WooKey: USB Devices Strike Back

Ryad Benadjila, Mathieu Renard, Philippe Trebuchet, Philippe Thierry, Arnauld Michelizza, Jérémy Lefaure
firstname.lastname@ssi.gouv.fr

ANSSI

Abstract. The USB bus has been a growing subject of research in recent years. In particular, securing the USB stack (and hence the USB hosts and devices) started to draw interest from the academic community since major exploitable flaws have been revealed by the BadUSB threat [41]. The work presented in this paper takes place in the design initiatives that have emerged to thwart such attacks. While some proposals have focused on the host side by enhancing the Operating System's USB sub-module robustness [53, 54], or by adding a proxy between the host and the device [12, 37], we have chosen to focus our efforts on the device side. More specifically, our work presents the WooKey platform: a custom STM32-based USB thumb drive with mass storage capabilities designed for user data encryption and protection, with a full-fledged set of in-depth security defenses. The device embeds a firmware with a secure DFU (Device Firmware Update) implementation featuring up-to-date cryptography, and uses an extractable authentication token. The runtime software security is built upon EwoK: a custom microkernel implementation designed with advanced security paradigms in mind, such as memory confinement using the MPU (Memory Protection Unit) and the integration of safe languages and formal methods for very sensitive modules. This microkernel comes along with MosEslie: a versatile and modular SDK that has been developed to easily integrate user applications in C, Ada and Rust. Another strength of this project is its core guiding principle: provide an open source and open hardware platform using off-the-shelf components for the PCB design to ease its manufacturing and reproducibility.

Disclaimer

The WooKey project is a work in progress. Most of the objectives described in the article have been implemented, but some features are still under development and testing.
For this reason, it is difficult to publish the code at the time of the SSTIC 2018 conference.

We are well aware that this can be confusing for a project that claims to be open source and open hardware.

We emphasize, however, that we focuse all our efforts to make the project available as soon as possible with end of Q2 2018 as a target.

1 Introduction

USB devices are nowadays ubiquitous and participate in a wide variety of use cases. Recent studies have exposed vulnerabilities in USB implementations [39], and among them the BadUSB [41] attacks are a serious threat against the integrity of USB devices. Firmwares, hosts Operating Systems, as well as user data confidentiality are at risk. As a matter of fact, this can have critical consequences knowing that USB mass storage devices are used to transfer public or confidential data between different machines, including in air-gapped networks.

Some proprietary devices [8, 9] are already sold as preventive solutions against the BadUSB class of attacks. They however lack code or architecture/design review, sometimes yielding crucial defects [44]. The academic community has, for its part, focused on the host side by enhancing the Operating Systems USB sub-module robustness [53, 54] and by developing filtering proxies [12, 37]. Such approaches, albeit interesting, can be non-portable and do not protect the USB device itself when it is lost or falls into the hands of adversaries.

Such limitations inclined us to prototype a secure and trustworthy USB mass storage device. This article compiles the results of this initiative and provides insight into how we designed and implemented WooKey (the prototype platform name) using off-the-shelf components with open source and open hardware objectives.

We first provide a security and threat model for USB mass storage thumb drives, thus highlighting the main challenges of designing a secure device. Then, we discuss the existing public products that adopt such functionalities from an end-user's perspective, introducing WooKey with what (we believe) brings new security features to embedded platforms that use off-the-shelf components.

This leads us to detail our hardware and software design choices in the light of the security expectations, bringing insights into our main contributions, namely:

- A full-featured USB dongle platform with *in-depth defense* in mind.
- Software isolation using a *microkernel* with a novel approach regarding MPU usage on constrained microcontrollers, embedding critical parts written in safe languages (Ada with SPARK).
- Secure *Device Firmware Update*.
- Two-factor *user authentication* using a smartcard and up-to-date cryptography.

Finally, the last section wraps up an analysis of the resulting security against our threat model and discusses the residual risks, thus assessing the limitations of an architecture based on off-the-shelf components.

1.1 Threats on USB devices

Fully understanding the threats on the USB stack requires a deep knowledge of the USB devices architecture. For the sake of simplicity, we will focus on the *mass storage class* (commonly named USB thumb drives) although the risks and concepts can be generalized.

USB devices hardware architecture: Two platform variants are distinguished in the USB protocol: *hosts* (a.k.a. masters) and *devices* (a.k.a. slaves). Both usually embed a controller chip in the form of a dedicated microcontroller or an Application-Specific Integrated Circuit (ASIC), whose role is to receive and transmit USB packets.

Microcontrollers are usually based on very low power Systems-on-a-Chip (SoC) with embedded persistent storage and usually reduced performance – few MHz CPUs, few kilobytes of RAM – compared to general purpose processors. They usually embed a firmware that can be reprogrammed either by logical means (e.g. through the USB protocol itself) or physical access (e.g. through flash banks hardware reflashing).

Among all USB devices, the mass storage class aims at storing user data on a dedicated high capacity persistent memory. Figure 1 provides a high level view of a thumb drive: the USB microcontroller interacts with the host USB stack on one side, and with flash storage banks on the other side.

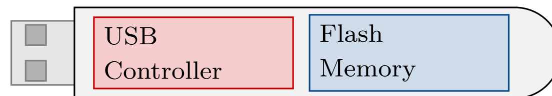

Fig. 1. USB thumb drives classical architecture

USB devices weaknesses: The flexibility of the USB bus is also a weakness. Since various device classes can be plugged into the same connectors, a device can impersonate another one without any user action or notification. This can be performed by reprogramming the USB controller embedded in the device as it has been shown in the BadUSB class of

attacks [41]: many USB controller chips lack protections against such reprogramming.

A common exploitation scenario is the *HID Payload Attack*: a malicious device is reprogrammed in order to act as a Human Interface Device (e.g. a keyboard) and enter custom keystrokes on the target machine to compromise it [26].

Another path for the attacker is to inject a crafted payload to exploit a (possibly zero-day) vulnerability in the host USB stack or any software layer using it, and adapting the payload to the host through fingerprinting [18, 23, 35].

Without specific hardware, the only way to protect critical host systems against such threats is to physically disable the USB ports. Any other software countermeasure is of limited efficiency: generic blacklists can be bypassed; USB filtering proxies [54] are based on complex USB stacks and are subject to the same risks. In addition, modern PCs contain low-level embedded firmwares that also integrate USB capabilities: countermeasures implemented at the Operating System level do not cover them. When targeting a computer's BIOS or UEFI, a successful exploitation opens a breach in the most privileged parts of the system.

A possible mitigation would be to only use *trusted devices*. This raises the following question: what is exactly a trusted device and how can one trust it? This inquiry has been the starting point of the WooKey project, and we tried to provide technical and well-founded solutions to this issue.

1.2 Secure USB thumb drive design

This section discusses all the elements that, according to us, should be used in secure hardware and software USB thumb drive architectures. The main goal is to provide the reader with a high-level view of the assumptions we make, the threats we try to protect against, and the security features we desire.

Although these elements are not based on formal foundations, we outline the fact that they represent what we believe is an interesting working framework to build a secure and trusted USB device. We also stress out that to our best knowledge, there is no detailed analysis of the desirable design features and threat model of USB thumb drives.

Functional specifications: The device must provide classical USB mass storage features with transparent *user data encryption and decryption*. It shall be detected as a thumb drive on any USB host (i.e. any classical Operating System) with no specific software installation.

Threat Model: We consider that the adversary has logical and/or physical access to the device:

1. The adversary may try to read the data simply by connecting the device to a host or by physically reading the mass storage cells, for example when the device is lost or stolen. This can be done either when the device is powered up, or when it is powered down.
2. The adversary may try to tamper with the device using logical attacks, for example when it is connected to an untrusted host. These attacks abuse potential weaknesses in protocols used for external communication such as the USB stack or the external data storage buses.
3. The adversary may open the device to physically tamper with the internal storage, firmware, or any other component present on the actual device.
4. We suppose that an *external authentication token* is used to validate the legitimate user presence. We will only consider physical attacks where the adversary does not possess the legitimate user's PIN code. In other words, side-channel and fault injection attacks on the device in a post-authentication phase are explicitly out of scope (even though we discuss them in section 4). Those kinds of attacks are considered during the pre-authentication phase though, either on the device, on the external token, or on the communication channel when these two exchange data.

Security expectations: We expect our device to provide the following main security features:

1. *User data protection*: all data at rest are encrypted, and their confidentiality protected. The data integrity is out of scope.
2. *Strong user authentication*: the legitimate user must be present when data is decrypted (implying a strong user authentication). When a user PIN code is used, attack vectors that can steal it must be limited.
3. *Secure device software update*: the device's software should be robustly upgradable for system maintenance (e.g. security patches). Update files must be authenticated and integrity checked with no rollback to (possibly buggy) old versions. A software upgrade must be a voluntary and authenticated action. The firmware updates must be reliable and must avoid bricking the platform.
4. *Firmware robustness against software attacks*: the firmware should guarantee that an adversary attacking the exposed software surface (on the USB bus for instance) is not able to get privileged access to

the platform, and does not gain access to critical material such as sensitive cryptographic keys. Software attacks must remain confined.

