           Practical Considerations and Implementation Experiences in
                        Securing Smart Object Networks

Abstract

   This memo describes challenges associated with securing resource-
   constrained smart object devices. The memo describes a possible
   deployment model where resource-constrained devices sign message
   objects, discusses the availability of cryptographic libraries for
   resource-constrained devices, and presents some preliminary
   experiences with those libraries for message signing on resource-
   constrained devices. Lastly, the memo discusses trade-offs involving
   different types of security approaches.

Status of This Memo

Copyright Notice

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.

   Section 4 discusses a potential deployment model for constrained
   environments. The model requires a minimal amount of configuration,
   and we believe it is a natural fit with the typical communication
   practices in smart object networking environments.

   Section 5 discusses the availability of cryptographic libraries.
   Section 6 presents some experiences in implementing cryptography on
   resource-constrained devices using those libraries, including
   information about achievable code sizes and speeds on typical
   hardware. Section 7 describes an example proof-of-concept prototype
   implementation that uses public-key cryptography on resource-
   constrained devices to provide end-to-end data authenticity and
   integrity protection.

   Finally, Section 8 discusses trade-offs involving different types of
   security approaches.

2. Related Work

   The Constrained Application Protocol (CoAP) [RFC7252] is a
   lightweight 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. The 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 certificates. The security model in all
   cases is mutual authentication, so while there is some commonality to
   HTTP [RFC7230] in verifying the server identity, in practice the
   models are quite different. The use of IPsec with CoAP is described
   with regards to the protocol requirements, noting that lightweight
   implementations of the Internet Key Exchange Protocol Version 2
   (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].

   [IoT-SECURITY] documents the different stages in the life cycle of a
   smart object. Next, it highlights the security threats for smart
   objects and the challenges that one might face to protect against

these threats. The document also looks at various security protocols
available, including IKEv2/IPsec [RFC7296], TLS/SSL [RFC5246], DTLS
[RFC6347], the Host Identity Protocol (HIP) [RFC7401], HIP Diet
EXchange [HIP-DEX], a Protocol for Carrying Authentication for
Network Access (PANA) [RFC5191], and the Extensible Authentication
Protocol (EAP) [RFC3748]. Lastly, [IoT-BOOTSTRAPPING] discusses
bootstrapping mechanisms available for resource-constrained Internet
of Things (IoT) devices.

[RFC6574] gives an overview of the security discussions at the March
2011 IAB workshop on smart objects. The workshop recommended that
additional work should be undertaken in developing suitable
credential management mechanisms (perhaps something similar to the
Bluetooth pairing mechanism), understanding the implementability of
standard security mechanisms in resource-constrained devices, and
conducting additional research in the area of lightweight
cryptographic primitives.

[HIP-DEX] defines a lightweight 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 all 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.

[IPV6-LOWPAN-SEC] makes a comprehensive analysis of security issues
related to IPv6 over Low-Power Wireless Personal Area Network
(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.

3. Challenges

   This section discusses three challenges: 1) implementation
   difficulties, 2) practical provisioning problems, and 3) layering and
   communication models.

   One of the most often discussed issues in the security for the
   Internet of Things relate to implementation difficulties. The desire
   to build resource-constrained, 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, a 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. It is important for developers
and product designers to determine what security threats they want to
tackle and the resulting security requirements before selecting the
hardware. Often, development work in the wild happens in the wrong
order: a particular platform with a resource-constrained
microcontroller is chosen first, and then the security features that
can fit on it are decided. Also, the most lightweight algorithms and
cryptographic primitives are useful but should not be the only
consideration in the design and development. 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.
This is perhaps the most fundamental and difficult issue and is
unfortunately often neglected in the design. There are several
problems in the provisioning and management of smart object networks:

o Resource-constrained 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 or display, and there may not even be buttons
   to press. Some devices may only have one interface, the interface
   to the network.

