Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 27, 2012

M. Sethi
Aalto / Ericsson Research
J. Arkko
A. Keranen
H. Rissanen
Ericsson

March 27, 2012

Practical Considerations and Implementation Experiences in Securing Smart Object Networks
draft-aks-crypto-sensors-02

Abstract

This memo describes challenges associated with securing smart object devices in constrained implementations and environments. The memo describes a possible deployment model suitable for these environments, discusses the availability of cryptographic libraries for small devices, presents some preliminary experiences in implementing small devices using those libraries, and discusses trade-offs involving different types of approaches.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 20, 2012.

Table of Contents

1. Introduction

This memo describes challenges associated with securing smart object devices in constrained implementations and environments (see Section 2).

Secondly, Section 3 discusses a deployment model that the authors are considering for constrained environments. The model requires minimal amount of configuration, and we believe it is a natural fit with the typical communication practices smart object networking environments.

Thirdly, Section 4 discusses the availability of cryptographic libraries. Section 5 presents some experiences in implementing small devices using those libraries, including information about achievable code sizes and speeds on typical hardware.

Finally, Section 7 discusses trade-offs involving different types of security approaches.

1.1. Related Work

Constrained Application Protocol (CoAP) [I-D.ietf-core-coap] is a light-weight protocol designed to be used in machine-to-machine applications such as smart energy and building automation. Our discussion uses this protocol as an example, but the conclusions may apply to other similar protocols. CoAP base specification [I-D.ietf-core-coap] outlines how to use DTLS [RFC5238] and IPsec [RFC4306] for securing the protocol. DTLS can be applied with group keys, pairwise shared keys, or with certificates. The security model in all cases is mutual authentication, so while there is some commonality to HTTP in verifying the server identity, in practice the models are quite different. The specification says little about how DTLS keys are managed. The IPsec mode is described with regards to the protocol requirements, noting that small implementations of IKEv2 exist [I-D.kivinen-ipsecme-ikev2-minimal]. However, the specification is silent on policy and other aspects that are normally necessary in order to implement interoperable use of IPsec in any environment [RFC5406].

[I-D.iab-smart-object-workshop] gives an overview of the security discussions at the March 2011 IAB workshop on smart objects. The workshop recommended that additional work is needed in developing suitable credential management mechanisms (perhaps something similar to the Bluetooth pairing mechanism), understanding the implementability of standard security mechanisms in small devices and additional research in the area of lightweight cryptographic primitives.

[I-D.moskowitz-hip-rg-dex] defines a light-weight version of the HIP protocol for low-power nodes. This version uses a fixed set of algorithms, Elliptic Curve Cryptography (ECC), and eliminates hash functions. The protocol still operates based on host identities, and runs end-to-end between hosts, protecting IP layer communications. [RFC6078] describes an extension of HIP that can be used to send upper layer protocol messages without running the usual HIP base exchange at all.

[I-D.daniel-6lowpan-security-analysis] makes a comprehensive analysis of security issues related to 6LoWPAN networks, but its findings also apply more generally for all low-powered networks. Some of the issues this document discusses include the need to minimize the number of transmitted bits and simplify implementations, threats in the smart object networking environments, and the suitability of 6LoWPAN security mechanisms, IPsec, and key management protocols for implementation in these environments.

[I-D.garcia-core-security] discusses the overall security problem for Internet of Things devices. It also discusses various solutions, including IKEv2/IPsec [RFC4306], TLS/SSL [RFC5246], DTLS [RFC5238], HIP [RFC5201] [I-D.ietf-hip-rfc5201-bis] [I-D.moskowitz-hip-rg-dex], PANA [RFC5191], and EAP [RFC3748]. The draft also discusses various operational scenarios, bootstrapping mechanisms, and challenges associated with implementing security mechanisms in these environments.

2. Challenges

This section discusses three challenges: implementation difficulties, practical provisioning problems, and layering and communication models.

The most often discussed issues in the security for the Internet of Things relate to implementation difficulties. The desire to build small, battery-operated, and inexpensive devices drives the creation of devices with a limited protocol and application suite. Some of the typical limitations include running CoAP instead of HTTP, limited support for security mechanisms, limited processing power for long key lengths, sleep schedule that does not allow communication at all times, and so on. In addition, the devices typically have very limited support for configuration, making it hard to set up secrets and trust anchors.

The implementation difficulties are important, but they should not be overemphasized. It is important to select the right security mechanisms and avoid duplicated or unnecessary functionality. But at

the end of the day, if strong cryptographic security is needed, the implementations have to support that. Also, the use of the most lightweight algorithms and cryptographic primitives is useful, but should not be the only consideration in the design. Interoperability is also important, and often other parts of the system, such as key management protocols or certificate formats are heavier to implement than the algorithms themselves.

