$\mathcal{P}$ARseL: Towards a Verified Root-of-Trust over seL4

Ivan De Oliveira Nunes, *RIT, USA*, ivanoliv@mail.rit.edu
Seoyeon Hwang, *UCI, USA*, seoyh1@uci.edu
Sashidhar Jakkamsetti, *UCI, USA*, sjakkams@uci.edu
Norrathep Rattanavipanon, *Prince of Songkla Univ., Thailand*, norrathep.r@phuket.psu.ac.th
Gene Tsudik, *UCI, USA*, gene.tsudik@uci.edu

2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD) | 979-8-3503-2225-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCAD57390.2023.10323832

***Abstract*—Widespread adoption and growing popularity of embedded/IoT/CPS devices make them attractive attack targets. On low-to-mid-range devices, security features are typically few or none due to various constraints. Such devices are thus subject to malware-based compromise. One popular defensive measure is Remote Attestation ($\mathcal{R}$A) which allows a trusted entity to determine the current software integrity of an untrusted remote device.**

For higher-end devices, $\mathcal{R}$A is achievable via secure hardware components. For low-end (bare metal) devices, minimalistic hybrid (hardware/-software) $\mathcal{R}$A is effective, which incurs some hardware modifications. That leaves certain mid-range devices (e.g., ARM Cortex-A family) equipped with standard hardware components, e.g., a memory management unit (MMU) and perhaps a secure boot facility. In this space, seL4 (a verified microkernel with guaranteed process isolation) is a promising platform for attaining $\mathcal{R}$A. HYDRA [1] made a first step towards this, albeit without achieving any verifiability or provable guarantees.

This paper picks up where HYDRA left off by constructing a $\mathcal{P}$ARseL architecture, that separates all user-dependent components from the TCB. This leads to much stronger isolation guarantees, based on seL4 alone, and facilitates formal verification. In $\mathcal{P}$ARseL, We use formal verification to obtain several security properties for the isolated $\mathcal{R}$A TCB, including: memory safety, functional correctness, and secret independence. We implement $\mathcal{P}$ARseL in F^* and specify/prove expected properties using Hoare logic. Next, we automatically translate the F^* implementation to C using $KaRaMeL$, which preserves verified properties of $\mathcal{P}$ARseL C implementation (atop seL4). Finally, we instantiate and evaluate $\mathcal{P}$ARseL on a commodity platform – a SabreLite embedded device.

***Index Terms*—Remote Attestation, Root-of-Trust, TCB, Embedded Devices, seL4 microkernel, Formal Verification**

I. Introduction

Internet-of-Things (IoT) and Cyber-Physical Systems (CPS) devices have become ubiquitous in modern life, including households, workplaces, factories, agriculture, vehicles, and public spaces. They often collect sensitive information and perform safety-critical tasks, such as monitoring vital signs in medical devices or controlling traffic lights. Given their importance and popularity, these devices are attractive targets for attacks, such as the Colonial Pipeline attack in the American energy grid [2] and Ukraine power grid hack [3].

Attacks are generally conducted via software exploits and malware infestations that result in device compromise. Remote Attestation ($\mathcal{R}$A) is a security service for detecting compromises on remote embedded devices. It allows a trusted entity ($\mathcal{V}$rf) to assess the software integrity of an untrusted remote embedded device ($\mathcal{P}$rv). $\mathcal{R}$A serves as an important building block for other security services, such as proof of execution [4], [5], control-flow and data-flow attestation [6], [7], [8], [9], [10], [11], and secure software updates [12], [13].

Many prior $\mathcal{R}$A techniques (e.g., [14], [15], [16], [17]) focused on low-end devices, that run one simple application atop "bare metal". For example, SANCUS [17] is a pure hardware-based $\mathcal{R}$A architecture for low-end devices. Whereas, *VRASED* [14] is a hybrid (hardware/software) $\mathcal{R}$A architecture, while PISTIS [18] is a software-only one. All these architectures are unsuited for higher-end devices that execute multiple user space processes in virtual memory.

At the other end of the spectrum, enclaved execution systems [19], [20] implement $\mathcal{R}$A for user-level sub-processes (called enclaves) on high-end systems, e.g., desktops, laptops, and cloud servers. However, they require substantial dedicated hardware support, thus making this approach unsuitable for the comparatively resource-constrained mid-range devices that we target in this work.