1.3 A survey of secure USB devices

When it comes to user data encryption in a secure USB device, various solutions already exist in both proprietary and open source products. This section discusses their design choices (when available), their advantages as well as their drawbacks.

Proprietary products: We will move fairly swiftly over commercial and proprietary solutions such as IronKey[1], Kingston DataTraveler[2], and all other similar devices [1, 5]. The reason is that the details about their internal architectures and the cryptography they use are usually very limited. This opacity does not allow us to put their security mechanisms under scrutiny.

For a broad overview of proprietary products, one can refer to [52]. It is worth noting that from outdated cryptography to unsafe external authentication methods, many of such products do not implement state of the art software and hardware security concepts, yielding in various attack vectors [24, 44].

Besides proprietary products, a few open source endeavors exist. We have only focused on what we believe are the most relevant solutions. Even though some of them do not aim at producing USB mass storage devices, many security features and/or security goals they target intersect with the threat model and the expectations that have been previously introduced. We discuss in the remaining of the section their benefits as well as their limitations.

USB Armory: USB Armory[3] is one of the first open source and open hardware USB stick with rich features that has been brought to public attention. The platform aims at embedding a small yet full-featured development board capable of booting a Linux distribution in a USB thumb drive form factor. It is built around a NXP Cortex-A8-based `i.MX53` SoC, and interestingly showcases advanced security features[4] based on

[1] `http://www.ironkey.com/en-US/`
[2] `https://www.kingston.com/en/usb`
[3] `https://inversepath.com/usbarmory`
[4] `https://github.com/inversepath/usbarmory/wiki/Hardware-security-features`

both the NXP HABv4 (High Assurance Boot) secure boot module and the ARM TrustZone isolation primitive. Even though such a platform has not been primarily designed as a data encryption device, one could easily build one using the Key Encryption Module.

However, these SoC advanced security features have not (at least publicly) been under the scrutiny of well-tried evaluation schemes such as Common Criteria: we cannot compare them to secure elements. Recent breaches discovered in the HABv4 [11] outline this matter of fact.

Finally, it is worth noting that the USB Armory device does not feature strong user authentication methods *per se*, but the flexibility of this platform allows to extend it and to add such modules through the exported buses and interfaces.

Ledger: this company develops USB hardware-based cryptocurrency wallets[5]. Their leitmotiv is to provide a device which ensures the end user that her wallet private keys are kept safe and are never stolen. The platform is based on a dual chips design: a STM32 general purpose MCU for USB communication, and a ST31 secure element. The sensitive cryptographic operations are performed in the ST31 enclave so that critical keys and assets never leave it. User identification requires a personal PIN code and the device either embeds a touch screen (Ledger Blue) or buttons and LCD screen (Ledger Nano S) for a safe authentication.

An innovative approach of Ledger is the introduction of BOLOS [34] (Blockchain Open Ledger Operating System), an OS built for software isolation between a *normal* world and a *secure* world through a controlled and dedicated API. The MPU isolation paradigm used by BOLOS to enforce application contexts is interesting: this will be detailed in section 3.6. Although Ledger has released some open source projects on GitHub[6], only the BOLOS API (the SDK to compile applets) seems to have been released. It is also worth noting that Ledger's products are not open hardware, and their detailed architecture cannot be thoroughly analyzed.

Without providing Common Criteria or equivalent certification results, it is quite difficult to evaluate the security level of such a platform compared to historical and time-tested smartcard embedded systems (though these two are not incompatible as described in [13]). Finally, the ST31 chip of Ledger devices is soldered on the PCB: we believe that a physical separation of the functional platform and the authentication token is crucial for our specific use cases as it will be argued in section 3.2.

[5] https://www.ledger.fr/hardware-wallets/

[6] https://github.com/ledgerhq

TREZOR: the TREZOR bitcoin wallet[7] is an open source project that suffers from neither using a secure element nor strong authentication such as Ledger's products. The lack of a secure element results in various attack vectors described in [2]. Weaknesses of the `STM32F205` (that stores all the TREZOR user secrets) against fault injection attacks have been exploited on the PIN verification implementation in [22].

Nitrokey: the Nitrokey family of devices[8] is probably the closest to our high-level expectations, at least in terms of advertised user functionality. These USB devices have emerged as a response to the BadUSB threat [40]. According to the authors, their main features consist of an open source and open hardware design with a firmware that cannot be updated through USB (hence preventing a BadUSB host to device attack vector). Moreover, a secure element in the form of a smartcard chip ensures a PIN based user authentication. The Nitrokey tokens are versatile: they offer an OpenPGP standard API through USB, along with mass storage and data encryption features.

On the hardware side, the main variants of the Nitrokey family use either a `STM32F103` or a Microchip AVR `AT32UC3A3256S`, both being USB oriented MCUs. For sensitive operations and keys protection, a secure element is explicitly used in the Nitrokey Pro, HSM and Storage variants. The reference of the chip depends on the Nitrokey product version, with at least a Common Criteria certified chip (and a certified Javacard platform for some of them) [4].

The Nitrokeys have the undeniable laudable advantage of being one of the first open source and open hardware initiatives against BadUSB. They however lack some crucial security features. First, they don't make use of an integrated user input system allowing secure PIN typing: the user authentication is performed on the host PC using the Nitrokey-App software, therefore allowing a compromised host to sniff it. Secondly, the firmware embedded in these products (at least the open source published versions) does not make use of dedicated kernel isolation and in-depth defense techniques: any software vulnerability leads to a complete compromise of the platform.

Furthermore, it seems that Nitrokeys do not enforce a *secure channel* between the USB MCU and the secure element: the secure element is not cryptographically personalized for a given platform and user (the only binding with the user is done with the PIN code, which is limited).

[7] `https://trezor.io/`
[8] `https://www.nitrokey.com/`

Finally, and as stated on Nitrokey's GitHub account, there is no secure firmware update using strong cryptography at this time.

1.4 Introducing WooKey

When compiling all the desirable security features that one wants for a secure USB device, no open source solution seems to offer a comprehensive answer. Proprietary solutions being out of scope since no architecture and code review are possible, the WooKey project has emerged. It aims at prototyping a secure and trusted USB mass storage device featuring user data encryption, with fully open source and open hardware foundations. We outline the fact that even though the prototype focuses on the *mass storage* USB class, all the security concepts we describe in the current article are easily portable to other USB device classes such as HID or CDC. The comparison between open source solutions and WooKey regarding security features is summarized in Table 1.

	USB Armory	Ledger	TREZOR	Nitrokeys	WooKey
Open Source	✓	~[1]	✓	✓	✓
Open Hardware	✓		✓	✓	✓
Secure Element		✓		✓	✓
Dedicated PIN pad		✓	✓		✓
Isolation Kernel	✓	✓			✓
Secure Firmware Update	✓[2]	✓	?	~[3]	✓
Secure Boot	✓[4]				

[1] Not all the elements are open source.
[2] Not implemented *per se*, but should inherit from open source projects.
[3] Firmware update is controlled, but not cryptographically sound.
[4] Although broken, see [11].

Table 1. Comparison between open source solutions and WooKey

Constraints related to WooKey: An important matter that the project pursues is that *any interested person* should be able to manufacture, flash and use its own device at will. This latter feature is very restrictive for our design: many interesting security components, such as secure boot and secure elements bare-metal development, are *under NDA (Non-Disclosure Agreements).*

Besides this embargo on security-related technologies, the "do it yourself" aspect would be refrained by the so-called *small scale issue*: many

hardware manufacturers do not retail small volumes, and will only deal with big companies that buy at least thousands of pieces. Such economical aspects are interesting, and we will discuss them later: we want WooKey to be manufactured in reasonable volumes for a reasonable price.

As it will be detailed in the next sections, we do not aim at *perfect security* as we believe that such an ideal paradigm is too difficult to achieve using only off-the-shelf components. However, our leitmotiv – underlined in the sequel – is to observe that a high level of security can be achieved notwithstanding some compromises. The crucial ingredient is to control the attack surface (i.e. attack scenarios and limits) that our platform covers, and to document the (un)achievable security features.

WooKey security overview: The security model is based on both hardware and software primitives designed to bring in-depth security. Hardware security relies on an extractable token embedding a secure element. This token is meant to provide *pre-boot authentication* as well as a secure storage area for the sensitive master keys of WooKey user data encryption.

Software security relies on a microkernel that enforces privilege separation, memory isolation, W⊕X principle, stack and heap anti-smashing techniques. The most sensitive parts are implemented with a safe language (SPARK/Ada).

The secure update mechanism over USB is based on the DFU (Device Firmware Update) protocol [29]. It also uses the pre-boot user authentication feature to strengthen the security of the platform. Firmware integrity and authenticity are based on state of the art cryptography.

2 Hardware Architecture

2.1 General hardware design

Hardware specifications: The functional and security inputs of the WooKey specifications lead to natural design choices and/or requirements when it comes to the hardware platform.

