Once upon a time in IoT: an industry-grade OS perspective for IoT security

Patrice Hameau, Victor Servant, Philippe Thierry and Florent Valette
firstname.lastname@ledger.fr

Ledger

Abstract. Last year [24] we started to work on a separated deported UI [1] designed to support an efficient secured and trusted display management with enhanced security level as alternative to technologies such as TrustZone. The goal was to be able to securely receive, manipulate and display requests from an eSE [2] in a separated, dedicated, control/data plane, with non-secure parts outside of this plane fully unaware of such a path.

In the meantime, we have worked on a more formal specification on how to properly support a deported UI in our products, while still including our initial use cases as defined in [24]. Our work has been focused on deported trusted and secured UI architectures where an eSE drives directly an auxiliary UI component. Considering also our needs for modern UI rendering, we have then started to look on how to implement such an architecture on various MCUs,[3] such as the STM32 family from STMicroelectronics, yet with portability in mind.

After an in-depth review of the state of art, no convincing open solution has been identified on MCUs for hosting the firmware pieces of such deported UI. From there is born a new secure and versatile Operating System (OS), denoted *Outpost OS*, conceived to support at the very same time code integration of various origins, runtime isolation, high level of robustness and security, and industrialization and maintenance constraints. This article presents this new OS and its main associated concepts.

1 Introduction

1.1 Deported UI needs

Following the work conducted and previously published in SSTIC 2023 [24] we have converged on a general architecture hardware design for our various secure and trusted deported UIs use cases. These use cases require a separated, dedicated UI controller, as described in

[1] User Interface
[2] embedded Secure Element
[3] Micro Controller's Unit

Figure 1. Such an architecture was tested with a bare metal firmware to validate its feasibility and performances, and the results were described in the previous year's article.

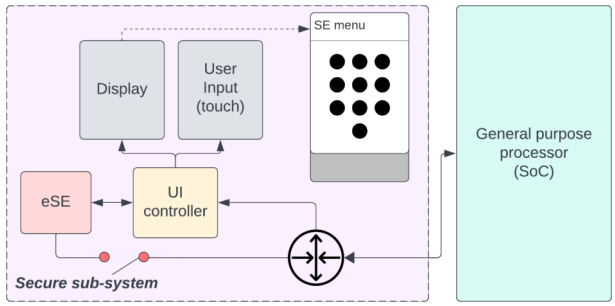

Fig. 1. High level view of secure and trusted deported UI architecture

Since then, we took time to review and formally specify the functionality, industrial and security needs and their impacts of an ideal software architecture for such a deported UI architecture. This step has been very valuable as it has lead us to formalize and refine the following aspects:

— Defining a strong and mature enough basis for our architectural and security model to enforce the system robustness, considered threats, and take into account the optional need of security certification process
— Easing the management of the technical debt associated to the MCU's software ecosystem during the natural evolution of the product lines
— Taking into account the constraints of industrially produced and maintained secure IoT devices (different developments environments, lifecycle and constraints for the parts composing it, and delivery management process)

From this study it has appeared that we need to use a secure Operating System (OS) to support the deported UI functionality on the MCU acting as the UI Controller. We have then defined more in details the requirements for this secure OS, which will cover in the next paragraph.

1.2 Secure operating system requirements

As per the high level specification for a deported UI (as well as for other future usages envisaged on our products we wanted to take into

account at the same time) we have refined our needs for a secure OS as per the following technical requirements list:

Requirement 1 *The OS guarantees a high level of security and robustness at runtime,*[4] *with predefined application assets, full isolation (memory, hardware resources...) of each application runtime, and resistance to certain logical and non-invasive threats.*

Requirement 2 *The OS relies on micro-kernel concepts to minimize its Trusted Code Base (TCB)*

Requirement 3 *The OS concepts, its build system and Application Programming Interface (API) are not dedicated to a specific market or usage*

Requirement 4 *The OS is open-source, with a non-contagious license model*

Requirement 5 *A SDK*[5] *tailored for a chosen OS configuration can be easily built and delivered to application developers*

Requirement 6 *The OS supports applications developed at least in the following programming languages: C11 and Rust (chosen to start as memory-safe language)*

Requirement 7 *The application developer can rely on supported parts of POSIX PSE-51-1 API [26] to ease application testing and portability*

Requirement 8 *Lifecycle, confidentiality, authenticity and traceability of each component composing the device image shall be natively supported as per OS architecture and integrated in the build system for both applications developers and product integrator*

Requirement 9 *The product integrator can rely on a dedicated and autonomous Integration Kit (IK) to build device images without needing to be able to access to the application developer development environment, and enforcing reproducible build.*

Requirement 10 *The product integrator's Integration Kit (IK) complies with the SCAP*[6] *framework (CPE*[7] *generation from delivery manifest) and allows notably SBOM*[8] *generation*

[4] What is aimed here is detailed in *Security Threat Model* hereafter
[5] Software Development Kit
[6] Security Content Automation Protocol [42]
[7] Common Platform Enumeration [41]
[8] Software Bill of Materials

The main reasons for these requirements is explained hereafter from the considerations for security threats and industrial approach.

Security Threat Model :
We have though from the start that such an OS, aiming at being used in small secure IoT devices, will be exposed to various attack scenarios. As the OS software implementation will run on a MCU with 'moderate' security level (at least considered as lower than the ones of an eSE [9]) some of these scenarios will be covered while others are voluntary kept out of scope (in general as unmanageable using pure software implementation). These scenarios have also been studied with a focus for our products ecosystem and associated security needs (and thus encompassing the link of the MCU with an eSE), but are however generic enough for most of secure IoT devices projects.

From our security analysis, we have considered the following threat model: [10]

Threat 1 *The adversary tries to tamper with the OS using logical attacks, either through external inputs it may control or using applicative code parts (bugs, trojan horses...)*

Threat 2 *The adversary tries to tamper the MCU using non-invasive hardware attack (side-channel attacks...)*

Threat 3 *The adversary tries to corrupt the boot sequence or the boot environment of the MCU before the start of the OS*

Threat 4 *The adversary tries to logically or physically tamper or replace the external parts (external peripherals, cut PCB line...) connected to the MCU onto which run the OS*

Threat 5 *The adversary tries to tamper the MCU with semi-invasive or invasive hardware attacks (silicon die access for micro-probing...)*

All threats can't be fully protected using software only counter-measures, moreover on MCUs that, even if some of them embed more and more efficient hardware security mechanisms, are not aiming to reach the security resistance of an eSE.

[9] embedded Secure Element
[10] For the sake of clarity, the threat model presented here has been simplified with only high level threats to keep this article short enough

Although, a well-designed OS, in the framework of a global system security architecture approach (notably if including the usage of an eSE connected to the MCU onto which run the OS), can include multiple defense in depth mechanisms increasing very significantly the necessary attacker technical skills, required equipments and time to perform a successful attack. This increase of attack complexity eases also the inclusion of some 'intelligent' attack detection mechanisms capable of identifying various attack scenarios (e.g. internal integrity checks, measure of response time of some services...). Such defense in depth mechanisms have been demonstrated in [3] and can be very efficient when setup as part of a complete defense system composed of hardware, software and architectural considerations.

We have separated in four main categories the support level of the OS security counter-measures for each of considered threats:

— *Fully Covered*: The OS must integrate proper counter-measures to be resistant against such threats (full coverage).
— *Partially Covered*: The OS must integrate counter-measures for at least a part of such threat cases. The overall threat scenario may not be covered by a software only approach in the OS, requiring a hybrid hardware/software/architectural response for full coverage.
— *Deferred*: While not aiming at defeating completely such a threat, the OS must include mechanisms to increase the necessary attack level, or make the attack path unpredictable, or the attack highly time-consuming. These counter-measures shall be seen here as part of a more complete global defense mechanism which can imply also other hardware and applicative counter-measures.
— *Out of Scope*: No dedicated software counter-measures is envisaged in the OS (either as inefficient against considered threat or because threat is not detectable at software level)

The Table 1 summarizes the support level of the OS counter-measures in regard to each threat defined in the threat model above.

Threat Nature	**Fully**	**Partially**	**Deferred**	**Out of Scope**
Threat 1: Logical attack	✓			
Threat 2: Non-invasive HW attack			✓	
Threat 3: Early boot attack				✓
Threat 4: External env. corruption		✓		
Threat 5: (Semi)Invasive HW attack				✓

Table 1. OS counter-measures support level vs considered threats

Threat 1 - Logical attack: Responding to this threat includes software architecture and defense in depth security features such as $W \oplus X$, Stack Smashing Protection, strict memory and resources partitioning, etc. These mechanisms are usual kernel-level security features (e.g. application in non-privileged mode, usage of MPU to provide only access to application resources, deterministic scheduler...), above multiple others described later. Response of such threat has also to be managed at application level by using proper software design architecture as notably ensuring efficient services separation based on the *separation of concerns* principle (e.g. creating smaller, separated services instead of macro-services, using separated SoCs for separated high level feature-sets, and so on). All these threat responses require proper design of OS API. But also some OS core properties (including efficient application switching and inter-process communication) and build-system level features (notably to allow efficient memory usage) to support such applicative software architecture based on separation of services.

