2022 21st ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)

Poster Abstract: Offloading Crypto Processing with RIOT

Lena Boeckmann
HAW Hamburg
lena.boeckmann@haw-hamburg.de

Peter Kietzmann
HAW Hamburg
peter.kietzmann@haw-hamburg.de

Thomas C. Schmidt
HAW Hamburg
t.schmidt@haw-hamburg.de

Matthias Wählisch
Freie Universität Berlin
m.waehlisch@fu-berlin.de

ABSTRACT

Secure elements allow for offloading complex crypto operations from embedded devices to external, protected hardware. In this poster, we present a concept for transparently accessing multiple secure elements behind a unified API as a feature of an IoT OS.

1 INTRODUCTION

The Internet of Things (IoT) needs to be secured and many standard protocols require the perpetual use of crypto operations [5]. Features of IoT nodes vary widely, and modern devices include hardware-based crypto assistance to improve the performance and security of keys. Figure 1 presents a classification of crypto-hardware features in the IoT: (1) devices without hardware assistance (*e.g.,* Microchip ATmega2560) which rely on crypto-software libraries to enable security; (2) devices with a peripheral accelerator for crypto operations (*e.g.,* Nordic nRF52840); (3) devices with a peripheral accelerator that operates on keys located in protected internal storage (*e.g.,* Nordic nRF9160); (4a) and (4b) external devices, so called 'secure elements' (SEs) that connect through a peripheral bus (*e.g.,* Microchip ATECC608A, STM STSAFE-A100). SEs are promising for augmenting the constrained IoT. They provide entropy and random numbers in hardware; key generation and storage in tamper proof key slots; SEs offload numerous cryptographic tasks from the main processor and execute in an isolated environment, enabling security even on very constrained platforms. As an example, the ATmega2560 microcontroller provides 256 kB ROM/8 kB RAM which is barely enough to operate a network stack *without* security. On this platform, crypto-software libraries consume too much memory [4] to coexist with the regular firmware.

Vendors commonly provide driver code to access their hardware, each of which exposes a vendor-specific API of special semantic. For example, an SE requires an ID-based access to an internal key, which never leaves the device. This is much unlike common crypto APIs that require the key as direct input. This harms usability, since (*i*) developers are confronted with numerous APIs for the same operation. (*ii*) These APIs are often specific to the hardware and algorithm, and (*iii*) not well established among developers (*e.g.,* unlike POSIX, PKCS, ...). Green *et al.* [3] emphasize that system security depends on the usability of the API, which covers aspects such as developer familiarity, high level access to crypto-primitives, secure default values, available example code, *etc.* . The *ARM Platform Security Architecture (PSA) Framework* [1] provides guidelines for developing secure IoT systems and contains a developer-centric design of the *PSA Crypto API* with tests and documentation.

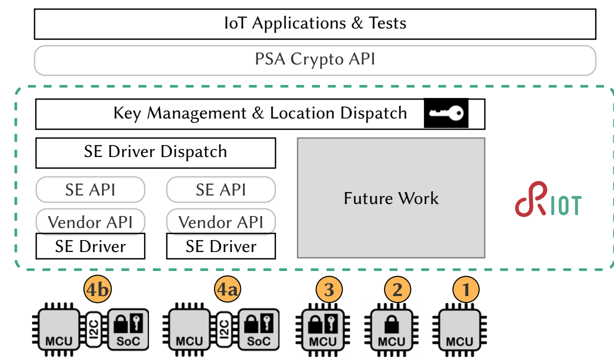

Figure 1: Integration concept of the PSA Crypto API in RIOT.

Recent deployments on the IoT increasingly make use of an operating system (OS) to benefit from hardware abstraction layers, as well as pre-provisioned drivers, (crypto-)libraries, and protocols. In this poster, we argue that the OS should provide transparent access to the plethora of crypto-backends through a unified system level API. We present our integration concept of the PSA Crypto API to RIOT [2], the operating system for low-end IoT devices. Thereby, we focus on SE access, and leave other crypto-hardware backends for future work. In Section 2, we introduce our device location- and SE dispatch mechanism that enables accessing (*i*) multiple SEs to increase the number of key slots, and (*ii*) different SEs, to exploit different hardware features. In Section 3, we measure the control overhead of our SE management on real-world hardware, comparing the bare vendor driver and the user access API.

2 INTEGRATING A SECURE ELEMENT

Indirect Key Management. The user facing PSA Crypto API bases on indirect key management. Keys are accessed through identifiers and never exposed to the user. Fig. 2 presents an overview of the internal PSA data structure storing a key in a *PSA Key Slot*. It is specified that a key shall be described by its *Key Attributes*. On generation, keys get assigned a unique *Key ID* under which its location can be described. The *Location* field provides 24 Bit to reference a key in local memory or some other location. Our specific SE integration reserves a range of values to use for SEs and requires a static assignment for each SE connected. We extend the RIOT startup function `auto_init` to automatically initialize and register each SE with the SE management module when booting the

978-1-6654-9624-7/22/$31.00 ©2022 IEEE
DOI 10.1109/IPSN54338.2022.00068

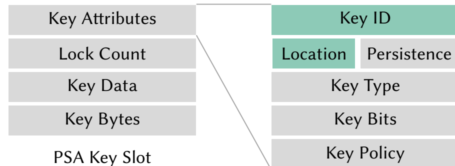

Figure 2: Data structures with key material (left) and info.

OS. During registration, a device handle and the assigned location values of all SEs are stored in a global driver list.

