo, JamesA. Malcolm, and Michael Roe, editors, *Security Protocols*, volume 4631 of *Lecture Notes in Computer Science*, pages 112–127. Springer Berlin / Heidelberg, 2007.

[174] Yang Xiao, Hsiao-Hwa Chen, Bo Sun, Ruhai Wang, and Sakshi Sethi. MAC Security and Security Overhead Analysis in the IEEE 802.15.4 Wireless Sensor Networks. *EURASIP Journal on Wireless Communications and Networking*, 2006(1):93830, 2006.

[175] Gergely V. Zaruba, Stefano Basagni, and Imrich Chlamtac. Bluetrees – scatternet formation to enable Bluetooth-based ad hoc networks. In *Proceedings of the 2001 IEEE International Conference on Communications (ICC 2001)*, volume 1, pages 273–277. IEEE, 2001.

[176] Xinwen Zhang, Yingjiu Li, and Divya Nalla. An Attribute-based Access Matrix Model. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, pages 359–363. ACM, 2005.

[177] ZigBee Alliance. ZigBee Specification. ZigBee Document 053474r17, 2008.

[178] ZigBee Alliance. ZigBee Smart Energy Profile Specification 1.1. ZigBee Document 075356r16ZB, 2011.

[179] ZigBee Alliance. ZigBee IP Specification. ZigBee Document 095023r34, 2014.