First, the processor at the heart of the design must embed a Memory Management Unit (MMU) or at least a Memory Protection Unit (MPU). These two hardware IPs provide a necessary privilege level separation between a supervisor mode and a user mode. A MMU usually isolates tasks memory using pagination, allowing two tasks to handle (virtual) pages with different access rights and pointing to the same physical memory

space. A MPU, usually implemented in embedded devices, allows a more coarse-grained isolation: physical memory is split in distinct regions with associated rights, and the number of simultaneous regions is often limited yielding in much less flexibility than pages.

Secondly, a cryptographic accelerator must be present to guarantee fast user data encryption. In order to achieve good performance on the USB side, the controller must be compatible with the USB High Speed (USB 2.0) specification.

Strong user authentication must be provided through the usage of an external token, which securely embeds the sensitive master keys of the platform. The firmware must remain authentic during the life-cycle of the product, and be only updated through controlled means: debug functionalities provided by the SoC manufacturer such as Joint Test Action Group (JTAG) or Serial Wire Debug (SWD) interfaces must be reliably deactivated.

Since the platform design will be open source, all components and their data-sheets must be publicly available. The platform should have a good security *versus* price ratio.

We detail in the next sections the rationale behind our specific choices for the hardware components.

2.2 USB controller choices

Using a microcontroller with an embedded firmware or an Application-Specific Integrated Circuit (ASIC) emerged as the optimal choice to properly implement a fully operational USB stack. The alternative of using a Field-Programmable Gate Array (FPGA) dedicated to the USB functions was too expensive compared to integrated SoCs, over and above their availability issues for small quantities. From this standing point, we eventually faced two options:

- Either use dedicated ASICs abstracting low-level protocol communication and exposing a simple interface [28].
- Or use microcontrollers (MCUs) with an embedded firmware.

We have chosen to use a MCU since it allows to reduce the complexity of the PCB thanks to the integration level (many functions are embedded on the same piece of silicon). Specifically, we have focused on the 32-bit ARM-based Cortex-M cores: they embed a MPU, are compact and energy efficient, and they provide desirable security features. Some of them allow us to override/prevent any proprietary code execution (BootROM): we

consider such privileged and unreviewed code as a possible important threat. This is all the more true when considering that any discovered vulnerability in this piece of software cannot be patched on the already deployed SoCs [11]. Some Cortex-M-based MCUs offer the possibility to disable debug interfaces in production and to prevent flash memory Read/Write/Erase operations (Read protection allows protecting against dumping sensitive data, Write/Erase protection allows preserving the integrity of the firmware).

We emphasize here that even though general purpose MCUs propose security features, they are not secure elements. A recent study has completely broken the NXP CRP (Code Read Protection) on the LPC microcontrollers family [10] using a power glitch during the bootROM code check of the CRP status. Another article [42] also circumvents `STM32F0` RDP (Readout Protection) to recover the firmware embedded in the flash using more invasive means (acid decapsulation to access the die and light-based fault injection).

Even though these attacks involve more or less intrusive vectors, they demonstrate the relative *frailty* of these features. This is why the WooKey platform does not rely *solely* on them, and includes such possibly broken features in the residual threats analysis.

Looking at the Cortex-M lines of the microcontrollers providers (Qualcomm/NXP, Atmel/Microship, STMicroelectronics) that include USB capabilities, we highlight several components families that would fit our need in Table 2. After examining their respective features, we chose to discard some of them. The NXP LPC43xx series do not include a bootROM override feature, and their Big/Little architecture increases the attack surface. The Cortex-M7-based Atmel SAMx7 and alike SoCs lack of OTP (One Time Programmable memory) and their JTAG seems to be non-lockable[9]. Finally, MCUs such as the NXP Kinetis K8x series or the newer ARMv8-M-based cores[10] did not exist back in 2014 during the hardware design phase.

We finally focused on the `STM32F439` as it fits most of our needs. Moreover, the Cortex-M4 SoCs have been widely studied in the recent years, and the `STM32F439` features a cryptographic coprocessor (the CRYP engine) as well as a TRNG (True Random Number Generator).

The power consumption of this SoC is rated as high as 98 mA when the core is running at maximum possibilities and all the peripherals are

[9] At least from the publicly available documentation.

[10] These SoCs seem very promising: they offer interesting security features such as a lightweight TrustZone mechanism implementation.

	Memory Protection Unit	USB Speed	Integrated USB PHY	Cryptographic coprocessor	JTAG/SWD deactivation	JTAG/SWD authentication	Data-sheets broadly available	SDcard (SDIO) interface	SPI Interface	OTP memory zone	Low power	Internal flash storage	BootROM inhibition	Flash R/W/Er protection
NXP LPC43Sxx	✓	FH[1]	FH[1]	✓	✓	✓	✓	✓	✓	✓	✓	✓[2]		✓
Atmel SAMx7	✓	FH[1]	FH[1]				✓	✓	✓		✓	✓[2]	✓	?[4]
NXP Kinetis K8x	✓	FH[1]	FH[1]	✓	?[4]	?[4]		✓	✓	✓	✓	✓[3]	?[4]	✓
STM32 Cortex-M4	✓	FH[1]	F[1]	✓	✓	✓	✓	✓	✓	✓	✓	✓[2]	✓	✓

[1] F: Full-Speed, H: High-Speed, FH: both speeds.
[2] >= 1 MBytes flash size.
[3] 256 KBytes flash size.
[4] Unknown information or value.

Table 2. Overview of available ARM Cortex-M SoCs with USB capabilities

enabled. This leaves room for the other components on the board to be powered even during the enumeration phase of the USB protocol where the maximum allowed power consumption is 150 mA. Though this SoC has an integrated USB Full Speed PHY (12 Mb/s capable), it needs an external PHY to achieve High Speed (480 Mb/s). The communication between the SoC and the PHY is done using ULPI, which is a standardized interface for USB 2.0.

2.3 Data storage

We have chosen to store the encrypted user data on an external SD card. This format has many advantages. It offers large storage capacities for an affordable cost with a possible expansion of the USB thumb drive capacity by switching the SD modules. Compared to raw flash modules, there is no need to handle complex FTL (Flash Translation Layer) software layers: the firmware embedded in the SD card takes care of this.

The fact that we use an *active* component that embeds a complex and uncontrolled firmware [17] to handle the data could be seen as a threat. This is, however, not the case since the SDIO protocol is simpler than the USB protocol. Furthermore, the SD card firmware only transfers encrypted data blocks on WooKey, which reduces the interest of a Man In The Middle (MITM) attack on the SDIO bus.

Nonetheless, since the SDIO driver is exposed to malleable user inputs, all the software modules handling it will run isolated from other sensitive modules (e.g. those manipulating secrets) using the MPU (see section 3.6).

2.4 Authentication tokens and secure elements

The need for an authentication token: Strong user authentication ensures that no sensitive cryptographic operation is performed without the legitimate user's presence (through a correct PIN code). This implies that all the cryptographic and authentication material must be handled securely: a *secure element* seems to be a suitable choice for this task.

Splitting the platform and the user authentication material yields in a strong two-factor authentication scheme. This is why we have chosen to use an external and extractable user token (instead of soldering it) in the form of a smartcard.

We emphasize the fact that the external token does not solely serve a *logical presence* purpose: this critical element is actively used as a safeguard, through cryptographic operations, when a sensitive action is initiated (user data decryption, firmware updates). The way we handle this strong adherence between the SoC of the platform and the external token will be thoroughly explained in section 3.2.

Using an external token with strong user authentication allows us to exclude many attack scenarios where the USB device is lost by the legitimate user. If the hardware and software design are sound and if the decryption master secrets are stored in the token, an adversary will only be able to observe and abuse the pre-authentication modules of WooKey, and hence all the post-authentication critical modules are safe. Moreover, since the external token is based on a *secure element*, we can also include lost tokens in our threat model.

Secure elements: Secure elements are the foundation of modern systems security: they are usually considered as a hardware root of trust in systems requiring strong authentication (payment and credit cards in the banking ecosystem, SIM cards in telecom, TPM in PCs' secure boot, etc.).

A secure element is in fact a microcontroller hardened against a wide range of logical and physical attacks (side-channels attacks and fault injections, cloning). It is validated by a formal process through Common Criteria: certified laboratories perform pentests and assess attacks difficulty and impacts so that the chip can be considered robust and certified at a so-called EAL level.

Finding a certified secure element that can be programmed without signing NDAs and that can be bought in small quantities is not an easy task. This is particularly true if one wants to have a bare-metal access to the chip, e.g. to implement her own OS. The situation between secure hardware suppliers and the Open Source community is thoroughly discussed in [25] and [13].

Using the Javacard framework: The attempts to bring secure elements technology to the public domain have emerged through VM-backed languages. The user code is confined in a Virtual Machine and the resources of the platform are abstracted with standard and documented APIs. This isolation serves various purposes: low-level layers are protected against tampering and the isolated applications cannot interfere with one another, the Virtual Machine API is standardized and proprietary information is not needed to implement useful algorithms.

To the best of our knowledge, the two main lines of products using publicly available secure elements are[11]:

- The Javacard platform [43]: this is a lightweight version of the Java language and runtime with embedded development constraints in mind. It has become the *de facto* standard adopted by the industry since the 90's. A major asset of Javacard certified products is that the EAL certification usually concerns the whole platform (from the low-level chip to the VM).
- The BasicCard platform [56]: a Basic interpreter is embedded in these smartcards. Though the underlying chips seem to be certified, this is not the case of the whole platform.

