Light-Weight Implementation Guidance
Internet-Draft
Intended status: Informational
Expires: May 4, 2017

M. Sethi
J. Arkko
A. Keranen
Ericsson
H. Back
Comptel

October 31, 2016

Practical Considerations and Implementation Experiences in Securing Smart Object Networks
draft-ietf-lwig-crypto-sensors-01

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 May 4, 2017.

Table of Contents

1. Introduction

This memo describes challenges associated with securing smart object devices in constrained implementations and environments. In Section 3) we specifically discuss three challenges: the implementation difficulties encountered on resource-constrained platforms, the problem of provisioning keys and making the choice of implementing security at the appropriate layer.

Secondly, Section 4 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 in smart object networking environments.

Thirdly, Section 7 discusses the availability of cryptographic libraries. Section 8 presents some experiences in implementing cryptography on small devices using those libraries, including

information about achievable code sizes and speeds on typical hardware.

Finally, Section 10 discusses trade-offs involving different types of security approaches.

2. Related Work

Constrained Application Protocol (CoAP) [RFC7252] 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 [RFC7252] outlines how to use DTLS [RFC6347] and IPsec [RFC4303] for securing the protocol. DTLS can be applied with pairwise shared keys, raw public 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 CoAP specification says little about how DTLS keys are managed. The use of IPsec with CoAP is described with regards to the protocol requirements, noting that small implementations of IKEv2 exist [RFC7815]. However, the CoAP specification is silent on policy and other aspects that are normally necessary in order to implement interoperable use of IPsec in any environment [RFC5406].

[RFC6574] 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-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.irtf-t2trg-iot-seccons] discusses the overall security problem for Internet of Things devices. It also discusses various solutions, including IKEv2/IPsec [RFC7296], TLS/SSL [RFC5246], DTLS [RFC6347], HIP [RFC7401] [I-D.moskowitz-hip-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.

3. 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.

4. 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) [RFC7401]. 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. | here denotes the concatenation operator.

5. Provisioning

As it is difficult to provision security credentials, shared secrets, and policy information, 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.

The above architecture is a perfect fit for sensor networks where information flows from large number of devices to small number of servers. But it is not sufficient alone for other types of applications. For instance, in actuator applications a large number of devices need to take commands from somewhere else. In such applications it is necessary to secure that the commands come from an authorized source. This can be supported, with some additional provisioning effort and optional pairing protocols. The basic provisioning approach is as described earlier, but in addition there must be something that informs the devices of the identity of the trusted server(s). There are multiple ways to provide this information. One simple approach is to feed the identities of the trusted server(s) to devices at installation time. This requires either a separate user interface, local connection (such as USB), or using the network interface of the device for configuration. In any case, as with sensor networks the amount of configuration information is minimized: just one short identity value needs to be fed in. Not both an identity and a certificate. Not shared secrets that must be kept confidential. An even simpler provisioning approach is that the devices in the device group trust each other. Then no configuration is needed at installation time. When both peers know the expected cryptographic identity of the other peer off-line, secure communications can commence. Alternatively, various pairing schemes can be employed. Note that these schemes can benefit from the already secure identifiers on the device side. For instance, the server can send a pairing message to each device after their initial power-on and before they have been paired with anyone, encrypted with the public key of the device. As with all pairing schemes that do not employ a shared secret or the secure dentity of both parties, there are some remaining vulnerabilities that may or may not be acceptable for the application in question. In any case, the secure identities help again in ensuring that the operations are as simple as possible. Only identities need to be communicated to the devices, not certificates, not shared secrets or IPsec policy rules.

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. In these deployment cases it is also possible to configure the identity of an entire group of devices, rather than registering the individual devices. For instance, many installations employ a kit of devices bought from the same manufacturer in one package. It is easy to provide an identity for such a set of devices as follows:

Idev = h(Pdev|Potherdev1|Potherdev2|...|Potherdevn)

Igrp = h(Pdev1|Pdev2|...|Pdevm)

where Idev is the identity of an individual device, Pdev is the public key of that device, and Potherdevi are the public keys of other devices in the group. Now, we can define the secure identity of the group (Igrp) as a hash of all the public keys of the devices in the group (Pdevi).