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

o 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.

o Smart object networks may rely on different radio technologies.
   Provisioning methods that rely on specific link-layer features may
   not work with other radio technologies in a heterogeneous network.

   o 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.

   In light of the above challenges, resource-constrained devices are
   often shipped with a single static identity. In many cases, it is a
   single raw public key. These long-term static identities makes it
   easy to track the devices (and their owners) when they move. The
   static identities may also allow an attacker to track these devices
   across ownership changes.

   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 [CoAP-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 relate 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

   [CoAP-SECURITY] recognizes the provisioning model as the driver of
   what kind of security architecture is useful. This section
   reintroduces 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. "|" (vertical bar) here denotes the
   concatenation operator.

4.1. 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
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 a large number of devices to a 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; however, 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 a separate user interface, a local
connection (such as USB), or use of 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
certificate or 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.

Once both the parties interested in communicating know the expected
cryptographic identity of the other offline, 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 identity of both parties, there
are some remaining vulnerabilities that may or may not be acceptable
for the application in question. For example, many pairing methods
based on "leap of faith" or "trust on first use" assume that the
attacker is not present during the initial setup. Therefore, they
are vulnerable to eavesdropping or man-in-the-middle (MitM) attacks.

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, shared secrets, or,
e.g., 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. One simple but not completely secure method is 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 (RFID) tags. 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 needed for provisioning security.

As discussed in the previous section, long-term static identities negatively affect the privacy of the devices and their owners. Therefore, it is beneficial for devices to generate new identities at appropriate times during their life cycle; an example is after a factory reset or an ownership handover. Thus, in our proposed deployment model, the devices would generate a new asymmetric key pair and use the new public-key P' to generate the new identity I'. It is also desirable that these identities are only used during the provisioning stage. Temporary identities (such as dynamic IPv6

addresses) can be used for network communication protocols once the
device is operational.

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, Potherdevi are the public keys of other
devices in the group, n is all the devices in the group except the
device with Pdev as its public key, and m is the total number of
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 cannot claim to be in the group, because the
group identity would change if any new keys were added to the
identity of the group (Igrp).

4.2. 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 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. [CoAP-SECURITY] 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, and more specifically, in the data formats transported in the payload part of CoAP. This approach provides the following benefits:

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

o 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.

o 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, it is generally not the typical arrangement.

5. Code Availability

For implementing public-key cryptography on resource-constrained environments, we chose the 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 RAM, and 32 kB of flash memory. Our choice of an 8-bit platform may seem surprising since cheaper and more energy-efficient 32-bit platforms are available. However, our intention was to evaluate the performance of public-key cryptography on the most resource-constrained platforms available. It is reasonable to expect better performance results from 32-bit microcontrollers.

For selecting potential asymmetric cryptographic libraries, we surveyed 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. It is advisable to 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:

o AVRCryptoLib [avr-cryptolib]: This library provides symmetric key
   algorithms such as AES. It provides RSA as an asymmetric key
   algorithm. Parts of the library were written in AVR 8-bit
   assembly language to reduce the size and optimize the performance.

o 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-Shacham signatures, and Boneh-Boyen short
   signatures. The library has also added support for curve25519
   (for Elliptic Curve Diffie-Hellman key exchange) [RFC7748] and
   edwards25519 (for elliptic curve digital signatures) [RFC8032].
   The toolkit provides an option to build only the desired
   components for the required platform.

o TinyECC [tinyecc]: TinyECC was designed for using elliptic-curve-
   based public-key cryptography on sensor networks. It is written
   in the nesC programming language [nesC] and as such is designed
   for specific use on TinyOS. However, the library can be ported to
   standard C either with tool chains or by manually rewriting parts
   of the code. It also has one of the smallest memory footprints
   among the set of elliptic curve libraries surveyed so far.

o 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 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 to test the code on Arduino Uno.

o MatrixSSL [matrix-ssl]: This library provides a low footprint
   implementation of several cryptographic algorithms including RSA
   and ECC (with a commercial license). The library in the original
   form takes about 50 kB of ROM and is intended for 32-bit
   platforms.