Since Javacard is the only widely available framework to offer a Common Criteria certification, we have chosen to focus on this platform. More specifically, we have developed and tested our applets on EAL 4 certified NXP JCOP `J3D081 2.4.2` smartcards [6]: we provide more details on this in section 3.2.

2.5 Touch screen

In order to limit the smartcard PIN code exposition and defeat Man In The Middle attacks on the USB bus or in a compromised host [24], we

[11] To a lesser extent, `.NET` VM-based smartcards also exist. We considered them out of scope because of proprietary framework and development tools.

have decided to include a user input interface directly on the platform. This allows confining the PIN code to the WooKey device.

Among possible input devices technologies, we have chosen the TFT-LCD `ILI9341` with a `AD7843` touch screen component. This allowed us to design a randomized PIN pad that makes movements observation attacks more complex [51]. We drive the touch screen from the STM32 SoC using the SPI bus where both the `ILI9341` and `AD7843` are slaves.

The residual risk is that the PIN code flows in clear text on the internal SPI bus: an adversary is able to recover it using hardware taps. We point out, however, that the PIN is only one of the two factors used for the authentication (the extractable token being the other one).

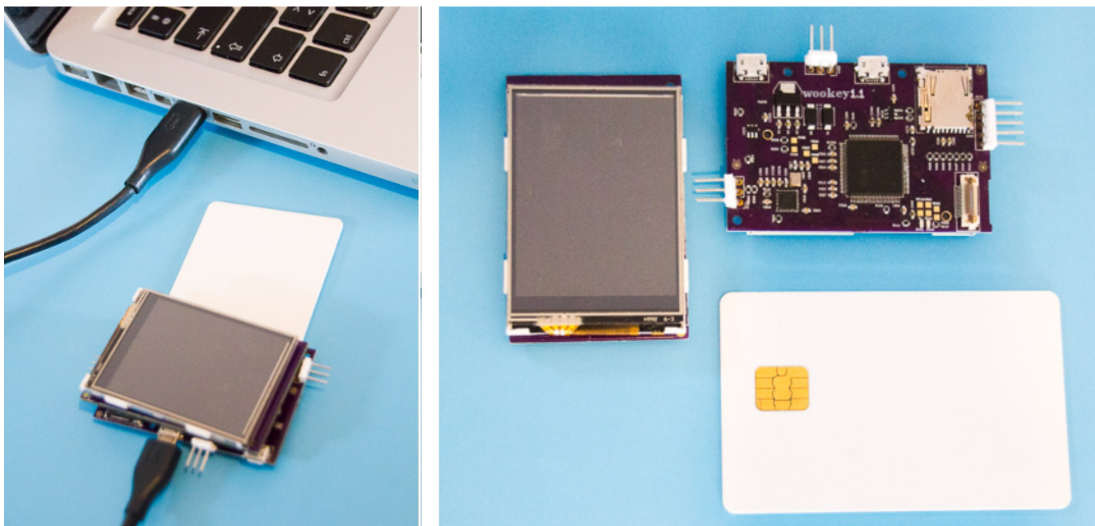

Fig. 2. WooKey hardware platform

2.6 Prototyping and cost estimations

All the elements described in the previous subsections are placed on two 4-layer PCB. The final dimensions are $44 \times 66 \times 8$ mm. An overview of the final design of WooKey is provided on Figure 2. At this stage of the project, it is difficult to have an accurate cost estimation for the device in a production context. We can nonetheless bring some feedbacks about our experience with hardware manufacturing during our prototyping phase.

For a batch of 10 boards with PCBs produced in China and assembly in France, the cost is around 300 € per board bundled with a 16 GB SD card. We also did a simulation for 10 boards produced and assembled in

the USA and we obtained an estimation of 174 €. The same company charges 44 € per board for 1,000 boards. All these estimations do not include the price of the circuit case.

The variations of the price are mainly due to three factors: the minimum order quantity for each part of the circuit, the setup time, and the number of boards and parts required for the batch.

Finally, the JCOP `J3D081` smartcards can be found at around 30 € per unit. Similarly to the PCB components, there is a scaling factor: the per unit price drops to 2 € for 1,000 pieces.

As a comparison basis, the price of commercial USB Encrypted Flash Drives usually fluctuates between 100 € and over 500 €.

3 Software architecture: towards a secure framework

3.1 General software design

Classical USB thumb drives need at least two main software components: the USB stack to exchange data with the host and the mass storage manager to store data. One of WooKey's main feature is to encrypt the data at rest, which requires a dedicated cryptographic module to encrypt/decrypt this data. WooKey must securely manage both the cryptographic and authentication materials along the user data path.

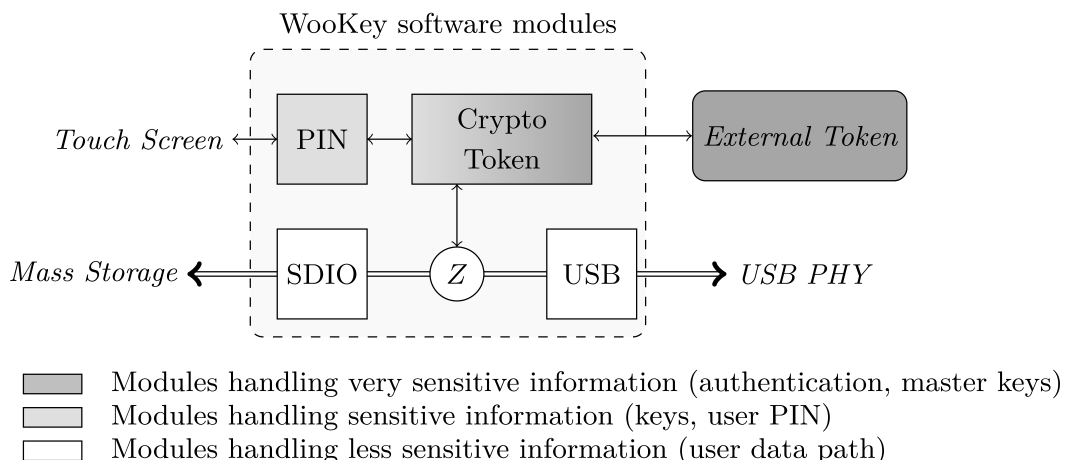

Fig. 3. General software architecture of WooKey

The data path goes through three logical modules to read and write data from/into the device.

- The USB module handles the USB communication with the host.

– The SD module manages the mass storage device and read/write access to encrypted data.
– The cryptographic module sits between these two modules. It encrypts and decrypts data when authentication has been performed using the external token.

The CRYP hardware module increases the cryptographic operations' performance: processing an AES block takes very few cycles, and the engine allows DMA (Direct Memory Access) transactions with the other modules (USB and SDIO).

3.2 Handling cryptographic material and authentication:

Cryptography is involved in the user data at rest confidentiality and the authentication token interactions on the WooKey platform. We discuss the issues related to these topics hereafter.

About user data confidentiality and integrity: Full Disk Encryption (FDE) has become a matter of concern and a topic of interest in applied cryptography these last years. This is mostly due to the development of nomadic devices and the emergence of privacy issues (all modern smartphones feature data encryption). From a pure cryptographic standpoint the situation is not ideal though: the high-level features an end user expects are both data *confidentiality* and *integrity*. Unfortunately, no ideal efficient solution exists nowadays to ensure both these assets with a perfect and proven security level. This is even more true since integrity expects extra data to be stored on the disk in addition to the encrypted blocks, and since fine-grained considerations such as local/global and temporal integrity must be taken into account. This inherent complexity explains why most devices acting as a transparent layer over the storage peripheral and under the OS (e.g. embedded encrypting USB devices) chose to only focus on user data confidentiality: this is also the case for WooKey.

User data confidentiality cipher: AES-XTS, standardized by the NIST [3], has become a popular AES tweakable mode for block device encryption. We have nonetheless decided to use AES-CBC-ESSIV [27] (used in dm-crypt and its implementation in Android FDE) because of performance reasons. Indeed, the CBC mode is accelerated on the `STM32F439`, and AES-XTS can be quite greedy for CPU cycle on general purpose MCUs because of operations over $GF(2^{128})$. A major advantage

of AES-XTS over AES-CBC-ESSIV is its better resilience against block malleability [7]. We however stress out that integrity is still at risk with AES-XTS: our approach is to clearly state that WooKey *does not ensure it*. Hence, getting back a lost device or SD card and using them must be considered dangerous. A straightforward solution for the end user is to handle integrity in a higher layer, e.g. at the file system level. Future work is planned to explore authenticated encryption schemes using the AES-GCM acceleration in the CRYP engine (the *tags* data extra storage and Initialization Vectors are still challenging issues in a FDE context).