Threat 2 - Non-Invasive attack: Such threats family encompasses attack that do not require alteration of the device, and can be subdivided in several classes: to simplify we consider here only the side-channel and fault injection ones. Side-channel threats (e.g. timing or power consumption measurement on USB cable) can be addressed by OS core mechanisms to mask with dedicated algorithms its internal operation and by offering some dedicated API for supporting resistant algorithms at application level (e.g. critical section). Fault injection threats (e.g. using power supply or electromagnetic glitches) are much more complex to address by software counter-measures only. Although, a well-defined software design with counter-measures using proper methodology and algorithm to detect and react to such threats can make them very complex or even impossible to materialize. Such counter-measures include CFI [11] on critical data paths such as system calls implementation, Hamming distance consideration or memory protection setting and update paths. Several components in the OS core shall include counter-measures against such fault injection threats (e.g. MPU management), but other typical security-critical components require also proper protection, such as the upgrade manager.

Threat 3 - Early boot attack: As this threat takes place before the execution of the OS core runtime, and that the secure boot mechanism is in charge of validating its integrity (and if needed its authenticity), the OS core code cannot have fully efficient response against it. In some configuration, the secure boot mechanism can leverage some secret token

[11] Control Flow Integrity

to the OS runtime that can be used afterwards to unlock some secret required by the OS core, thus limiting access to sensitive data if boot mechanism is completely bypassed. Such threat is indeed considered to be covered by the secure boot mechanisms of the MCU that include ROM and fuse usage [43]. We have thus considered for our model that the OS integrity is fully in charge of the secure boot of the MCU. And then that the OS core has to enforce the application integrity upon each boot, and to validate the OS core and application authenticity upon an upgrade.

Threat 4 - External environment corruption: This kind of threat is quite complex to thwart, but counter-measures such as timing measurements and behaviors analysis can be considered as a good staring point to detect unexpected alterations of unsecured peripheral components that cannot include defense mechanisms such as authentication (touchscreens, lcd panels...). Most of the counter-measures to be implemented there will be at peripheral driver level in application code. However, the OS core shall provide enough support through its API to implement them (e.g. precise timing measurements of some events, access to frequency or power consumption data...).

Threat 5 - Invasive attack: Such threats family are almost impossible to be countered by software, as most of the time they cannot be detected by the runtime code alone. Either these attack aimed at performing silent measurements (e.g. micro-probing on the MCU die) or they consist in altering temporary or permanently the MCU hardware itself, thus making the runtime OS code behave in an unexpected and unpredictable way, and then having the implementation of its security counter-measures ineffective. Even if software may help to react to such attack once detected, the detection and prevention of such attack requires to have dedicated hardware mechanisms which most of MCUs do not have.

In the end, all the threats not out of scope in the Table 1 (indeed a much more detailed version from our internal analysis), as well as further review of MCUs hardware counter-measures and known attack paths, have been used to precisely define what we wanted to encompass in 'high level of security and robustness at runtime' in requirement 1, and has allowed us to define a complete threat model and associated security objectives wished for such an OS.

Industrial Model Approach :
From an industrial point of view, the requirements 8 and 9 are the consequence of multiple industrial hypothesis, such as development lifecycle and collaborative / non-collaborative considerations, with respect for the

generic model defined in Figure 2. Typically, the overall software functionality may be delivered thanks to multiple teams or contractors, requiring the usage of external statements of work or internal team repartition. And as such it should not be constrained by a monolithic project-based software design and should allow different entity to make independent build of applications and integration in the device image.

Requirement 10 is a natural consequence of the previous ones, as it allows one to automate the survey of vulnerabilities and ease the dispatch of software update requirements to teams, contractors, etc., through a formally defined and widely used component identification mechanism.

The requirements 6 and 7 have been defined to help in the inclusion of C11 existing code, either as OSS,[12] MCU manufacturer's drivers and library, or commercial code, while encouraging the writing of critical applicative components in a memory safe language as Rust. The support of a subset of POSIX PSE51-1 has also been added as an efficient helper for business logic native testing on any POSIX-compliant host (e.g. linux) and also as a possible enabler for easing the integration and the training of developers.

Future-proofing is mostly ensured by requirement 3, with the aim of ensuring that any new product can reuse as much as possible the OS without having to generate per-project any specific codeblock in core components. All project-specific parts are deported to user-space tasks, and any common functionalities to any (sub-)product line can then be shared through either a dedicated application or library without impacting other projects that do not require it at all.

As open-sourcing is part of Ledger production model and user trust, we also wanted to go for an open-source OS, as defined in 4. This requires that the OS, with the constraints of such a model, be capable of handling also strictly confidential value-added software blocks and ensure that they may be kept confidential if needed. Nevertheless, proprietary but free to use software, such as ThreadX [12] are problematic as soon as we wish to include some missing security-critical value added at core level.

As soon as reusability of common components is a critical need, the application developer's environment and the product integrator's one need to be uncorrelated. Such a paradigm allows the application developer to use an easy, potentially multi-products, SDK, while at the same time to develop various common functionality that can be integrated as reusable blocks for different products. Typical examples are the upgrade subsystem

[12] Open-Source Softwares

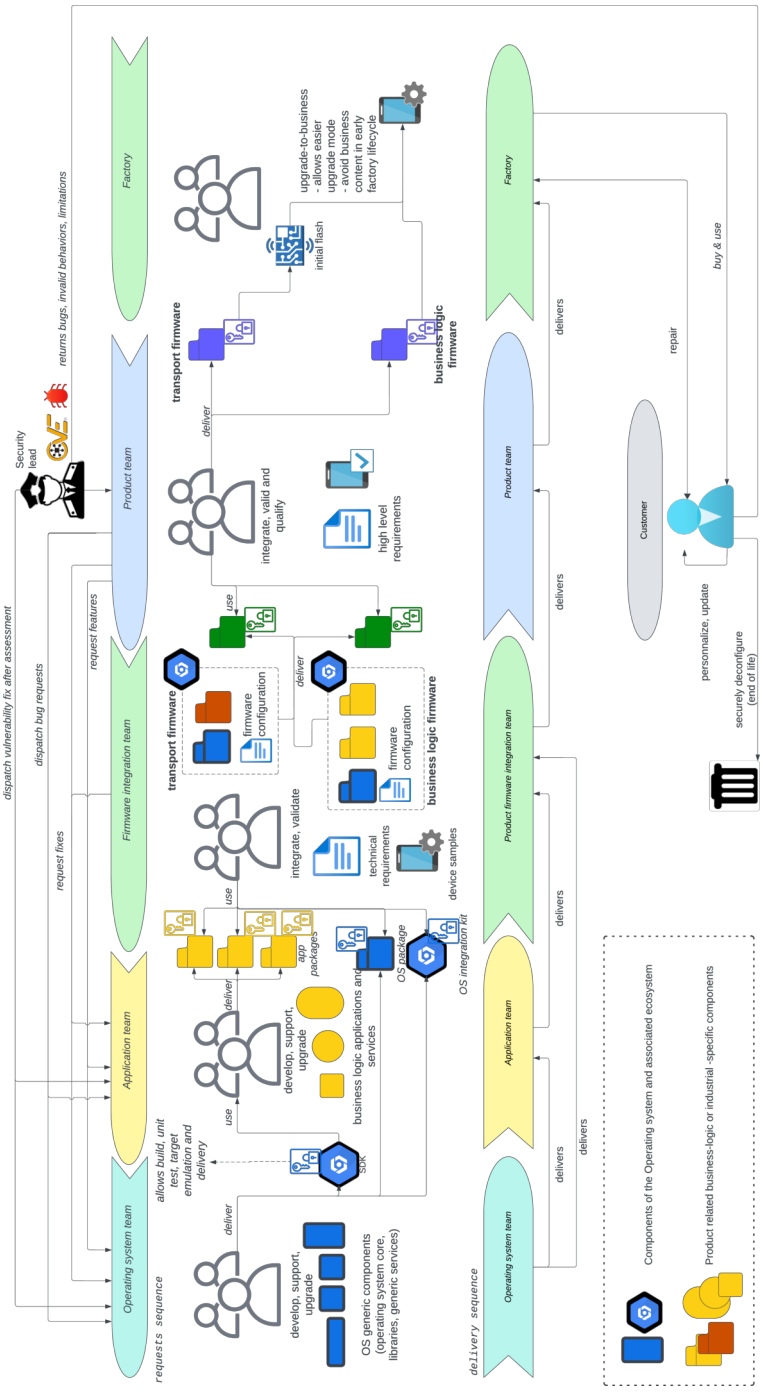

Fig. 2. Targeted typical generic embedded software delivery sequence

or the key management service, but it can be also useful to other various services. To support this separation, we have defined requirements 5 and 9.