This is by no means an exhaustive list, and there exists other
cryptographic libraries targeting resource-constrained devices.

There are also a number of operating systems that are specifically
targeted for resource-constrained devices. These operating systems
may include libraries and code for security. Hahm et al. [hahmos]
conducted a survey of such operating systems. The ARM Mbed OS [mbed]
is one such operating system that provides various cryptographic
primitives that are necessary for SSL/TLS protocol implementation as
well as X509 certificate handling. The library provides an API for
developers with a minimal code footprint. It is intended for various
ARM platforms such as ARM Cortex M0, ARM Cortex M0+, and ARM Cortex
M3.

6. 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 7 required only the
   signature generation operation on the resource-constrained platforms.
   We have summarized the initial results of RSA private-key
   exponentiation performance using AVRCryptoLib [avr-crypto-lib] 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
   execution time for a key size of 2048 bits was inordinately long and
   would be a deterrent in real-world deployments.

   +---------------+-------------------------+----------------------------+
   | Key length | Execution time (ms); | Memory footprint (bytes); |
   | (bits) | key in RAM | key in RAM |
   +---------------+-------------------------+----------------------------+
   | 2048 | 1587567 | 1280 |
   +---------------+-------------------------+----------------------------+

                Table 1: RSA Private-Key Operation Performance

   The code size was about 3.6 kB with potential 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 RAM, we believe that 2048-bit operations can be performed in
   much less time as has been shown in [rsa-8bit].

   In Table 2, we present the results obtained by manually porting
   TinyECC into the C99 standard and running the Elliptic Curve Digital
   Signature Algorithm (ECDSA) on the Arduino Uno board. TinyECC
   supports a variety of SEC-2-recommended elliptic curve domain
   parameters [sec2ecc]. The execution time and memory footprint are

shown next to each of the curve parameters. These results were
obtained by turning on all the optimizations and using assembly code
where available.

The results from the performance evaluation of ECDSA in the following
tables also contain a column stating the approximate comparable RSA
key length as documented in [sec2ecc]. It is clearly observable that
for similar security levels, elliptic curve public-key cryptography
outperforms RSA.

Curve parameters	Execution time (ms)	Memory footprint (bytes)	Comparable RSA key length
secp160k1	2228	892	1024
secp160r1	2250	892	1024
secp160r2	2467	892	1024
secp192k1	3425	1008	1536
secp192r1	3578	1008	1536

          Table 2: Performance of ECDSA Sign Operation with TinyECC

We also performed experiments by removing the assembly optimization
and 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 is
available. The memory footprint remains the same with or without
assembly code. The tables contain the maximum RAM that is used when
all the possible optimizations are on. However, if 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
secp160k1	3795	892	1024
secp160r1	3841	892	1024
secp160r2	4118	892	1024
secp192k1	6091	1008	1536
secp192r1	6217	1008	1536

          Table 3: Performance of ECDSA Sign Operation with TinyECC
                      (No Assembly Optimizations)

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 all the 128-, 160-, and 192-bit SEC 2 elliptic
curves. The ROM size for all the experiments was less than 16 kB.

Curve parameters	Execution time (ms)	Memory footprint (bytes)	Comparable RSA key length
secp160k1	10957	842	1024
secp160r1	10972	842	1024
secp160r2	10971	842	1024
secp192k1	18814	952	1536
secp192r1	18825	952	1536

Table 4: Performance ECDSA Sign Operation with Wiselib

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. Arduino Mega has the same 8-bit architecture as
Arduino Uno, but it has a much larger RAM/ROM. We used Arduino Mega
for experimenting with the relic-toolkit. Again, it is important to
mention that we used Arduino as it is a convenient prototyping
platform. Our intention was to demonstrate the feasibility of the
entire architecture with public-key cryptography on an 8-bit
microcontroller. However, it is important to state that 32-bit
microcontrollers are much more easily available, at lower costs, and
are more power efficient. Therefore, real deployments are better off
using 32-bit microcontrollers that allow developers to include the
necessary cryptographic libraries. There is no good reason to choose
platforms that do not provide sufficient computing power to run the
necessary cryptographic operations.