The data path encryption module: Since we want the encryption and decryption along the data path to be very efficient during USB and SDIO transfers, we must avoid reconfiguring the AES CRYP engine (and key schedule) at each transaction while preventing the USB and SDIO tasks to steal and leak the sensitive data encryption key. Fortunately, we can isolate the registers configuring and holding the AES key using the MPU. This yields in the following split of the WooKey cryptographic task in two modules:

- An untrusted cryptographic module: it shares its memory space with the USB and SDIO tasks, and its job is to trigger AES-CBC encryption and decryption in the CRYP and handle DMA transfers. This module uses the CRYP with the key already setup, and never accesses the secret value.
- A trusted cryptographic module: this module is confined and isolated from the other tasks. It is in charge of setting up the CRYP key registers with the secret AES key derived from the external authentication token. It is also in charge of managing all communications with this token.

Authentication with the external token: The trusted cryptographic module running on the STM32 SoC communicates with the external smartcard (aka authentication token) through an ISO-7816-3 bus using Application Protocol Data Units (APDUs). The main SoC and the token embed (personalized) ECDSA key pairs. The first thing that is performed by the two peers when the token is inserted is mutual authentication. This is performed with an ephemeral ECDH (Elliptic Curve Diffie-Hellman key exchange), then AES-CTR and HMAC-SHA-256 session keys are derived, as well as a random IV (Initialization Vector) value. This allows to establish a secure channel with confidentiality, integrity and anti-replay properties. Forcing mutual authentication as a mandatory first step allows limiting the attack surface (against malicious tokens or a malicious ISO-7816 master).

On the platform side, we use the open source `libecc`[12] that was designed with embedded constraints in mind. On the token side, ECDSA and ECDH are part of the Javacard 3.0.1 framework. AES-CTR and HMAC-SHA-256 were not fully supported, so we have implemented our own Javacard classes/applet over the built-in hardware accelerated AES-ECB and SHA-256. One could wonder why we have decided to implement our own secure channel while the Global Platform framework offers this feature. None of the proposed schemes were adequate on our JCOP Javacard: they are either broken [50] and/or make use of symmetric key cryptography (compromising a platform would yield in breaking the token secret key as well, which is prevented by asymmetric cryptography in our case).

Whenever the PIN is entered on the touch screen, the cryptographic module gets it and sends it to the token. The token checks the PIN, and if the PIN is OK (locked after configurable n failures) it derives a key using the PIN value and a master secret stored in the token. This key is sent back to the main SoC and serves as the AES-CBC-ESSIV data master key. The PIN also participates in secure channel session keys diversification to bind the session to this authentication instance.

The CRYP engine is not certified (i.e. could be attacked through side-channels), and since the secure channel is established and used before the PIN is provided, we have chosen to use a dedicated software *masked* AES for our AES-CTR [19, 21, 48]. The masked AES is secure but slow, which is actually not an issue here because of the relatively limited baud rate of the ISO-7816 channel and the small size of the data packets.

Finally, the external token (actually a token dedicated to firmware updates) also participates in our DFU implementation as it will be described in section 3.4.

3.3 About the WooKey personalization phase:

It is assumed in the threat model that the initial firmware upload and configuration of the platform are performed in a *trusted environment*. The security insurance brought by the defense-in-depth mitigations during the life cycle of the product inherently depends on this critical phase.

The main steps that are handled during personalization are:

– Flash the initial firmware on a virgin and open device (i.e. with JTAG/SWD unlocked).

[12] `https://github.com/ANSSI-FR/libecc`

- Flash the Javacard applets on virgin and open smartcards (one for user authentication, one for firmware signing, one for device firmware update).
- Generate and deploy (on the platform and the external tokens) all the master cryptographic keys, namely the ECDSA key pairs for mutual authentication with user and update tokens, keys handling firmware updates, as well as user data encryption master key.
- Deactivate JTAG/SWD, activate the flash RDP protection (level 2) on the `STM32F439` MCU in order to lock the platform in production mode.
- Lock the external token smartcards in production mode (modify the default Global Platform keys).

3.4 Designing an efficient and secure DFU mode

An often underrated security feature is the ability to maintain a product in secure and working conditions. However, updating a USB device in a safe way is not an easy task because such devices are often not self-powered and may be disconnected at any time.

Because microcontrollers have very little memory space in volatile memory, firmware upload and checking have to be performed in-place in the flash area where it will be executed. Hence, this requires a flip-flop mechanism ensuring software redundancy in order to handle any file corruption (hazardous disconnection, data flow corruption, invalid cryptographic signature, etc.). Such implementations are in general proprietary, but are sometimes based on standards like the USB DFU protocol [29] that allows device update through USB.

Allowing patching of device firmware is dangerous, as malware might use this feature to replace the genuine firmware with a malicious one. Therefore, DFU should not be accessible without explicit user activation and authentication. The DFU implementation itself could contain bugs, and as a consequence it should also be upgradable.

To support such features, we have decided that the DFU flip/flop implementation should be separated from the standard firmware; it is executable only after voluntary physical button toggling at boot time.

When the user willingly activates the DFU mode, the bootloader expects a specific external token to be present. Mutual authentication is performed with the token (the details of this protocol are provided in section 3.2), and a specific PIN code (dedicated to updates) is asked by the main SoC and checked on the token side. If the PIN is valid, firmware upload can begin. The firmware is encrypted using a session key

at production time, and this session key is decrypted using the token so that its presence is indeed enforced during the update.

Once uploaded to the device, the new firmware integrity and authenticity are checked using a cryptographic ECDSA signature validation, and the default bootloader pointer is switched. The update version is also validated in comparison to the current firmware version to avoid any downgrade with an older and buggy (but signed) firmware [20].

All these constraints and security features impact the overall device software mapping and reduce the available space for each software component. Figure 4 shows the content of the device with all the required components.

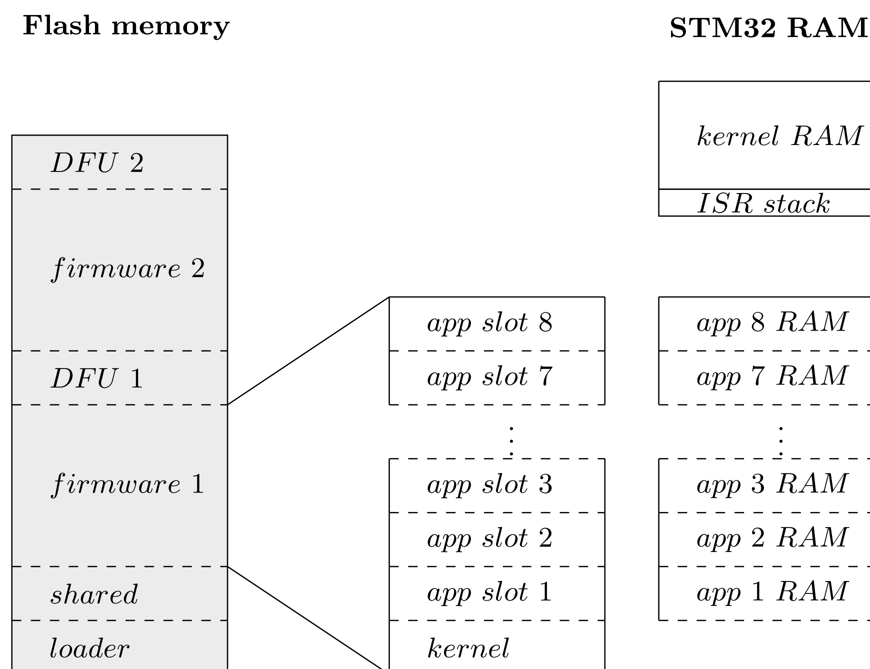

Fig. 4. Overview of the embedded software mapping

Splitting the device software into two independent firmwares and into two DFU-dedicated firmwares is an efficient way to bring some resilience and protection against the risks previously described. However, even if it is a way to protect the *offline* devices, it does not prevent any *online* software attack. For instance, without any further protection mechanism, a flaw in the USB stack implementation may allow an adversary to fully compromise the device. Such escalations can be thwarted by enforcing memory segregation between I/O applications (USB stack, etc.) and other parts

of the device. In the following sections, we describe the countermeasures implemented to avoid this kind of software exploitation.

3.5 Toward a highly secured embedded software

Most firmwares embedded in microcontrollers do not enforce any security at all. In particular, the lack of isolation between software components hinder enforcement of crucial security principles: e.g. *least privilege* or *privilege separation*. Therefore, it is not uncommon that each component of the firmware can access the whole memory space and that any bug in the smallest piece of code can corrupt the entire system.

Overview of the software security requirements: We retained several security requirements to bring the WooKey platform to a security level nearing the state of the art, while respecting the inherent flash and RAM small footprints:

1. Using the MPU to enforce the least privilege principle and to protect the most sensitive assets.
2. Formal verification of critical code, or at least the usage of a safe language to harden the implementation.
3. Advanced in-depth mitigation mechanisms (stack-smashing protection, heap protection, `W⊕X`, etc.).
4. Being open source to permit peer review.

The MPU, a crucial confinement primitive: Most of modern 32-bit microcontrollers have a Memory Protection Unit (MPU) and a processor with at least two CPU privilege levels (the so-called *user mode* and *supervisor mode*). The MPU is a programmable unit that allows privileged software, often a *kernel*, to define memory access permissions in order to isolate memory regions. It can be used to enforce confinement and privilege separation between unprivileged components, like *tasks* executed in *user mode*. These tasks must not break out of their address space.