HYDRA [1] is an $\mathcal{R}$A architecture aimed at such mid-range devices. It does not require additional hardware support other than an (often present) memory management unit (MMU) and a secure boot facility. HYDRA relies on a formally verified microkernel, seL4 [21], to provide strong inter-process memory isolation. However, neither HYDRA's implementation nor its integration with seL4, is formally verified. Also, as discussed in Sections II-B and IV-A, HYDRA implements both attestation and untrusted application-defined functionalities in the same runtime process. Thus, HYDRA's trusted computing base (TCB) implementation is application-dependent, and whenever an application changes, errors can be introduced within the TCB. As a consequence, even if the $\mathcal{R}$A component in HYDRA were verified, application bugs could still undermine its security due to the lack of guaranteed isolation. Unfortunately, moving away from this model also introduces non-trivial architectural challenges (see Section IV-B), requiring a clean-slate trust model.

Motivated by the above, this paper re-visits HYDRA trust model and proposes $\mathcal{P}$ARseL: Provable Attestation Root-of-Trust over seL4 Microkernel – a design that separates user-dependent components from the $\mathcal{R}$A TCB. This new model addresses the aforementioned challenges, leading to proper isolation, and facilitates formal verification. Specifically, we use formal verification to prove security properties for the (now isolated) root-of-trust in $\mathcal{P}$ARseL. Proven properties include memory safety, functional correctness, and secret independence. We then deploy and evaluate $\mathcal{P}$ARseL verified C implementation (atop seL4) on a commodity prototyping board, SabreLite [22]. $\mathcal{P}$ARseL implementation is publicly available at [23].

Organization: Section II overviews background, followed by our goals and assumptions in Section III. $\mathcal{P}$ARseL design is presented in Section IV and its implementation details are in Section V, along with formal verification. $\mathcal{P}$ARseL security analysis follows in Section VI and limitations are discussed in Section VII. The paper concludes with the related work overview in Section VIII.

II. Background

This section provides background information on seL4, $\mathcal{R}$A, and formal verification tools. Given familiarity with these topics, it can be skipped with no loss of continuity.

A. seL4 *Microkernel [21]*

seL4 is a member of the L4 family of microkernels. Functional correctness of its implementation, including the C code translation [24], is formally verified, i.e., the behavior of seL4 C implementation strictly adheres to its specification. To provide provable

memory isolation between processes, seL4 implements a *capability*-based access control model. A capability is an unforgeable token that represents a set of permissions that define what operations can be performed on the associated object at which privilege level. This enables fine-grained access control by granting or revoking specific permissions to individual components or threads. Also, user-space applications cannot directly access or modify their own capabilities, because each capability is stored in *Capability Space* (CSpace) which is managed by seL4. User applications interact with seL4 through system calls and operate on their capabilities indirectly. Since seL4 enforces strict access control and authorization checks for system calls, seL4 retains the ultimate authority over capabilities and their allocation, revocation, and manipulation.

As a micro-kernel, seL4 provides minimal functionality to user-space applications. For example, inter-processes' data sharing requires the establishment of *inter-process communication* (IPC) by invoking *endpoint* objects, that act as general communication ports. Each endpoint is given a capability by assigning it a unique identifier, called a "*badge*", which identifies the sender process during communication. Each process is represented in seL4 by its *Thread Control Block* object which includes its associated CSpace and *Virtual-address Space* (VSpace) and (optionally) an *IPC buffer*. CSpace contains the capabilities owned by the process. VSpace represents the virtual memory space of the process, defining the mappings between virtual addresses (used by the process) and physical memory. IPC buffer is a fixed region of memory reserved for IPC. To send or receive messages, a process places them in its *message registers* which are put in the IPC buffer and then it invokes the capabilities within its CSpace via seL4 system calls.

B. $\mathcal{R}$A & HYDRA

As mentioned earlier, the goal of $\mathcal{R}$A is for a trusted $\mathcal{V}$rf to securely assess the software integrity of an untrusted remote $\mathcal{P}$rv. To do so, $\mathcal{V}$rf issues a unique challenge to $\mathcal{P}$rv. Using the received challenge, $\mathcal{P}$rv computes an authenticated measurement of its own software state. This measurement is computed using either a $\mathcal{P}$rv-$\mathcal{V}$rf shared secret or a $\mathcal{P}$rv-unique private key for which $\mathcal{V}$rf knows the corresponding public key. $\mathcal{P}$rv returns the measurement to $\mathcal{V}$rf which authenticates it and decides on $\mathcal{P}$rv's state (i.e., compromised or not).

