A Performance Study of Crypto-Hardware in the Low-end IoT

Peter Kietzmann[1], Lena Boeckmann[1], Leandro Lanzieri[1], Thomas C. Schmidt[1], Matthias Wählisch[2]
[1]HAW Hamburg, [2]Freie Universität Berlin
{peter.kietzmann, lena.boeckmann, leandro.lanzieri, t.schmidt}@haw-hamburg.de,
m.waehlisch@fu-berlin.de

Abstract

In this paper, we contribute a comprehensive resource analysis for widely used cryptographic primitives across different off-the-shelf IoT platforms, and quantify the performance impact of crypto-hardware. This work builds on the newly designed crypto-subsystem of the IoT operating system RIOT, which provides seamless crypto support across software and hardware components. Our evaluations show that (i) hardware-based crypto outperforms software by considerably over 100 %, which is crucial for nodal lifetime. Despite, the memory consumption typically increases moderately. (ii) Hardware diversity, driver design, and software implementations heavily impact resource efficiency. (iii) External crypto-chips operate slowly on symmetric crypto-operations, but provide secure write-only memory for private credentials, which is unavailable on many platforms.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Measurement, Performance, Security

Keywords. Internet of Things, Embedded Security

1 Introduction

Security is an essential building block for the Internet of Things (IoT). Data confidentiality, integrity, and availability rely on crypto-operations that are often resource-intensive and in conflict with device constraints. Nevertheless, software updates, access management, and data encryption rely on these crypto-operations. To enable usable security in the low-end IoT, cryptographic primitives should be highly optimized and utilize the constrained hardware most efficiently—including possible crypto-extensions. This poses a severe challenge, since hardware support is heterogeneous and ranges from extended instruction sets to complete implementations of popular algorithms such as AES.

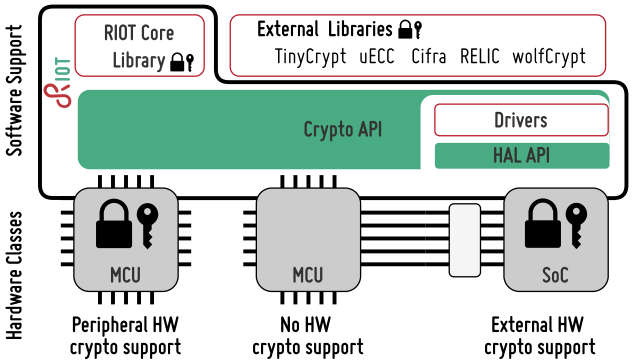


Figure 1. The software support layer of RIOT integrating crypto-peripherals, external crypto-devices, and crypto-libraries using a common crypto API.

Figure 1 presents three common options to enable security in the constrained IoT. (*i*) cryptographic software libraries that try to cope with embedded constraints, (*ii*) microcontrollers that include a crypto-peripheral, (*iii*) external crypto-devices that connect to the microcontroller using a communication bus. Software libraries do not exploit crypto-hardware for portability reasons, and manufacturer SDKs (on bare metal) reduce flexibility towards a vendor lock-in.

More and more IoT deployments utilize an operating system (OS) to keep applications portable while gaining near-optimal hardware support via an abstraction layer. A key motivation of this work is to make heterogeneous hardware components uniformly accessible for both crypto-libraries and applications, and to quantify its resource gain. Up until now, platform-agnostic support of crypto-hardware is rarely available at an IoT system level.

In this paper, we argue that an IoT OS should provide unified APIs to grant access of available hardware without sacrificing performance nor functionality. We will report about the various challenges that derive from heterogeneous hardware concepts as well as tight resource constraints and how to over these by trading software for hardware without sacrificing efficiency.

The contributions of this paper are the following:

International Conference on Embedded Wireless Systems and Networks (EWSN) 2021
17–19 February, Delft, The Netherlands
ISBN: 978-0-9949886-5-2

1. We present design tradeoffs for integrating different types of crypto-drivers, and introduce our crypto-subsystem to the IoT operating system RIOT (§ 2).
2. A brief overview of hardware crypto platforms (§ 3).
3. A comparative performance study of five software libraries (§ 4), basic symmetric and asymmetric cryptography implemented on four hardware platforms (§ 5), and advanced Elliptic Curve Cryptography (ECC) (§ 6). Our results indicate that hardware is not always the most efficient solution.
4. A detailed system benchmark of vendor drivers indicate optimization potentials (§ 7).
5. Our software is available on `https://github.com/inetrg/EWSN-2021`.

2 A Crypto-Subsystem in RIOT

We now introduce the design and implementation of a crypto-subsystem that integrates hardware with software components and allows for a fair comparison across multiple platforms and libraries. We base our implementation on RIOT [8], an open-source operating system for low-end IoT microcontrollers.

We decided for RIOT because it runs on many architectures (from 8-bit over 16-bit to 32-bit processors), provides multi-threading with a scheduler supporting fixed priorities and preemption, power management [36], and a powerful hardware abstraction layer. Security protocols utilize cryptographic functions, which are currently implemented as software solutions at the system level [18].

Alternatively, the `package` system can be used to integrate external libraries. RIOT includes wolfCrypt [43], an embedded library for symmetric and asymmetric crypto, Cifra [10] which implements common building blocks for symmetric crypto, TinyCrypt [20] and micro-ecc (uECC) [21], both particularly minimizing memory, and Relic [4], which contributes a comprehensive list of symmetric and asymmetric cryptographic schemes with particular support for many elliptic curves. As such, these third-party libraries are not implemented against any OS APIs, yet. Our design concept integrates these components in a generic fashion and extends to further hardware platforms and libraries in a straightforward manner.

2.1 Design Considerations

The operating system grants access to cryptographic hardware. A driver controls the device and implements an agnostic OS API. In five sample use cases of this paper, vendors provide a library to access low-level operations. We now consider design aspects of how to integrate these and future cryptographic components.

2.1.1 Vendor Driver Integration

Capabilities of vendor drivers vary widely. We argue for using these implementations, though, to take advantage of specific vendor knowledge, testing, and to allow for sustainable maintenance. The `package` subsystem in RIOT clones, builds, and links external repositories during firmware compilation. In this way, third-party software does not require maintenance within the OS and can easily be updated. We implement vendor libraries as RIOT packages and provide software wrappers to integrate external code into the subsystem. It is noteworthy, that vendor libraries do not always perform at maximum performance since they are commonly implemented in a generic way, as we will show in Section 7.

2.1.2 Context Abstraction

Cryptographic functions operate on an internal state (context `struct`). It is allocated for each driver instance and depends on the exposed state by a vendor implementation which includes hardware specific elements internally. When facing the OS, a context `struct` must abstract vendor specifics and implement common OS interfaces. Hence, every driver defines a common context `struct`, containing vendor specifics and optional elements to facilitate the OS integration. Consequently, users of the API must not dereference the context `struct` since it changes with different backends. This design decision prevents parallel operation of different backends for the same function. We argue that this is in line with common IoT deployments for three reasons: (*i*) Single-core OSes are not optimized for parallel processing because real-world IoT firmware is tailored to a single application. Consequently, excessive parallelization of crypto-operations is not expected. (*ii*) Computational and memory resources are scarce on constrained IoT devices. Our context abstraction keeps complexity low. (*iii*) The performance of crypto-peripherals increases software solutions by one order of magnitude (see Section 5), thus, successive hardware operations already outperform software notably.

2.1.3 State Handling

Applications of cryptographic primitives are manifold, including security protocols or pseudo-random number generators, and may require individual maintenance of a crypto-functions state. We enable external memory allocation and state handling in our approach. Software implementations operate on an allocated `struct` in RAM. Cryptographic processors may reduce operations to one at a time, provided they rely on a single hardware state. This inflicts to read, save, and restore the hardware state between operations to achieve state independence. Especially for external processors, this increases completion time and RAM requirements to replay and store hardware contexts. We implement an optional read–save–restore behavior for such devices and evaluate the overhead in Section 7

2.1.4 Concurrent Access

Cryptographic processors need protection against concurrent access. Certain vendor drivers implement mutexes internally, while others require protection by the OS. It is noteworthy, that hardware devices require protection and not a single crypto-function, since processors are commonly single resources. We lock/unlock a mutex per device before and after every hardware access. Few microcontrollers provide hardware acceleration units with more than one crypto-peripheral, which can operate in parallel. In that case, each device must be protected separately against concurrency. Dual-accelerators promise throughput enhancements when crypto is heavily used. We implement a management instance to the driver which is requested internally before every crypto-operation and delivers the next free device.