Microkernels paradigm: Microkernels architecture (e.g. QNX, Fiasco.OC, SeL4, OKL4, etc.) goes back to the 1970's [47,55] and enforce the least privilege principle by isolating the drivers in their own address space. Vulnerabilities in a driver are kept confined and cannot compromise other parts of the system. Another advantage over monolithic kernels is that the Keep It Simple and Stupid (KISS) paradigm is also applied. By keeping

microkernels minimal, with fewer functionalities, they are in theory simpler to design, implement, debug, and maintain, and they are therefore less error-prone. Another consequence of shrinking the functionalities and the number of syscalls is that the exposed code to untrusted applications (and hence the attack surface) is drastically reduced. Drivers still have to be implemented, but as userspace tasks, with limited access rights.

Safe languages: Most kernels and microkernels are written in C with a pinch of assembly. The major drawback of the C language is its proneness to coding errors. Out-of-bound array accesses, integer overflows and dangling pointers are difficult to avoid due to the weakly enforced typing. Such bugs can become nonetheless devastating when exploited in a privileged context.

A way to prevent such vulnerabilities is to use a safe language. Pierce [45] defines a safe language as one that protects and guarantees the integrity of its own abstractions, which can be achieved by static checking, but also by run-time checks. Many high-level languages share this feature, but very few are actually suitable for operating systems programming, let alone embedded bare-metal programming.

Using a safe language for implementing low-level kernel code is an approach that goes back to the early 1970's [32, 46]. Nowadays, such initiatives are not widespread, and usually use languages such as Rust (for example Redox[13] or Tock[14]) or Ada [16, 38].

Ada is designed for building high-confidence and safety-critical applications [15, 49]. It is a strongly typed language that supports bare-metal programming, and can circumvent most well-known vulnerabilities like buffer overflows or invalid pointers management by enforcing type checking at compile time and at run time.

SPARK is an Ada subset that can be used with the *GNATprove* tool to prove that the written code is free of any type violations. SPARK and *GNATprove* bring confidence in the soundness of the code, therefore allowing to remove run-time Ada type checks. This yields in better performance, smaller memory footprint and no run-time exception breaking the execution flow.

Rust is a rather new promising language with increasing popularity. It aims at enforcing strong static type checking and memory safety.

[13] https://github.com/redox-os/redox
[14] https://www.tockos.org/

Language	*Safe language*	*Formal proof*	*Memory footprint*	*Well-known*
C	✗	✗	✓ low	✓
Ada	✓	✓(SPARK)	✓ low[1]	✗
Rust	✓	✗	✗ high	~[2]

[1] Ada can embed runtime checks. SPARK code reduces such checks size.
[2] Recent language, but with a growing and active community.

Table 3. Comparison between the languages used in the WooKey project

Formal methods: Formal methods allow proving functional correctness and soundness of a design and/or of an implementation with respect to some predefined properties. This approach fits well with microkernels.

It was successfully applied to SeL4 [33], proving that its implementation is free of several classes of vulnerabilities (deadlocks, buffer overflows and arithmetic exceptions).

This approach has nonetheless some drawbacks. It is complex, and it is not uncommon that the number of proof code lines goes well beyond the number of code lines to prove (SeL4's 8,500 lines of C code induced 200,000 lines of proof and 11 man-years of work [33]). Moreover, inner constraints limit the scope of the properties that can be proven.

Defense-in-depth security mitigations: Software exploit mitigation uses many techniques: stack canaries, ASLR (Address Space Randomization), page guards, heap protection, memory isolation, non-executable data regions, `W⊕X`, kernel side checks, data sanitization, etc. Thus, even if an attacker may find a flaw in one defense, combining many of them multiplies by orders of magnitude the efforts needed to bypass all of them.

3.6 EwoK, a driver-oriented microkernel

Enforcing memory protection, tasks isolation and providing access control to the assets and to the hardware requires a kernel.

A brief survey of embedded kernels (summarized in table 4) shows that none of them met our security requirements. We discarded non-free and closed source kernels, despite some of them have interesting security features (BOLOS operating system, INTEGRITY, ProvenCore, etc.). We also discarded most of open source embedded kernels: their real-time driven design is barely compatible with the overhead produced by security mechanisms (Contiki, FreeRTOS, etc.).

TockOS [36] is a new kernel written in Rust that benefits from the memory protection mechanisms offered by this language in order to secure

the drivers execution. The major drawback is its memory footprint that does not fit within our hardware limitations.

L4 microkernels have some interesting security properties but they target bigger devices. Among the L4 family, F9-kernel [31] is designed to be executed on microcontrollers (such as the `STM32F4` family). However, it is written in C, with no specific security properties.

Therefore, we decided to develop our own microkernel.

Features and security properties: EwoK is a microkernel that provides the necessary functionalities to execute device drivers as user tasks. It implements all the security requirements exposed above.

It enforces at *build time* a strict memory partitioning between tasks, despite the inner limitations of the MPU that only allows 8 memory regions at the same time [30].

Registered devices are mapped only when needed, enforcing the least privilege principle. The drivers (running as tasks) can claim resources, like GPIO, DMA and MMIO devices, during their initialization phase.

Tasks may communicate using IPC or shared memory. For example, the untrusted cryptographic task (handling the user data path) and the trusted cryptographic task (handling the smartcard and the master keys) are synchronized using IPC.

Permissions are statically defined at build time to avoid improper access or information leakage.

A user task supports two contexts: a main context and an Interrupt Servicing Routine (ISR) context. The kernel manages the processor interrupts in the so-called ARMv7m privileged handler mode. Then execution is dispatched to the registered ISR handlers to be executed in user mode.

Critical parts of the kernel and the components belonging to the Trusted Computing Base (TCB) are developed in SPARK/Ada (see Figure 5). Criteria for identifying these components are their role in the security of the platform as well as their exposure to untrusted applications and to user inputs.

EwoK is also hardened with defense-in-depth mechanisms: usage of the $\mathtt{W} \oplus \mathtt{X}$ paradigm and stack guards (inspired from stack canaries). Due to the low amount of RAM and the lack of virtual memory, we abandoned the idea of implementing ASLR.

The software implementation suffered from two major hurdles: the small amount of available RAM memory and the IPCs performance. Executing drivers as user tasks implies that function calls are replaced with IPCs, which generates an overhead that can be a real burden for

	Software requirement								
name	*designed for security*	*partitionned drivers*	*Supports Cortex-M*	*Based on a safe language*	*Fast & safe driver exec*	*Portable*	*Small enough*	*Static permissions*	*Open*
Zircon	✓	✓			✓	✓			✓
SeL4	✓	✓		✓	✓	✓	✓	✓(capabilities)	✓
f9-kernel			✓				✓		✓
TokOS	✓	✓	✓	✓	~[3]	✓		?	✓
Ravenscar Ada profile		~[1]	✓	✓	✓	✓	✓		✓
FreeRTOS						✓	✓		✓
RIOT-OS			✓			✓	✓		✓
QNX		✓(pure μkernel)				✓	✓		
Minix3		✓			?	✓	✓		
PikeOS		~[2]			~[2]	✓	✓	✓	
ProvenCore	✓	✓	✓	✓	✓	✓	✓	✓	
VxWorks			✓	?	?	✓	✓		
INTEGRITY	✓	✓		?	?	✓	✓	?	
BOLOS	✓	?	✓	?	?	?	?	?	
Contiki			✓			✓	✓		✓
EwoK	✓	✓	✓	✓	✓	✓	✓	✓	✓

[1] Depends on the developer's choice.
[2] Depends on integrator's choice.
[3] Speed may vary depending on the driver/application interactions.

Table 4. Existing embedded kernels and EwoK features *versus* WooKey constraints

low-power devices [14]. The kernel uses a mixed collaborative/preemptive scheduler, with an event-based threshold support for high reactivity of ISRs and tasks: tasks can yield and ask (under certain circumstances and permissions) for voluntary reschedule of their main threads, but a preemptive time slotting and a priority management is maintained by the kernel scheduler. Using (controlled) shared memory and DMA transactions also participates in improving performance.

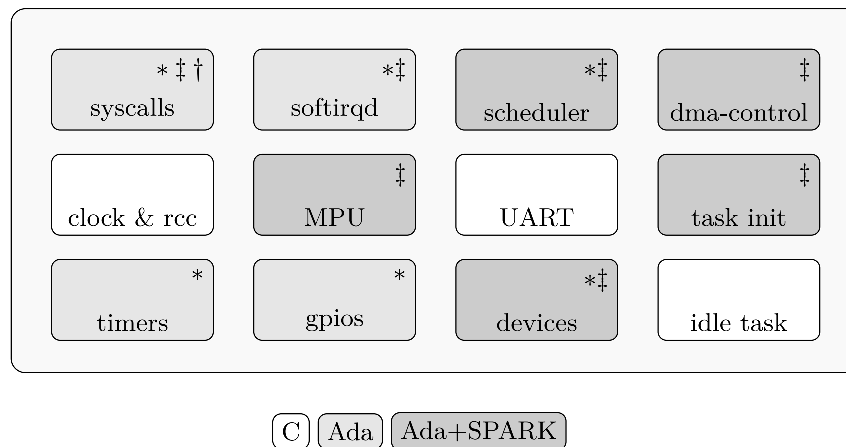

Fig. 5. Block diagram of the software components of the EwoK microkernel

EwoK, the WooKey gate keeper: The WooKey project aims at protecting user sensitive assets against their stealing by adversaries. In order to do so, the main cryptographic secrets are stored in an external smartcard which needs to be connected to the device at boot time to allow the data read and write actions.