The second challenge relates to practical provisioning problems. These are perhaps the most fundamental and difficult issue, and unfortunately often neglected in the design. There are several problems in the provisioning and management of smart object networks:

- Small devices have no natural user interface for configuration that would be required for the installation of shared secrets and other security-related parameters. Typically, there is no keyboard, no display, and there may not even be buttons to press. Some devices may only have one interface, the interface to the network.

- Manual configuration is rarely, if at all, possible, as the necessary skills are missing in typical installation environments (such as in family homes).

- There may be a large number of devices. Configuration tasks that may be acceptable when performed for one device may become unacceptable with dozens or hundreds of devices.

- Network configurations evolve over the lifetime of the devices, as additional devices are introduced or addresses change. Various central nodes may also receive more frequent updates than individual devices such as sensors embedded in building materials.

Finally, layering and communication models present difficulties for straightforward use of the most obvious security mechanisms. Smart object networks typically pass information through multiple participating nodes [I-D.arkko-core-sleepy-sensors] and end-to-end security for IP or transport layers may not fit such communication models very well. The primary reasons for needing middleboxes relates to the need to accommodate for sleeping nodes as well to enable the implementation of nodes that store or aggregate information.

3. Proposed Deployment Model

[I-D.arkko-core-security-arch] recognizes the provisioning model as the driver of what kind of security architecture is useful. This

section re-introduces this model briefly here in order to facilitate the discussion of the various design alternatives later.

The basis of the proposed architecture are self-generated secure identities, similar to Cryptographically Generated Addresses (CGAs) [RFC3972] or Host Identity Tags (HITs) [RFC5201]. That is, we assume the following holds:

$$I = h(P|O)$$

where I is the secure identity of the device, h is a hash function, P is the public key from a key pair generated by the device, and O is optional other information.

3.1. Provisioning

As provisioning security credentials, shared secrets, and policy information is difficult, the provisioning model is based only on the secure identities. A typical network installation involves physical placement of a number of devices while noting the identities of these devices. This list of short identifiers can then be fed to a central server as a list of authorized devices. Secure communications can then commence with the devices, at least as far as information from from the devices to the server is concerned, which is what is needed for sensor networks. Actuator networks and server-to-device communication is covered in Section 4.4 of [I-D.arkko-core-security-arch].

Where necessary, the information collected at installation time may also include other parameters relevant to the application, such as the location or purpose of the devices. This would enable the server to know, for instance, that a particular device is the temperature sensor for the kitchen.

Collecting the identity information at installation time can be arranged in a number of ways. The authors have employed a simple but not completely secure method where the last few digits of the identity are printed on a tiny device just a few millimeters across. Alternatively, the packaging for the device may include the full identity (typically 32 hex digits), retrieved from the device at manufacturing time. This identity can be read, for instance, by a bar code reader carried by the installation personnel. (Note that the identities are not secret, the security of the system is not dependent on the identity information leaking to others. The real owner of an identity can always prove its ownership with the private key which never leaves the device.) Finally, the device may use its wired network interface or proximity-based communications, such as Near-Field Communications (NFC) or Radio-Frequency Identity tags

(RFIDs). Such interfaces allow secure communication of the device identity to an information gathering device at installation time.

No matter what the method of information collection is, this provisioning model minimizes the effort required to set up the security. Each device generates its own identity in a random, secure key generation process. The identities are self-securing in the sense that if you know the identity of the peer you want to communicate with, messages from the peer can be signed by the peer's private key and it is trivial to verify that the message came from the expected peer. There is no need to configure an identity and certificate of that identity separately. There is no need to configure a group secret or a shared secret. There is no need to configure a trust anchor. In addition, the identities are typically collected anyway for application purposes (such as identifying which sensor is in which room). Under most circumstances there is actually no additional configuration effort from provisioning security.

Groups of devices can be managed through single identifiers as well. See Section 4.2 in [I-D.arkko-core-security-arch] for further information.

3.2. Protocol Architecture

As noted above, the starting point of the architecture is that nodes self-generate secure identities which are then communicated out-of-band to the peers that need to know what devices to trust. To support this model in a protocol architecture, we also need to use these secure identities to implement secure messaging between the peers, explain how the system can respond to different types of attacks such as replay attempts, and decide at what protocol layer and endpoints the architecture should use.