To the best of our knowledge, the only relevant prior result that attempted to fuse $\mathcal{R}$A with seL4 is HYDRA [1]. It operates in three phases: *Boot*, seL4 *Setup*, and *Attestation*. In Boot phase, $\mathcal{P}$rv executes a ROM-resident secure boot procedure that verifies seL4 binary. Upon verification, $\mathcal{P}$rv loads all executables into RAM and passes control to the kernel. In seL4 Setup phase, the kernel sets up the user space and initializes the first process, *attestation process* (AP). The kernel then hands control to AP after assigning all capabilities for all available memory locations to AP and verifying AP's binary. AP is then responsible for spawning all user processes with lower scheduling priorities and user-defined capabilities, initializing the network interface, and waiting for subsequent attestation requests. Finally, in Attestation phase (which comprises the rest of the runtime), upon receiving a $\mathcal{V}$rf-issued attestation request for a particular user-space process, AP computes an HMAC [25] of the memory region of that process, using a symmetric key pre-shared with $\mathcal{V}$rf, and returns the result to $\mathcal{V}$rf.

HYDRA AP implements several system functions that are unrelated to $\mathcal{R}$A functionality. While this approach simplifies *Boot* and seL4 *Setup* phases, it also makes HYDRA verification challenging. We further discuss this in Section IV-A.

C. F^*, Low^*, and $KaRaMeL$

F^* [26] is a general-purpose functional programming language with an effect system facilitating program verification. Developers can write a program and its specifications in F^*, representing that program's computational and side effects, and then formally verify that it adheres to those specifications using automated theorem-proving techniques. The type system of F^* includes dependent types, monad effects, refinement types, and the weakest precondition calculus, which together allow describing precise and compact specifications for programs using Hoare logic [27]. For example, Fig. 1 shows two simple functions in F^*. While both take an integer as input and output its absolute value, `abs_pos` "requires" the input integer to be positive as **pre-condition** and "ensures" that the result equals the absolute value of `x` as **post-condition**. The pre-condition of `abs_pos` can be instead written with refinement type input: `(x : int {x > 0})`. Both have the `Pure` effect, meaning that they are stateless functions, guaranteeing deterministic results and no side effects. `Tot` is a special type of `Pure` with no pre-condition, i.e., it is defined for all possible values of input so that it terminates and returns an output.

```
1  let abs (x : int) : Tot int
2    = if x >= 0 then x else -x
3
4  let abs_pos (x : int) : Pure int
5  (requires x > 0) (ensures fun y -> y = abs x) = x
```

Fig. 1: Example Functions in F^*

To support stateful programs, F^* provides `ST` effect with the form:

```
ST (a:Type) (pre:s→Type) (post:s→a→s→Type)
```

This means: for a given initial memory "`h0:s`" that satisfies pre-condition "`(pre h0)` is true", a computation "`e`" of type "`ST a (requires pre) (ensures post)`" outputs a result "`r:a`" and updates existing memory to final memory "`h1:s`", which satisfies the post-condition "`(post h0 r h1)` is true".

One notable feature of F^* is *machine integers* and arithmetic operations on them. Machine integers model (un)signed integers with a fixed number of bits, e.g., `uint32` and `int64`, while `FStar.Int.Cast` module offers conversions between these types. Using machine integers ensures that input and computation result values fit in the given integer bit-width, preventing an unintentional arithmetic overflow. In addition, one can express their secrecy level, denoted by '`PUB`' or '`SEC`'. The former is considered public and can be safely shared, while the latter is considered secret, i.e., F^* guarantees no leaks for them. Specifically, it prevents information leakage from timing side-channels and clears all memory that contains `SEC`-level integers when they are no longer needed.

Low^* [28] is a subset of F^*, targeting a carefully curated subset of `C` features, such as the `C` memory model with stack- and heap-allocated arrays, machine integers, `C` string literals, and a few system-level functions from the `C` standard library. To support these features, Low^* refines the memory model in F^* by adding a distinguished set of regions modeling `C` call stack – so-called *hyper-stack* memory model. For modeling `C` stack-based memory management mechanism, Low^* introduces a region called `tip` to represent the currently active stack frame and relevant operations, such as `push` and `pop`. Low^* also introduces the `Stack` effect with the form below, to ensure that the stack tip remains unchanged after any pushed frame is popped and the final memory is the same as the initial memory:

```
Stack a pre post = ST a (requires pre) (ensures
```

```
(λ h0 r h1 → post h0 r h1 ∧
(tip h0 = tip h1) ∧ (∀ x. x ∈ h1 ⇔ x ∈ h0)))
```

Programmers writing code in Low^* can utilize the entire F^* for proofs and specifications. This is because proofs are erased at compile-time and only low-level Low^* code is left and compiled to C code. Verified Low^* programs can be efficiently extracted to readable and idiomatic C code using the $KaRaMeL$ [29] compiler tool (previously known as *KreMLin*). $KaRaMeL$ implements a translation scheme from a formal model of Low^*, λow*, to CompCert Clight [30]: a subset of C. This translation preserves trace equivalence with respect to the original F^* semantics. Thus, it preserves the functional behavior of the program without side channels due to memory access patterns that could be introduced by the compiler. The resulting C programs can be compiled with CompCert or other C compilers (e.g., GCC, Clang).