Device Driver Interface. Each type of SE implements the internal OS interface for secure elements. We utilize the interface proposed by ARM[1]. We provide the glue code that maps the vendor drivers to the SE interface, as well as the mechanism to switch between multiple devices. A structure containing pointers to the driver methods is stored in the driver list, together with the location value and device handle.

Location and Driver Dispatch. We contribute a dispatching unit (see Fig. 1) to mediate the crypto operation request of a user to a backend. API calls are dispatched in two steps, utilizing the *Location* value of the key in use. First, our location dispatch distinguishes between internal or external source, the latter of which exploits the existing SE interface which allows us to operate multiple SEs in parallel. Second, the SE driver dispatch selects the SE behind the *Location*. This is obligatory since an operation must be executed on the SE that holds the key.

Example Flow. To generate an elliptic curve key pair an application passes *Key Attributes* with the *Location* of the destination SE to the key generation function. Our location dispatcher checks the location, fetches the associated SE from the global list and invokes a key generation method implemented by that device. A key is generated, stored in a free key slot on the SE and the slot number is returned. Our key management module stores the *Key Attributes* along with the slot number and assigns an ID, which is returned to the application. To use that key, the application passes the ID to the cryptographic function and our SE integration mediates the call to the corresponding SE backend.

3 EVALUATION ON IOT PLATFORMS

Tab. 1 compares the execution time and memory sizes of our RIOT implementation on two platforms that connect the ATECC608A SE via I2C to perform crypto operations. (*i*) ATmega2560 (8-bit AVR) provides 8 kB RAM/256 kB ROM and operates at 16 MHz. (*ii*) nRF52840 (32-bit ARM Cortex-M4) provides 256 kB RAM/1 MB ROM and operates at 64 MHz. Our device choice reflects the lower and upper end of common IoT platforms.

The single threaded measurement application utilizes the PSA Crypto API to generate an elliptic curve key pair and performs an ECDSA signature/verification on the SE. The execution time of elliptic curve key generation takes ≈88 ms when connected to the nRF52840; calculating a signature takes ≈95 ms, and verification ≈49 ms. With the ATmega2560, each operation requires roughly 1 ms longer due to slower device control. Note, this overhead is independent of our SE management. Our SE management adds 14 μs (key generation) and 9 μs (signature and verification) on the nRF52840. This includes retrieving the SE from the driver list. On the ATmega2560, the overhead increases by a factor of ten, due to the lower operation frequency. Still, in comparison to the execution time of the cryptographic algorithms, the management overhead remains negligible on both platforms. We further analyze the overhead of multiple SEs connected to the nRF52840 which increases by only 1 μs. We excluded these results, which do not contribute additional insights.

[1]https://armmbed.github.io/mbed-crypto/psa/se/

Table 1: Time & memory consumption of PSA Crypto. Overheads indicate additions to pure crypto-operations on an SE.

	Processing Time [ms]			Memory [B]	
Platform	Key Gen	Sign	Verify	RAM	ROM
nrf52840 + SE	88.140	95.327	49.435	108	6700
PSA Overhead	↑ 0.014	↑ 0.009	↑ 0.009	↑ 389	↑ 4862
ATmega2560 + SE	89.269	96.17	50.385	763	11734
PSA Overhead	↑ 0.133	↑ 0.095	↑ 0.095	↑ 689	↑ 10026

Memory analyses in Tab. 1 display the pure crypto overhead and ignore the OS offset. On the nRF52840, PSA adds 389 Bytes of RAM. Besides operational memory, this includes two *PSA Key Slots* (108 Bytes each) to hold the references (compare Sec. 2) to private and public keys on the SE. 32 Bytes of memory are needed for each SE driver instance in the global driver list. ROM requirements (6.7 kB driver + 4.8 kB PSA) are more notable, however, the platform provides enough flash so the overhead remains negligible. Surprisingly the memory overhead on the ATmega2560 doubles. We contribute this to more complex low level instructions on the 8-bit architecture and further investigate this in future work. It is noteworthy, however, that all operations of the SE can be executed (*e.g.,* random number generation, symmetric ciphers, hashes) without additional memory. As a counter-example, a comparable firmware with software-crypto already requires ≈3.3 kB RAM/22 kB ROM on the ATmega2560. Hence, an overhead of 689 Bytes in RAM reveals efficient SE utilization, alongside additional security of keys.

4 CONCLUSIONS AND OUTLOOK

Our PSA integration for SEs adds seamless hardware-crypto support to the open source OS RIOT and fosters portability of applications and tests. This enables security even on very constrained IoT nodes that cannot operate crypto-software libraries. Our measurements show that the management overhead is negligible in runtime and small in memory, prospecting resource-independent operation of multiple SEs. Following our research agenda, we will (*i*) integrate internal crypto-accelerators to our concept, and (*ii*) enable transparent access to operate in trusted execution environments such as the ARM TrustZone or RISC-V PMP.

REFERENCES

[1] ARM. Platform Security Architecture. https://developer.arm.com/architectures/architecture-security-features/platform-security, last accessed 09-28-2021.
[2] Baccelli, et al. 2018. RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT. *IEEE Internet of Things Journal* 5, 6 (Dec. 2018), 4428–4440.
[3] Matthew Green and Matthew Smith. 2016. Developers are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security and Privacy* 14, 5 (2016), 40–46.
[4] Kietzmann, Boeckmann, Lanzieri, Schmidt, and Wählisch. 2021. A Performance Study of Crypto-Hardware in the Low-end IoT. In *EWSN '21*. 12.
[5] Lanzieri, Kietzmann, Schmidt, and Wählisch. 2022. Secure and Authorized Client-to-Client Communication for LwM2M. In *Proc. of ACM/IEEE IPSN '22*.