The deployment itself is suitable for a variety of design choices regarding layering and protocol mechanisms. [I-D.arkko-core-security-arch] was mostly focused on employing end-to-end data object security as opposed to hop-by-hop security. But other approaches are possible. For instance, HIP in its opportunistic mode could be used to implement largely the same functionality at the IP layer. However, it is our belief that the right layer for this solution is at the application layer. More specifically, in the data formats transported in the payload part of CoAP. This approach provides the following benefits:

- Ability for intermediaries to act as caches to support different sleep schedules, without the security model being impacted.

- Ability for intermediaries to be built to perform aggregation, filtering, storage and other actions, again without impacting the security of the data being transmitted or stored.

- Ability to operate in the presence of traditional middleboxes, such as a protocol translators or even NATs (not that we recommend their use in these environments).

However, as we will see later there are also some technical implications, namely that link, network, and transport layer solutions are more likely to be able to benefit from sessions where the cost of expensive operations can be amortized over multiple data transmissions. While this is not impossible in data object security solutions either, it is not the typical arrangement either.

4. Code Availability

For implementing public key cryptography on resource constrained environments, we chose Arduino Uno board [arduino-uno] as the test platform. Arduino Uno has an ATmega328 microcontroller, an 8-bit processor with a clock speed of 16 MHz, 2 kB of SRAM, and 32 kB of flash memory.

For selecting potential asymmetric cryptographic libraries, we did an extensive survey and came up with a set of possible code sources, and performed an initial analysis of how well they fit the Arduino environment. Note that the results are preliminary, and could easily be affected in any direction by implementation bugs, configuration errors, and other mistakes. Please verify the numbers before relying on them for building something. No significant effort was done to optimize ROM memory usage beyond what the libraries provided themselves, so those numbers should be taken as upper limits.

Here is the set of libraries we found:

- AvrCryptolib [avr-cryptolib]: This library provides a variety of different symmetric key algorithms such as DES/Triple DES/AES etc. and RSA as an asymmetric key algorithm. We stripped down the library to use only the required RSA components and used a separate SHA-256 implementation from the original AvrCrypto-Lib library [avr-crypto-lib]. Parts of SHA-256 and RSA algorithm implementations were written in AVR-8 bit assembly language to reduce the size and optimize the performance. The library also takes advantage of the fact that Arduino boards allow the programmer to directly address the flash memory to access constant data which can save the amount of SRAM used during execution.

- Relic-Toolkit [relic-toolkit]: This library is written entirely in C and provides a highly flexible and customizable implementation of a large variety of cryptographic algorithms. This not only includes RSA and ECC, but also pairing based asymmetric cryptography, Boneh-Lynn-Schacham, Boneh-Boyen short signatures and many more. The toolkit provides an option to build only the desired components for the required platform. While building the library, it is possible to select a variety mathematical optimizations that can be combined to obtain the desired performance (as a general thumb rule, faster implementations require more SRAM and flash). It includes a multi precision integer math module which can be customized to use different bit-length words.

- TinyECC [tinyecc]: TinyECC was designed for using Elliptic Curve based public key cryptography on sensor networks. It is written in nesC programming language and as such is designed for specific use on TinyOS. However, the library can be ported to standard C99 either with hacked tool-chains or manually rewriting parts of the code. This allows for the library to be used on platforms that do not have TinyOS running on them. The library includes a wide variety of mathematical optimizations such as sliding window, Barrett reduction for verification, precomputation, etc. It also has one of the smallest memory footprints among the set of Elliptic Curve libraries surveyed so far. However, an advantage of Relic over TinyECC is that it can do curves over binary fields in addition to prime fields.

- Wiselib [wiselib]: Wiselib is a generic library written for sensor networks containing a wide variety of algorithms. While the stable version contains algorithms for routing only, the test version includes many more algorithms including algorithms for cryptography, localization , topology management and many more. The library was designed with the idea of making it easy to interface the library with operating systems like iSense and Contiki. However, since the library is written entirely in C++ with a template based model similar to Boost/CGAL, it can be used on any platform directly without using any of the operating system interfaces provided. This approach was taken by the authors to test the code on Arduino Uno. The structure of the code is similar to TinyECC and like TinyECC it implements elliptic curves over prime fields only. In order to make the code platform independent, no assembly level optimizations were incorporated. Since efficiency was not an important goal for the authors of the library while designing, many well known theoretical performance enhancement features were also not incorporated. Like the relic-toolkit, Wiselib is also Lesser GPL licensed.

- MatrixSSL [matrix-ssl]: This library provides a low footprint implementation of several cryptographic algorithms including RSA and ECC (with a commercial license). However, the library in the original form takes about 50 kB of ROM which is not suitable for our hardware requirements. Moreover, it is intended for 32-bit systems and the API includes functions for SSL communication rather than just signing data with private keys.