D. HACL* *Cryptographic Library [31]*

HACL* [31] is a formally verified cryptographic library written in Low^* and compiled to readable C using $KaRaMeL$. Each cryptographic algorithm specification is derived from the published standard and covers a range of properties, including:

- *Memory safety*: verified software never violates memory abstractions so that it is free from common vulnerabilities due to reads/writes from/to invalid memory addresses, e.g., buffer overflow, null-pointer dereferences, and use-after-free.
- *Type safety*: software is well-typed and type-related operations are enforced, i.e., HACL* code respects interface, and all the operations are performed on the correct types of data.
- *Functional correctness*: input/output of the software for each primitive conform to simple specifications derived from published standards.
- *Secret independence*: observations of the low-level behavior, such as execution time or accessed memory addresses, are independent of secrets used in computation, i.e., the implementation is free of timing side-channels.

III. Goals & Assumptions

A. System Model

We consider $\mathcal{P}$rv to be a mid-range embedded device equipped with an MMU and a secure boot facility[1]. Devices in this class include I.MX6 Sabre Lite [22] and HiFive Unleashed [32] (on which seL4 is fully formally verified [33]). Following seL4 verification axioms, $\mathcal{P}$rv is limited to one active CPU core, i.e., it schedules multiple user-space processes, though only one process is active at a time. We assume that secure boot is correctly enabled prior to device deployment.

$\mathcal{P}$ARseL TCB consists of seL4 microkernel, the first process loaded by the microkernel in user-space, called *Root Process* (RP), and *Signing Process* (SP), also in user-space (details in Section IV). $\mathcal{V}$rf wants to use $\mathcal{R}$A to establish a secure channel with a particular attested user-space process. To facilitate this, $\mathcal{P}$ARseL attestation response can also include a unique public key associated with the process. $\mathcal{V}$rf can then use the secure channel to communicate sensitive data with the attested process, after verifying its integrity through $\mathcal{R}$A.

[1]Although common in mid-range embedded devices, secure boot requirement can be relaxed with weaker adversary model where $\mathcal{A}$dv does not have physical access to $\mathcal{P}$rv and the initial deployment of seL4 and $\mathcal{P}$ARseL TCB on $\mathcal{P}$rv is trusted.

$\mathcal{P}$ARseL provides a *static* root of trust for measurement of user-space process, i.e., the binary of processes are measured at their loading time. This is plausible because $\mathcal{P}$ARseL, by design, enforces that no new user process is spawned during runtime and no modifications on code occur without rebooting the device. On the other hand, $\mathcal{P}$ARseL design allows the user-process updates without modifying $\mathcal{P}$ARseL TCB. However, any updates require the device to reboot to re-measure the updated programs, which limits the scalability. We further discuss this limitation and possible alternatives in Section VII.

$\mathcal{P}$ARseL design is agnostic to the choice of cryptographic primitives. In fact, $\mathcal{P}$ARseL can support both (1) symmetric-key cryptography where $\mathcal{P}$rv and $\mathcal{V}$rf share a master secret from which a subsequent symmetric key can be derived, or (2) public-key cryptography where $\mathcal{P}$rv has a private signing key whose public counterpart is securely provisioned to $\mathcal{V}$rf. In both cases, the required keys can be hard-coded as part of the $\mathcal{P}$ARseL TCB prior to $\mathcal{P}$rv deployment.

B. Adversary Model

Based on the $\mathcal{R}$A taxonomy in [34], four main types of $\mathcal{A}$dv are:

1) *Remote*: exploits vulnerabilities in $\mathcal{P}$rv software and injects malware over the network;
2) *Local*: controls $\mathcal{P}$rv's local communication channels; may attempt to learn secrets leveraging timing side-channels;
3) *Physical non-intrusive*: has physical access to $\mathcal{P}$rv and attempts to overwrite its software through legal programming interfaces (e.g., via J-TAG or by replacing an SD card).
4) *Physical intrusive*: performs invasive physical attacks, physical memory extraction, firmware tampering, and invasive probing, e.g., via various physical side-channels.