Appendix A

Full StarfishNet model

```
THEORY StarfishNet
BEGIN

//Builtins for handling hashing, signatures, and DH.
BUILTINS: hashing, signing, diffie-hellman

/* This function represents the integrity-protection
 * provided by AES-CCM*.
 */
FUNCTIONS: MAC/2

//Base PKI rule
RULE Register_key:
    [ Fr(~ltk) ]
  -->
    [ !Ltk($A, ~ltk), !Pk($A, pk(~ltk)), Out(pk(~ltk)) ]

//Protocol description rules

/* 1. Associate_Request (A sends DHE public key.)
 *    Context: Initiator
 */
RULE Associate_Request:
    LET pdhI = 'g' ^ ~sdhI
    IN
    [ !Ltk($I, ltkI) //initiator's signing key
    , !Pk($R, pk(ltkR)) //responder's public key
    , Fr(~sdhI) //generate initiator's DHE keys
    ]
  -->
    [ Associate_Request($I, $R, ~sdhI) //say A_Req has occurred
    , Out(<pdhI, sign{pdhI}ltkI>) //tx A_Req
    ]
```

```
/* 2. Associate_Reply
 *    (B sends DHE public key, and challenge1 == H(H(k)).)
 *    Context: Responder
 */
RULE Associate_Reply:
    LET pdhR = 'g' ^ ~sdhR
        k = H(<pdhI ^ ~sdhR, pk(ltkI), pk(ltkR)>)
            //generate session key
        chal1 = H(H(k)) //generate chal1
    IN
    [ !Ltk($R, ltkR) //responder's signing key
    , !Pk($I, pk(ltkI)) //initiator's public key
    , FR(~sdhR) //generate responder's DHE keys
    , IN(<pdhI, SIGN{pdhI}ltkI>) //rx A_Req
    ]
  --[ SessionKeyRProvisional($I, $R, k)
      //R has now calculated (but not confirmed) k
    ]->
    [ Associate_Reply($I, $R, k) //say A_Rep has occurred
    , OUT(<pdhR, chal1, SIGN{<pdhR, chal1>}ltkR>) //tx A_Rep
    ]


/* 3. Associate_Finalise (A sends challenge2 == H(k).)
 *    Context: Initiator
 */
RULE Associate_Finalise:
    LET k = H(<pdhR ^ sdhI, pk(ltkI), pk(ltkR)>)
        chal1 = H(H(k)) //check chal1
        chal2 = H(k)    //generate chal2
    IN
    [ !Pk($R, pk(ltkR)) //responder's public key
    , !Pk($I, pk(ltkI)) //initiator's public key
    , Associate_Request($I, $R, sdhI) //ensure A_Req has occurred
    , IN(<pdhR, chal1, SIGN{<pdhR, chal1>}ltkR>) //rx A_Rep
    ]
  --[ SessionKeyI($I, $R, k)
      //I now thinks that a session key has been established
    ]->
    [ OUT(<chal2, MAC(chal2, k)>) //tx A_Fin
    ]
```

```
/* Final rule for reception of A_Fin.
 * Context: Responder
 */
RULE Associate_Finalise_Recv:
    LET chal2 = H(k) //check chal2
    IN
    [ Associate_Reply($I, $R, k) //ensure A_Rep has occurred
    , IN(<chal2, MAC(chal2, k)>) //rx A_Req
    ]
  --[ SessionKeyR($I, $R, k)
      //R now thinks that a session key has been established
    ]->
    [ !SessionKey($I, $R, k)
      //assert that session key has been established
    ]


//Attack rules
RULE Ltk_Reveal:
    [ !Ltk($A, ltkA) ]
  --[ LtkReveal($A, ltkA) ]->
    [ OUT(ltkA) ]

RULE Session_Key_Reveal:
    [ !SessionKey($A, $B, k) ]
  --[ SessionKeyReveal($A, $B, k) ]->
    [ OUT(k) ]

//Security lemmas
LEMMA Key_Agreement_I: "
    ALL I R sessKey #i.
        //If I thinks it's established a session key with R...
        SessionKeyI(I, R, sessKey) @ i &

        //... and they have not lost their signing keys...
        NOT (Ex sk #e. LtkReveal(I, sk) @ e) &