Table 1. Overview of typical on- and off-chip IoT hardware with their crypto-acceleration features that we analyze.

Feature \ MCU/ Board/ Library	**nRF52840** (@64 MHz) Nordic nRF52840dk CryptoCell	**EFM32**(PG12) (@40 MHz) Silicon Labs Pearl Gecko EMLIB	**MKW22D** (@48 MHz) Phytec IoT Kit 2 mmCAU	**ATECC608A** (I2C@400kbps) Adafruit ATECC608 CryptoAuthLib
TRNG	✓	✓	✓	✓
SHA-256	✓	✓	✓	✓
HMAC-SHA256	✓	✘	✘	✓
AES-128	ECB, CTR, CBC,CCM(*) CMAC/CBC-MAC	ECB, CTR, CBC,CCM(*),CFB CBC-MAC, PCBC,GMAC,GCM	✓	ECB, GCM
AES-256	✓	✓	✘	✘
ChaCha20/Poly1305	✓	✘	✘	✘
ECC	secp160k/r1, secp192k/r1,secp224r/k1, **secp256k/r1**, secp384r1, secp521r1 Ed25519, Curve25519	secp192r1, secp224r1, **secp256r1**, sect163k1, sect163r2, sect233k/r1,	✘	**secp256r1** (**P-256**)
ECDSA / ECDH	✓	✘	✘	✓
RSA	✓	✘	✘	✘
Secure memory	128-bit	5 x 256-bit	256-bit	16 Key Slots

2.1.5 Low Power Management

Low energy consumption is a core requirement in the IoT. Active peripherals prevent devices from sleep and consume energy. It is an obvious design choice to keep active time of a crypto-device at minimum, hence, the OS integration of a driver should enable hardware only when used. This performs efficient on peripherals that turn on and off fast. Other devices (*e.g.,* external chips), however, require additional resources during initialization, which is inefficient when requested excessively. To deal with on/off patterns, we follow two different approaches. (*i*) Devices that turn on fast are powered only during operation. This aligns well with the concept of other peripheral drivers in RIOT. (*ii*) Devices with long wakeup sequences are not unconditionally set to sleep after usage, since concurrent applications might require crypto access. Instead, we implement a user counter that increments on device allocation and decrements on release. The device is turned off when the counter decrements to zero. Vendor drivers commonly operate synchronously, hence, our approach only affects preempted driver calls. On the downside, successive requests from a single context will not benefit and require a manual power switch which we analyze in Section 7.

2.2 Integration of Crypto Modules

RIOT currently supports 208 boards and 117 different microcontrollers, all of which exhibit varying crypto-hardware capabilities. To allow for the implementation and use of crypto-based applications without considering the current hardware setup upfront, we design (*i*) a hardware-agnostic API and (*ii*) the dynamic configuration of the crypto-subsystem. We use a feature model to represent crypto-hardware capabilities to the build system. Our approach selects and compiles hardware features where possible, and additionally provides an extended configuration interface to the user. This generic approach is capable of handling any hardware component, provided it is correctly modeled.

2.2.1 Module Design

Our layered approach to interface with different crypto-backends introduces two types of APIs that are exposed to the user: (*i*) The *basic cryptographic API* provides direct access to low level functions, for example, a single AES block encryption. (*ii*) The *cryptographic mode API* grants access to operation modes of crypto primitives, for example, configuring AES in Cipher Block Chaining (CBC), or Electronic Code Book (ECB) mode. Different backends are modeled as modules, each of them is a translation unit that provides an implementation of one public API, which allows specific selection by the build system.

A backend module (*i.e.,* a driver) can support one of the three levels of crypto-acceleration in hardware: (*i*) Full hardware acceleration, (*ii*) partial hardware acceleration, and (*iii*) no hardware acceleration. The first level is given by peripheral-, or external devices that provide full hardware support for a cryptographic mode (*e.g.,* AES CBC). The second level occurs when hardware support is only available for basic cryptographic operations. In that case, the operation mode (*e.g.,* CBC) is performed in software and needs access to basic cryptographic primitives (*e.g.,* AES block encryption). The software component, however, is agnostic to the specific cipher or hash in use. The third level represents the case with missing hardware acceleration units. Our abstraction of the cryptographic API allows to switch backend implementations seamlessly.

2.2.2 Feature Model

We use Kconfig [40] to select compiled modules during the build process. Kconfig allows to define symbols which specify dependencies and conditional default values. A user can interact with Kconfig using existing tools (*e.g., menuconfig*) to configure the values of a symbol. Each block in our crypto-stack is modeled as a Kconfig symbol. Hardware capabilities are represented as non-visible boolean Kconfig symbols that indicate the availability of crypto-hardware. Crypto APIs are implemented as boolean choice symbols, which allows for a transparent replacement of crypto-backends. Consequently, Kconfig provides a list of exchangeable modules, which are mutually exclusive. The default activation of a module is handled by Kconfig which selects the module combination, given the hardware capabilities of the underlying platform. A user can still manually overwrite the default selection.

Table 2. Performance of SHA-256 on 64 Byte inputs implemented in different software on the nRF52840.

Impl.	Processing Time			Memory			
	Init [μs]	Update [μs]	Final [μs]	Ctx [B]	Stack [B]	RAM [B]	ROM [kB]
Relic	1.00	97.25	90.25	116	1000	110	1.3
RIOT Core	1.08	112.40	119.40	104	600	74	1.3
wolfCrypt	5.13	69.25	68.87	112	600	74	1.8
TinyCrypt	13.75	97.75	97.25	112	600	74	1.2
Cifra	12.62	85.75	113.40	104	616	74	1.2

Table 3. Performance of a single block AES-128 operation implemented in different software on the nRF52840. RAM is 38 Byte for all platforms.

Impl.	Processing Time			Memory		
	Init [μs]	Enc. [μs]	Dec. [μs]	Ctx [B]	Stack [B]	ROM [kB]
Relic	3.00	57.42	88.03	577	1476	13.8
RIOT Core	3.25	38.17	71.17	20	508	4.7
wolfCrypt	0.67	51.50	86.42	284	780	11.7
TinyCrypt	-	225.70	659.90	176	668	2.6
Cifra						
unprotect.	49.37	60.37	77.25	180	732	1.7
protect.	1617.10	6338.12	6353.87	180	732	1.9

3 Measurement Setup

3.1 Platform Overview

Table 1 summarizes the hardware and its features that we use in our evaluation. We omit legacy algorithms, and deploy our experiments on state-of-the-art (nRF52840 and EFM32) as well as older (MKW22D) microcontrollers with *peripheral* crypo-acceleration. nRF52840 and EFM32 are a good representation of the current generation of devices with advanced crypto-peripherals that were designed for flexible deployment in general use cases. Furthermore, we use the *external* crypto-chip (ATECC608A), connected via an I2C bus. ATECC608A represents a series of external security devices that are protected against side channel attacks.

All platforms provide a true random number generator (TRNG), of which all except the MKW22D comply with NIST standards (cf., [22] for background on embedded random number generation). The ATECC608A implements a crypto-secure pseudo-random number generator (CSPRNG), which is seeded from a true random source in hardware. EFM32 and MKW22D deploy software assisted HMAC SHA-256 that utilizes hardware hashing. nRF52840 and EFM32 support multiple cipher modes in hardware, contrasting MKW22D and ATECC608A, which need software assistance. ECC is available on all platforms but MKW22D.

The nRF52840 and EFM32 offer secured key registers for performing crypto-operations, and the MKW22D has a tamper protected register. The ATECC608A has 16 non-volatile, write-only memory slots to store secret elements. Keys are generated and maintained on the external device to prevent unauthorized access and erase on tamper detection.

3.2 Measured Resources

We measure processing time, energy consumption, and memory overhead and repeat experiments 1000 times presenting averages, if not mentioned otherwise. To allow for a fair comparison, all software runs on RIOT version 2020.07.

Processing time. We evaluate the processing time with a logic analyzer that samples at 12 MS/s by toggling an I/O pin via direct register access on the test device. Using this setup the measurement overhead remains negligible.