We consider type (1) and (2) adversaries. Type (3) can be supported if $\mathcal{P}$rv hardware offers protection to prevent access to $\mathcal{P}$rv's secret key via programming interfaces. Protection against type (4) adversaries is orthogonal and typically obtained via standard physical security measures [35]. This scope is in line with related work on trusted hardware architectures for embedded systems [36], [16], [14], [15]. In terms of capabilities, if $\mathcal{A}$dv compromises a user-space process in $\mathcal{P}$rv, it takes full control of that user-space process, i.e., it can freely read and write its memory and diverge its control flow. We assume user-space processes as untrusted and therefore compromisable, except for $\mathcal{P}$ARseL TCB. Finally, we assume that $\mathcal{A}$dv can trigger interrupts at any time.

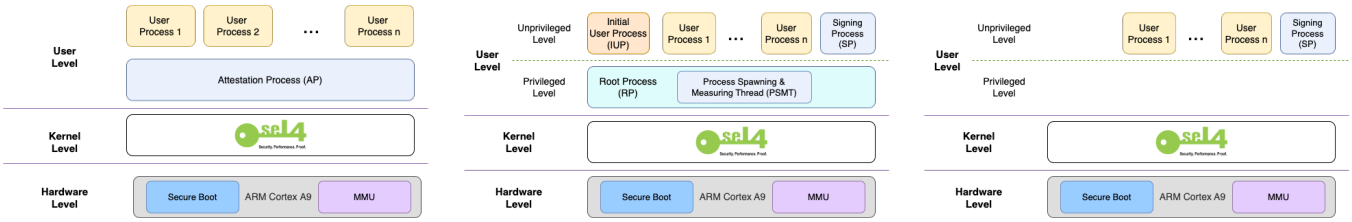
IV. Verified Root-of-Trust over seL4 ($\mathcal{P}$ARseL)

This section starts by describing HYDRA and its limitations. It then justifies our approach and discusses how $\mathcal{P}$ARseL realizes it.

A. HYDRA *& Its Limitations*

As mentioned above, HYDRA is composed of *Boot*, seL4 *Setup*, and *Attestation* phases. AP is the very first user-space process to run after seL4 *Setup*. As such, AP possesses all capabilities for all available memory and system resources. It is responsible for creating and managing all other processes, ensuring proper configuration of capabilities for them, and performing $\mathcal{R}$A.

We argue that this design results in an excessive and application-dependent TCB. First, formally verifying the implementation of AP is extremely challenging since it requires a giant manual proof effort that might not be achievable in practice. However, without formal verification, there is no guarantee that AP is vulnerability-free and correct. Since AP has all user-space capabilities, its compromise would lead to a breach of all seL4 isolation guarantees provided. Even



(a) HYDRA Execution Levels (b) $\mathcal{P}$ARseL on Boot (c) $\mathcal{P}$ARseL at Runtime

Fig. 2: Comparison of HYDRA (left) and $\mathcal{P}$ARseL Execution Levels on Boot (middle) and at Runtime (right)

assuming the feasibility of AP formal verification, process-spawning component of AP strictly depends on the specific user application configuration. This is so that AP can properly assign custom (user-defined) access control configurations to each application process. Thus, whenever an application changes, AP implementation needs to be adjusted accordingly. Doing so modifies the AP's previously verified TCB. It is clearly infeasible to re-verify AP implementation for all possible application-dependent configurations.

B. Design Rationale

To enable verifiability, the TCB size at runtime must be reduced, by identifying and removing unnecessary functionalities from the privileged AP process. HYDRA AP functionalities are:

① Spawning all user processes with memory/capability settings;
② Communication with $\mathcal{V}$rf over the network interface for $\mathcal{R}$A;
③ Attestation of all user processes;

First, we observe that including ② in the TCB yields no benefit since the security of $\mathcal{R}$A does not depend on the availability/integrity of the communication interface. Thus, we move this functionality out of the TCB and handle $\mathcal{P}$rv $\leftrightarrow$ $\mathcal{V}$rf communication in a separate user-space process. Second, ① performing initialization tasks that are not needed at runtime (i.e., post-boot). Third, further sub-dividing ③:

③-(a) Measuring (reading) the code binary for each user process;
③-(b) Signing the measurement with a private key and a challenge from $\mathcal{V}$rf;

③-(a) can be also done once, assuming that the code does not change post-boot (as mentioned in Section III-A). Thus, these components can be terminated after completion, at boot time, which effectively limits these components' exploitable time window to boot time.