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 to prove
which curves perform the fastest but rather to show that asymmetric
cryptography is possible on resource-constrained devices.

The results from relic-toolkit are documented separately in Tables 5
and 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
sect163k1 (assembly math)	261	2804	1024
sect163k1	932	2750	1024
sect163r2	2243	2444	1024
sect233k1	1736	3675	2048
sect233r1	4471	3261	2048

Table 5: Performance of ECDSA Sign Operation with
relic-toolkit (Fast)

Curve parameters	Execution time (ms)	Memory footprint (bytes)	Comparable RSA key length
sect163k1 (assembly math)	592	2087	1024
sect163k1	2950	2215	1024
sect163r2	3213	2071	1024
sect233k1	6450	2935	2048
sect233r1	6100	2737	2048

Table 6: Performance of ECDSA Sign Operation with relic-toolkit
(Low Memory)

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

o Some boards (e.g., Arduino Uno) do not provide a hardware random
   number generator. On such boards, obtaining cryptographic-quality
   randomness is a challenge. Real-world deployments must rely on a
   hardware random number generator for cryptographic operations such
   as generating a public-private key pair. The Nordic nRF52832
   board [nordic], for example, provides a hardware random number
   generator. A detailed discussion on requirements and best
   practices for cryptographic-quality randomness is documented in
   [RFC4086]

o 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.

Tschofenig and Pegourie-Gonnard [armecdsa] have also evaluated the
performance of ECC on an ARM Coretex platform. The results for the
ECDSA sign operation shown in Table 7 are performed on a Freescale
FRDM-KL25Z board [freescale] that has an ARM Cortex-M0+ 48MHz
microcontroller with 128 kB of flash memory and 16 kB of RAM. The
sliding window technique for efficient exponentiation was used with a
window size of 2. All other optimizations were disabled for these
measurements.

Curve parameters	Execution time (ms)	Comparable RSA key length
secp192r1	2165	1536
secp224r1	3014	2048
secp256r1	3649	2048

       Table 7: Performance of ECDSA Sign Operation with an ARM Mbed TLS
                      Stack on Freescale FRDM-KL25Z

Tschofenig and Pegourie-Gonnard [armecdsa] also measured the
performance of curves on an ST Nucleo F091 (STM32F091RCT6) board
[stnucleo] that has an ARM Cortex-M0 48 MHz microcontroller with 256
kB of flash memory and 32 kB of RAM. The execution time for the
ECDSA sign operation with different curves is shown in Table 8. The
sliding window technique for efficient exponentiation was used with a
window size of 7. Fixed-point optimization and NIST curve-specific
optimizations were used for these measurements.

+--------------------+----------------------+---------------------------+
| Curve parameters | Execution time (ms) | Comparable RSA key |
| | | length |
+--------------------+----------------------+---------------------------+
secp192k1	291	1536
secp192r1	225	1536
secp224k1	375	2048
secp224r1	307	2048
secp256k1	486	2048
secp256r1	459	2048
secp384r1	811	7680
secp521r1	1602	15360
+--------------------+----------------------+---------------------------+

       Table 8: ECDSA Signature Performance with an ARM Mbed TLS Stack on ST
                      Nucleo F091 (STM32F091RCT6)

Finally, Tschofenig and Pegourie-Gonnard [armecdsa] also measured the
RAM consumption by calculating the heap consumed for the
cryptographic operations using a custom memory allocation handler.
They did not measure the minimal stack memory consumption. Depending
on the curve and the different optimizations enable or disabled, the
memory consumption for the ECDSA sign operation varied from 1500
bytes to 15000 bytes.