5. Implementation Experiences

We have summarized the initial results of RSA private key performance using AvrCryptolib in Table 1. All results are from a single run since repeating the test did not change (or had only minimal impact on) the results. The keys were generated separately and were hard coded into the program. All keys were generated with the value of the public exponent as 3. The performance of encryption with private key was faster for smaller key lengths as was expected. However the increase in the execution time was considerable when the key size was 2048 bits. It is important to note that two different sets of experiments were performed for each key length. In the first case, the keys were loaded into the SRAM from the ROM (flash) before they were used by any of the functions. However, in the second case, the keys were addressed directly in the ROM. As was expected, the second case used less SRAM but lead to longer execution time.

Key length (bits)	Execution time (ms); key in SRAM	Memory footprint (bytes); key in SRAM	Execution time (ms); key in ROM	Memory footprint (bytes); key in ROM
64	66	40	70	32
128	124	80	459	64
512	25,089	320	27,348	256
1,024	199,666	640	218,367	512
2,048	1,587,559	1,280	1,740,267	1,024

RSA private key operation performance

Table 1

The code size was less than 3.6 kB for all the test cases with scope for further reduction. It is also worth noting that the implementation performs basic exponentiation and multiplication operations without using any mathematical optimizations such as Montgomery multiplication, optimized squaring, etc. as described in

[rsa-high-speed]. With more SRAM, we believe that 1024/2048-bit operations can be performed in much less time as has been shown in [rsa-8bit]. 2048-bit RSA is nonetheless possible with about 1 kB of SRAM as is seen in Table 1.

In Table 2 we present the results obtained by manually porting TinyECC into C99 standard and running ECDSA signature algorithm on the Arduino Uno board. TinyECC supports a variety of SEC 2 recommended Elliptic Curve domain parameters. The execution time and memory footprint are shown next to each of the curve parameters. SHA-1 hashing algorithm included in the library was used in each of the cases. It is clearly observable that for similar security levels, Elliptic Curve public key cryptography outperforms RSA. These results were obtained by turning on all the optimizations. These optimizations include - Curve Specific Optimizations for modular reduction (NIST and SEC 2 field primes were chosen as pseudo-Mersenne primes), Sliding Window for faster scalar multiplication, Hybrid squaring procedure written in assembly and Weighted projective Co-ordinate system for efficient scalar point addition, doubling and multiplication. We did not use optimizations like Shamir Trick and Sliding Window as they are only useful for signature verification and tend to slow down the signature generation by precomputing values (we were only interested in fast signature generation). There is still some scope for optimization as not all the assembly code provided with the library could be ported to Arduino directly. Re-writing these procedures in compatible assembly would further enhance the performance.

Curve parameters	Execution time (ms)	Memory Footprint (bytes)	Comparable RSA key length
128r1	1,858	776	704
128r2	2,002	776	704
160k1	2,228	892	1,024
160r1	2,250	892	1,024
160r2	2,467	892	1,024
192k1	3,425	1008	1,536
192r1	3,578	1008	1,536

ECDSA signature performance with TinyECC

Table 2

We also performed experiments by removing the assembly code for hybrid multiplication and squaring thus using a C only form of the

library. This gives us an idea of the performance that can be achieved with TinyECC on any platform regardless of what kind of OS and assembly instruction set available. The memory footprint remains the same with our without assembly code. The tables contain the maximum RAM that is used when all the possible optimizations are on. If however, the amount of RAM available is smaller in size, some of the optimizations can be turned off to reduce the memory consumption accordingly.

Curve parameters	Execution time (ms)	Memory Footprint (bytes)	Comparable RSA key length
128r1	2,741	776	704
128r2	3,086	776	704
160k1	3,795	892	1,024
160r1	3,841	892	1,024
160r2	4,118	892	1,024
192k1	6,091	1008	1,536
192r1	6,217	1008	1,536

ECDSA signature performance with TinyECC (No assembly optimizations)

Table 3

Table 4 documents the performance of Wiselib. Since there were no optimizations that could be turned on or off, we have only one set of results. By default Wiselib only supports some of the standard SEC 2 Elliptic curves. But it is easy to change the domain parameters and obtain results for for all the 128, 160 and 192-bit SEC 2 Elliptic curves. SHA-1 algorithm provided in the library was used. The ROM size for all the experiments was less than 16 kB.