Also, ① can be sub-divided into:

①-(a) Storing access control capabilities for all processes to be spawned;
①-(b) Spawning the user processes based on given access control capabilities;

To separate all user-dependent components from the TCB, a separate user process can perform ①-(a) and communicate with AP for ①-(b). Or it can be even just a configuration file that AP can read from. Finally, ③-(b) must be active at runtime to process $\mathcal{R}$A requests from $\mathcal{V}$rf, which represents the only potential remaining entry point for $\mathcal{A}$dv. To close this gap, this operation can be assigned to a tiny dedicated process, called *Signing Process* (SP). Due to its small size and independence of user-defined components, verifying SP is now relatively easier.

C. $\mathcal{P}$ARseL Design

Combining all the above, Fig. 2 shows $\mathcal{P}$ARseL execution levels at boot- and at run-time, as compared to HYDRA. $\mathcal{P}$ARseL sub-divides seL4 user-space into two execution levels: *Privileged* and *Unprivileged*. We refer to the privileged initial user process as *Root Process* (RP) which has a thread (for the roles of ①-(b) and ③-(a)), called *Process Spawning & Measuring Thread* (PSMT). In contrast, the processes at the unprivileged level have restricted capabilities assigned by RP. Unprivileged processes include *Initial User Process* (IUP) (for ①-(a)), SP, and user-defined processes (UP-s). Capabilities of any process at the unprivileged level do not allow access to any memory not explicitly assigned to that process. RP (including PSMT) and IUP are terminated at the end of boot phase, and only UP-s and SP remain during run-time, as shown in Fig. 2(c).

D. $\mathcal{P}$ARseL Execution Phases

$\mathcal{P}$ARseL has seven execution phases in total: three on boot and four at runtime. Three phases in the boot-time are:

(Secure) Boot: The boot-loader verifies, loads, and passes control to, seL4. Thereafter, seL4 verifies the integrity of $\mathcal{P}$ARseL TCB, i.e., the software that runs in RP, and passes control to RP, once verification succeeds.

Process Spawn: RP spawns PSMT as a thread. PSMT spawns IUP as an unprivileged process and establishes an IPC channel with it. Once spawned, IUP sends the configuration of user processes and their process ID-s (P_{ID}-s) to PSMT via IPC. Upon receiving a request, PSMT spawns a new process according to received capabilities. It also ensures that these capabilities are valid, not containing the write capability for its own code segment. Finally, it spawns and sets up an IPC channel with SP. Once all processes are spawned, PSMT sets up an IPC *endpoint* for each user process, assigns a unique *badge* for each endpoint, and associates this unique badge with P_{ID}.

Measurement: While spawning each user process, PSMT also measures (via hashing) its code segment, and stores the results in *measurement map* (mmap) with the P_{ID} as the lookup key. Once all measurements are complete, PSMT sends the entire mmap to SP through IPC, and RP (including PSMT) is terminated.

Once $\mathcal{P}$rv is booted and in a steady state, it repeatedly executes the remaining four phases at runtime:

Listen: SP listens to receive messages from user processes through the endpoint set up in the boot phase.

Request: Once a user process, UP, receives an attestation request from $\mathcal{V}$rf with a fresh challenge, Chal, UP transmits the request to SP through IPC system calls. The request message includes Chal and the public key of UP, pk.

Sign: Upon receiving a request, SP identifies the sender process, UP, from the activated endpoint badge and derives P_{ID}[2]. It then

[2]Note that seL4 guarantees that UP cannot forge its own endpoint badge. Therefore, the attested UP is the same process that provides pk to SP.

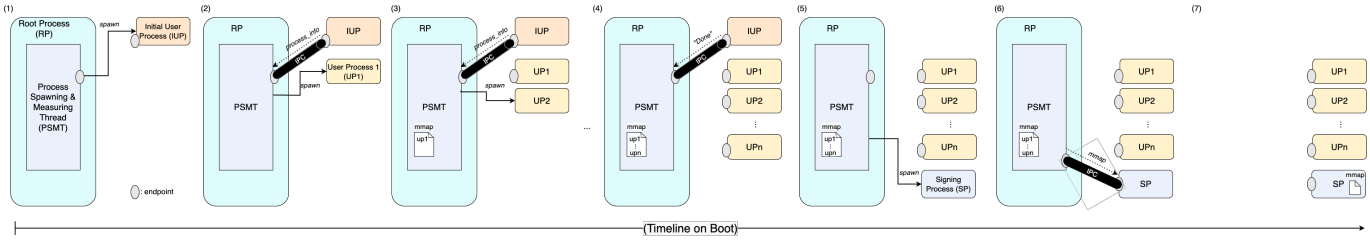


Fig. 3: Sequence of $\mathcal{P}$ARseL Execution Phases at Boot (After Secure Boot Checks)