Energy consumption. We connect our test platform to a regulated voltage supply (Siglent SPD3303C) and evaluate the current consumption of each operation using a digital sampling multimeter (Keithley DMM7510 7 1/2) at 1 MS/s. A measurement period is marked by toggling I/O pins. To bypass unrelated current flows, we connect our probes in series with the MCU and turn off unused hardware components (by hardware switches or in software). We measure the current of ATECC608A on the external chip, only, for better compatibility. In practice, the nRF52840 is used to operate the device, which might sleep during crypto-operations.

Memory requirements. We evaluate the memory consumption and consider compile- and runtime properties. Our compile time measurements show the overhead on top of a minimal RIOT build by analyzing the ELF file. We accumulate all linked objects that are associated with a crypto-implementation. Numbers are differentiated w.r.t. to RAM and ROM memory. Runtime memory requirements include the size of data structures and the maximum amount of stack memory that has been used during execution.

4 The Impact of a Software Implementation

Table 2 and 3 compare the impact of different SHA-256 and AES-128 software implementations provided by commonly available crypto-libraries in RIOT that we ran on the nRF52840 platform. In both cases, we applied the cryptographic function to an input vector with the size of one internal block (*i.e.,* 64 Byte for SHA-256 and 16 Byte for AES-128).

Relic, TinyCrypt, and Cifra require 190–210 μs for an *init-update-final* sequence to process a SHA-256 digest. Cifra is faster during *update* on the price of longer finalization caused by an extra copy of the hash value. RIOT Core is 20 μs slower because it involves repeated modulo operations during state update (FIPS PUB 180-4). Furthermore, RIOT includes multiple endianness conversions to operate on 32-bit arithmetic. wolfCrypt provides a highly optimized implementation with an unrolled mixing loop, at higher memory consumption. Disabling this optimization increases the processing overhead of *update* and *final* to approximately 100 μs each, but it reduces ROM requirements by 500 Byte. Context sizes and stack usage are similar in all implementations, except for Relic that uses 400 Byte additional stack for a global array with initial hash state values.

The impact of different software implementations becomes more pronounced for AES-128. Initialization is fast in RIOT, wolfCrypt, and Relic taking at most about 3 μs to initiate state and the AES key length. TinyCrypt does not expose a separate API call for initializing but handles it internally. In contrast, Cifra contains the key schedule (FIPS PUB 197) already on initialization which takes up to 50 μs.

wolfCrypt, Relic, and TinyCrypt provide a dedicated API to trigger AES key expansion, while RIOT handles the expansion on every en-/decrypt call. In Table 3 we include key expansion overhead in the en- and decrypt column, except for Cifra.

Encryption is by a factor of 1.5–3 faster than decryption for the additional key inversion during decrypt [12]. RIOT provides the fastest implementation for a single block, followed by wolfCrypt and Relic. Their implementations base on pre-calculated look-up tables (T-tables) which is a speed optimization for 32-bit platforms. Cifra (w/o protection) and TinyCrypt implement the default algorithm based on a substitution table (S-box). Surprisingly, the S-box implementation in Cifra (w/o protection) scales similar to the T-table approach, whereas TinyCrypt operates 4–10 times slower. This is due to multiple copies between the internal and externally provided state, as well as explicit clearing of the internal memory. Lookup table implementations are vulnerable to side channel attacks [7, 41], especially cache attacks, why Cifra (w/ protection) provides countermeasures by default, that increase the runtime by a factor of 100.

Context sizes vary widely. RIOT allocates only 20 Byte for one context, of which 16 Byte represent the internal AES-128 state. Cifra and TinyCrypt keep the expanded key (multiple "round keys") in their context structure, which increases memory by up to 180 Byte. In that way, round keys do not have to be re-calculated when encrypting repeatedly on one AES context. In contrast, working on multiple contexts heavily increases RAM consumption. wolfCrypt and Relic follow a similar approach, though, both `structs` are not tailored to 128 Bit keys. wolfCrypt unconditionally allocates memory for 256 Bit keys and Relic additionally stores raw keys and initialization vectors that are used for cipher modes. The stack usage correlates to the context sizes. Memory overhead in ROM is more distinctive. wolfCrypt and Relic store complete T-tables (10 buffers of 1024 Byte), whereas RIOT only stores half of them and generates the remaining values on demand. TinyCrypt and Cifra attain the smallest ROM footprint based on minimized lookup tables.

5 Basic Crypto-Hardware Acceleration

Next we compare the performance of basic cryptographic operations between hardware and software, using the crypto-hardware discussed in Section 3.1. Software results are obtained from RIOT core on the same platforms but with crypto-hardware turned off.

5.1 Processing Time

Figure 2 shows the processing time for short (32 Byte) and long (512 Byte) input data, separated into cryptographic operations. We use a randomly chosen 128 Bit AES key and a random initialization vector for the CBC mode. The HMAC SHA-256 is initialized with a random 256 Bit key. As occasionally recommended, we also conducted experiments with 512 Bit keys, but without further insights.

Short input data. Hardware accelerated operations scale similar on nRF52840 and EFM32 for short inputs (Fig. 2(a)) and require less than 70 μs for AES ECB/CBC during initialization, encryption, and decryption as well as SHA-256 hashing. HMAC SHA-256 is more complex, for repeated internal hash computations, and takes at most 250 μs on both platforms. Hash *updates* are small for all configurations due to the short input sequence. The *update* function collects 64 Byte of data (SHA-256 internal state) before starting a block operation, which is first triggered by *final* in this case. The MKW22D operates at minimal overhead for all functions. AES CBC encryption takes longer than decryption on that platform because of the software chaining of hardware accelerated AES blocks. An additional copy of the input buffer during encryption avoids overwriting it, which is omitted during decrypt.

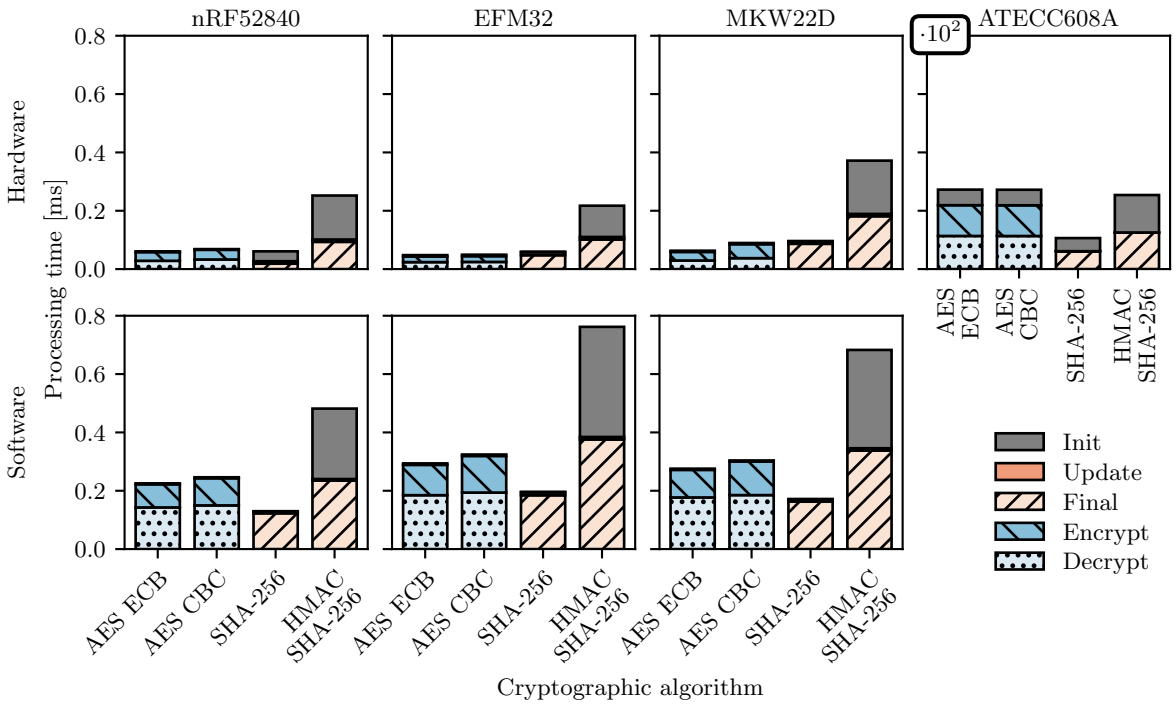
When implemented in hardware, ciphers gain more—a factor 4–6—over software than hashes (factor 2–4). A comparison of software and hardware measurements for the EFM32 shows the particular power of that platform. It operates at minimal cost using hardware accelerated operations, in contrast to software, for which it performs slower than nRF52840 and MKW22D, since it operates at lowest CPU frequency. In software, AES ECB/CBC decryption is two times slower than encryption due to the additional key inversion (see Section 4). This overhead disappears on hardware.