At the time of performing these measurements and this study, it was
unclear which exact elliptic curve(s) would be selected by the IETF
community for use with resource-constrained devices. However,
[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). [RFC8032] 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 the signing of data using the curve Ed25519 from the
NaCl library needs only 23216241 cycles on the same microcontroller
that we used for our evaluations (Arduino Mega ATmega2560). This
corresponds to about 1451 milliseconds of execution time. When
compared to the results for other curves and libraries that offer a

similar level of security (such as sect233r1 and sect233k1), this
implementation far outperforms all others. As such, it is
recommended that the IETF community use these curves for protocol
specification and implementations.

A summary library flash memory use is shown in Table 9.

Library	Flash memory footprint (kilobytes)
AVRCryptoLib	3.6
Wiselib	16
TinyECC	18
Relic-toolkit	29
NaCl Ed25519 [naclavr]	17-29

Table 9: Summary of Library Flash Memory Consumption

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. By no means are these
numbers the final source for measurements, and some curves presented
here may no longer be acceptable for real in-the-wild deployments.
For example, Mosdorf et al. [mosdorf] and Liu et al. [tinyecc] also
document the performance of ECDSA on similar resource-constrained
devices.

7. Example Application

We developed an example application on the Arduino platform to use
public-key cryptography, 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 stay on at all times. They wake up
periodically and transmit their readings.

Such sensor nodes can be supported in various ways. [CoAP-SENSORS]
was an early multicast-based approach. In the current application,
we have switched to using resource directories [CoRE-RD] and publish-
subscribe brokers [CoAP-BROKER] 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 publish-subscribe
brokers, 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 publish-subscribe broker and

periodically update the readings. All queries from the rest of the
world go to the publish-subscribe broker.

We constructed a system with four entities:

Sensor: This is an Arduino-based device that runs a CoAP publish-
   subscribe broker client and relic-toolkit. Relic takes 29 kB of
   flash memory, and the simple CoAP client takes roughly 3 kB.

Publish-Subscribe Broker: This is a publish-subscribe broker 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 publish-subscribe brokers 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
   publish-subscribe broker.

The security of this system relies on a secure-shell-like approach.
In Step 1, upon first boot, sensors generate keys and register
themselves in the publish-subscribe broker. 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 measurement, it sends an update to
the publish-subscribe broker and signs the message contents with a
JSON Object Signing and Encryption (JOSE) signature on the used JSON
[RFC7515] and Sensor Measurement List (SenML) payload [MT-SenML].
The sensor can also alternatively use CBOR Object Signing and
Encryption (COSE) [RFC8152] for signing the sensor measurement.

In Step 3, any other device in the network -- including the publish-
subscribe broker, 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 publish-subscribe
broker and the Resource Directory (RD) reside on the same physical
host. A 64-bit x86 Linux machine serves as the broker and the RD,
while a similar but physically distinct 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 sect163k1 curve
parameters. 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 publish-
subscribe broker and Resource Directory Representational State
Transfer (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 RAM requirements and
uses stack-based allocation only. It is interoperable with the Java
implementation of CoAP running on the gateway. The location of the
resource directory was configured into the smart object sensor by
hardcoding the IP address. A real implementation based on this
prototype would instead use the domain name system for obtaining the
location of the resource directory.

Our intention was to demonstrate that it is possible to implement 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 RAM
consumption is relatively high because some of the Arduino libraries
were used out of the box, and there are several functions that can be
removed. Similarly, we used the fast version of the Relic library in
the prototype instead of the low-memory version. However, it is
important to note that this was only a research prototype to verify
the feasibility of this architecture and, as stated elsewhere, most
modern development boards have a 32-bit microcontroller since they
are more economical and have better energy efficiency.

8. 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. In particular, this analysis looks at layering,
   freshness, and the choice of symmetric vs. asymmetric cryptography.

8.1. Feasibility

   The first question is whether using cryptographic security and
   asymmetric cryptography in particular is feasible at all on resource-
   constrained 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 a 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 resource constrained and difficult as possible.