        NOT (Ex sk #e. LtkReveal(R, sk) @ e)

        //... then...
        ==>

        //... R has calculated the same session key when
        // replying to I.  (Note that we can't make a stronger
        // guarantee than that without knowing that the
        // Associate_Finalise is received.)
        (Ex #r. SessionKeyRProvisional(I, R, sessKey) @ r)
"
```

```
lemma Key_Agreement_R: "
    All I R sessKey #i.
        //If R thinks it's established a session key with I...
        SessionKeyR(I, R, sessKey) @ i &

        //... and they haven't lost their signing keys...
        not (Ex sk #e. LtkReveal(I, sk) @ e) &
        not (Ex sk #e. LtkReveal(R, sk) @ e)

        //... then...
        ==>

        //... I thinks it's established the same session key
        // with R.
        (Ex #r. SessionKeyI(I, R, sessKey) @ r)
"

/* Key agreement establishes that if there's a SessionKeyR
 * event, then there's also a matching SessionKeyI event,
 * which means I can just reason about SessionKeyR events
 * from here on in.  That also means that !SessionKey events
 * mean what they're supposed to mean.  In other words, a
 * SessionKeyR event now means "a shared secret has been
 * established".
 */

lemma Key_Secrecy: "
    All I R sessKey #i.
        //If a session key has been established...
        SessionKeyR(I, R, sessKey) @ i &

        //... and it hasn't been disclosed...
        not(Ex #r. SessionKeyReveal(I, R, sessKey) @ r) &

        //... and I and R haven't lost their signing keys...
        not
            ( (Ex sk #r. LtkReveal(I, sk) @ r)
            | (Ex sk #r. LtkReveal(R, sk) @ r)
            )

        //... then...
        ==>

        //... the attacker doesn't know the session key.
        not (Ex #k. K(sessKey) @ k)
"
```

```
LEMMA Perfect_Forward_Secrecy: "
    ALL I R sessKey #i #k.
        //If a session key has been established...
        SessionKeyR(I, R, sessKey) @ i &

        //... and the attacker knows it...
        K(sessKey) @ k

        //... then...
        ==>

        //... either the session key itself was leaked...
        (Ex #r. SessionKeyReveal(I, R, sessKey) @ r) |

        //... or one of the nodes' signing keys was leaked,
        // AND this happened *before* establishment of the
        // session key.
        (Ex sk #r. LtkReveal(I, sk) @ r & r < i) |
        (Ex sk #r. LtkReveal(R, sk) @ r & r < i)
"

LEMMA Honest_I_And_R_Still_Work: EXISTS-TRACE "
    //There exists a trace...
    Ex I R sessKey #p #i #r.
        //... in which R calculates a session key...
        SessionKeyRProvisional(I, R, sessKey) @ p &

        //... I thinks it's that key with R...
        SessionKeyI(I, R, sessKey) @ i &

        //... and R agrees...
        SessionKeyR(I, R, sessKey) @ r &

        //... and the protocol occurs in the correct order...
        p < i &
        i < r &

        //... with no key leakage of any kind.
        NOT (Ex #k. K(sessKey) @ k) &
        NOT (Ex #r. SessionKeyReveal(I, R, sessKey) @ r) &
        NOT (Ex A sk #l. LtkReveal(A, sk) @ l)
"

END
```