Curve parameters	Execution time (ms)	Memory Footprint (bytes)	Comparable RSA key length
128r1	10,744	732	704
128r2	10,615	732	704
160k1	20,164	842	1,024
160r1	20,231	842	1,024
160r2	20,231	842	1,024
192k1	34,486	952	1,536
192r1	34,558	952	1,536

ECDSA signature performance with Wiselib

Table 4

For testing the relic-toolkit we used a different board because it required more RAM/ROM and we were unable to perform experiments with it on Arduino Uno. We decided to use the Arduino Mega which has the same 8-bit architecture like the Arduino Uno but has a much larger RAM/ROM for testing relic-toolkit. The relic-toolkit supports curves over prime as well as binary fields. Since it is shown in [binary-prime] that Kolobitz curves perform better over binary fields than prime fields, we have experimented with curves over binary fields only. We leave the comparison of ordinary curves over prime and binary fields on 8-bit platforms as future work. The results from relic-toolkit are documented in two separate tables shown in Table 5 and Table 6. The first set of results were performed with the library configured for high speed performance with no consideration given to the amount of memory used. For the second set, the library was configured for low memory usage irrespective of the execution time required by different curves. By turning on/off optimizations included in the library, a trade-off between memory and execution time between these values can be achieved.

Curve parameters	Execution time (ms)	Memory Footprint (bytes)	Comparable RSA key length
NIST K163 (assembly math)	300	2,804	1024
NIST K163	966	2,750	1024
NIST B163	2,408	2,444	1024
NIST K233	1,828	3,675	2,048
NIST B233	4,834	3,261	2,048

ECDSA signature performance with relic-toolkit (Fast)

Table 5

Curve parameters	Execution time (ms)	Memory Footprint (bytes)	Comparable RSA key length
NIST K163 (assembly math)	1,128	2,087	1024
NIST K163	3,178	2,215	1024
NIST B163	3,464	2,071	1024
NIST K233	6,995	2,935	2,048
NIST B233	6,535	2,737	2,048

ECDSA signature performance with relic-toolkit (Low Memory)

Table 6

It is important to note the following points about the elliptic curve measurements:

- The arduino board only provides pseudo random numbers with the random() function call. In order to create private keys with a better quality of random number, we can use a true random number generator like the one provided by TrueRandom library [truerandom], or create the keys separately on a system with a true random number generator and then use them directly in the code.

- For measuring the memory footprint of all the ECC libraries, we used the Avrora simulator [avrora]. Only stack memory was used to easily track the RAM consumption.

A summary library ROM use is shown in Table 7.

Library	ROM Footprint (Kilobytes)
AvrCryptolib	3.6
Wiselib	16
TinyECC	18
Relic-toolkit	29

Summary of library ROM needs

Table 7

6. Example Application

We developed an example application on the Arduino platform to use public key crypto mechanisms, data object security, and an easy provisioning model. Our application was originally developed to test different approaches to supporting communications to "always off" sensor nodes. These battery-operated or energy scavenging nodes do not have enough power to be stay on at all times. They wake up periodically and transmit their readings.

Such sensor nodes can be supported in various ways. [I-D.arkko-core-sleepy-sensors] was an early multicast-based approach. In the current application we have switched to using resource directories [I-D.shelby-core-resource-directory] and mirror proxies [I-D.vial-core-mirror-proxy] instead. Architecturally, the idea is that sensors can delegate a part of their role to a node in the network. Such a network node could be either a local resource or something in the Internet. In the case of CoAP mirror proxies, the network node agress to hold the web resources on behalf of the sensor, while the sensor is asleep. The only role that the sensor has is to register itself at the mirror proxy, and periodically update the readings. All queries from the rest of the world go to the mirror proxy.

We constructed a system with four entities:

Sensor

This is an Arduino-based device that runs a CoAP mirror proxy client and Relic-toolkit. Relic takes 29 Kbytes of ROM, and the simple CoAP client roughly 3 kilobytes.

Mirror Proxy

This is a mirror proxy that holds resources on the sensor's behalf. The sensor registers itself to this node.

Resource Directory

While physically in the same node in our implementation, a resource directory is a logical function that allows sensors and mirror proxies to register resources in the directory. These resources can be queried by applications.

Application

This is a simple application that runs on a general purpose computer and can retrieve both registrations from the resource directory and most recent sensor readings from the mirror proxy.

The security of this system relies on an SSH-like approach. In Step 1, upon first boot, sensors generate keys and register themselves in the mirror proxy. Their public key is submitted along with the registration as an attribute in the CORE Link Format data [I-D.ietf-core-link-format].

In Step 2, when the sensor makes a sensor reading update to the mirror proxy it signs the message contents with a JOSE signature on the used JSON/SENML payload [I-D.ietf-jose-json-web-signature] [I-D.jennings-senml].

In Step 3, any other device in the network -- including the mirror proxy, resource directory and the application -- can check that the public key from the registration corresponds to the private key used to make the signature in the data update.