The ATECC608A operates two orders of magnitude slower than the other platforms (see Figure 2(a)). The reason for this overhead is twofold. First, the vendor library maintains device power levels and wakes up the device prior to every operation. Second, control commands and data need to traverse the I2C bus with a copy to resp. from the microcontroller. AES initialization takes proportionally longer because the encryption and decryption key has to be sent to the device before use. The difference between cipher and hash based algorithms is higher on the ATECC608A in comparison to crypto-peripheral and software support, because AES-128 encryption of 32 Byte involves two block operations, the transport of which adds an overhead.

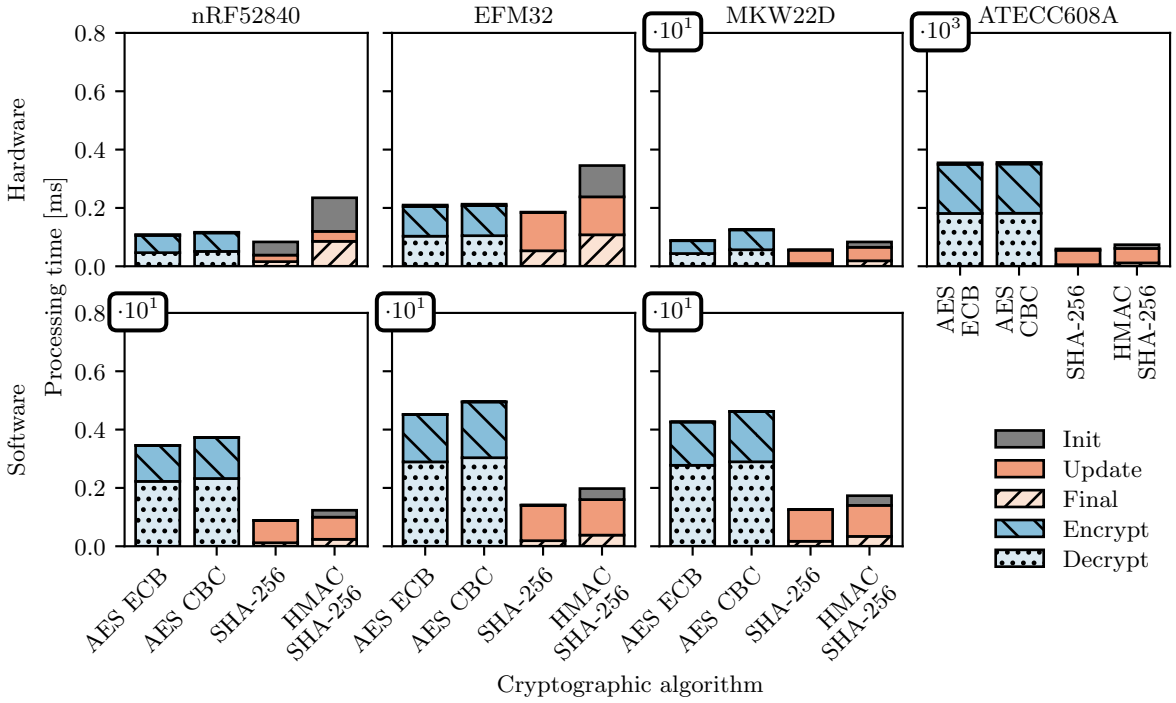
Long input data. Hardware crypto performance gains over software with longer input strings. We display results for 512 Byte input in Figure 2(b). Hardware based hash computation now operates 5–10 times faster than software and ciphers speed up by a factor of 20 to 30. The processing time of crypto-peripherals still operates on a similar scale compared to short inputs, on nR52840 and EFM32. Surprisingly, accelerated operations on nRF52840 outperform EFM32 for long inputs—in contrast to short inputs.

The EFM32 requires a manual iteration over blocks, which involves a copy of intermediate data to a temporary buffer. The nRF52840 vendor library hides this iteration under its API and the internal operations are closed source. We expect advantages for nRF52840 here. Hardware accelerated AES on the MKW22D increases by one order of magnitude in comparison to short inputs, for the hybrid software-hardware chaining mode. The performance overhead for hashes is less dominant. Processing times for ciphers in software increase by one order of magnitude due to the complexity of block chaining and repeated key schedules. Hash computations are more comparable in software. The effort of *update* becomes visible since long input buffers update the internal state before calculating a final digest.

Most notable is a severe performance overhead on the ATECC608A, which operates three orders of magnitude



(a) 32 Byte input data



(b) 512 Byte input data

Figure 2. Processing time of different crypto-algorithms on different platforms for short (32 Byte) and long (512 Byte) input data, separated into cryptographic operations. Crypto-operations are either accelerated in hardware on off-the-shelf microcontrollers (nRF52840, EFM32, MKW22D) or on an external cryptographic chip (ATECC608A connected via nRF52840), or purely implemented in software.

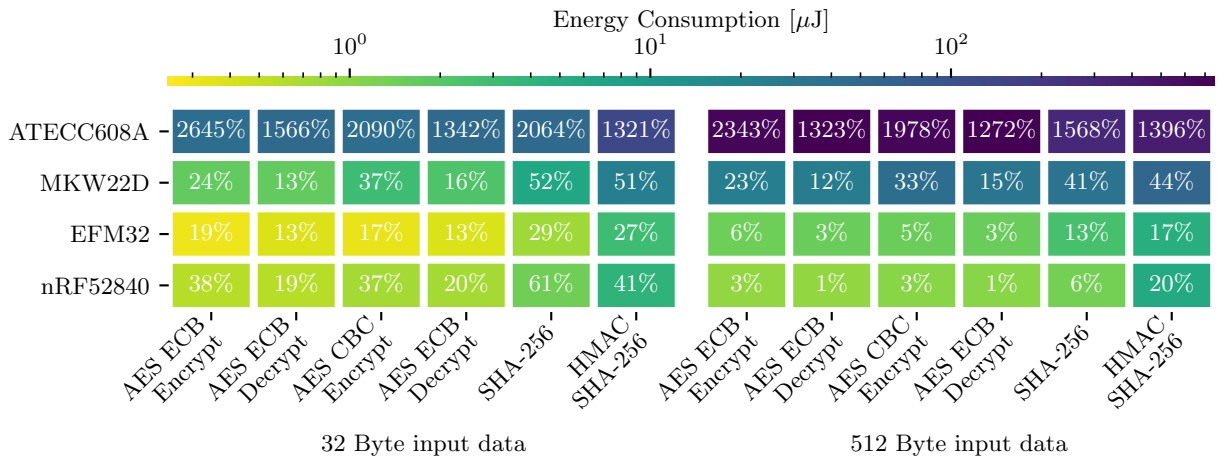


Figure 3. Energy consumption of hardware accelerated cryptographic operations on different platforms. Percentages show relative overhead compared to crypto-implementations in software running on the same devices.

slower than peripheral accelerators, and two orders slower than software. AES suffers from the length of output data, which equals the input length. In contrast, HMAC- and SHA-256 only return a 32 Byte digest which relieves the bus and device control overhead.

5.2 Energy Consumption

Figure 3 depicts the absolute energy consumed by basic crypto-operations on hardware (colors) and displays the relative energy compared to software (percentages). AES encryption and decryption include the initialization, and hash based operations include *init-update-final*. These results roughly correlate with the processing time.

Short input data. The peripheral EFM32 and nRF52840 consume 0.25–4 µJ. EFM32 requires 4 mA and nRF52840 6 mA. The MKW22D is the most expensive peripheral and consumes 1–20 µJ due to a high current of up to 20 mA peak. The external ATECC608A consumes 45–130 µJ which is the highest energy demand despite its small average current of 1.2 mA—a consequence of the long execution time. HMAC SHA-256 is the most expensive operation on all platforms.

Long input data. The energy consumption of EFM32 and nRF52840 increase marginally over short inputs. nRF52840 now outperforms EFM32 despite the higher current of ≈ 1 mA. The MKW22D increases the consumption by roughly 10x, which is the overhead of software assisted acceleration. The most expensive device is still the ATECC608A whose consumption also increases by roughly 10x due to additional device management and bus utilization. AES based operations are the most expensive operations due to the high amount of encrypted data transmitted.