Separating the business logic developer environment from the integrator environment has multiple impacts. Component delivering, ABI [13] compliance (in case of binary integration) and developer authentication mechanisms (such as usage of GPG signature to sign code, builds and images) become there critical for enforcing end to end security of integration and delivery chains. This requires the developer's environment and project Integration Kit (IK) of the OS to have such security mechanisms (signature generation and checks, manifest, role definition...) integrated by design. The requirement 8 is aiming at covering all these aspects.

1.3 Operating system state of the art

We took some time to perform a survey as exhaustive as possible on existing OS candidates suitable in regard to our needs. It has been found at the time of this article that mature and secure enough OSs for IoT (based on ARM® Cortex-M or RISC-V RV32E CPUs) designed with both high security and industrial-level considerations are mostly closed-source or proprietary solutions: ThreadX [12], ProvenCore-M [35], PikeOS [16] or SeL4 [30]. We have thus excluded them from our survey (as per our requirement 4). Careful review of well known open source OSs with security features brought us to the conclusion that they were not fulfilling our robustness and security level requirements, or were not tailored for industrial usage. We have thus finally not retained any of them (for example Riot [8] based on previous analysis already done in [10, 11]). Others potentially promising ones such as Muen [15] or Redox [33] have also not been considered as targeting mainly hardware out of our scope for IoT.

As a summary, the main results of our OSs survey are listed in Table 2. The following symbols have been used in this table:

— ✓ : the requirement is fully supported
— ~ : the requirement is partially supported (plugin, partial support)
— ✗ : the requirement is not supported
— ∉: the requirement is not in the scope of the target

One of the point standing out from the open source OSs reviewed is a lack of differentiation between the developer environment and the product integrator environment, or even the absence of product integration mechanism. Without such mechanism it is complicated to manage the

[13] Application Binary Interface

label	**MbedOS**	**TockOS**	**FreeRTOS**	**Wookey**	**Zephyr**
R. 1: Secure	✗	~‡	✗	~	✗
R. 2: μkernel	✗⋆	✓	✗	✓	✗
R. 3: COTS	✓	✓	✓	✓	✓
R. 4: OSS	✓	✓	~	✓	✓
R. 5: SDK	✓	✓	✓	✓	✓
R. 6: C, Rust	✗	✓	~	✗	~
R. 7: known API	~†	✓	✓	✓	✓
R. 8: components authentication	∉	∉	∉	∉	∉
R. 9: integration kit (IK)	∉	∉	∉	∉	∉
R. 10: IK: SCAP & SBOM	∉	∉	∉	∉	∉

‡ No SSP support
† CMSIS API is not POSIX and is not portable, but is yet a defacto standard API
⋆ partitioning but no core kernel functions delivered

Table 2. secure Open-Source Operating systems for small IoT survey

separation of COTS [14] with internal developments, the sharing of various core functionalities between applications, the separation of developments teams, the release lifecycle management, and the maintenance (tracking of bugs, CVEs...).

In order not to exclude most of the found solutions, we have considered there to lower our requirements for production integrations, by considering redeveloping a set of proper product management and integration tools on top of provided build toolchain. But it was no question however to lower our robustness and security level requirements (including considering future security certification needs), and here none of the analyzed open source solution was fulfilling them. In the end, from this survey we came unfortunately to the conclusion that no open source solution fulfilling our requirements was available.

As a response to this lack of corresponding technical solution, we have started to work on the specification and implementation of a new OS denoted *Outpost OS*. From our previous experiences, we were conscious that it would be a real challenge. But also that it will respond to an unmet need to have at disposal a generic secure OS for various IoT devices types that will support state-of-the-art security architecture principles (present already in a few open source solutions [10, 22]), the support of the high level security constraints as defined in 1.1, as well as industrial and maintenance lifecycle processes.

[14] Commercial off-the-shelf

Also, in order to avoid some traps we may encounter during a new design, the architecture and implementation of Outpost OS is undergoing a continuous review of our security team for both logical and hardware attack paths on the MCUs we are using. This includes external logical attack paths, as well as state-of-the-art hardware based attack paths (non-invasive and invasive; e.g. power supply, electromagnetic, and laser perturbations).

The remaining of this article explains more in depth what is Outpost OS. In Section 2, we describe the general concept of Outpost OS, starting with how we defined the global architecture and build system in order to achieve our high robustness and security level requirements 1-2, as well as industrial requirements 3-8. We then present in more details the overall security architecture, explaining the various security considerations that have been integrated to the OS in order to respond to the security threats considered in Section 1.1 and requirements 1 and 10.

In Section 3 we then walk through our first concrete use case initially described in [24], showing how Outpost OS responds to our needs for a secure deported UI. We conclude with the current development state of the OS, and the next planned features.

2 OutpostOS: a versatile and secure OS for small embedded systems

2.1 Design and concepts

From a fully specified build model... In order to ensure C and Rust hybrid build, efficient SDK delivery, build manifest and license management, Outpost OS is based on Meson [32] as build system and Kconfig [21] as configuration system. Subcomponents can also use CMake or Cargo for example, while their configuration is still based on Kconfig. Some build system restrictions have been set to allow automatic generation of firmware manifest files. Such files allow to deliver a SBOM and software CPE upon each firmware delivery time, providing a complete software traceability. They also allow SDK-based independent configuration (Kconfig-based), build and delivery.

The Outpost Operating system and tooling is also considered, in terms of licensing, in order to supports:

— Closed source applications, through the usual dual-licensing model that include BSD license family in userspace.
— Open-source applications using the (L)GPL licenses family with the same dual licensing pattern.

— Closed source projects (such as military or other specific domains), through the usage of dual licensing and Apache 2.0 license at kernel level.
— Applications and kernel are never linked together, keeping potential heterogeneous license model feasible at project level.

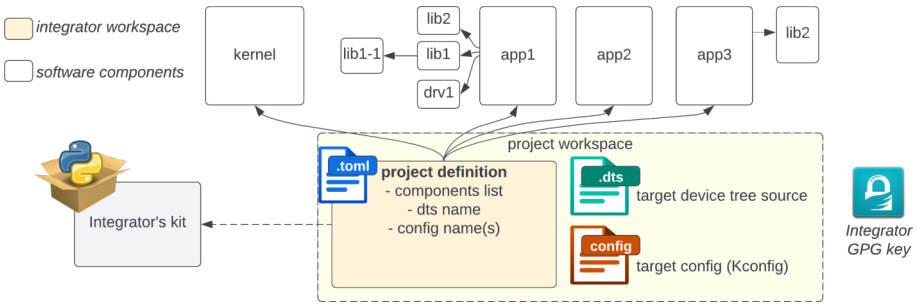

Fig. 3. Outpost OS Integrator's kit

While the Outpost OS is natively designed in order to be decomposed in a Software Development Kit (SDK) and an Integration Kit (IK), multiple additional development considerations have been taken. First it has been decided that any library, driver or application is an independent component which can be built independently. Libraries, like drivers, can be reused in multiple projects, using multiple boards, with the very same VCS [15] tag if needed. Secondly including prebuilt objects, like library binaries (for example in the case of NDA [16] restrictions) is also allowed.

These environments are described in Figure 3 and 4. All this allows to support Requirements 5 and 9.

Outpost OS applications, which are implemented using the Outpost SDK, are defined as services that can be either hardware-relative (interacting directly with a hardware peripheral) or being a portable business service (value added algorithm or function without direct hardware dependency).

In the first case, the application needs inputs from the project integration configuration to be properly configured. This typically includes peripheral pin-muxing configuration, mapping, and so on.

[15] Version Control System
[16] Non-Disclosure Agreement

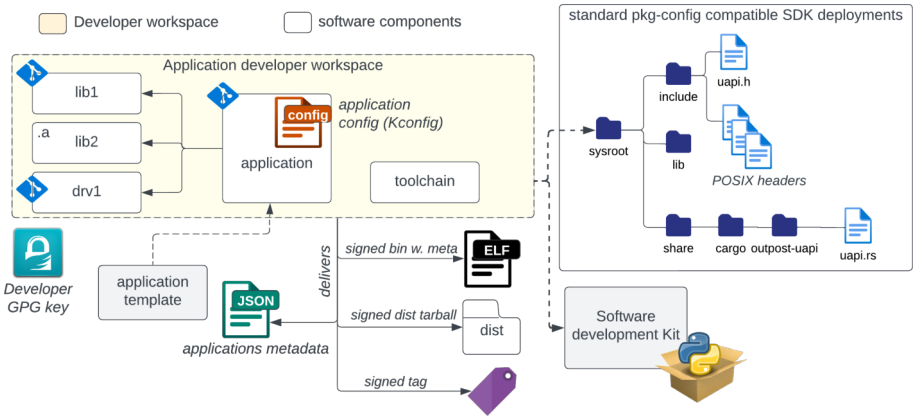

Fig. 4. Outpost OS Software development kit

In the later case, applications can be included in the Outpost IK as binary components, allowing prebuilt services and NDA restriction compliance for value added algorithms if needed.