Note that checks can be done at any time and there is no need for the sensor and the checking node to be awake at the same time. In our implementation, the checking is done in the application node. This demonstrates how it is possible to implement end-to-end security even with the presence of assisting middleboxes.

7. Design Trade-Offs

This section attempts to make some early conclusions regarding trade-offs in the design space, based on deployment considerations for various mechanisms and the relative ease or difficulty of implementing them. This analysis looks at layering and the choice of symmetric vs. asymmetric cryptography.

7.1. Feasibility

The first question is whether using cryptographic security and asymmetric cryptography in particular is feasible at all on small devices. The numbers above give a mixed message. Clearly, an implementation of a significant cryptographic operation such as public key signing can be done in surprisingly small amount of code space. It could even be argued that our chosen prototype platform was unnecessarily restrictive in the amount of code space it allows: we chose this platform on purpose to demonstrate something that is as small and difficult as possible.

In reality, ROM memory size is probably easier to grow than other parameters in microcontrollers. A recent trend in microcontrollers is the introduction of 32-bit CPUs that are becoming cheaper and more easily available than 8-bit CPUs, in addition to being more easily programmable. In short, the authors do not expect the code size to be a significant limiting factor, both because of the small amount of code that is needed and because available memory space is growing rapidly.

The situation is less clear with regards to the amount of CPU power needed to run the algorithms. The demonstrated speeds are sufficient for many applications. For instance, a sensor that wakes up every now and then can likely spend a fraction of a second for the computation of a signature for the message that it is about to send. Or even spend multiple seconds in some cases. Most applications that use protocols such as DTLS that use public key cryptography only at the beginning of the session would also be fine with any of these execution times.

Yet, with reasonably long key sizes the execution times are in the seconds, dozens of seconds, or even longer. For some applications this is too long. Nevertheless, the authors believe that these algorithms can successfully be employed in small devices for the following reasons:

- With the right selection of algorithms and libraries, the execution times can actually be smaller. Using the Relic-toolkit with the NIST K163 algorithm (roughly equivalent to RSA at 1024 bits) at 0.3 seconds is a good example of this.

- As discussed in [wiman], in general the power requirements necessary to send or receive messages are far bigger than those needed to execute cryptographic operations. There is no good reason to choose platforms that do not provide sufficient computing power to run the necessary operations.

- Commercial libraries and the use of full potential for various optimizations will provide a better result than what we arrived at in this paper.

- Using public key cryptography only at the beginning of a session will reduce the per-packet processing times significantly.

7.2. Layering

It would be useful to select just one layer where security is provided at. Otherwise a simple device needs to implement multiple security mechanisms. While some code can probably be shared across such implementations (like algorithms), it is likely that most of the code involving the actual protocol machinery cannot. Looking at the different layers, here are the choices and their implications:

link layer

This is probably the most common solution today. The biggest benefits of this choice of layer are that security services are commonly available (WLAN secrets, cellular SIM cards, etc.) and that their application protects the entire communications.

The main drawback is that there is no security beyond the first hop. This can be problematic, e.g., in many devices that communicate to a server in the Internet. A Withings scale [Withings], for instance, can support WLAN security but without some level of end-to-end security, it would be difficult to prevent fraudulent data submissions to the servers.

Another drawback is that some commonly implemented link layer security designs use group secrets. This allows any device within the local network (e.g., an infected laptop) to attack the communications.

network layer

There are a number of solutions in this space, and many new ones and variations thereof being proposed: IPsec, PANA, and so on. In general, these solutions have similar characteristics to those in the transport layer: they work across forwarding hops but only as far as to the next middlebox or application entity. There is plenty of existing solutions and designs.

Experience has shown that it is difficult to control IP layer entities from an application process. While this is theoretically easy, in practice the necessary APIs do not exist. For instance, most IPsec software has been built for the VPN use case, and is

difficult or impossible to tweak to be used on a per-application basis. As a result, the authors are not particularly enthusiastic about recommending these solutions.

transport and application layer

This is another popular solution along with link layer designs. SSL, TLS, DTLS, and HTTPS are examples of solutions in this space, and have been proven to work well. These solutions are typically easy to take into use in an application, without assuming anything from the underlying OS, and they are easy to control as needed by the applications. The main drawback is that generally speaking, these solutions only run as far as the next application level entity. And even for this case, HTTPS can be made to work through proxies, so this limit is not unsolvable. Another drawback is that attacks on link layer, network layer and in some cases, transport layer, can not be protected against. However, if the upper layers have been protected, such attacks can at most result in a denial-of-service. Since denial-of-service can often be caused anyway, it is not clear if this is a real drawback.

data object layer

This solution does not protect any of the protocol layers, but protects individual data elements being sent. It works particularly well when there are multiple application layer entities on the path of the data. The authors believe smart object networks are likely to employ such entities for storage, filtering, aggregation and other reasons, and as such, an end-to-end solution is the only one that can protect the actual data.