Software versus hardware. The performance gain from hardware acceleration increases with longer input data. The EFM32 and nRF52840 reduce the consumption on long input data down to 1 % of the software variants. The MKW22D equally profits for short and long input data. Energy demands of the ATECC608A extension are much higher than for software—an increase by factors from 13 to 25 with only a slight gain in efficiency for longer input data.

5.3 Memory Requirements

Table 4 shows the memory consumption for basic crypto-operations on our reference hardware and compares the results to an equivalent software implementation.

Context. On nRF52840, the overhead of context structures ranges from 76 to 168 Byte, which are introduced by a buffer to represent hardware state for internal use by the vendor driver. AES CBC introduces less overhead since cipher chaining requires additional memory in software which is absent in hardware. In contrast, the EFM32 and MKW22D platforms require the same context sizes as our reference software as they do not mirror the hardware state.

Stack. The stack sizes allocated with the crypto-hardware are similar to the base crypto-software. Most notably for HMAC SHA-256, several accelerators can reduce stack size, *e.g.,* by not requiring a second SHA-256 context.

RAM and ROM. For ciphers, ROM overheads mostly decrease for the EFM32 and MKW22D due to the absence of static lookup tables in software (T-tables, see Section 4). RAM sizes remain moderate on these platforms with increases originating from initialized variables of driver libraries.

In contrast, on nRF52840 ROM and RAM overheads are dominant. Its crypto-library maintains an internal hardware abstraction layer, basic synchronization primitives, and interrupt routines. This software representation adds 6.4 kByte to all operations and cannot be disabled, even if features are not required for basic operations.

External device drivers. The external ATCC608A device interacts only via I2C communication and operates differently via its drivers. For example, encryption keys have to be written to a key slot and indexed for use, instead of passing a pointer to an allocated key structure. RAM and ROM usage are fairly independent of the selected crypto-operation and merely reflects the abstracted message transfer and invocation of the different hardware functions.

Table 4. Memory consumption of different crypto-algorithms on different platforms. Crypto-operations are hardware accelerated. The overhead shows more (↑) or less (↓) hardware resources required in software (RIOT Core).

Platform	AES ECB				AES CBC				SHA-256				HMAC SHA-256			
	Ctx [B]	Stack [B]	RAM [B]	ROM [B]	Ctx [B]	Stack [B]	RAM [B]	ROM [B]	Ctx [B]	Stack [B]	RAM [B]	ROM [B]	Ctx [B]	Stack [B]	RAM [B]	ROM [B]
nRF52840	96	600	6407	7107	96	632	6455	7263	240	744	6391	5691	376	880	6455	6699
Overhead	↑76	↑36	↑6348	↑2105	↑76	↑20	↑6348	↑2117	↑136	↑136	↑6348	↑4477	↑168	↓16	↑6284	↑5213
EFM32	20	524	84	4208	20	556	132	4288	104	600	64	4220	208	704	160	4492
Overhead	0	↓40	↑16	↓804	0	↓56	↑16	↓878	0	↓8	↑16	↑2990	0	↓192	↓16	↑2990
MKW22D	20	540	196	2838	20	556	244	2976	104	628	372	1692	208	924	468	1964
Overhead	0	↓24	↑136	↓2152	0	↓56	↑136	↓2184	0	↑20	↑328	↑480	0	↑28	↑296	↑464
ATECC608A	56	676	138	4175	56	740	154	4361	136	780	90	4089	136	748	154	4249
Overhead	↑36	↑112	↑79	↓827	↑36	↑128	↑47	↓785	↑32	↑172	↑47	↑2875	↓72	↓148	↓17	↑2763

6 ECC Hardware Acceleration

We present analyses of elliptic curve cryptography running on hardware and software implementations. Hardware performance measurements consider peripheral crypto-acceleration of the nRF52840 and the external crypto-chip ATECC608A. Software measurements include Relic, a feature-rich library, and uECC, a minimal highly optimized library, on the same device. Relic operates in default configuration, which uses a precomputation table for scalar multiplication to improve runtime performance. uECC is deployed with optimization level two, which is the default to achieve a balanced speed-size tradeoff. We operate on the NIST P-256 elliptic curve with 256 Bit sized keys that is supported by all hardware and software platforms and evaluate keypair generation, signature generation and signature verification (ECDSA), as well as generation of a shared secret, based on preceding key exchange between two parties (ECDH). Signatures are computed on a 32 Byte message digest and secrets are 32 Byte (256 Bit) in size. Keypair generation and signing rely on random numbers and hardware accelerators use a build-in TRNG. Our software measurements use a seeded SHA-256 based CSPRNG. We also configured both libraries to use a hardware generator, but the advantage remains negligible. As the results do not contribute to additional insights, we excluded these experiments.

6.1 Processing Time

Figure 4 presents the average and the min/max processing time of different elliptic curve operations. Results now scale from tens to hundreds of milliseconds.

Hardware crypto support. The nRF52840 crypto-peripheral performs best, analogously to the results of Section 5. All operations require ≈ 20 ms, which is one order of magnitude below the software. The ATECC608A operates 2 to 4 times slower than the nRF52840 peripheral. Verification as well as secret generation still outperform software by a factor of 2–3. In contrast to basic crypto-operations (cf., Section 5), the external device reveals distinct performance benefits because reduced device access contributes to lower control overhead.

Software versus hardware. Relic creates a precomputation table during initialization. This step, which takes up to 140 ms, is not required on hardware, and it is absent in uECC because lookup tables are statically compiled. Key generation and signing benefit from the precomputation table when applying the COMBS method [26] optimization for multiplying a prime elliptic curve point by an integer. This reduces processing times to less than 100 ms and leads to performance results that are on par with hardware accelerated operation on ATECC608A. Verification, however, exhibits a higher processing demand of 260 ms. This variable base scalar multiplication is complex because Relic uses the window-non-adjacent form method [33]. The same applies to the shared secret generation. The ATECC608A, in contrast, performs inversely, which shows that a dedicated multiplication circuit is effective.

Compared to Relic, uECC operates at the same scale but benefits from selected algorithmic choices without using a precomputation table. Secret creation in uECC is 50 ms faster than in Relic because the co-Z Montgomery Ladder [29] outperforms the window-non-adjacent form for

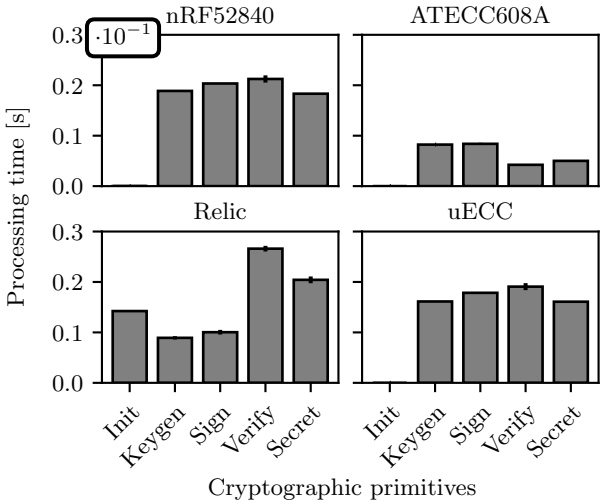


Figure 4. Processing time of different elliptic curve algorithms. Crypto-operations are either accelerated in hardware on the microcontroller (nRF52840), on an external chip (ATECC608A), or in software (Relic, uECC).

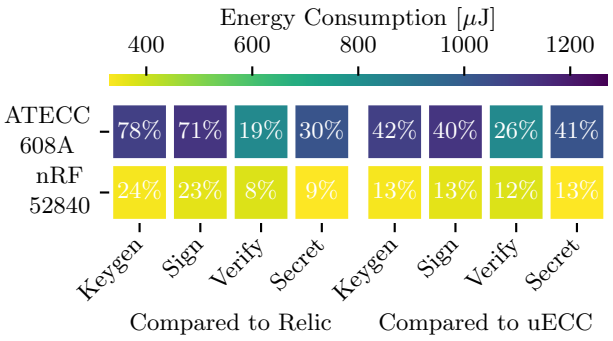


Figure 5. Energy consumption of different elliptic curve algorithms on the nRF52840 platform. Percentages show relative overhead compared to crypto-implementations in software running on the same device.

scalar multiplication over the elliptic curve [23]. Verification is equally fast in uECC taking advantage of the Strauss-Shamir method [39] for double scalar multiplication. Signature creation is 50 % slower, though. Considering the sum of signing and verification uECC and Relic are on par.