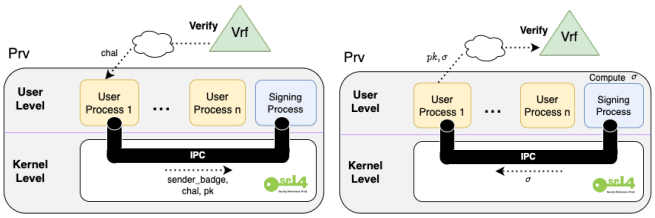


Fig. 4: Sequence of $\mathcal{P}$ARseL Execution Phases at Runtime

retrieves UP's measurement $\mathtt{m}_{\mathsf{UP}}$ from mmap using P_{ID} and signs $\mathtt{m}_{\mathsf{UP}}$ along with the request message using its secret key, $\mathcal{K}$. i.e.,

$$\sigma := Sign(\mathcal{K}, Hash(\mathsf{Chal}||\mathsf{pk}||\mathtt{m}_{\mathsf{UP}})) \quad (1)$$

Response: SP responds σ to UP via IPC. UP forwards σ and pk to $\mathcal{V}$rf. Finally, after successful σ verification, $\mathcal{V}$rf establishes a secure channel with UP using the received pk.

Fig.s 3 and 4 show the aforementioned $\mathcal{P}$ARseL execution phases on boot and at run-time, respectively.

V. $\mathcal{P}$ARseL Implementation & Verification

A. Implementation Details

1) Implementation of RP: Once seL4 passes control to RP, RP initializes user space by creating necessary boot-time objects, such as CSpace, VSpace, and a memory allocator. Then, it initializes PSMT by creating a new thread control block object, a memory frame for its IPC buffer, a new page table, and a new endpoint object. Next, RP maps the page table and IPC buffer frame into the VSpace and configures a badge for the endpoint and thread control block priority. RP then sets up the thread-local storage (for its own storage area) and spawns PSMT. Finally, it waits for PSMT to complete and send ACK.

2) Implementation of PSMT: Once spawned, PSMT creates SP by assigning it a new set of virtual memory, configuring it with two endpoints, and associating a unique badge for each endpoint. SP uses one endpoint for IPC with SP and the other for UP-s. PSMT similarly creates IUP, establishes an IPC between itself and IUP, and spawns IUP. Then, PSMT waits for a request from IUP.

A request includes all the specifications of UP to be spawned, such as P_{ID}, binary location, and capabilities to system resources. Once receiving the request, PSMT first ensures that the requested capabilities do not contain the write capability to UP's binary and then initializes UP accordingly. Next, PSMT computes its measurement, using a hash algorithm (e.g., SHA2-256 [37]) in HACL*, and stores it in mmap in order. PSMT uses a counter to make sure the number of spawned processes does not exceed the size of mmap and assigns a badge based on the counter to make it unique per UP endpoint. Finally, PSMT spawns UP and waits for the next request. Once receiving the "Done" signal from IUP, PSMT sends the entire mmap to SP via IPC, waits for IUP to finish its tasks (if any), and sends an ACK to RP.

3) Implementation of IUP: In $\mathcal{P}$ARseL, all the user process information is consolidated into a configuration file at compile-time. IUP first parses this file and loads its information to a local object. Then, for each UP, IUP sends a spawn request to PSMT with its P_{ID}, and waits for an acknowledgment. After all the UP-s are spawned, IUP sends the "Done" signal to PSMT and finishes its remaining tasks (if any), before terminating itself. Note that if IUP contains no tasks other than requesting to spawn UP-s, then PSMT can directly read the configuration file and spawn UP-s, instead of having a separated IUP.

4) Implementation of SP: SP has two roles: (1) collecting all the UP-s measurements from PSMT at boot-time, and (2) repeatedly processing $\mathcal{R}$A requests at runtime. Once SP is spawned by PSMT during boot-time, SP uses seL4 system calls to receive the entire mmap via IPC in the following way:

1) Using seL4_Recv(), SP listens for measurement message (P_{ID}, m) from PSMT's badge.
2) SP uses seL4_GetMR() to unmarshal the message and copies (P_{ID}, m) to mmap.
3) Using seL4_Reply(), SP sends '0' (as a ACK).

This process is repeated until all the measurements are received from PSMT. In the following section, we describe the verified implementation of SP's runtime phase.