The downside is that the lower layers are not protected. But again, as long as the data is protected and checked upon every time it passes through an application level entity, it is not clear that there are attacks beyond denial-of-service.

The main question mark is whether this type of a solution provides sufficient advantages over the more commonly implemented transport and application layer solutions.

7.3. Symmetric vs. Asymmetric Crypto

The second trade-off that is worth discussing is the use of plain asymmetric cryptographic mechanisms, plain symmetric cryptographic mechanisms, or some mixture thereof.

Contrary to popular cryptographic community beliefs, a symmetric crypto solution can be deployed in large scale. In fact, the largest

deployment of cryptographic security, the cellular network authentication system, uses SIM cards that are based on symmetric secrets. In contrast, public key systems have yet to show ability to scale to hundreds of millions of devices, let alone billions. But the authors do not believe scaling is an important differentiator when comparing the solutions.

As can be seen from the Section 5, the time needed to calculate some of the asymmetric crypto operations with reasonable key lengths can be significant. There are two contrary observations that can be made from this. First, recent wisdom indicates that computing power on small devices is far cheaper than transmission power [wiman], and keeps on becoming more efficient very quickly. From this we can conclude that the sufficient CPU is or at least will be easily available.

But the other observation is that when there are very costly asymmetric operations, doing a key exchange followed by the use of generated symmetric keys would make sense. This model works very well for DTLS and other transport layer solutions, but works less well for data object security, particularly when the number of communicating entities is not exactly two.

8. Security Considerations

This entire memo deals with security issues.

9. IANA Considerations

There are no IANA impacts in this memo.

10. References

10.1. Normative References

[I-D.ietf-core-coap]
Shelby, Z., Hartke, K., Bormann, C., and B. Frank, "Constrained Application Protocol (CoAP)", draft-ietf-core-coap-06 (work in progress), May 2011.

[arduino-uno]
"Arduino Uno", <http://arduino.cc/en/Main/arduinoBoardUno>.

[relic-toolkit]

"Relic Toolkit",
<http://code.google.com/p/relic-toolkit/>.

[avr-crypto-lib]
"AVR-CRYPTO-LIB",
<http://www.das-labor.org/wiki/AVR-Crypto-Lib/en>.

[avr-cryptolib]
"AVR CRYPTOLIB", <http://www.emsign.nl/>.

[avrora] "AVR CRYPTOLIB", <http://compilers.cs.ucla.edu/avrora/>.

[truerandom]
"AVR CRYPTOLIB",
<http://code.google.com/p/tinkerit/wiki/TrueRandom>.

[wiselib] "", <www.wiselib.org/>.

[tinyecc] North Carolina State University and North Carolina State University, "TinyECC",
<http://discovery.csc.ncsu.edu/software/TinyECC/>.

[matrix-ssl]
PeerSec Networks, "Matrix SSL",
<http://www.matrixssl.org/>.

[rsa-high-speed]
RSA Labs, "High-Speed RSA Implementation",
<http://cs.ucsb.edu/~koc/docs/r01.pdf>.

[binary-prime]
University of Waterloo, Auburn University, University of Valle, and Certicom Research, "High-Speed RSA Implementation", <http://cs.ucsb.edu/~koc/docs/r01.pdf>.

[rsa-8bit]
Sun Microsystems, "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs".

10.2. Informative References

[RFC3748] Aboba, B., Blunk, L., Vollbrecht, J., Carlson, J., and H. Levkowetz, "Extensible Authentication Protocol (EAP)", RFC 3748, June 2004.

[RFC3972] Aura, T., "Cryptographically Generated Addresses (CGA)", RFC 3972, March 2005.

[RFC4306] Kaufman, C., "Internet Key Exchange (IKEv2) Protocol", RFC 4306, December 2005.

[RFC5191] Forsberg, D., Ohba, Y., Patil, B., Tschofenig, H., and A. Yegin, "Protocol for Carrying Authentication for Network Access (PANA)", RFC 5191, May 2008.

[RFC5201] Moskowitz, R., Nikander, P., Jokela, P., and T. Henderson, "Host Identity Protocol", RFC 5201, April 2008.

[RFC5238] Phelan, T., "Datagram Transport Layer Security (DTLS) over the Datagram Congestion Control Protocol (DCCP)", RFC 5238, May 2008.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC5406] Bellovin, S., "Guidelines for Specifying the Use of IPsec Version 2", BCP 146, RFC 5406, February 2009.