The installation personnel can scan the identity of the group from the box that the kit came in, and this identity can be stored in a server that is expected to receive information from the nodes. Later when the individual devices contact this server, they will be able to show that they are part of the group, as they can reveal their own public key and the public keys of the other devices. Devices that do not belong to the kit can not claim to be in the group, because the group identity would change if any new keys were added to Igrp.

6. 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.

7. 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 AES 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.

This is by no ways an exhaustive list and there exist other cryptographic libraries targeting resource-constrained devices.

8. Implementation Experiences

While evaluating the implementation experiences, we were particularly interested in the signature generation operation. This was because our example application discussed in Section 9 required only the signature generation operation on the resource-constrained platforms. 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 signing 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.

More importantly, any RSA key size less than 2,048-bit should be considered legacy and insecure. The performance measurements for these keys are provided here for reference only.

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	64	40	69	32
128	434	80	460	64
512	25,076	320	27348	256
1,024	199688	640	218367	512
2,048	1587567	1,280	1740258	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. The measurements reflect the performance of elliptic curve signing only and not the SHA-1 hashing algorithm. SHA-1 is now known to be insecure and should not be used in real deployments. 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 Coordinate 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	1858	776	704
128r2	2002	776	704
160k1	2228	892	1024
160r1	2250	892	1024
160r2	2467	892	1024
192k1	3425	1008	1536
192r1	3578	1008	1536

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	2741	776	704
128r2	3086	776	704
160k1	3795	892	1024
160r1	3841	892	1024
160r2	4118	892	1024
192k1	6091	1008	1536
192r1	6217	1008	1536

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 measurements reflect the performance of elliptic curve signing only and not the SHA-1 hashing algorithm. SHA-1 is now known to be insecure and should not be used in real deployments. 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	5615	732	704
128r2	5615	732	704
160k1	10957	842	1024
160r1	10972	842	1024
160r2	10971	842	1024
192k1	18814	952	1536
192r1	18825	952	1536

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. Again, SHA-1 hashing algorithm included in the library was used in each of the cases. The measurements reflect the performance of elliptic curve signing only and not the SHA-1 hashing algorithm. SHA-1 is now known to be insecure and should not be used in real deployments. The library does provide several alternatives with such as SHA-256.

The relic-toolkit supports Koblitz curves over prime as well as binary fields. We have experimented with Koblitz curves over binary fields only. We do not run our experiments with all the curves available in the library since the aim of this work is not prove which curves perform the fastest, and rather show that asymmetric crypto is possible on resource-constrained devices.

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)	261	2,804	1024
NIST K163	932	2750	1024
NIST B163	2243	2444	1024
NIST K233	1736	3675	2048
NIST B233	4471	3261	2048

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)	592	2087	1024
NIST K163	2950	2215	1024
NIST B163	3213	2071	1024
NIST K233	6450	2935	2048
NIST B233	6100	2737	2048

ECDSA signature performance with relic-toolkit (Low Memory)

Table 6

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

- As with the RSA measurements, curves giving less that 112-bit security are insecure and considered as legacy. The measurements are only provided for reference.

- 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.

At the time of performing these measurements and study, it was unclear which exact elliptic curve(s) would be selected by the IETF community for use with resource-constrained devices. However now, [RFC7748] defines two elliptic curves over prime fields (Curve25519 and Curve448) that offer a high level of practical security for Diffie-Hellman key exchange. Correspondingly, there is ongoing work to specify elliptic curve signature schemes with Edwards-curve Digital Signature Algorithm (EdDSA). [I-D.irtf-cfrg-eddsa] specifies the recommended parameters for the edwards25519 and edwards448 curves. From these, curve25519 (for elliptic curve Diffie-Hellman key exchange) and edwards25519 (for elliptic curve digital signatures) are especially suitable for resource-constrained devices.