Appendix B

Public StarfishNet API

```
/* Crypto */

#define SN_PK_key_bits 160
#define SN_PK_key_size (SN_PK_key_bits/8)

typedef struct SN_Public_key {
    uint8_t data[SN_PK_key_size + 1]; //in packed format
} SN_Public_key_t;

/* Alternate streams */

//sized to fit an IPv6 address + a UDP port number
#define SN_MAX_ALT_STREAM_IDX_BITS (128 + 16)
#define SN_MAX_ALT_STREAM_IDX_SIZE (SN_MAX_ALT_STREAM_IDX_BITS/8)

typedef struct SN_Altstream {
    uint8_t  stream_idx_length;
    uint8_t* stream_idx;
} SN_Altstream_t;

/* Networking */

typedef enum {
    SN_ENDPOINT_NULL,
    SN_ENDPOINT_LONG_ADDRESS,
    SN_ENDPOINT_SHORT_ADDRESS,
    SN_ENDPOINT_PUBLIC_KEY
} SN_Endpoint_type_t;

typedef struct SN_Endpoint {
    SN_Endpoint_type_t type;
    union {
        uint8_t long_address[8];
        uint16_t short_address;
```

```
        SN_Public_key_t public_key;
    };
    SN_Altstream_t* altstream;
} SN_Endpoint_t;

#define SN_COORDINATOR_ADDRESS 0x0000

typedef struct SN_Network_config {
    //routing tree configuration
    uint8_t         routing_tree_branching_factor;
    uint16_t        leaf_blocks;

    //router information
    uint8_t         routing_tree_position;
    uint16_t        router_address;
    SN_Public_key_t router_public_key;
} SN_Network_config_t;

typedef struct SN_Network_descriptor {
    //MAC information
    uint16_t        pan_id;
    uint8_t         radio_channel;

    SN_Network_config_t* network_config;
} SN_Network_descriptor_t;

/* Messages */

typedef enum SN_Message_type {
    //NULL marker
    SN_No_message,

    /* Used by the network layer to signal a dissociation
     * request from another node.  Implicitly invalidates
     * any short address(es) we've taken from or given to
     * it, forcing a recursive address revocation if
     * needs be.
     */
    SN_Dissociation_request,

    /* Used by the network layer to signal an association
     * request from another node.
     */
    SN_Association_request,

    //Standard data message.
    SN_Data_message,
```

```
    //Send a certificate to a StarfishNet node.
    SN_Explicit_Evidence_message,
} SN_Message_type_t;

//StarfishNet messages
typedef union SN_Message {
    SN_Message_type_t type;

    struct {
        SN_Message_type_t type;
        uint8_t*          payload;
        uint8_t           payload_length;
    } data_message;

    struct {
        SN_Message_type_t type;
        SN_Certificate_t* evidence;
    } explicit_evidence_message;
} SN_Message_t;

#define SN_MAX_DATA_MESSAGE_LENGTH 101

//Contiki network stack
#include "net/netstack.h"

extern const struct network_driver starfishnet_driver;

//Send a (usually data) message to a remote node.
int8_t SN_Send(const SN_Endpoint_t *dst_addr,
        const SN_Message_t *message);

//Perform an association with a remote node.
int8_t SN_Associate(const SN_Endpoint_t *dst_addr);

//Register callback for receiving messages.
typedef void (SN_Receive_callback_t)
    (SN_Endpoint_t* src_addr, SN_Message_t* message);
void SN_Receive(SN_Receive_callback_t* callback);

/* Perform a scan for StarfishNet networks.
 * This call returns immediately, and will call the
 * callback function provided once for each network
 * discovered, and once with network == NULL at the
 * end of the discovery period.
 */
typedef void (SN_Discovery_callback_t)
```

```
    (SN_Network_descriptor_t* network, void* extradata);
int8_t SN_Discover(
    SN_Discovery_callback_t *callback,
    uint32_t channel_mask,
    clock_time_t timeout, //in ms
    bool show_full_networks, //0 means ignore full networks
    void *extradata //arbitrary, passed to callback
);

//Start a new StarfishNet network as coordinator.
int8_t SN_Start(const SN_Network_descriptor_t *network);

/* Tune the radio to a StarfishNet network and listen for
 * packets with its PAN ID.
 * To join a network, do SN_Join(network) to tune the radio,
 * followed by
 * SN_Associate(network->network_config->router_public_key)
 * to get an address.
 */
int8_t SN_Join(const SN_Network_descriptor_t *network,
        bool disable_routing);

//Possible return codes for all API calls.
typedef enum {
    SN_OK               , //Success
    SN_ERR_NULL         , //Unexpected NULL pointer
    SN_ERR_UNEXPECTED   , //Operation was unexpected at this time
    SN_ERR_UNIMPLEMENTED, //Operation has not been implemented
    SN_ERR_INVALID      , //Argument or operand was invalid
    SN_ERR_UNKNOWN      , //Neighbor table lookup failure
    SN_ERR_RADIO        , //Error while operating radio
    SN_ERR_END_OF_DATA  , //Packet size mismatch
    SN_ERR_RESOURCES    , //Out of relevant resource
    SN_ERR_SECURITY     , //Security checks failed
    SN_ERR_SIGNATURE    , //Signature verification failed
    SN_ERR_KEYGEN       , //Key generation failed
    SN_ERR_DISCONNECTED , //Contact lost with remote node
} SN_Status;
```