In order to protect these critical assets (the master keys and the PIN code), the kernel segregates the data plane (USB to/from mass storage) and the authentication plane.

Figure 6 describes such a logical partitioning. Mutual authentication with the smartcard is controlled by a process dedicated to the secure channel management. Another dedicated process manages the embedded screen and touchpad. The data encryption/decryption is done using cryptographic content accessible only when the external token authentication has been successfully performed. The USB and SDIO stacks have access

to the current session data, as they manage the transfers to and from the external host, but have no direct access to the master cryptographic assets inside the smartcard.

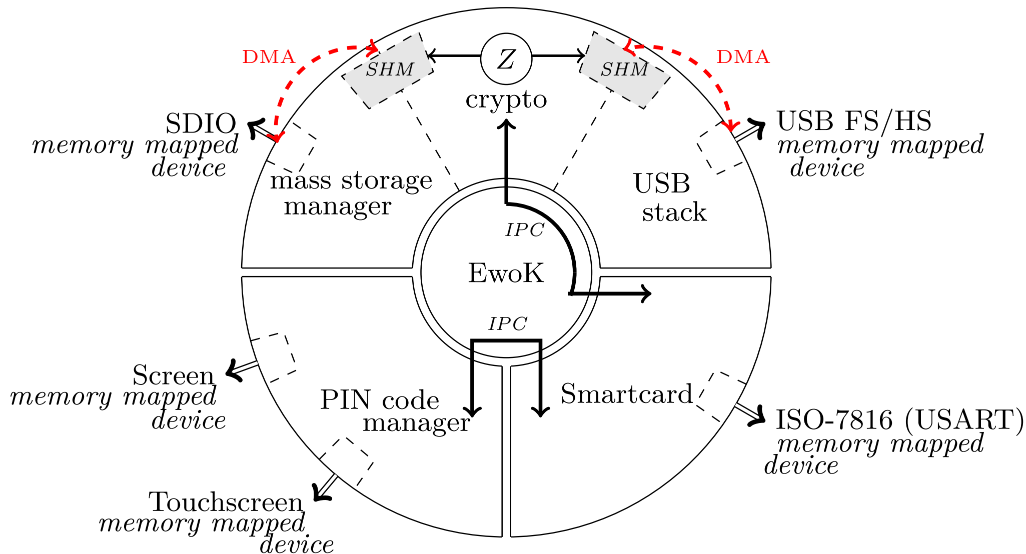

Fig. 6. Usage of EwoK in WooKey

Completely separating the USB stack, the untrusted cryptographic subsystem and the mass storage manager in terms of processes and shared memory is still an ongoing investigation. The final design will heavily depend on performances trade-offs.

The bottleneck that would emerge is in the data path between the USB and the SDIO sides. In any case, the data plane and the cryptographic module handling the external token and the sensitive assets are confined in distinct memory spaces: their interactions do not need high speed and can therefore be handled using IPCs.

3.7 MosEslie: towards a versatile SDK

In order to easily integrate new software, drivers and tasks in the WooKey platform and over the EwoK microkernel, we have designed a dedicated SDK (Software Development Kit): MosEslie. It uses only widespread open source tools (GCC, GNAT, Kconfig, Makefiles, etc.) without any external dependencies.

The SDK helps configuring the flash and RAM partitioning for the applications, as well as the shared memory slots and the IPC control flow matrix. The memory protection layouts applied by the MPU and

the whole permissions are automatically generated by the configuration subsystem. If no proper layout can be produced (i.e. inconsistent MPU configuration at runtime), the SDK displays a comprehensive error.

Finally, MosEslie makes integrating software written in safe languages (Ada and Rust) very easy thanks to their binding interfaces with C, hence providing a mixed languages firmware.

4 WooKey: a security and threat analysis

The expected requirements of the platform (regarding hardware, software and security) are recalled in Table 5.

Hardware	Software	Security
MPU	Resilient storage	Strong authentication
Crypto-processor	Performance	Signed firmware update
Conforming to USB2.0	Open	Confinement of exposed interfaces
Open, COTS and cheap	Modular and evolutive	Static and checkable permissions
JTAG/flash protections	Versatile SDK	Safe languages for critical parts

Table 5. Platform security requirements overview

We have chosen to build the WooKey hardware platform around the `STM32F439` SoC featuring a MPU for memory confinement, AES co-processor and TRNG random generator. Only off-the-shelf and widely available components are used on the PCB.

The strong authentication requirement is ensured using an extractable secure element (see section 2.4): an affordable Javacard smartcard with a dedicated applet. The PIN code is entered on a TFT touch screen (see section 2.5) ensuring no leak to the host.

The firmware's integrity and authenticity are serviced using a robust and strengthened DFU mode involving digital signature and authentication token interactions (explained in section 3.4).

The EwoK microkernel provides efficient isolation of distinct firmware parts, static configuration of the applications and their memory layout, Ada usage on sensitive modules and SPARK formal verification on very critical ones (such as the MPU driver) as detailed in section 3.5.

Hence, most of the hardware and software requirements listed in Table 5 are met. The performance benchmarking is still an *ongoing work*: many modules have been individually tested and no major issue should arise once the complete integration is performed. The integration and the fine tuning are in progress, nonetheless not finished yet.

We leverage the implemented two-factor authentication method (user PIN and smartcard). The fact that WooKey does almost nothing during pre-authentication allows preventing some attack scenarios:

- Adversaries that use side-channel attacks or fault injection without the authentication token won't be able to do much. This still stands if they have the token without the PIN: since all the cryptographic sensitive material remains locked in a secure element, this should resist to such a class of attacks (under the assumption of the CC EAL certification of the secure element of course). The only part that is exposed is the secure channel (using ECDSA and ECDH, AES-CTR and HMAC-SHA-256) established in the pre-authentication phase: failed attempts could yield in locking the platform by using a counter stored in flash.
- Try to mimic hardware and software: unintelligent attempts will fail thanks to the strong cryptography (secure channel) performed in the pre-authentication phase (since the adversaries do not have the ECDSA private keys). The only asset that an attacker will be able to get is the user PIN by deceiving the legitimate user in entering it. However, the adversary still has to achieve physical possession of the external token.

On the features that are **lacking** *by design* on WooKey, we have:

- Trojan firmware, aka evil maid style attacks. Our DFU ensures that firmware updates are sound and authentic, but there is no check at boot time that the firmware has not been altered by other means: either physically by manipulating the flash (JTAG/SWD and so on), or logically through exploiting a buffer overflow for instance. We try to limit the physical modifications by using ST RDP (Readout Protection) that locks the platform in production, but section 2.2 and [10, 42] have shown how such countermeasures could become fragile against aggressive attack vectors (decapping and faulting). On the software part, EwoK is precisely dedicated to intrusion mitigation and confinement.
  Against trojan firmware, *secure boot* technologies could come to the rescue, but they might not be the silver bullet we expect to thwart such threats [11]. Newer MCUs also integrate transparent flash encryption using a dedicated cryptographic co-processor (a technology inherited from the secure elements and the FPGA worlds). However, such recent improvements of the publicly available MCU lines have still to prove their robustness against physical and logical threats.
- Trojan hardware. This kind of attack is devastating and we cannot do much against it. If an attacker has been able to somehow recover

(e.g. via physical attacks) WooKey sensitive private keys from the SoC's flash, and build a full platform resembling the genuine one, this is the perfect crime. A much weaker variant of such hardware mimic attacks is when the adversary does not know the platform private keys, this falls in the previously described *unintelligent* scenarios.

On a side note, the main advantage for an adversary when implanting a trojan firmware or hardware is to steal the legitimate user PIN, and ultimately to steal the master decryption key from the token (using the PIN) in order to be able, in the future, to decrypt user data without the token.

We emphasize the fact that achieving a protection level preventing these two latter powerful – and rather costly – kinds of adversaries is very complex (if not impossible) using off-the-shelf affordable components.

Regarding cryptography, as it has been thoroughly detailed in section 3.2, we only protect user data confidentiality: the integrity of the SD card is out of scope and the user must be informed of this and use complementary solutions (such as integrity at the file system level).

5 Conclusion

WooKey aims at being an open source, open hardware, secure and affordable USB encrypting mass storage device using off-the-shelf components.