   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. The flash memory size
   is probably easier to grow than other parameters in microcontrollers.
   Flash memory size is not expected to be the most significant limiting
   factor. Before picking a platform, developers should also plan for
   firmware updates. This would essentially mean that the platform
   should at least have a flash memory size of the total code size * 2,
   plus some space for buffer.

   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, or even spend
   multiple seconds in some cases, for the computation of a signature
   for the message that it is about to send. 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, these algorithms can successfully be
   employed in resource-constrained devices for the following reasons:

   o With the right selection of algorithms and libraries, the
      execution times can actually be very small (less than 500 ms).

o As discussed in [wiman], in general, the power requirements
   necessary to turn the radio on/off and sending or receiving
   messages are far bigger than those needed to execute cryptographic
   operations. While there are newer radios that significantly lower
   the energy consumption of sending and receiving messages, there is
   no good reason to choose platforms that do not provide sufficient
   computing power to run the necessary cryptographic operations.

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

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

While we did not do an exhaustive performance evaluation of
asymmetric key-pair generation on resource-constrained devices, we
did note that it is possible for such devices to generate a new key
pair. Given that this operation would only occur in rare
circumstances (such as a factory reset or ownership change) and its
potential privacy benefits, developers should provide mechanisms for
generating new identities. However, it is extremely important to
note that the security of this operation relies on access to
cryptographic-quality randomness.

8.2. Freshness

In our architecture, if implemented as described thus far, messages
along with their signatures sent from the sensors to the publish-
subscribe broker can be recorded and replayed by an eavesdropper.
The publish-subscribe broker 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:

o Communication is mostly unidirectional to save energy.

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

o Network time synchronization protocols such as the 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 publish-subscribe broker
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 publish-subscribe broker 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; 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
publish-subscribe broker can still end up in a discordant state where
the sequence number received by the publish-subscribe broker 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 publish-subscribe broker 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
resynchronize with the publish-subscribe broker if a reset message is
received. The sensors resynchronize by sending a new registration
message with the current sequence number.

Although sequence numbers protect the system from replay attacks, a
publish-subscribe broker 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 that include
timestamps to allow the publish-subscribe broker to determine the
time when the update was created. For example, when the publish-

subscribe broker 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 publish-
subscribe broker 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 publish-subscribe broker 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 trust the publish-subscribe broker 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 mobile network or the
Global Positioning System (GPS), and they can be used obtain the
current time. Finally, if the sensors need to coordinate their sleep
cycles, or if the publish-subscribe broker 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.

8.3. 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
   primary 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 smart home weighing
   scale, 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 the link layer, network layer, and in some cases, transport
   layer, cannot 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. 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.

8.4. 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
cryptographic solution can be deployed in large scale. In fact, one
of the largest deployments of cryptographic security, the cellular
network authentication system, uses Subscriber Identification Module
(SIM) cards that are based on symmetric secrets. In contrast,
public-key systems have yet to show an 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 Section 6, the time needed to calculate some of
the asymmetric cryptographic 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 resource-constrained devices is far cheaper than transmission
power [wiman], and it 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 it works less
well for data-object security, particularly when the number of
communicating entities is not exactly two.

9. Summary

   This document makes several security recommendations based on our
   implementation experience. We summarize some of the important ones
   here:

   o Developers and product designers should choose the hardware after
      determining the security requirements for their application
      scenario.

   o ECC outperforms RSA-based operations; therefore, it is recommended
      for resource-constrained devices.

   o Cryptographic-quality randomness is needed for many security
      protocols. Developers and vendors should ensure that the
      sufficient randomness is available for security critical tasks.

   o 32-bit microcontrollers are much more easily available, at lower
      costs, and are more power efficient. Therefore, real-world
      deployments are better off using 32-bit microcontrollers.

   o Developers should provide mechanisms for devices to generate new
      identities at appropriate times during their life cycle, for
      example, after a factory reset or an ownership handover.

   o Planning for firmware updates is important. The hardware platform
      chosen should at least have a flash memory size of the total code
      size * 2, plus some space for buffer.