[RFC6078] Camarillo, G. and J. Melen, "Host Identity Protocol (HIP) Immediate Carriage and Conveyance of Upper-Layer Protocol Signaling (HICCUPS)", RFC 6078, January 2011.

[I-D.arkko-core-sleepy-sensors]
Arkko, J., Rissanen, H., Loreto, S., Turanyi, Z., and O. Novo, "Implementing Tiny COAP Sensors", draft-arkko-core-sleepy-sensors-01 (work in progress), July 2011.

[I-D.arkko-core-security-arch]
Arkko, J. and A. Keranen, "CoAP Security Architecture", draft-arkko-core-security-arch-00 (work in progress), July 2011.

[I-D.daniel-6lowpan-security-analysis]
Park, S., Kim, K., Haddad, W., Chakrabarti, S., and J. Laganier, "IPv6 over Low Power WPAN Security Analysis", draft-daniel-6lowpan-security-analysis-05 (work in progress), March 2011.

[I-D.garcia-core-security]
Garcia-Morchon, O., Keoh, S., Kumar, S., Hummen, R., and R. Struik, "Security Considerations in the IP-based Internet of Things", draft-garcia-core-security-02 (work in progress), July 2011.

[I-D.iab-smart-object-workshop]

Tschofenig, H. and J. Arkko, "Report from the 'Interconnecting Smart Objects with the Internet' Workshop, 25th March 2011, Prague", draft-iab-smart-object-workshop-01 (work in progress), July 2011.

[I-D.ietf-hip-rfc5201-bis]
Moskowitz, R., Heer, T., Jokela, P., and T. Henderson, "Host Identity Protocol Version 2 (HIPv2)", draft-ietf-hip-rfc5201-bis-06 (work in progress), July 2011.

[I-D.kivinen-ipsecme-ikev2-minimal]
Kivinen, T., "Minimal IKEv2", draft-kivinen-ipsecme-ikev2-minimal-00 (work in progress), February 2011.

[I-D.moskowitz-hip-rg-dex]
Moskowitz, R., "HIP Diet EXchange (DEX)", draft-moskowitz-hip-rg-dex-05 (work in progress), March 2011.

[I-D.vial-core-mirror-proxy]
Vial, M., "CoRE Mirror Proxy", draft-vial-core-mirror-proxy-00 (work in progress), March 2012.

[I-D.shelby-core-resource-directory]
Shelby, Z. and S. Krco, "CoRE Resource Directory", draft-shelby-core-resource-directory-00 (work in progress), June 2011.

[I-D.ietf-jose-json-web-signature]
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature-01 (work in progress), March 2012.

[I-D.ietf-core-link-format]
Shelby, Z., "CoRE Link Format", draft-ietf-core-link-format-11 (work in progress), January 2012.

[I-D.jennings-senml]
Jennings, C., "Media Type for Sensor Markup Language (SENML)", draft-jennings-senml-05 (work in progress), March 2011.

[Withings]

Withings, "The Withings scale", February 2012, <http://www.withings.com/en/bodyscale>.

[wiman] "Impact of Operating Systems on Wireless Sensor Networks (Security) Applications and Testbeds. In International Conference on Computer Communication Networks (ICCCN'2010) / IEEE International Workshop on Wireless Mesh and Ad Hoc Networks (WiMAN 2010), 2010, Zuerich. Proceedings of ICCCN'2010/WiMAN'2010", 2010.

Appendix A. Acknowledgments

The authors would like to thank Mats Naslund, Salvatore Loreto, Bob Moskowitz, Oscar Novo, Vlasios Tsiatsis, Daoyuan Li, Muhammad Waqas, Eric Rescorla and Tero Kivinen for interesting discussions in this problem space. The authors would also like to thank Diego Aranha for helping with the relic-toolkit configurations and Tobias Baumgartner for helping with questions regarding wiselib.

Authors' Addresses

Mohit Sethi
Aalto / Ericsson Research
Ericsson: 02420 Jorvas, Finland
Aalto: Tietotekniikan laitos, Tietoliikenneohjelmistot, PL 15400, 00076 Aalto, Finland

Email: mohit.sethi@aalto.fi / mohit.m.sethi@ericsson.com

Jari Arkko
Ericsson
Jorvas 02420
Finland

Email: jari.arkko@piuha.net

Ari Keranen
Ericsson
Jorvas 02420
Finland

Email: ari.keranen@ericsson.com

Heidi-Maria Rissanen
Ericsson
Jorvas 02420
Finland

Email: heidi-maria.rissanen@ericsson.com