6.2 Energy Consumption

Figure 5 surveys the absolute energy consumption of hardware accelerated operations on elliptic curves (colors) and compares the relative excess over software (percentages). The ATECC608A consumes 700–1200 μJ and the peripheral nRF52840 requires less than 400 μJ for every operation. The most expensive operation on nRF52840 is signature verification (10 μJ more than signing), which is in contrast to the ATECC608A, which requires more energy for signature creation but is most frugal when doing verification. These results roughly correlate with the processing time.

The performance gain of the ATECC608A over Relic surprises most. Both implementations are on par in terms of processing overhead but the specific hardware implementation requires only 71%-78% energy of corresponding operations implemented in pure software. Creating the precomputation table in Relic consumes approx. 500 μJ and presents pure software overhead.

6.3 Memory Requirements

Table 5 shows the memory consumption for different ECC schemes implemented in hardware and in comparison to the two software libraries.

Peripheral crypto support. To store private and public keys the nRF52840 implements one data structure for each key, which require 816 and 884 Byte respectively. Considering private (32 Byte) and public (64 Byte) key sizes of the P-256 curve, this is a significant overhead and illustrates a design decision that favors flexibility at the expense of memory resources. Each data structure mirrors hardware state as well as the elliptic curve domain parameters (*i.e.,* elliptic curve modulus, equation parameters, and co-factors of a key). The latter contributes most to the overhead. Storing the domain parameters for each key (pairs) separately, however, allows not only different elliptic curve configurations in parallel but also to change parameters at runtime.

Table 5. Memory consumption of different elliptic curve algorithms on different platforms. Crypto-operations are hardware accelerated. The overhead shows more (↑) or less (↓) resources required in comparison to software.

Platform	Keys		ECDSA			ECDH		
	Priv. [B]	Pub. [B]	Stack [B]	RAM [kB]	ROM [kB]	Stack [B]	RAM [kB]	ROM [kB]
nRF52840	816	884	5472	8.08	21.20	3880	8.10	15.06
Ovrh. uECC	↑784	↑820	↑4352	↑7.70	↑14.28	↑3064	↑7.72	↑9.73
Ovrh. Relic	↑540	↑784	↓480	↑2.53	↓2.48	↓1136	↑2.55	↓6.22
ATECC608A	–	64	998	0.12	4.85	676	0.12	3.52
Ovrh. uECC	↓32	0	↓122	↓0.26	↓2.06	↓140	↓0.26	↓1.81
Ovrh. Relic	↓276	↓36	↓4954	↓5.43	↓18.82	↓4340	↓5.43	↓17.75

In contrast to basic crypto-operations (cf., Section 5.3), the stack of the nRF52840 operates ECCs at a scale of kilobytes because the crypto-driver library requires temporary buffers. A user context to sign and verify introduces additional overhead in ECDSA. RAM (8 kByte) and ROM (14–20 kByte) requirements of ECDSA/ECDH are large for off-the-shelf microcontrollers but relatively low (3 %) with respect to the overall memory available on the nRF52840.

Software versus peripheral crypto support. The nRF52840 crypto-hardware introduces the key `struct` and with it a significant overhead (520–820 Byte per key pair) compared to uECC and Relic. uECC requires only 64 Byte resp. 32 Byte for public and private keys. In Relic, the public key is represented as an elliptic curve point of 100 Byte, and the private key is contained in a multi-precision integer structure, which allocates 276 Byte per instance.

The nRF52840 requires more RAM, ROM, and stack than uECC. This is not surprising since uECC is tailored to a low memory footprint [38]. uECC is able to store key material and CSPRNG data in 380 Byte of RAM but provides only a limited set of features. Lower performance penalties are visible when comparing the nRF52840 with Relic. Similar to the nRF52840 crypto support, Relic is feature-rich and flexible. To implement this, Relic maintains a global context in the library, which leads to a similar performance overhead compared to the storage of domain parameters at the nRF52840. The nRF52840 still needs to run specific device drivers to control crypto-acceleration.

External crypto-chip support. The ATECC608A operates most frugal compared to the hardware crypto-peripheral and both software implementations. The low memory requirements enable its use even with tightly constrained microcontrollers. A core design advantage of this device is a dedicated, secure memory slot to store the private key. Only the public key data structure allocates a minimum of 64 Byte, while driver support impacts stack usage, RAM, and ROM only slightly. On the downside, ATECC608A limits crypto-operations to the P-256 curve.

7 The Impact of Driver Implementations

7.1 Vendor Driver and Concurrent Access

The EFM32 (V. PG12) provides two crpyto-peripherals that can be operated independently. This concurrent feature is managed to organize the peripheral access with a driver API that must be asynchronous in a single-core system. The

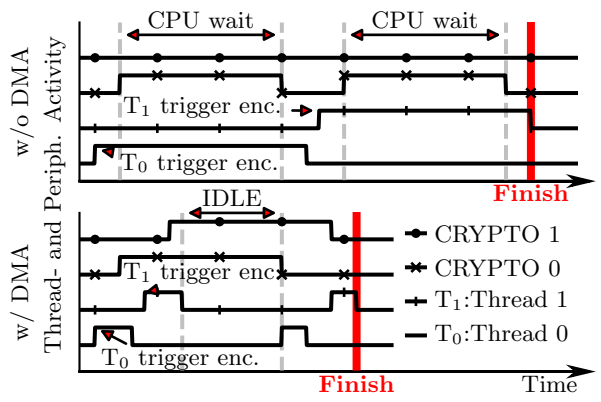


Figure 6. Qualitative comparison of thread and crypto-peripheral activity with (bottom) and without (top) CPU offloading using DMA.

vendor implementation, however, blocks the CPU during crypto-operation. Figure 6 (top) visualizes the progress of our test application using the vendor driver. We start two threads of the same priority, each of which encrypts data periodically. Thread 0 (T_0) triggers encryption on peripheral CRYPTO0. The CPU is acquired by T_0 until completion of the hardware. Thereafter, T_1 is scheduled and triggers encryption, operating CRYPTO 0 again since is not busy anymore. CRYPTO 1 is never used, since the vendor driver hinders parallelism.

We implement an asynchronous driver that exploits DMA to offload the CPU. Each crypto-device uses two DMA channels, one for copying input data to peripheral registers, and the other to return encrypted data. Figure 6 (bottom) visualizes the progress of our test application with an optimized driver. T_0 triggers encryption on CRYPTO 0 and relieves the CPU while the peripheral operates. When T_1 is scheduled, it triggers encryption on CRYPTO 1 and relieves CPU access. Note that both peripherals operate in parallel now, while the CPU stays idle. During that time, the OS can schedule other tasks, or switch to an energy-saving state. After completion of the peripheral tasks, each thread is notified.

Table 6 presents results of a test program that encrypts data concurrently, using AES ECB with and without optimization. Comparing the results for 32 Byte input data, our driver introduces an overhead of 20 ms for the DMA synchronization. For 512 Byte input data, the advantage gets visible. The vendor driver increases the progressing time by a factor of five, while our DMA fix remains unaffected and outperforms the vendor driver by 400 %.

7.2 Power Management and State Handling

Crypto-peripherals and external devices consume energy, why they should be disabled when not in use. Power-cycling a peripheral (*e.g.,* nRF52840) is fast, whereas external devices such as the ATECC608A have a longer, costly startup time. The ATECC608A vendor library turns off the device after every command, which is not desired on successive requests.

We implement a manual switch in the power manage-

Table 6. Processing time for 2000 AES-128 encryptions from two threads (1000 encryptions each) on the EFM32 with different diver implementations.

	32 Byte input	512 Byte input
Implementation	Time [ms]	Time [ms]
DMA off, blocking	37.10	205.20
DMA on, non-blocking	56.60	56.90

Table 7. Performance of SHA-256 on the ATECC608A Platform with different driver properties.