Figure 5 points the different application types and the way they are used in Outpost.

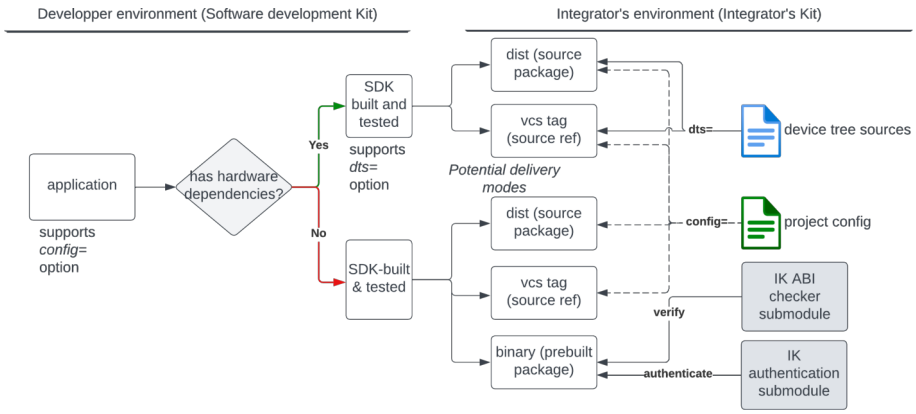

Fig. 5. Outpost application families

Such a paradigm is highly impacting on the overall OS build system security. To achieve that, Outpost IK voluntary delivers three independent binary blocks:

— Applications binary blobs

— Kernel binary blob, containing Outpost micro-kernel, denoted *Sentry*
— Applications meta-information binary table

At application build time, the SDK always delivers a relocatable ELF binary hosting both executable content **and** all required meta-information with a GPG-signed authentication mechanism.

In the case of relocatable ELF binary, Meta-information signing works in the same way as the Debian package signing model used in *dsc* and *changes* files [2, 9]. In Outpost though, metadata are stored in the `.note` section of the ELF instead of a separated file, in a similar way Linux does for signed external modules [1]. Nevertheless, these fields are not run-time checked (like Linux does) but project integration time checked.

On the other hand, dist packages and VCS tags being standard delivery formats, these last ones, containing reproducible sources, delivers all the required configuration to guarantee a reproducible build. The Dist package also delivers a GPG-signed manifest in order to authenticate the developer, while VCS sources use the standard VCS GPG-based authentication scheme.

Such meta-information generation is a production of the SDK and complies with SystemD package notes specification [17] [38], and is used as an input by the IK (Integrator's Toolkit) to rebuild source application for dist or VCS deliveries, generate a valid layout for the target, and to forge the firmware application meta-information binary table. This table is then always signed by the project's integrator's key, and is authenticated at runtime, as explained later, in Section 2.3. Figure 7 describes the way metadata is included in binary deliveries, while source deliveries are equivalent GPG-authenticated JSON files. Figure 6 describes the overall authentication sequence.

To achieve that in the IK, input binary applications authentication is validated against the project GPG repository, in order to authenticate the developer. This step also validates the relocatable binary integrity, using the GPG-signed hashes fields in the generated metadata.

It is to note that the IK is also responsible for validating that the SDK used comply with the current project ABI, using the SDK version set in the metadata.

In SDK-based application build, the application metadata also hold generic, application-hosted information. In order to produce the final ELF file with final metadata, some will be superseded by the project integrator

[17] Package metadata requires '–package-metadata' linker flags
linker required version: bfd, gold $\geq$ 2.39, mold $\geq$ 1.3.0, lld $\geq$ 15.0

in order to correspond to the project configuration, while others will be kept.

Typically, superseded fields are the one that impact the overall project integration, such as application priority or quantum, while application-local fields such as owned configuration, stack or heap size are kept.

The result of such a process enables:

— authenticated relocatable pre-built ELF file with developer's metadata, signed by the developer's GPG key
— authenticated final ELF file with integrator superseded metadata, signed by the integrator's key
— source-based deliveries, authenticated through VCS developer's GPG signature, producing ELF authenticated with integrator's GPG key

To finish, the IK then delivers a Integrator's key signed SBOM and build manifest, that ensure traceability for all input artefact references, being pre-built or not.

At the end, all input artifacts are authenticated, having their integrity checked by the IK. This makes Requirement 8 also supported by Outpost OS build system.

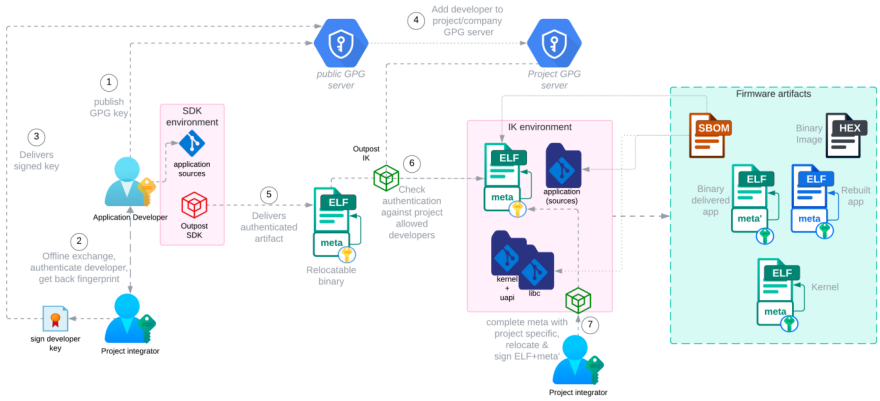

Fig. 6. Outpost SDK and IK application authentication process using GPGl

Based on the ELF integrator's signed metadata and sections information, the meta-information table is forged. This table is a binary blob that is read by the kernel at startup.

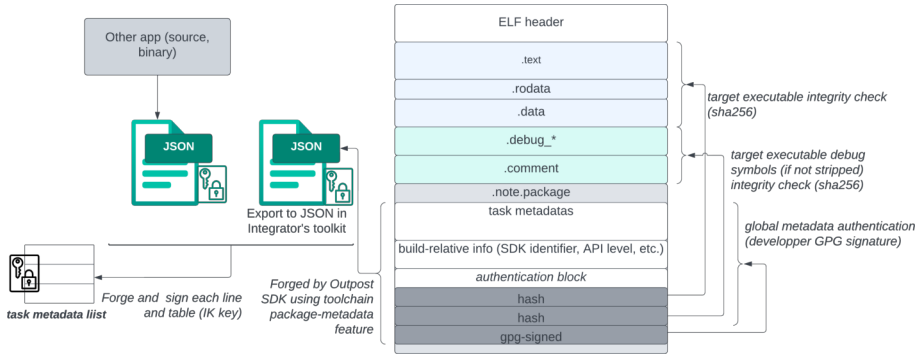

Fig. 7. Outpost application metadata forge and authentication model

Its structure is strictly defined using JSON schemas that describes each field type and field order. To ensure valid compliance to the current kernel ABI, the kernel is delivered with associated tooling and schemas as IK inputs, allowing validation of ABI compliance.

. . . with portability and maintainability in mind. . . To avoid any MCU-specific or project specific content in drivers, kernel or other OS components, the platform specification uses the standard Open-Firmware *device-tree source* [28] format. In Outpost OS though device-trees are not compiled to be included at runtime, but instead are used to generate source files with all device-trees information, similarly to what Zephyr [34] does. This allows to support device-tree based configuration while keeping a small target footprint.

Drivers implementations can hold their own device-tree file(s), which can be overloaded at project integration time if needed. It enables an easy and efficient way to allow both autonomous and project-based driver built using the same VCS tag, with the device-tree passed as an input parameter.

In order to deliver to the application developer an easy-to-use build environment, Outpost OS permits the usage of a fully independent SDK that does not require either the source of Sentry micro-kernel or others applications. Thanks to the usage of POSIX API, the C application business logic can be easily unit tested natively on the build host, while Rust code can be easily checked with Clippy [27]. This allows the usage of multiple independent project integration environments, for multiple product(s).

The Outpost SDK aims to delivers all the required tooling to ensure ABI compliance checking for binary deliveries. It is done by including various information on the SDK version used in the application metadata, which can be checked at integration time, with respect for the semantic versioning principles [19].