We found that the NaCl [nacl] and MicoNaCl [micronacl] libraries provide highly efficient implementations of Diffie-Hellman key exchange with curve25519. The results have shown that these libraries with curve25519 outperform other elliptic curves that provide similar levels of security. Hutter and Schwabe [naclavr] also show that signing of data using the curve Ed25519 from the NaCl library needs only 23,216,241 cycles on the same microcontroller that we used for our evaluations (Arduino Mega ATmega2560). This corresponds to about 1,4510 milliseconds of execution time. When compared to the results for other curves and libraries that offer similar level of security (such as NIST B233, NIST K233), this implementation far outperforms all others. As such, it is recommend that the IETF community uses these curves for protocol specification and implementations.

A summary library ROM use is shown in Table 7.

Library	ROM Footprint (Kilobytes)
AvrCryptolib	3.6
Wiselib	16
TinyECC	18
Relic-toolkit	29
NaCl Ed25519 [naclavr]	17-29

Summary of library ROM needs

Table 7

All the measurements here are only provided as an example to show that asymmetric-key cryptography (particularly, digital signatures) is possible on resource-constrained devices. These numbers by no way are the final source for measurements and some curves presented here may not be acceptable for real in-the-wild deployments anymore. For example, Mosdorf et al. [mosdorf] and Liu et al. [tinyecc] also document performance of ECDSA on similar resource-constrained devices.

9. 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.ietf-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 agrees 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 [RFC6690].

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 [RFC7515] [I-D.jennings-core-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.

To verify the feasibility of our architecture we developed a proof-of-concept prototype. In our prototype, the sensor was implemented using the Arduino Ethernet shield over an Arduino Mega board. Our implementation uses the standard C99 programming language on the

Arduino Mega board. In this prototype, the Mirror Proxy (MP) and the Resource Directory (RD) reside on the same physical host. A 64-bit x86 linux machine serves as the MP and the RD, while a similar but physically different 64-bit x86 linux machine serves as the client that requests data from the sensor. We chose the Relic library version 0.3.1 for our sample prototype as it can be easily compiled for different bit-length processors. Therefore, we were able to use it on the 8-bit processor of the Arduino Mega, as well as on the 64-bit processor of the x86 client. We used ECDSA to sign and verify data updates with the standard NIST-K163 curve parameters (163-bit Koblitz curve over binary field). While compiling Relic for our prototype, we used the fast configuration without any assembly optimizations.

The gateway implements the CoAP base specification in the Java programming language and extends it to add support for Mirror Proxy and Resource Directory REST interfaces. We also developed a minimalistic CoAP C-library for the Arduino sensor and for the client requesting data updates for a resource. The library has small SRAM requirements and uses stack-based allocation only. It is inter-operable with the Java implementation of CoAP running on the gateway. The location of the mirror proxy was pre-configured into the smart object sensor by hardcoding the IP address. We used an IPv4 network with public IP addresses obtained from a DHCP server.

Some important statistics of this prototype are listed in table Table 8. Our straw man analysis of the performance of this prototype is preliminary. Our intention was to demonstrate the feasibility of the entire architecture with public-key cryptography on an 8-bit microcontroller. The stated values can be improved further by a considerable amount. For example, the flash memory and SRAM consumption is relatively high because some of the Arduino libraries were used out-of-the- box and there are several functions which can be removed. Similarly we used the fast version of the Relic library in the prototype instead of the low memory version.

Flash memory consumption (for the entire prototype including Relic crypto + CoAP + Arduino UDP, Ethernet and DHCP Libraries)	51 kB
SRAM consumption (for the entire prototype including DHCP client + key generation + signing the hash of message + COAP + UDP + Ethernet)	4678 bytes
Execution time for creating the key pair + sending registration message + time spent waiting for acknowledgement	2030 ms
Execution time for signing the hash of message + sending update	987 ms
Signature overhead	42 bytes