Protection against the BadUSB class of attacks is achieved using strong cryptography with two-factor authentication of the legitimate user (PIN and smartcard using a secure element), as well as a robust DFU dedicated to firmware integrity and authenticity insurance. Software classes of attacks (e.g. buffer overflows) are mitigated using EwoK, a novel microkernel designed with security in mind, enforcing memory isolation using the MPU and providing more confidence by using the Ada safe language along with SPARK for formal verification of critical parts. Beyond the mere USB key itself, we also provide MosEslie: an easy-to-use SDK that simplifies the integration of user applications on the platform as well as the possible mixed usage of safe languages (Ada and Rust).

Thanks to these characteristics, the overall security of WooKey is strong against software attack vectors, and some pre-authentication hardware attacks (side-channel observation and fault injections). The residual adversaries that endanger the platform integrity involve aggressive and costly attacks on the STM32 MCU (decapping and light-based fault injection) to recover the private keys in flash. On a similar note, WooKey only

protects user data confidentiality: user data integrity is not covered (which is the case for most of USB thumb drives with transparent encryption).

Though such residual threats directly inherit from the constraint of using available and affordable off-the-shelf components for WooKey, we feel that there is still room for improvement. Newer ARMv8-M architectures offer interesting features such as TrustZone and a more advanced MPU (for better isolation of EwoK applications), some recent MCUs also integrate tamper detection and transparent flash encryption using dedicated hardware, etc. Many of the key concepts already developed during the project are easily portable on these platforms, and improving the defense-in-depth layers is future work.

— "It's not wise to upset a WooKey."

— "I suggest a new strategy, Artoo: let the WooKey win."

References

1. Encrypted Drive. `https://www.kingston.com/fr/usb/encrypted_security`.
2. Hardware Wallet Vulnerabilities. `https://blog.gridplus.io/hardware-wallet-vulnerabilities-f20688361b88`.
3. NIST: Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices.
4. Nitrokey Secure Elements. `https://www.nitrokey.com/documentation/frequently-asked-questions#is-nitrokey-common-criteria-or-fips-certified`.
5. USB encryption. `https://www.hacker10.com/usb-encryption/`.
6. NXP JCOP 2.4.2 R2 CC, 2013.
7. Practical malleability attack against CBC-Encrypted LUKS partitions, 2013.
8. BadUSB vs. DataLocker Sentry 3.0. `http://www.ireo.com/fileadmin/img/Fabricantes_y_productos/datalocker/BadUSBWP.pdf`, 2015.
9. Get BadUSB protection from IronKey USB Flash drives. `https://media.kingston.com/images/usb/pdf/BADUSB_us.pdf`, 2016.
10. Breaking Code Read Protection on the NXP LPC-family Microcontrollers, 2017.
11. USB armory security advisory. `https://github.com/inversepath/usbarmory/blob/master/software/secure_boot/Security_Advisory-Ref_QBVR2017-0001.txt`, 2017.
12. Sebastian Angel, Riad S Wahby, Max Howald, Joshua B Leners, Michael Spilo, Zhen Sun, Andrew J Blumberg, and Michael Walfish. Defending against Malicious Peripherals with Cinch. In *USENIX Security Symposium*, pages 397–414, 2016.
13. Nicolas Bacca. *Secure Hardware and Open Source*, 2016. `https://www.ledger.fr/2016/06/09/secure-hardware-and-open-source/`.
14. Brian N Bershad. The increasing irrelevance of ipc performance for micro-kernel-based operating systems. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 205–212, 1992.
15. Carl Brandon and Peter Chapin. The Use of SPARK in a Complex Spacecraft. *ACM SIGAda Ada Letters*, 36(2):18–21, 2017.

16. Reto Buerki and Adrian-Ken Rueegsegger. Muen-an x86/64 separation kernel for high assurance. *University of Applied Sciences Rapperswil (HSR), Tech. Rep*, 2013.
17. Xobs Bunnie. Lecture: The Exploration and Exploitation of an SD Memory Card. Chaos Communication Congress 2013.
18. Benoît Camredon. USBiquitous: USB intrusion toolkit. In *SSTIC 2016*. SSTIC, 2016.
19. S. Chari, C.S. Jutla, J.R. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. pages 398–412.
20. Yue Chen, Yulong Zhang, Zhi Wang, and Tao Wei. Downgrade Attack on TrustZone. *arXiv preprint arXiv:1707.05082*, 2017.
21. Christophe Clavier and Kris Gaj, editors. *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*. Springer, 2009.
22. Josh Datko, Chris Quartier, and Kirill Belyayev. Breaking Bitcoin Hardware Wallet. *DEF CON 2017*, 2017.
23. Andy Davis. Revealing embedded fingerprints: Deriving intelligence from usb stack interactions. *Blackhat USA*, 2013.
24. Matthias Deeg and Schreiber Sebastian. *Cryptographically Secure? SySS Cracks a USB Flash Drive*, 2009. `https://www.syss.de/fileadmin/dokumente/Publikationen/2009/SySS_Cracks_SanDisk_USB_Flash_Drive.pdf`.
25. Jakob Ehrensvärd. *Secure Hardware vs. Open Source*, 2016.
26. Darren Kitchen et al. Hack5 USB Rubber Ducky Part 1.
27. Clemens Fruhwirth. *New Methods in Hard Disk Encryption.* na, 2005.
28. FTDI. *FT600Q-FT601Q IC Datasheet (USB 3.0 to FIFO Bridge).*
29. Trenton Henry, David Rivenburg, and Dan Stirling. Universal Serial Bus Device Class Specification for Device Firmware Upgrade. *Aug*, 5:47, 2004.
30. ARM Holdings. ARMv7-M Architecture Reference Manual, 2010.
31. George Kang et al. Jim Huang. F9-Microkernel implementation, 2012.
32. Paul A. Karger and Roger R. Schell. Thirty Years Later: Lessons from the Multics Security Evaluation. In *Proceedings of the 18th Annual Computer Security Applications Conference*, ACSAC '02, pages 119–, Washington, DC, USA, 2002. IEEE Computer Society.
33. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.
34. Ledger. Ledger BOLOS. `https://www.ledger.fr/2016/03/02/introducing-bolos-blockchain-open-ledger-operating-system/`, 2017.
35. Lara Letaw, Joe Pletcher, and Kevin Butler. Host Identification via USB Fingerprinting. In *Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on*, pages 1–9. IEEE, 2011.
36. Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. The case for writing a kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, page 1. ACM, 2017.
37. E. L. Loe, H. C. Hsiao, T. H. J. Kim, S. C. Lee, and S. M. Cheng. SandUSB: An installation-free sandbox for USB peripherals. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 621–626, Dec 2016.
38. Arnauld Michelizza. Programmation d'un noyau sécurisé en Ada. *SSTIC*, 2013.

39. Nir Nissim, Ran Yahalom, and Yuval Elovici. USB-based attacks. *Computers & Security*, 70:675–688, 2017.
40. Nitrokey. How Nitrokey's Firmware is Protected Against BadUSB and NSA. `https://www.nitrokey.com/news/2015/how-nitrokeys-firmware-protected-against-badusb-and-nsa`.
41. Karsten Nohl and Jakob Lell. *BadUSB - On accessories that turn evil*, 2014. `https://srlabs.de/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf`.
42. Johannes Obermaier and Stefan Tatschner. Shedding too much Light on a Microcontroller's Firmware Protection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017. USENIX Association.
43. Oracle. Java Card 3 Platform Runtime Environment Specification, Classic Edition Version 3.0.5, 2015.
44. Jean-Michel Picod, Rémi Audebert, Sven Blumenstein, and Elie Bursztein. Attacking encrypted USB keys the hard(ware) way. *Black Hat USA*, 2017.
45. Benjamin C Pierce. *Types and programming languages*, 2002.
46. Gerald J Popek, Mark Kampe, Charles S Kline, Allen Stoughton, Michael Urban, and Evelyn J Walton. UCLA secure Unix. In *afips*, page 355. IEEE, 1979.
47. Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. *SIGOPS*, December 1981.
48. Matthieu Rivain, Emmanuel Prouff, and Julien Doget. Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers. In Clavier and Gaj [21], pages 171–188.
49. José F Ruiz. Going real-time with Ada 2012 and GNAT. *ACM SIGAda Ada Letters*, 33(1):45–52, 2013.
50. Mohamed Sabt and Jacques Traoré. Cryptanalysis of GlobalPlatform Secure Channel Protocols. Cryptology ePrint Archive, Report 2017/032, 2017. `https://eprint.iacr.org/2017/032`.
51. Alireza Sahami Shirazi, Peyman Moghadam, Hamed Ketabdar, and Albrecht Schmidt. Assessing the vulnerability of magnetic gestural authentication to video-based shoulder surfing attacks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2045–2048. ACM, 2012.
52. Bruce Schneier, Kathleen Seidel, and Saranya Vijayakumar. A worldwide survey of encryption products. 2016.
53. Dave Jing Tian, Adam Bates, and Kevin Butler. Defending Against Malicious USB Firmware with GoodUSB. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 261–270, New York, NY, USA, 2015. ACM.
54. Dave Jing Tian, Nolen Scaife, Adam Bates, Kevin Butler, and Patrick Traynor. Making USB great again with USBFILTER. In *USENIX Security Symposium*, 2016.
55. W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *ACM*, June 1974.
56. Zeitcontrol. BasicCard Developer Manual V8.15, 2013.