... down to the product deployment and lifecycle The Outpost OS allows the usage of business-function centric applications. They could be separated into micro-services [39,44] in order to separate various hardware backends, such as communication buses, cryptographic backend and so on. This is done naturally by creating small tasks, each one corresponding to a given application, that are strictly separated in terms of memory using the MPU, and delivering higher level interfaces to others. Of course, such a paradigm is not always applicable at its full extent due to the constrained footprint and performances.

Spatial and temporal isolation is also considered between task sets as in next releases, through the already considered notion of task domains, allowing hierarchical scheduling policy (scheduling the sets with differentiated scheduler) for strictly separated tasks sets.

It is to note that, by new, the kernel scheduling policy is based on a Round-Robin multi-queues with fixed priority and quantum management. Nevetheless, the scheduling model is built to support easy scheduling substitution, allowing, for example, RM [18] or TDM [19] schedulers. Although, the overall real-time compliance consideration is not, at this step of the Operating System core functions implementation, considered as a high priority feature.

To draw near such a design in a performant enough manner, the Outpost API must be efficient and allow efficient application scheduling and optimized memory footprint. This is why performance, deployment and product lifecycle have been dully considered at each stage of the Outpost OS specification:

1. *The Sentry kernel*: Upon startup, the *Sentry* micro-kernel has no idea of the list of applications that will be executed on top of it. It will discover the applications list through a strict parsing of a dedicated area containing the metadata of all the applications present. A maximum threshold is defined in order to ensure a strictly bounded maximum number of applications (and thus size of associated dedicated area).

[18] Rate Monotonic scheduling
[19] Time division Multiplexing scheduling

Sentry micro-kernel is quite similar to the EwoK kernel [10], in terms of UAPI, but several modifications and enhancements have been added:

— Shared memory management, with ownership and permissions management
— Support of signals
— Enhanced one-copy IPC implementation
— Enhanced single call burst-compliant user ISR support
— Enhanced scheduler with dual priorities and quantum support
— Support of MCU low power modes
— Dedicated API for core-controlled events, allowing support of attack detection and post-mortem usage

Its implementation, at the time of this article, is made in a hybrid C and Rust implementation, including some formal proof on some part of the C code. The whole syscalls gate and UAPI library being written in Rust.

2. *List of Applications Metadata*: The applications metadata list is defined as a table. Each entry contains a pointer to a dedicated area placed at after the binary of each application:

— A dedicated 64 bits magic, specific to the current product
— Kernel ABI related versioning information, ensuring that next fields are properly parsed
— All application-related information and configuration (capabilities, layout, scheduling quantum, resources such as peripherals and shared memory blocks...)
— A SHA256 hash enforcing integrity of all application binary part (text, data, rodata)
— A SHA256 hash enforcing integrity of all application debug information
— A GPG signature on all the dedicated area contents above, enforcing the authenticity of all application binaries, information and configuration data.

The public key used to verify the GPG signature above is generated by the IK tools for each Outpost OS build. It is for now, included in the binary of Sentry micro-kernel itself (the integrity and authenticity of the Sentry micro-kernel being enforced by the MCU Secure Boot). We may as part of future work support other authentication schemes and IK tools keys integration in binaries. The applications binaries being placed in memory by the IK tool,

they must be compiled either in PIE [20] mode (application metadata having a field for referencing a GOT [21]) or partially linked position dependent executable as described in Section 2.2.

3. *The applications*: Each application is a binary blob that is memory-mapped by the Outpost IK at a valid position, in association with a correct metadata information in the metadata list. By now, upgrading an application requires the IK to re-generate an up to date memory placement and thus a new firmware image.

   As part of future work, application update could be either done on-place, using free space, or dual-slotting mechanism. This will allow the usage of resilient upgrade methods in MCUs that do not support several flash banks, and can be used for increasing the availability of flash memory in such MCUs.

   By now, the metadata list is not duplicated using memory slotting but it can be considered as part of future work.

   There are two specific limitations to application positioning:

   — the memory protection unit mapping constraints
   — the non-volatile memory structure (usually sectors) when delta-upgrade is required

   MPU regions constraints may vary from one MCU to another [4, 5]. Such placement constraint is under the control of the Outpost IK, and can be considered with specific padding, alignment and slotting support to allow efficient delta upgrade management if needed. While no collision due to high increasing of a specific application happen, only the new application blob and the associated metadata need to be deployed. This should be the case for minor upgrades while major would require a full upgrade, in the very same way other OSes do [45].

   In our use cases, the number of concurrent applications on the target is small enough to allow a smart enough layouting that resolve both constraints. Such a number is considered, for our given integration project, between 6 and 16 separated applications depending on the various products.

[20] Position Independent Executable
[21] Global Offset Table

2.2 Application build and link time considerations

Prerequisite The IK is written in Python [22] and uses The Meson build system [23] and Ninja as build backend. The required C language revision is, at least C11 with a linker with package notes support [38].

As written earlier, application may be in position dependent or position independent executable. According to build mode, the application upgrade capabilities may be restricted. For instance, a per application upgrade model requires position independent. By now, memory placement and relocation are done by IK at integration time. Note that in the following description applies to the use case target architecture (Armv8M-Mainline) and GCC suite. In both case there is currently no support for dynamic linkage and thus shared objects.

Static Position Independent Executable (PIE): In PIE mode, the build flow perform by IK is the following:

— Application build in PIE
— Firmware memory layout/application memory placement
— Application relocation
— Firmware image generation

In this mode, data address is fetch from a GOT which contains data absolute addresses, only the offset in the table is known at application build time.

Application must be build with GCC flags '–single-pic-base' and '–no-pic-data-is-text-relative' as data memory relative placement compare to text is not known at application build. Those flags will generate code that complies with EABI by using 'r9' register as global data offset register [6]. At link time, '-static-pie' flag must be used this is expanded to '-static -pie –no-dynamic-linker' linker flags by GCC [23]. Those link flags tell linker to not produce any dynamic relocation section and all data only requires the GOT to be patched and thus, a dynamic linker at runtime is not needed.

In the next build step, after applications memory placement, the IK will patch application GOT entries with the right data addresses and loadable sections LMA [24]/VMA [25] are updated. At runtime, the initial task frame is initialized with GOT address.

For future work, in order to support per application upgrade, GOT might be fixed up at runtime.

[22] Python $\geq$ 3.10
[23] Meson $\geq$ 1.4.0
[24] Load Memory Address
[25] Virtual Memory Address

Static position dependent executable: In noPIE mode, the build flow perform by IK is the following:

— Application build and partially linked
— Firmware memory layout/application memory placement
— Final, per application, linker script generation based on memory placement
— Application linkage pass.
— Firmware image generation

In noPIE build mode, application can't be fully linked as the generated code is position dependent. Thus application must be partially linked before memory placement (e.g. GCC '-r' link flags). After memory placement, a dedicated, per application, linker script is generated all definitive LMA/VMA for text and data section and then the application is linked using **only** the partially linked elf as input and the generated linker script.

Note that this build mode does not allow per application upgrade but only whole firmware image upgrade.

2.3 Run-time security considerations

The Outpost OS security model has been considered over the overall lifecycle, starting with the components build and delivery time down to the target install, upgrade, repair and destruction time.

For the sake of simplicity, all the security considerations will not be presented in this single paper, as it would be too long to describe. Although, we try to focus on the core security concepts.

To start with, the micro-kernel based architecture has been defined as the initial OS usual best practice. Indeed, such architecture allows multiple efficient security considerations, such as supervisor code strict analysis and attack surface reduction [25, 31]. It also allows to support separation of concerns principle, that has already been demonstrated in both safety and security critical industrial systems [13]. To this initial architectural requirements, a set of runtime defense-in-depth considerations have also been included to respond to our typical technical threats scenarios in an efficient manner, complexifying and delaying as much as possible considered threats.

As a consequence, besides the usual security best practices (stack smashing protection (SSP), MPU-enforced W⊕X, strict and controlled application and kernel partitioning), some others defense-in-depth are also considered, such as potential shadow stacking support such as described

in [17], CFI,[26] or active software or hardware corruption detection, in order to answer security requirements defined in Section 1.1.

Moreover, the kernel is implemented with specific fault-injection protection schemes (hamming distance, no comparison to 0, critical checks duplication, etc.).

Nevertheless, it must not be forgotten that security is not the main target of the OS. As a consequence, we have defined, in a similar way to networking concepts [40] the notions of *slow path* and *fast path* at syscall level.

This has been achieved by defining a performance level consideration for each syscall, based on the usual, secure and performant embedded usage of devices. In our model, we consider that:

— all system wide events, external I/O and device manipulation should comply with high performances, allowing as small as possible business logic latency, including various timing consideration such as graphical VSYNC/VBLANK periods, network stacks performances or all DMA-related usage
— all platform initialization API (platform discovering, opaques discovering) can has its performances reduced
— power management support such as entering or leaving hardware stop-modes is also not considered as time critical

In the same time, syscall are considered as Finite State Machines. The FSM states are associated to a given manager execution, a syscall being a consecutive call to multiple managers (task, memory, etc.) that interact to deliver the corresponding service. Managers API can be called directly or through managers inter-dependencies.