Prototype Performance

Table 8

To demonstrate the efficacy of this communication model we compare it with a scenario where the smart objects do not transition into the energy saving sleep mode and directly serve temperature data to clients. As an example, we assume that in our architecture, the smart objects wake up once every minute to report the signed temperature data to the caching MP. If we calculate the energy consumption using the formula W = U * I * t (where U is the operating voltage, I is the current drawn and t is the execution time), and use the voltage and current values from the datasheets of the ATmega2560 (20mA-active mode and 5.4mA-sleep mode) and W5100 (183mA) chips used in the architecture, then in a one minute period, the Arduino board would consume 60.9 Joules of energy if it directly serves data and does not sleep. On the other hand, in our architecture it would only consume 2.6 Joules if it wakes up once a minute to update the MP with signed data. Therefore, a typical Li-ion battery that provides about 1800 milliamps per hour (mAh) at 5V would have a lifetime of 9 hours in the unsecured always-on scenario, whereas it would have a lifetime of about 8.5 days in the secured sleepy architecture presented. These lifetimes appear to be low because the Arduino board in the prototype uses Ethernet which is not energy efficient. The values presented only provide an estimate (ignoring the energy required to transition in and out of the sleep mode) and would vary depending on the hardware and MAC protocol used. Nonetheless, it is evident that

our architecture can increase the life of smart objects by allowing them to sleep and can ensure security at the same time.

10. 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.

11. 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.

12. Freshness

In our architecture, if implemented as described thus far, messages along with their signatures sent from the sensors to the mirror proxy can be recorded and replayed by an eavesdropper. The mirror proxy has no mechanism to distinguish previously received packets from those that are retransmitted by the sender or replayed by an eavesdropper. Therefore, it is essential for the smart objects to ensure that data updates include a freshness indicator. However, ensuring freshness on constrained devices can be non-trivial because of several reasons which include:

- Communication is mostly unidirectional to save energy.

- Internal clocks might not be accurate and may be reset several times during the operational phase of the smart object.

- Network time synchronization protocols such as Network Time Protocol (NTP) [RFC5905] are resource intensive and therefore may be undesirable in many smart object networks.

There are several different methods that can be used in our architecture for replay protection. The selection of the appropriate choice depends on the actual deployment scenario.

Including sequence numbers in signed messages can provide an effective method of replay protection. The mirror proxy should verify the sequence number of each incoming message and accept it only if it is greater than the highest previously seen sequence number. The mirror proxy drops any packet with a sequence number

that has already been received or if the received sequence number is greater than the highest previously seen sequence number by an amount larger than the preset threshold.

Sequence numbers can wrap-around at their maximum value and, therefore, it is essential to ensure that sequence numbers are sufficiently long. However, including long sequence numbers in packets can increase the network traffic originating from the sensor and can thus decrease its energy efficiency. To overcome the problem of long sequence numbers, we can use a scheme similar to that of Huang [huang], where the sender and receiver maintain and sign long sequence numbers of equal bit-lengths but they transmit only the least significant bits.

It is important for the smart object to write the sequence number into the permanent flash memory after each increment and before it is included in the message to be transmitted. This ensures that the sensor can obtain the last sequence number it had intended to send in case of a reset or a power failure. However, the sensor and the mirror proxy can still end up in a discordant state where the sequence number received by the mirror proxy exceeds the expected sequence number by an amount greater than the preset threshold. This may happen because of a prolonged network outage or if the mirror proxy experiences a power failure for some reason. Therefore it is essential for sensors that normally send Non-Confirmable data updates to send some Confirmable updates and re-synchronize with the mirror proxy if a reset message is received. The sensors re-synchronize by sending a new registration message with the current sequence number.

Although sequence numbers protect the system from replay attacks, a mirror proxy has no mechanism to determine the time at which updates were created by the sensor. Moreover, if sequence numbers are the only freshness indicator used, a malicious eavesdropper can induce inordinate delays to the communication of signed updates by buffering messages. It may be important in certain smart object networks for sensors to send data updates which include timestamps to allow the mirror proxy to determine the time when the update was created. For example, when the mirror proxy is collecting temperature data, it may be necessary to know when exactly the temperature measurement was made by the sensor. A simple solution to this problem is for the mirror proxy to assume that the data object was created when it receives the update. In a relatively reliable network with low RTT, it can be acceptable to make such an assumption. However most networks are susceptible to packet loss and hostile attacks making this assumption unsustainable.