Appendix C

Public BottleCap API

```
//TYPES

#define CAP_MAGIC_TOP    0xCA9A8171
#define CAP_MAGIC_BOTTOM 0x1718A9AC

typedef union {
  struct {
    uint32_t magic_top;  //0xCA9A8171
    uint32_t urights;    //rights mask, used by issuer
    uint64_t oid;        //object-ID field, used by issuer
    aeskey_t issuer;     //issuer's AES-128 key
    uint64_t expiry;     //cap expiry date
    uint32_t srights;    //BottleCap system rights
    uint32_t magic_bottom; //0x1718A9AC
  };
  uint8_t bytes[48];
} cap_t;

#define BOTTLE_MAGIC_TOP  0x80771ECA
#define BOTTLE_MAGIC_BOTTOM 0x909ACE17

typedef struct {
  uint32_t magic_top; //0x80771ECA

  //AES128 cryptographic stuff
  uint128_t  biv;   //Bottle Initialization Vector
  tpm_aeskey_t bek; //{Bottle Encryption Key}_SRK (sealed)

  //bottle configuration stuff
  uint32_t flags; //currently unused
  uint32_t size;  //number of slots in the bottle

  //matching/integrity check
  sha1hash_t header_hmac; //HMAC_SHA1(BEK, this data structure)
```

```
  sha1hash_t table_hmac;  //HMAC_SHA1(BEK, {main table}_BEK)

  uint32_t magic_bottom; //0x909ACE17
} bottle_header_t;

typedef struct {
  bottle_header_t* header; //bottle header structure
  aeskey_t         bek;    //placeholder for decrypted BEK
  cap_t*           table;  //{Caps}_BEK
} bottle_t;

//BOTTLE CREATION/DELETION

/**
 * Initialises a new bottle, in the memory provided.
 * bottle.header.size and bottle.header.flags should be
 * filled in by the caller.
 *
 * @param bottle Memory where the bottle is to be created.
 * @return       Error code.
 */
int32_t bottle_init(bottle_t* bottle);

/**
 * Destroys the given bottle.
 * At the moment simply zeros the relevant memory, but could
 * in future reclaim resources consumed by this bottle, such
 * as TPM monotonic counters.
 *
 * @param bottle Memory where the bottle is to be created.
 * @return       Error code.
 */
int32_t bottle_destroy(bottle_t* bottle);

//BOTTLE STATE FUNCTIONS

/**
 * Returns the number of free slots in the bottle.
 *
 * @param bottle The bottle to operate on.
 * @param slots  Output: Number of free slots in bottle.
 * @return       Error code.
 */
int32_t bottle_query_free_slots(bottle_t* bottle,
        uint32_t* slots);
```

```
/**
 * Deletes all caps in the bottle whose expiry dates
 * are less than or equal to time.
 *
 * @param bottle The bottle to operate on.
 * @param time   The current time.
 * @param slots  Output: The new number of free bottle slots.
 * @return       Error code.
 */
int32_t bottle_expire(bottle_t* bottle,
        uint64_t time, uint32_t* slots);

//CAP INSERTION/DELETION FUNCTIONS

/**
 * Inserts a capability into the first free slot in the bottle.
 *
 * @param bottle The bottle to operate on.
 * @param cap    {The new cap.}_BRK (bound)
 * @param slot   Output: The slot into which cap was inserted.
 * @return       Error code.
 */
int32_t bottle_cap_add(bottle_t* bottle,
        tpm_encrypted_cap_t* cap, uint32_t* slot);

/**
 * Deletes a capability from the specified slot.
 * If the slot is empty, this is a no-op.
 *
 * @param bottle The bottle to operate on.
 * @param slot   The slot to clear.
 * @return       Error code.
 */
int32_t bottle_cap_delete(bottle_t* bottle, uint32_t slot);

//CAP INVOCATION FUNCTIONS

#define ATTEST_MAGIC 0xA77E57EDCA900000ULL
```

```
/**
 * Ticket data structure, filled by bottle_cap_attest --
 * read the comment for that funtion before going any further.
 */
typedef struct {
  uint128_t nonce; //nonce, in plaintext
  union {
    struct {
      uint64_t  oid;     //cap OID
      uint64_t  expiry;  //Ticket expiry date
      uint64_t  amagic;  //0xA77E57EDCA90000
      uint32_t  urights; //Ticket rights mask
      uint32_t  cmagic;  //0xCA9A817 -- CAP_MAGIC_TOP
    };
    unsigned char bytes[32];
  } authdata; //encrupted under issuer key, using nonce as IV

  sha1hash_t hmac;  //HMAC_SHA1(issuer, authdata)

  uint64_t expiry;  //Repeat of the same fields as above, in
  uint32_t urights; // plaintext, for easy introspection.
                    // Must not be used by any
                    // security-sensitive code.
} cap_attestation_block_t;

/**
 * Generates a proof of posession of a capability.
 *
 * @param bottle  The bottle to operate on.
 * @param slot    The slot containing the cap to attest.
 * @param nonce   Public nonce value.
 * @param expiry  Expiry time of the issued authority block.
 * @param urights A mask of user rights to authorise for this
 *                block; must be a subset of the cap rights.
 * @param result  Output: cap attestation block described above.
 * @return        Error code.
 */
int32_t bottle_cap_attest(bottle_t* bottle, uint32_t slot,
        uint128_t nonce, uint64_t expiry,
        uint32_t urightsmask, cap_attestation_block_t* result);
```