The global syscalls repartition and design is then considered in the way defined in 8.

With such a model properly defined, it is possible to trigger differentiated security functions, so that slow paths and fast path can hold integrity checks with consideration for the overall system performances.

In one hand, as performances degradation associated to slow paths checks is accepted, this allows us to include more efficient security considerations for various logical and hardware exploitations:

— moving from a state to another being predictable, a CFI based on double sequence calculation is added, detecting any abnormal transition.

[26] Control Flow Integrity

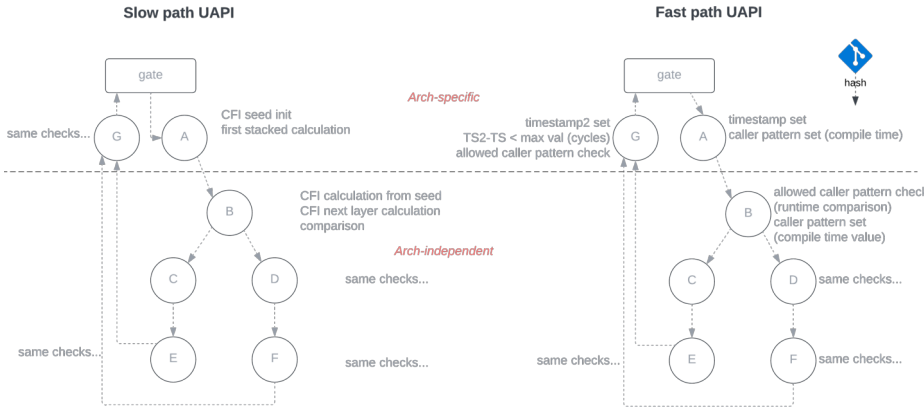

Fig. 8. Outpost syscall slow and fast paths repartition model

— hardware checks can be added, once per syscall execution, to detect potential hardware IP state corruption (MPU state, NVIC configuration, etc.).

In the other hand, fast path syscalls can't have their execution time such increased. The security checks is then reduced:

— the overall syscall duration in cycles is calculated for a given sub-architecture, in cycles.

— as syscalls are unpreemptible, the syscall execution time is measured down to the potential preemption sequence (for asynchronous syscall) or to the handler return (for synchronous syscalls) to detect any abnormal behavior (too short or too long), based on all execution paths measurement.

— instead of an effective CFI, a compile-time pattern forge for main functions is made and checked by callee. The check is a high performance numerical comparison. patterns vary with new releases, as using the commit hash as input seed, with respect for reproductible build.

In micro-kernel based systems, the logical attack surface is hosted by user-space applications, starting with those which communicate with external environment, as a consequence of Threat 1. To increase the overall system protection, the Sentry kernel runtime security model is based on multiple defense-in-depth mechanisms to reduce such an attack path impact. To start with, the kernel code as a particular kernelspace/userspace way to communicate. The kernel handlers always start with a reconfiguration of the application memory region. The goal here is to reduce the

accessible application memory from the kernel to a small, fixed, dedicated memory space denoted `svc exchange area`, through which applications and kernel communicate. The main impacts are:

— the `svc exchange area` is a build-time fixed, canary protected, subsection of the application memory, mapped at the beginning of the application memory layout, allowing fast and easy region resizing and bound-checking
— kernel syscall gate never manipulates pointers. The UAPI implementation is responsible for fulfilling this zone before calling the syscall handler, and for getting kernel result(s) back from it
— the area canary is updated each time a slow-path syscall is executed, and checked each time the syscall gate is called
— Sentry kernel code is unable to access application business data. Only this exchange zone is kept mapped during kernel execution context

To support all these defense-in-depth mechanisms, a set of security events has been defined, as well as a storing mechanism in a dedicated area of the volatile memory, with write before reset capability. The goal is to be able to react to such consecutive events as per a security policy, as for example erasing all assets and generating a report for auditing.

3 Outpost OS in deported-UI use case

3.1 Project specific construct

To answer to the project-specific architecture of our deported UI, a set of applications has been defined, as described in Figure 9:

— *User Interface Management*: this application is responsible for the *View* part of the standard MVC [14] [27] pattern for graphical interfaces. While this application uses OS generic components (mostly graphic related peripheral drivers), it also includes some OSS software notably for the graphical library, such as LVGL [37]. The application main loop is a product-specific component.
— *SE Communication Gate*: this application is the deported UI entrypoint from the eSE. It is responsible for handling the link and transport layers for all the exchanges performed with the eSE. It notably includes the usual flow control management and service identification between peers. Communication security (authentication, integrity, confidentiality) is supported in interaction with the

[27] Model-View-Controller

Key Storage and Management Service. The physical layer for communication is implemented using OS generic drivers (SPI, I2C. . .). The application itself is not defined as a generic remote gateway service but may, in the future, be defined as such.

— *Key Storage and Management Service*: This generic service is responsible for storing and manipulating the local cryptographic assets to support cryptography-relative functions, such as authentication, signature and encryption/decryption. It can be addressed using bare shared memories, signals or IPC in our first implementation (but we plan to use higher level API later on). The shared memory ownership and access request handling is explained in this section.

— *OS upgrade service*: This generic service is the foundation of the OS security. It is uncorrelated from the way the upgrade is delivered, but implies some specific security considerations, including chunk encryption and global upgrade content authentication (for example using HMAC). This service relies on the *Key Storage and Management Service* for securing the keys used for the upgrade process. In its initial implementation, only a whole firmware upgrade will be supported, based on dual flash banking A/B.

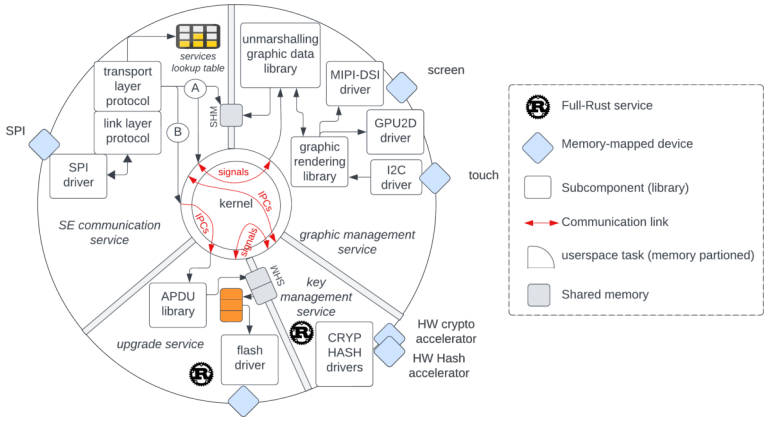

Fig. 9. Outpost OS-based deported UI use case

Listing 1: deported UI sample project configuration

```
name = 'extUI Project'
license = 'Apache-2.0'
license_file = 'LICENSE.txt'
devicetree = 'product_board_ref0.dts'
[sentry]
url = 'git@outpost-repo-url/sentry-kernel.git'
revision = 'v0.1'
method = 'git'
config_file = 'configs/project_depUI_debug_defconfig'
gpgsig = B9AD793E[...]0F6530D5E920F5C65
[app.secom]
url = 'git@my-repo-url/secom.git'
revision = 'v1.3.5'
method = 'git'
config_file = 'configs/project_depUI_defconfig'
gpgsig = 73D1790[...]E3C16097C115470EF8
# continuing...
```

Such a software architecture allows a relative straightforward project definition thanks to the TOML syntax used by the project integration's toolkit, as shown in Listing 1. In Outpost OS, each application local dependencies, such as user-space drivers, protocol stack and so on, are under the responsibility of the application build system, and mostly derive from the `config_file` line that use the Kconfig language to define the overall application metadata. In the same time, dependencies are pushed in the global delivery manifest so that no indirect dependency is lost. In our implementation, it is done using the meson build system manifest delivery mechanism.

Next to the project software configuration is also added the project target board configuration, that holds the board description using the device-tree syntax (DTS). In our project, we use VCS-based applications integration, where the project's device-tree overloads the project configuration.

3.2 Configuring user-space drivers and communication

enditemi

In Figure 9, user-space applications can use and own several MCU peripherals. MCU peripheral drivers are in user-space (excepted the ones managed by Sentry kernel at system level). It is up to the developer to provide for each service a *dtsi* fragment for the configuration of each

peripheral and to list all enabled peripherals in its package-metadata 2 through the Kconfig-based configuration mechanism.