10. Security Considerations

   This entire memo deals with security issues.

11. IANA Considerations

   This document has no IANA actions.

12. Informative References

   [arduino-uno]
             Arduino, "Arduino Uno REV3",
             <http://arduino.cc/en/Main/arduinoBoardUno>.

   [armecdsa] Tschofenig, H. and M. Pegourie-Gonnard, "Performance
             Investigations", March 2015,
             <https://www.ietf.org/proceedings/92/slides/
             slides-92-lwig-3.pdf>.

   [avr-crypto-lib]
             Das Labor, "AVR-Crypto-Lib", February 2014,
             <http://www.das-labor.org/wiki/AVR-Crypto-Lib/en>.

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

   [avrora] Avora, "The AVR Simulation and Analysis Framework",
             <http://compilers.cs.ucla.edu/avrora/>.

   [CoAP-BROKER]
             Koster, M., Keranen, A., and J. Jimenez, "Publish-
             Subscribe Broker for the Constrained Application Protocol
             (CoAP)", Work in Progress, draft-ietf-core-coap-pubsub-04,
             March 2018.

   [CoAP-SECURITY]
             Arkko, J. and A. Keranen, "CoAP Security Architecture",
             Work n Progress, draft-arkko-core-security-arch-00, July
             2011.

   [CoAP-SENSORS]
             Arkko, J., Rissanen, H., Loreto, S., Turanyi, Z., and O.
             Novo, "Implementing Tiny COAP Sensors", Wok in Progress,
             draft-arkko-core-sleepy-sensors-01, July 2011.

   [CoRE-RD] Shelby, Z., Koster, M., Bormann, C., Stok, P., and C.
             Amsuess, "CoRE Resource Directory", Work in Progress,
             draft-ietf-core-resource-directory-13, March 2018.

   [freescale]
             ARM Mbed, "FRDM-KL25Z",
             <https://developer.mbed.org/platforms/KL25Z/>.

   [hahmos] Hahm, O., Baccelli, E., Petersen, H., and N. Tsiftes,
             "Operating systems for low-end devices in the internet of
             things: a survey", IEEE Internet of Things Journal,
             Vol. 3, Issue 5, DOI 10.1109/JIOT.2015.2505901, October
             2016.

   [HIP-DEX] Moskowitz, R., Ed. and R. Hummen, "HIP Diet EXchange
             (DEX)", Work in Progress, draft-ietf-hip-dex-06, December
             2017.

   [huang]   Huang, C., "LOFT: Low-overhead freshness transmission in
             sensor networks", IEEE, DOI 10.1109/SUTC.2008.38, June
             2008.

[IoT-BOOTSTRAPPING]
          Sarikaya, B., Sethi, M., and A. Sangi, "Secure IoT
          Bootstrapping: A Survey", Work in Progress,
          draft-sarikaya-t2trg-sbootstrapping-03, February 2017.

[IoT-SECURITY]
          Garcia-Morchon, O., Kumar, S., and M. Sethi,
          "State-of-the-Art and Challenges for the Internet of
          Things Security", Work in Progress,
          draft-irtf-t2trg-iot-seccons-14, April 2018.

[IPV6-LOWPAN-SEC]
          Park, S., Kim, K., Haddad, W., Chakrabarti, S., and J.
          Laganier, "IPv6 over Low Power WPAN Security Analysis",
          Work in Progress, draft-daniel-6lowpan-security-
          analysis-05, March 2011.

[matrix-ssl]
          Inside Secure, "GUARD TLS Toolkit (formerly Matrix SSL)",
          <http://www.matrixssl.org/>.

[mbed]   ARM Mbed, "Mbed TLS",
          <https://www.mbed.com/en/technologies/security/mbed-tls/>.