Depending on the hardware used by the smart objects, they may have access to accurate hardware clocks which can be used to include

timestamps in the signed updates. These timestamps are included in addition to sequence numbers. The clock time in the smart objects can be set by the manufacturer or the current time can be communicated by the mirror proxy during the registration phase. However, these approaches require the smart objects to either rely on the long-term accuracy of the clock set by the manufacturer or to trust the mirror proxy thereby increasing the potential vulnerability of the system. The smart objects could also obtain the current time from NTP, but this may consume additional energy and give rise to security issues discussed in [RFC5905]. The smart objects could also have access to a GSM network or the Global Positioning System (GPS), and they can be used obtain the current time. Finally, if the sensors need to co-ordinate their sleep cycles, or if the mirror proxy computes an average or mean of updates collected from multiple smart objects, it is important for the network nodes to synchronize the time among them. This can be done by using existing synchronization schemes.

13. 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. TLS with HTTP (HTTPS) and DTLS with CoAP 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.

14. 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, one of 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 8, 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.

15. Security Considerations

This entire memo deals with security issues.

16. IANA Considerations

There are no IANA impacts in this memo.

17. Informative references

[arduino-uno]
Arduino, "Arduino Uno", September 2015,
<http://arduino.cc/en/Main/arduinoBoardUno>.

[avr-crypto-lib]
AVR-CRYPTO-LIB, "AVR-CRYPTO-LIB", September 2015, <http://www.das-labor.org/wiki/AVR-Crypto-Lib/en>.

[avr-cryptolib]
Van der Laan, E., "AVR CRYPTOLIB", September 2015, <http://www.emsign.nl/>.

[avrora] Titzer, Ben., "Avrora", September 2015, <http://compilers.cs.ucla.edu/avrora/>.

[huang] Huang, C., "Low-overhead freshness transmission in sensor networks", 2008.

[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.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.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.ietf-core-resource-directory]
Shelby, Z., Koster, M., Bormann, C., and P. Stok, "CoRE Resource Directory", draft-ietf-core-resource-directory-08 (work in progress), July 2016.

[I-D.irtf-cfrg-eddsa]
Josefsson, S. and I. Liusvaara, "Edwards-curve Digital Signature Algorithm (EdDSA)", draft-irtf-cfrg-eddsa-08 (work in progress), August 2016.

[I-D.irtf-t2trg-iot-seccons]
Garcia-Morchon, O., Kumar, S., and M. Sethi, "Security Considerations in the IP-based Internet of Things", draft-irtf-t2trg-iot-seccons-00 (work in progress), October 2016.

[I-D.jennings-core-senml]
Jennings, C., Shelby, Z., Arkko, J., and A. Keranen, "Media Types for Sensor Markup Language (SenML)", draft-jennings-core-senml-06 (work in progress), April 2016.

[I-D.moskowitz-hip-dex]
Moskowitz, R. and R. Hummen, "HIP Diet EXchange (DEX)", draft-moskowitz-hip-dex-05 (work in progress), January 2016.

[I-D.vial-core-mirror-proxy]
Vial, M., "CoRE Mirror Server", draft-vial-core-mirror-proxy-01 (work in progress), July 2012.

[matrix-ssl]
PeerSec Networks, "Matrix SSL", September 2015, <http://www.matrixssl.org/>.

[micronacl]
MicroNaCl, "The Networking and Cryptography library for microcontrollers", <http://munacl.cryptojedi.org/>.

[mosdorf] Mosdorf, M. and W. Zabolotny, "Implementation of elliptic curve cryptography for 8 bit and 32 bit embedded systems time efficiency and power consumption analysis", 2010.

[nacl] NaCl, "Networking and Cryptography library", <http://nacl.cr.yp.to/>.

[naclavr] Hutter, M. and P. Schwabe, "NaCl on 8-Bit AVR Microcontrollers", International Conference on Cryptology in Africa , Springer Berlin Heidelberg , 2013.

[relic-toolkit]
Aranha, D. and C. Gouv, "Relic Toolkit", September 2015, <http://code.google.com/p/relic-toolkit/>.

[RFC3748] Aboba, B., Blunk, L., Vollbrecht, J., Carlson, J., and H. Levkowetz, Ed., "Extensible Authentication Protocol (EAP)", RFC 3748, DOI 10.17487/RFC3748, June 2004, <http://www.rfc-editor.org/info/rfc3748>.