At integration time, the Outpost IK will check that:

— A peripheral is enabled by exactly only one service
— No enabled peripheral is orphan (i.e. associated neither to the Sentry kernel nor to a service)

For security and portability reasons, user-space applications cannot have access at runtime to some peripherals configuration related to MCU core and managed by Sentry kernel, such as clock configuration or pin-mux [10,11]. Those are defined in a MCU-based *dtsi* fragment in a *dts* file at project top level. Other less sensitive MCU core peripherals configuration such as baudrate, clock signal polarity, word size, and so on, are defined in user-space fragments and can be overridden at project level if needed. For example a typical MCU *dtsi* fragment for a SPI bus peripheral description is shown in Listing 2. The application may there add peripheral specific configuration (e.g. clock polarity, required by the underlying protocol), as shown in Listing 3. As a last resort, the project is configuring the pin used for SPI1 for the board used in the project and can override any other fields if needed, as shown in Listing 4.

Note that, in order to enforce proper memory isolation, all bus master (as DMA controllers) are considered as MCU core peripherals and are under the strict control of the Sentry kernel, in the way EwoK kernel does [11].

At build time a table of mappable peripherals is built by Outpost IK with all application enabled peripherals and MCU-specific configuration if any. At runtime, before its usage, a peripheral must be pre-configured (it is called 'probing') and then memory mapped. The Sentry kernel validates for all operations relative to peripheral management that the application has the proper ownership and capability compliance. The peripheral management is done using build-time forged opaque identifiers, based on the project device-tree peripherals listing. The Sentry kernel is fully in charge of the probing of each peripheral: setup of the peripheral clock-tree (as per fragments defined at project top level), setup of required GPIO pins configuration and muxing, and setup of associated interrupts. Once the peripheral probing is done, its address range is then added to the application's allowed memory layout and the peripheral can be mapped upon application request with a dedicated syscall. Note that our implementation is made in a way to prevent that Sentry kernel can never access peripheral that it does not own itself.

As each mapped peripheral requires a dedicated MPU region, the number of peripherals mapped simultaneously at a given time for an application is limited: it is up to the application to handle peripheral map/unmap policy in function of its needs at a given time (best security practice being to have unused peripheral unmapped as soon as unused).

Listing 2: SPI1 MCU definition

```
spi1: spi@40013000 {
  compatible = "st,stm32-spi";
  #address-cells = <1>;
  #size-cells = <0>;
  reg = <0x40013000 0x400>;
  clocks = <&rcc STM32_CLOCK_BUS_APB2 0x00001000>;
  interrupts = <35 5>;
  status = "disabled";
};
```

Listing 3: SPI1 service definition

```
&spi1 {
  st,spi-clock-pol-inv;
  status = "okay";
};
```

Listing 4: SPI1 project definition

```
&spi1 {
  pinctrl-0 = <&spi1_mosi_gpio>, <&spi1_miso_gpio>;
  status = "okay";
};
```

In Figure 9, some user-space applications use shared memory in order to exchange data. Each shared memory resource required by an application has to be declared in its metadata. Such resource is seen as a peripheral (aka a 'shared memory block peripheral') in the device-tree with the following characteristics:

— They have an unique owner
— They can't be 'released', but can be (un)mapped when needed by the application
— They are mapped read-write
— Synchronization mechanism is left to applications

The shared memory notion in Outpost OS is managed with the principle of a reserved memory pool at product integration time, as shown in

Listing 5. This memory pool is initialized by the Sentry kernel at boot time. An application can always map a shared memory block it owns, in the very same way a peripheral is mapped, by using a strictly unique identifier forged at build time. The owning application can also allow the access of shared memory block it owns to one other application using a dedicated set of syscalls. The information that an access has been given has to be managed by application developers using messaging (IPC/Signals) between applications.

The Outpost IK is in charge of collecting all required shared memory blocks resources, and to perform various verifications, as notably that the amount of memory reserved for shared memory is sufficient (avoiding race condition during runtime), and MPU restrictions. Even if the pool of all shared memory blocks is built by Outpost IK, and thus fixed before runtime, the memory base address of each shared memory block is not known by the application before runtime, as Sentry kernel can perform some scrambling between blocks to enhance security.

Listing 5: Sentry shared memory pool

```
reserved-memory {
  #address-cells = <1>;
  #size-cells = <1>;
  ranges;

  shared_memory: shm@2001000 {
    reg = <0x2001000 0x4000>;
  };
};
```

3.3 Measurements and performances

Our current Outpost OS implementation supports three build modes with different objectives:

1. *Release*: Mode for production with debug disabled (no debug related code included) and all security features activated
2. *Debug*: Mode for development with some security features deactivated, debug support and log levels added in the OS code
3. *KSelfTest*: Mode for auto-test with a dedicated user-space application for kernel UAPI regression and security testing

As shown in Table 3, our current Outpost OS footprint is quite small whatever the build mode used.

Build Mode	**Release**		**Debug**		**KSelfTest**	
	flash	SRAM	flash	SRAM	flash	SRAM
kernel	12,064	7,492	18,132	7,917	17,844	7,917
task metadata list (max=4)	800	0	800	0	–	
idle (user app)	588	260	588	260	588	260
autotest-for-ktest (user app)	–				3,372	4,424

Table 3. Size in bytes of kernel-related components on STM32F4 target

The boot time is around a couple of milliseconds in Release build mode (without considering standard security checks such as integrity and potentially authenticity verifications which are independent of OS implementation). And approximately a hundred times longer in Debug build mode as shown in table 4. Note that this penalty is due to the time taken by synchronous logging over UART at 115200 bauds speed.

Build Mode	**Release**	**Debug***
cycles	60,887	11,727,204
milliseconds	1.439 ms	1,407.265 ms

* with synchronous log on uart at 115200 bauds

Table 4. Outpost OS boot time (to first application start) on STM32F401 target

Context switch timing is measured using the `sys_yield()` use case, dimensioning the overall UAPI and handler mode traversal, and including as well the scheduler elect call and internal context switching. Results are shown in table 5.

MCU	**STM32F401**		**STM32F429**		**STM32U5A5**	
Architecture	ARMv7-M		ARMv7-M		ARMv8-M	
Core	Cortex-M4		Cortex-M4		Cortex-M33	
Frequency used	84Mhz		144Mhz		140Mhz	
DMIPS/Mhz [7]	1.26		1.26		1.57	
	cycles	μs	cycles	μs	cycles	μs
context switch*†‡	1,313	15.75	1,313	9.19	1,173	8.37

‡ with round robin multi-queues with quantum scheduler policy
† mean cycle count over 10^5 yield syscall
* build with GCC 12.3, *-Os* and resp. *-mcpu=cortex-m4* and *-mcpu=cortex-m33*

Table 5. Outpost OS context switch time on ARMv7-M/ARMv8-M targets

3.4 Limitation and Future Work

The Outpost OS and its associated ecosystem is still in its early stage of development, leaving us a huge set of features and improvements we aim to include at various levels.

At the time of writing this article, the OS already supports two ARM® architectures: *ARMv7-M* and *ARMv8-M*, and multiple STM32-based MCUs, from high performances (STM32F4xx) to newest low power ones (STM32U5xx).

Most of described security features are already fully or partially implemented, and designed to be adapted easily to the targeted security level and the considered threats. One of our upcoming work are features related to industrial-grade OS, with notably the support for logging in a dedicated secure storage in NVM [28] critical events, allowing product self-analysis of its integrity and security state, as well as support of secured post-mortem checks.

Other core features are also part of our short plans: low power management, integration with chip secure boot, as well as in-depth security considerations using in-code counter-measures against faults.

ARMv8-M TrustZone support is also only basically supported: we consider to fully support it later with all the restrictions associated to each runtime world [29].

In the meanwhile, we want to offer various C and Rust-based standard functionality at application level in order to offer to developers what they expect off the shelves from an OS used as foundation for generic multipurpose products. We will start with drivers for common MCU peripherals (using for example Rust traits), and then with support of some standard protocols and cryptographic stacks.

The DMA controller (DMA2D) used for graphical operation is fully in user-space application as a first quick implementation of our User Interface Management application. We are still analyzing how to adapt at best the syscalls required to guarantee full security properties (as made for standard DMAs controllers which are master on the bus), while providing all its capabilities for graphic operations.

We have also in mind some security improvements of the built chain itself, by including the complete forge of the SDK with the toolchain build (in the same way as Yocto does). It will make it possible to take advantage of features such as compiler-level hardening, as the ones provided recently by AdaCore [18]. We plan also to improve the verification and issuance of

[28] Non-Volatile Memory

cryptographic signatures processes for building the deliverables with the IK, by leveraging notably the support of other signature methodologies in addition to the GPG one we support for now.

The post-integration security compliance checker is also an evolution of the IK we want to work on. The goal here is to provide a security compliance analysis of a built image to assist the verification of the project configuration in regards of a given security profile, by providing complete product integrator's information and output deliveries metadata in various formats allowing easy compliance check. This work is still under definition and not yet bootstrapped, but we are convinced of its added value for industrial products.