[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", Pomiary
          Automatyka Kontrola, 2010.

[MT-SenML] Jennings, C., Shelby, Z., Arkko, J., Keranen, A., and C.
          Bormann, "Sensor Measurement Lists (SenML)", Work in
          Progress, draft-ietf-core-senml-15, May 2018.

[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, Computer Science, Vol. 7918, pp.
          156-172, February 2013,
          <https://doi.org/10.1007/978-3-642-38553-7_9>.

   [nesC]   Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E.,
               and D. Culler, "The nesC language: A holistic approach to
               networked embedded systems", ACM SIGPLAN Notices, Vol. 38,
               Issue 5, DOI 10.1145/781131.781133, 2003.

   [nordic]   Nordic Semiconductor, "nRF52832 Product Specification
               v1.3", March 2017, <http://infocenter.nordicsemi.com/pdf/
               nRF52832_PS_v1.3.pdf>.

   [relic-toolkit]
               "relic", March 2017,
               <https://github.com/relic-toolkit/relic>.

   [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,
               <https://www.rfc-editor.org/info/rfc3748>.

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

   [RFC4086]   Eastlake 3rd, D., Schiller, J., and S. Crocker,
               "Randomness Requirements for Security", BCP 106, RFC 4086,
               DOI 10.17487/RFC4086, June 2005,
               <https://www.rfc-editor.org/info/rfc4086>.

   [RFC4303]   Kent, S., "IP Encapsulating Security Payload (ESP)",
               RFC 4303, DOI 10.17487/RFC4303, December 2005,
               <https://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, <https://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,
               <https://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, <https://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,
              <https://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, <https://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, <https://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,
              <https://www.rfc-editor.org/info/rfc6574>.

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

   [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer
              Protocol (HTTP/1.1): Message Syntax and Routing",
              RFC 7230, DOI 10.17487/RFC7230, June 2014,
              <https://www.rfc-editor.org/info/rfc7230>.

   [RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained
              Application Protocol (CoAP)", RFC 7252,
              DOI 10.17487/RFC7252, June 2014,
              <https://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, <https://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,
              <https://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, <https://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, <https://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,
              <https://www.rfc-editor.org/info/rfc7815>.

   [RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital
              Signature Algorithm (EdDSA)", RFC 8032,
              DOI 10.17487/RFC8032, January 2017,
              <https://www.rfc-editor.org/info/rfc8032>.

   [RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)",
              RFC 8152, DOI 10.17487/RFC8152, July 2017,
              <https://www.rfc-editor.org/info/rfc8152>.

   [rsa-8bit] Gura, N., Patel, A., Wander, A., Eberle, H., and S.
              Shantz, "Comparing Elliptic Curve Cryptography and RSA on
              8-bit CPUs", DOI 10.1007/978-3-540-28632-5_9, 2004.

   [rsa-high-speed]
              Koc, C., "High-Speed RSA Implementation", November 1994,
              <http://storage.jak-stik.ac.id/rsasecurity/tr201.pdf>.

   [sec2ecc] Certicom Research, "SEC 2: Recommended Elliptic Curve
              Domain Parameters", Version 2.0, January 2010.

   [stnucleo] STMicroelectronics, "NUCLEO-F091RC",
              <http://www.st.com/en/evaluation-tools/
              nucleo-f091rc.html/>.

   [tinyecc] Liu, A. and P. Nig, "TinyECC: A Configurable Library for
              Elliptic Curve Cryptography in Wireless Sensor Networks
              (Version 2.0)", NCSU College of Engineering, February
              2011, <http://discovery.csc.ncsu.edu/software/TinyECC/>.

   [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", Proceedings of the 19th
              International Conference on Computer Communciations and
              Networks, DOI 10.1109/ICCCN.2010.5560028, 2010.

   [wiselib] "wiselib", February 2015,
              <https://github.com/ibr-alg/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
   Nokia
   Helsinki 00181
   Finland

   Email: heidi.back@nokia.com