[RFC3972] Aura, T., "Cryptographically Generated Addresses (CGA)", RFC 3972, DOI 10.17487/RFC3972, March 2005, <http://www.rfc-editor.org/info/rfc3972>.

[RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <http://www.rfc-editor.org/info/rfc4303>.

[RFC5191] Forsberg, D., Ohba, Y., Ed., Patil, B., Tschofenig, H., and A. Yegin, "Protocol for Carrying Authentication for Network Access (PANA)", RFC 5191, DOI 10.17487/RFC5191, May 2008, <http://www.rfc-editor.org/info/rfc5191>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc5246>.

[RFC5406] Bellovin, S., "Guidelines for Specifying the Use of IPsec Version 2", BCP 146, RFC 5406, DOI 10.17487/RFC5406, February 2009, <http://www.rfc-editor.org/info/rfc5406>.

[RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, DOI 10.17487/RFC5905, June 2010, <http://www.rfc-editor.org/info/rfc5905>.

[RFC6078] Camarillo, G. and J. Melen, "Host Identity Protocol (HIP) Immediate Carriage and Conveyance of Upper-Layer Protocol Signaling (HICCUPS)", RFC 6078, DOI 10.17487/RFC6078, January 2011, <http://www.rfc-editor.org/info/rfc6078>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <http://www.rfc-editor.org/info/rfc6347>.

[RFC6574] Tschofenig, H. and J. Arkko, "Report from the Smart Object Workshop", RFC 6574, DOI 10.17487/RFC6574, April 2012, <http://www.rfc-editor.org/info/rfc6574>.

[RFC6690] Shelby, Z., "Constrained RESTful Environments (CoRE) Link Format", RFC 6690, DOI 10.17487/RFC6690, August 2012, <http://www.rfc-editor.org/info/rfc6690>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <http://www.rfc-editor.org/info/rfc7252>.

[RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October 2014, <http://www.rfc-editor.org/info/rfc7296>.

[RFC7401] Moskowitz, R., Ed., Heer, T., Jokela, P., and T. Henderson, "Host Identity Protocol Version 2 (HIPv2)", RFC 7401, DOI 10.17487/RFC7401, April 2015, <http://www.rfc-editor.org/info/rfc7401>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <http://www.rfc-editor.org/info/rfc7515>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <http://www.rfc-editor.org/info/rfc7748>.

[RFC7815] Kivinen, T., "Minimal Internet Key Exchange Version 2 (IKEv2) Initiator Implementation", RFC 7815, DOI 10.17487/RFC7815, March 2016, <http://www.rfc-editor.org/info/rfc7815>.

[rsa-8bit] Gura, N., Patel, A., Wander, A., Eberle, H., and S. Shantz, "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs", 2010.

[rsa-high-speed] Koc, C., "High-Speed RSA Implementation", November 1994, <http://cs.ucsb.edu/~koc/docs/r01.pdf>.

[tinyecc] North Carolina State University and North Carolina State University, "TinyECC", 2008, <http://discovery.csc.ncsu.edu/software/TinyECC/>.

[truerandom] Drow, C., "Truerandom", September 2015, <http://code.google.com/p/tinkerit/wiki/TrueRandom>.

[wiman] Margi, C., Oliveira, B., Sousa, G., Simplicio, M., Paulo, S., Carvalho, T., Naslund, M., and R. Gold, "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, Zurich. Proceedings of ICCCN'2010/WiMAN'2010", 2010.

[wiselib] Baumgartner, T., Chatzigiannakis, I., Fekete, S., Koninis, C., Kroller, A., and A. Pyrgelis, "Wiselib", 2010, <www.wiselib.org/>.

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

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
Ericsson
Jorvas 02420
Finland

EMail: mohit@piuha.net

Jari Arkko
Ericsson
Jorvas 02420
Finland

EMail: jari.arkko@piuha.net

Ari Keranen
Ericsson
Jorvas 02420
Finland

EMail: ari.keranen@ericsson.com

Heidi-Maria Back
Comptel
Helsinki 00181
Finland

EMail: heidi.back@comptel.com