4 Conclusion

In this article, we have presented how we have moved forward from the requirements of an initial deported User Interface proof of concept towards an industrial-grade secure OS with a complete dedicated toolchain.

After unsuccessful review of the state of the art, we have specified functional, security and industrial requirements with scalability and maintainability in mind for such an OS for small/medium IoT devices using MCUs. We have started its implementation, based on micro-kernel architecture, on market standard MCU, integrating multiple robustness and security mechanisms, from the delivery of security model down to OS runtime security checks, and capable of supporting several languages for applications developments. We have also worked on the setup of a toolchain at today's best standard level, with the configuration and integration level similar to what is found in Yocto [36] or Buildroot [20] projects. This toolchain allows one to properly configure, build and deliver software components and associated bill of material, while supporting separate developments with proper security and trust.

This OS will be used in Ledger's future products for supporting a secure deported UI where a MCU drives high resolution color displays under control of our eSE (as follow up of previously exposed deported UI use case). This industrial integration implies further planned improvements such as upgrade, power-management and high security requirements. Considering such work may also be of interest for building various industrial products where high level of security and robustness are required, we aim to share it with the Open-Source community to make it evolve considering various other needs and expertise.

References

1. Linux kernel module signing. https://www.kernel.org/doc/html/v4.15/admin-guide/module-signing.html.
2. Debian keysigning model, 2024. https://wiki.debian.org/Keysigning.
3. Amossys ANSSI, LETI EDSI, Oppida Lexfo, SERMA Quarkslab, and Thales Synacktiv. Trusted labs. inter-cesti: Methodological and technical feedbacks on hardware devices evaluations. *SSTIC 2020*, page 165, 2020.
4. ARM. Arm cortex-m3 processor technical reference manual, 2016. https://developer.arm.com/documentation/100165/0201?lang=en.
5. ARM. Armv8-m memory protection unit (mpu) programming using arm ds and cmsis, 2023. https://developer.arm.com/documentation/ka005771/1-0/?lang=en.
6. ARM. Procedure call standard for the arm® architecture, 2023. https://github.com/ARM-software/abi-aa/blob/main/aapcs32/aapcs32.rst.
7. ARM. Arm cortex-m processor comparison table, 2024. https://developer.arm.com/documentation/102787/latest/.
8. Emmanuel Baccelli, Oliver Hahm, Mesut Günes, Matthias Wählisch, and Thomas C Schmidt. Riot os: Towards an os for the internet of things. In *2013 IEEE conference on computer communications workshops (INFOCOM WKSHPS)*, pages 79–80. IEEE, 2013.
9. Andreas Barth, Adam Di Carlo, Raphaël Hertzog, Christian Schwarz, and Ian Jackson. Debian developer's reference, 2005.
10. Ryad Benadjila, Arnauld Michelizza, Mathieu Renard, Philippe Thierry, and Philippe Trebuchet. Wookey: Designing a trusted and efficient usb device. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 673–686, 2019.
11. Ryad Benadjila, Mathieu Renard, Philippe Trebuchet, Philippe Thierry, Arnauld Michelizza, and Jérémy Lefaure. Wookey: Usb devices strike back. *Proceedings of SSTIC*, 2018.
12. Marcelo Borges, Sofia Paiva, António Santos, Bruno Gaspar, and Jorge Cabral. Azure rtos threadx design for low-end nb-iot device. In *2020 2nd International Conference on Societal Automation (SA)*, pages 1–8. IEEE, 2021.
13. Jens Braband. Safety vs. security–why separation of concerns is a good strategy for safety-critical systems. In *Applicable Formal Methods for Safe Industrial Products: Essays Dedicated to Jan Peleska on the Occasion of His 65th Birthday*, pages 85–95. Springer, 2023.
14. James Bucanek. Model-view-controller pattern. *Learn Objective-C for Java Developers*, pages 353–402, 2009.
15. Reto Buerki and Adrian-Ken Rueegsegger. Muen-an x86/64 separation kernel for high assurance. *University of Applied Sciences Rapperswil (HSR), Tech. Rep*, 2013. http://muen.sk/.
16. Cédric Cazanove, Frédéric Boniol, and Jérôme Ermont. A linux container-based architecure for partitioning real-time tasks sets on arm multi-core processors.

17. Wonwoo Choi, Minjae Seo, Seongman Lee, and Brent Byunghoon Kang. Sum: Efficient shadow stack protection on arm cortex-m. *Computers & Security*, 136:103568, 2024.
18. Fabien Chouteau. Adacore enhances gcc security with innovative features, 2024. `https://blog.adacore.com/adacore-enhances-gcc-security-with-innovative-features`.
19. Alexandre Decan and Tom Mens. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, 47(6):1226–1240, 2019.
20. John Diamond and Kevin Martin. Managing a real-time embedded linux platform with buildroot. Technical report, Fermi National Accelerator Lab.(FNAL), Batavia, IL (United States), 2015.
21. Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. Analysing the kconfig semantics and its analysis tools. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 45–54, 2015.
22. Amit Levy *et al.* Tock - programmable iot starts at the edge, 2015. `https://www.tockos.org/`.
23. GCC. Gcc options for linking, 2023. `https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html#index-static-pie`.
24. Patrice Hameau, Philippe Thierry, and Florent Valette. From dusk till dawn: toward an effective trusted ui. *Proceedings of SSTIC*, 2023. `https://www.sstic.org/2023/presentation/from_dusk_till_dawn_toward_an_effective_trusted_ui/`.
25. Allison Husain, Mengzhu Sun, and Evgeny Pobachienko. zvisor: A micro-kernel for monolithic kernels.
26. Portable Operating System Interface (POSIX™) Base Specifications, Issue 8. Standard, IEEE, 2003.
27. Chunmiao Li, Yijun Yu, Haitao Wu, Luca Carlig, Shijie Nie, and Lingxiao Jiang. Unleashing the power of clippy in real-world rust projects. *arXiv preprint arXiv:2310.11738*, 2023.
28. Grant Likely and Josh Boyer. A symphony of flavours: Using the device tree to describe embedded hardware. In *Proceedings of the Linux Symposium*, volume 2, pages 27–37, 2008.
29. Lan Luo, Yue Zhang, Clayton White, Brandon Keating, Bryan Pearson, Xinhui Shao, Zhen Ling, Haofei Yu, Cliff Zou, and Xinwen Fu. On security of trustzone-m-based iot systems. *IEEE Internet of Things Journal*, 9(12):9683–9699, 2022.
30. Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: from general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy*, pages 415–429. IEEE, 2013.
31. Olivier Nicole. Automatically proving microkernel security. *RESSI*, 2020.
32. Meson project developers. The meson build system, 2024. `https://github.com/mesonbuild/meson`.
33. Redox project developers. Redox: A rust operating system, 2017. `https://github.com/redox-os/redox`.

34. Zephyr project developers. The zephyr operating system: device-trees usage, 2023. https://docs.zephyrproject.org/latest/build/dts/index.html.

35. ProvenRun SA. Provenrun proven-core m. https://provenrun.com/provencore-m/.

36. Otavio Salvador and Daiane Angolini. *Embedded Linux Development with Yocto Project*. Packt Publishing Ltd, 2014.

37. Lassi Syri. Generating uis at runtime for embedded devices using lvgl. Master's thesis, L. Syri, 2022.

38. Systemd. Package metadata for core files, 2023. https://systemd.io/ELF_PACKAGE_METADATA/.

39. Miroslav Tomić, Vladimir Dimitrieski, Marko Vještica, Radovan Župunski, Aleksandar Jeremić, and Hannes Kaufmann. Towards applying api gateway to support microservice architectures for embedded systems, 2022.

40. Nguyen Van Tu, Jae-Hyoung Yoo, and James Won-Ki Hong. Accelerating virtual network functions with fast-slow path architecture using express data path. *IEEE Transactions on Network and Service Management*, 17(3):1474–1486, 2020.

41. David Waltermire and Brant Cheikes. Forming common platform enumeration (cpe) names from software identification (swid) tags. Technical report, National Institute of Standards and Technology, 2015.

42. David Waltermire, Stephen Quinn, Harold Booth, Karen Scarfone, and Dragos Prisaca. The technical specification for the security content automation protocol (scap): Scap version 1.3. Technical report, National Institute of Standards and Technology, 2016.

43. Rui Wang and Yonghang Yan. A survey of secure boot schemes for embedded devices. In *2022 24th International Conference on Advanced Communication Technology (ICACT)*, pages 224–227. IEEE, 2022.

44. Siao Wang, Chenglie Du, Jinchao Chen, Ying Zhang, and Mei Yang. Microservice architecture for embedded systems. In *2021 IEEE 5th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, volume 5, pages 544–549. IEEE, 2021.

45. Ellinor Westerberg. Efficient delta based updates for read-only filesystem images: An applied study in how to efficiently update the software of an ecu, 2021.