		I2C@100 kbps		I2C@400 kbps	
Impl.	Ctx [B]	Rate [kB/s]	Energy [μJ]	Rate [kB/s]	Energy [μJ]
Auto on/off	136	1.73	85	3.01	48
Copy state	235	0.56	243	0.99	129
Man. on/off	136	2.70	59	6.46	30
Copy state	235	0.88	171	2.17	75

ment to prevent redundant power cycles. The ATECC608A requires to sleep in predefined intervals for clearing register and RAM values intermittently. This is enforced by a hardware watchdog timer, which can be configured to 10 s at most. Wakeup is triggered by a 100 μs low pulse on the I2C SDA line which can be generated by sending a 0-Byte when the bus is operated at 100 kbps. The ATECC608A, however, is capable of 400 kbps bus speed. To exploit the maximum performance, the OS needs reconfiguration capabilities for the I/Os to toggle the SDA pin independently of the I2C operation. We implemented this feature on the microcontroller that drives the ATECC608A. This implementation was used already in the previous sections.

The ATECC608A operates on a single hardware state (see Section 3.1) and requires an atomic *init-update-final* during hashing, though, certain use-cases operate on multiple hash states to be updated independently, before a final message digest is calculated. We implement an alternative SHA-256 function that replays the hardware state for every operation.

Table 7 shows evaluations for repeated SHA-256 operations on the ATECC608A. We measure the impact of manual power management, the I2C bus speed, and the overhead of preserving hardware state. Our evaluation presents context sizes, data rates for periodic hashing, and the energy consumption for a single hash. Manually powering the ATECC608A almost doubles the rate due to the reduced wakeup time. Conversely, the energy consumption reduces by a factor of 1.5. Increasing the I2C from 100 to 400 kbps increases the rate by a factor of two and decreases the energy consumption respectively. Copying the internal state, however, costs performance. The state needs to be stored in the context `struct`, which adds 99 Byte. The overhead of this mechanism reduces the rate by a factor of three and affects the energy consumption similarly. All together energy and speed can vary by on order of magnitude.

8 Related Work

Crypto support in operating systems. mbed OS [5] is the ARM operating system for the Cortex-M family. It includes the SSL library mbed TLS [6], which implements symmetric and asymmetric cryptographic algorithms in software. mbed OS allows replacement of cryptographic functions by crypto-hardware implementations. An analysis of the performance advantages is missing, though. zephyr [44] does not include crypto-hardware, but support is planned for future releases[1]. Software-based crypto support is inherited from mbed TLS and TinyCrypt [20]. Similarly, mynewt [3] uses mbed TLS and TinyCrypt. A rudimentary crypto API provides `encrypt`/`decrypt` functions that leverage hardware capabilities for basic AES. Contiki-NG [2] provides sparse crypto support, though. With our crypto-extensions in RIOT we aim to bring more diversity in terms of hard- and software support to evaluate and deploy security in the IoT.

Performance of crypto-software. On very constrained devices, Gura *et al.* [19] compare RSA with ECC, Zhou *et al.* [46] present optimized implementations of SM2 and the NIST P-256 elliptic curve. These evaluations run bare metal based on software dedicated to specific hardware platforms which is in contrast to our study. We focus on a multi-purpose operating system and common crypto-libraries, and find that results are on par with bare metal implementations albeit our system favors more flexibility with respect to supported microcontroller platforms and peripherals. In the context of an operating system, processing overhead, memory, and energy were measured for symmetric cryptography (*i.e.,* AES) and secure hashing (*i.e.,* SHA and MD5) by Passing *et al.* [34] on NutOS and Tsao *et al.* [42] on Contiki [14].

Mössinger *et al.* [30] present runtime, memory, and energy consumption of elliptic curve cryptography in Contiki. Frimpong *et al.* [16] present an ECDH and ECDSA [1] implementations in Contiki-NG, using TinyCrypt. Kim *et al.* [24] ported the mbed TLS crypto-library to RIOT and FreeRTOS, and evaluate the processing time of ECDSA signature and verification on two platforms. Optimizations of elliptic curve cryptography are presented in [9, 13, 27]. In a comparative study of different elliptic curve libraries, Silde [38] shows that distinct optimizations for elliptic curves are vulnerable to side-channel attacks. This is one reason why we focus on common ECC.

Performance of crypto-hardware. Munoz *et al.* [31] present time and energy measurements for AES, running an SDK for software and hardware support on two platforms. Pearson *et al.* [35] compare the performance of peripheral and external crypto support for different symmetric and asymmetric operations, deploying Espressif and Arduino code, because a multi-platform OS was missing. Lachner *et al.* [25] assess time measurements for different ciphers and one asymmetric signature algorithm, operating three devices with Arduino firmware. wolfCrypt [43] includes crypto-hardware drivers and analyzed throughput of selected platforms. The library does not abstract hardware and thus cannot benefit from crypto-hardware on the OS level.

Gerez *et al.* [17] compare the power consumption of a TLS session using RSA and ECDHE between a Rapberry Pi and an IoT device with crypto-hardware. Mades *et al.* [28] compare the battery runtime of a TLS stack with and without hardware acceleration. Nofal *et al.* [32] analyse the TLS handshake and record layer and present the energy consumption of three elliptic curves and two RSA configurations on two hardware platforms, with and without crypto-acceleration. Schläpfer *et al.* [37] provide a brief performance comparison between new secure elements and DTLS, using mbed TLS as a software platform. Durand *et al.* [15] quantify the energy demands of OSCORE. Zhou *et al.* [45] argue for a reprogrammable FPGA approach to implement optimized cryptographic algorithms. Conti *et al.* [11] present a novel IoT platform architecture with AES acceleration and evaluate the benefit of hardware over software cryto. Their findings are on par with our study, however, the specific design contrasts our approach that focuses on off-the-shelf hardware and software implementations.

[1] https://docs.zephyrproject.org/latest/security/security-overview.html

9 Conclusion and Outlook

In this paper, to the best of our knowledge, we presented the first comprehensive comparison of multiple symmetric and asymmetric cryptographic algorithms, implemented in hard- and software and consistently evaluated on multiple constrained common IoT devices. For a representative set of crypto-peripherals as well as an external security device, we showed detailed system benchmarks to reveal design trade-offs when implementing secure crypto-hardware support on a multi-purpose operating system for constrained devices. Our results include: (*i*) *Crypto-peripherals outperform software in runtime and energy. The benefit increases with longer input lengths.* This contributes to node lifetime. On the downside, drivers introduce memory overhead. (*ii*) *Context sizes and stack utilization of crypto-hardware operate at a similar scale as crypto-software.* Device complexity unsurprisingly increases the overhead. (*iii*) *External crypto-devices are slow on symmetric crypto-operations, but their performance advances are notably on asymmetric crypto.* A small memory footprint enables cryptographic operations on very constrained devices. Furthermore, a collection of hardware based side-channel countermeasures provides additional resistance against attacks. On the downside, the I2C communication introduces an attack surface. (*iv*) *Special care is required with crypto-drivers. We found several vendor implementations with large optimization potentials.* Furthermore, different levels of hardware crypto support require a configurable environment with different layers of abstraction and software assistance. This is provided by the OS and contributes to code reusability and portability while exploiting hardware features. We hope that our results will help to prevent performance pitfalls in the future.

In future work, we will extend our analysis on hardware architectures that provide dedicated security subsystems such as TrustZone and measure their performance in a widened experimental setup. Part of this will be a focus on the abstraction of secure memory technologies and a performance evaluation using the novel security protocol standards ACE-OAuth and LAKE, which will make heavy use of hardware-accelerated crypto-operations.

10 References

[1] M. Al-Zubaidie, Z. Zhang, and J. Zhang. Efficient and Secure ECDSA Algorithm and its Applications: A Survey. *Int. Journal of Communication Networks and Information Security (IJCNIS'19)*, 11(1), 2019.

[2] Apache Software Foundation. Contiki-NG: The OS for Next Generation IoT Devices. `https://github.com/contiki-ng/contiki-ng`, last accessed 10-11-2020.

[3] Apache Software Foundation. Apache Mynewt. `https://mynewt.apache.org`, last accessed 07-17-2020.

[4] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. `https://github.com/relic-toolkit/relic`, last accessed 11-25-2020.

[5] ARM Ltd. Mbed OS. `https://www.mbed.com`, last acc. 07-17-2020.

[6] ARM Ltd. Mbed TLS. `https://tls.mbed.org`, l. acc. 07-17-2020.

[7] C. Ashokkumar, B. Roy, B. S. V. Mandarapu, and B. Menezes. "S-Box" Implementation of AES Is Not Side Channel Resistant. *Journal of Hardware and Systems Security*, 4:86–97, 2019.

[8] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch. RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT. *IEEE Internet of Things Journal*, 5(6):4428–4440, December 2018.

[9] D. J. Bernstein and T. Lange. Faster Addition and Doubling on Elliptic Curves. In K. Kurosawa, editor, *Advances in Cryptology — ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer, Berlin, Heidelberg, Germany, 2007.

[10] Cifra. A collection of cryptographic primitives targeted at embedded use. `https://github.com/ctz/cifra`, last acc. 10-11-2020.

[11] F. Conti, R. Schilling, P. D. Schiavone, et al.. An IoT Endpoint System-on-Chip for Secure and Energy-Efficient Near-Sensor Analytics. *IEEE Trans. on Circuits and Systems I*, 64(9):2481–2494, 2017.

[12] J. Daemen and V. Rijmen. AES Proposal: Rijndael, 1999.

[13] R. de Clercq, L. Uhsadel, A. Van Herrewege, and I. Verbauwhede. Ultra Low-Power Implementation of ECC on the ARM Cortex-M0+. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 1–6, New York, NY, USA, 2014. ACM.

[14] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *IEEE Local Computer Networks (LCN)*, pages 455–462, 2004. IEEE ComSoc.

[15] A. Durand, P. Gremaud, J. Pasquier, and U. Gerber. Trusted Lightweight Communication for IoT Systems Using Hardware Security. In *9th International Conference on the Internet of Things (IoT '19)*, pages 1–4, New York, NY, USA, 2019. ACM.

[16] E. Frimpong and A. Michalas. SeCon-NG: Implementing a Lightweight Cryptographic Library Based on ECDH and ECDSA for the Development of Secure and Privacy-Preserving Protocols in Contiki-NG. In *35th Symposium on Applied Computing (SAC '20)*, pages 767–769, New York, NY, USA, 2020. ACM.

[17] A. H. Gerez, K. Kamaraj, R. Nofal, Y. Liu, and B. Dezfouli. Energy and Processing Demand Analysis of TLS Protocol in Internet of Things Applications. In *International Workshop on Signal Processing Systems (SiPS '18)*, pages 312–317. IEEE, 2018.

[18] C. Gündogan, C. Amsüss, T. C. Schmidt, and M. Wählisch. IoT Content Object Security with OSCORE and NDN: A First Experimental Comparison. In *Proc. of 19th IFIP Networking Conference*, pages 19–27, Piscataway, NJ, USA, June 2020. IEEE Press.

[19] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, vol. 3156 of *LNCS*, pages 119–132, 2004.

[20] Intel Corporation. TinyCrypt Cryptographic Library. `https://github.com/intel/tinycrypt`, last accessed 07-17-2020, 2017.

[21] Ken MacKay. micro-ecc. `http://kmackay.ca/micro-ecc/`, last accessed 10-11-2020.

[22] P. Kietzmann, T. C. Schmidt, and M. Wählisch. A Guideline on Pseudorandom Number Generation (PRNG) in the IoT. Technical Report arXiv:2007.11839, Open Archive: arXiv.org, July 2020.

[23] K. H. Kim, J. Choe, S. Y. Kim, N. Kim, and S. Hong. Speeding up Elliptic Curve Scalar Multiplication without Precomputation. *IACR Cryptol. ePrint Arch.*, (Report 2017/669), 2017.

[24] Y.-S. Kim and G. Kim. A Performance Analysis of Lightweight Cryptography Algorithm for Data Privacy in IoT Devices. In *Inform. and Comm. Techn. Convergence (ICTC '18)*, pages 936–938. IEEE, 2018.

[25] C. Lachner and S. Dustdar. A Performance Evaluation of Data Protection Mechanisms for Resource Constrained IoT Devices. In *Intern. Conf. on Fog Computing (ICFC '19)*, pages 47–52. IEEE, 2019.

[26] C. H. Lim and P. J. Lee. More Flexible Exponentiation with Precomputation. In *CRYPTO'94*, pages 95–107, 1994. Springer.

[27] P. Longa and C. Gebotys. Novel Precomputation Schemes for Elliptic Curve Cryptosystems. In *Applied Cryptography and Network Security (ACNS'09)*, pages 71–88, Berlin, Heidelberg, 2009. Springer.

[28] J. Mades, G. Ebelt, B. Janjic, F. Lauer, C. C. Rheinländer, and N. Wehn. TLS-Level Security for Low Power Industrial IoT Network Infrastructures. In *Design, Automation Test in Europe Conference Exhibition (DATE '20)*, pages 1720–1721. IEEE, 2020.

[29] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[30] M. Mössinger, B. Petschkuhn, J. Bauer, et al.. Towards quantifying the cost of a secure IoT: Overhead and energy consumption of ECC signatures on an ARM-based device. In *World of Wireless, Mobile and Multim. Networks (WoWMoM'16)*, pages 1–6. IEEE, 2016.

[31] P. S. Munoz, N. Tran, B. Craig, B. Dezfouli, and Y. Liu. Analyzing the Resource Utilization of AES Encryption on IoT Devices. In *Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC '18)*, pages 1200–1207. IEEE, 2018.

[32] R. A. Nofal, N. Tran, C. Garcia, Y. Liu, and B. Dezfouli. A Comprehensive Empirical Analysis of TLS Handshake and Record Layer on IoT Platforms. In *22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWIM '19)*, pages 61–70, New York, NY, USA, 2019. ACM.

[33] K. Okeya and T. Takagi. The Width-w NAF Method Provides Small Memory and Fast Elliptic Scalar Multiplications Secure against Side Channel Attacks. In M. Joye, editor, *Topics in Cryptology — CT-RSA 2003*, vol. 2612 of *LNCS*, pages 328–343. Springer, 2003.

[34] M. Passing and F. Dressler. Experimental Performance Evaluation of Cryptographic Algorithms on Sensor Nodes. In *IEEE Mobile Ad Hoc and Sensor Systems (MASS'06)*, pages 882–887. IEEE, 2006.

[35] B. Pearson, L. Luo, Y. Zhang, R. Dey, Z. Ling, M. Bassiouni, and X. Fu. On Misconception of Hardware and Cost in IoT Security and Privacy. In *53rd International Conference on Communications (ICC '19)*, pages 1–7. IEEE, 2019.

[36] M. Rottleuthner, T. C. Schmidt, and M. Wählisch. Eco: A Hardware-Software Co-Design for In Situ Power Measurement on Low-end IoT Systems. In *ACM SenSys, WS ENSsys*, pp 22–28, Nov. 2019. ACM.

[37] T. Schläpfer and A. Rüst. Security on IoT Devices with Secure Elements. Technical report, 2019.

[38] T. Silde. Comparative Study of ECC Libraries for Embedded Devices. `https://tjerandsilde.no/files/Comparative-Study-of-ECC-Libraries-for-Embedded-Devices.pdf`, last accessed 10-09-2020, 2019.

[39] E. G. Straus. Addition chains of vectors (problem 5125). *American Mathematical Monthly*, 70:806–808, 1964.

[40] The Linux Kernel Development Community. Kconfig Language. `https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html`, last accessed 28-09-2020.

[41] E. Tromer, D. A. Osvik, and S. Adi. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[42] B. Tsao, Y. Liu, and B. Dezfouli. Analysis of the Duration and Energy Consumption of AES Algorithms on a Contiki-Based IoT Device. In *Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous '19)*, pages 483–491, NY, USA, 2019. ACM.

[43] wolfSSL Inc. wolfCrypt Embedded Crypto Engine. `https://www.wolfssl.com/products/wolfcrypt/`, last acc. 10-11-2020.

[44] Zephyr Project. Zephyr. `https://www.zephyrproject.org`, last accessed 07-17-2020, 2020.

[45] B. Zhou, M. Egele, and A. Joshi. High-performance low-energy implementation of cryptographic algorithms on a programmable SoC for IoT devices. In *High Performance Extreme Computing Conference (HPEC'17)*, pages 1–6. IEEE, 2017.

[46] L. Zhou, C. Su, Z. Hu, S. Lee, and H. Seo. Lightweight Implementations of NIST P-256 and SM2 ECC on 8-bit Resource-Constraint Embedded Device. *ACM TECS.*, 18(3):1–13, 2019.