B. Formally Verification of $\mathcal{P}$ARseL Runtime Implementation

We describe the implementation of runtime $\mathcal{P}$ARseL TCB in Low^*, with verified properties, and how to convert it to C code, preserving the verified properties, using $KaRaMeL$.

1) Verifying Properties: Recall that SP runs the infinite loop of (*Listen, Request, Sign, Response*) phases (see Section IV-D). To verify SP, we prove the following invariant properties for this infinite loop: **functional correctness, memory safety**, and **secret independence**.

Functional correctness ensures that each loop iteration performs all the functionalities as intended. In this context, it means each iteration of SP correctly computes the signature according to Equation (1) for the given input and returns the computed result without modifying SP internal states. *Memory safety* and *secret independence* guarantee that no additional information beyond the signature result is leaked from SP. This applies to both memory-based leakages as well as timing side channels. In Section VI, we show that these three properties are sufficient to provide secure $\mathcal{R}$A in $\mathcal{P}$ARseL.

2) Runtime SP Implementation in Low^* and C: To prove these properties, we first specify all seL4 APIs used by SP in Low^*. Then, we implement the Low^* code for all SP execution phases and integrate it with the Low^*-specified seL4 APIs and HACL* verified cryptographic functions. Next, we formally verify the combined implementation via Low^* memory model, intermediate assertions,

and post-condition of the SP execution. Finally, we convert the final *Low** code to C using the verified *KaRaMeL* compiler.

[Specifying seL4 APIs in *Low**] While SP is implemented in *Low**, the functional correctness of seL4 implementation (including system calls) is verified with a different formal specification language called Isabelle/HOL [38]. Hence, we represent them as axioms, using the construct '`assume val`' in F^*. F^* type checker accepts the given assumption without attempting to verify it, and these axioms are converted to '`extern`' in the generated C code. We specify the input/output of each seL4 system call with required type definitions.

For example, Fig. 5 shows in order, the original C code for a system call, `seL4_GetMR` from seL4 APIs, corresponding *Low** implementation as an axiom, and the generated C code using *KaRaMeL*. `seL4_GetMR` has an integer input `i` and simply outputs the `i`-th element of `msg` array in `seL4_IPCBuffer` with type `seL4_Word`. Including the new type `seL4_Word` for `uint64`[3], all the definitions or structs in seL4 (lines 1-12 of original C code) are properly converted into *Low** (lines 1-17 of *Low** code). Note that since there is no concept of the *global variable* in functional programming, all global variables or structs used in SP are represented in `state` type (lines 5-7 of *Low** code), initialized in `st_var` (lines 8-15) and defined in function `st` (lines 16-17). Once the *Low** axiom is compiled with *KaRaMeL*, generated C code only contains one line of declaration (line 12 of generated C code) without implementation. The rest of seL4 system calls used in SP, `seL4_Recv`, `seL4_Reply`, and `seL4_SetMR`, are similarly written as axioms.

[Writing SP in *Low**, combining HACL* library] The *Sign* phase is implemented using cryptographic operations in HACL* which is also implemented in *Low** and formally verified according to their specification. Thus, three HACL* functions for concatenation, hash, and sign, are integrated into one signing function for Equation (1). We use HMAC [25] for the symmetric signing algorithm with SHA2-256 [37] hash function and EdDSA [39] for the asymmetric one. Runtime SP with the four execution phases is implemented by combining this signing function and the seL4 axioms.

First, to receive/send a message through the IPC buffer or store intermediate computation results, we need some local C arrays in *Low**. For representing C arrays, *Low** provides the `Buffer` module. In *Low**, a buffer is a reference to a sequence of memory with a starting `index` and a `length`. We use `alloca` (or `create` from HACL*) for stack allocation, and retrieve/update the buffer contents using `index`/`upd` with the proper indices.

Then, since the *Sign* phase is in between two seL4 system calls for *Request* and *Response* phases, proper type conversions are required. Specifically, seL4 system calls use the type `seL4_Word` and HACL* functions require the `uint8` input type. To safely convert back and forth between `uint8` buffer and `seL4_Word` buffer (with big-endian), we use `uints_to_bytes_be` and `uints_from_bytes_be` of the `Lib.ByteBuffer` module in HACL*.

[Formal Verification] To verify the *functional correctness* of runtime SP, we first specify necessary pre-/post-conditions for each seL4 axiom. For example, the *Low** code in Fig. 5 shows that the function `seL4_GetMR` correctly returns with the `i`-th element of `msg` array in `seL4_IPCBuffer` (line 22). Also, some properties are needed to be specified to verify that SP internal states are not modified. In Fig. 5, the post-condition `B.(modifies loc_none h0 h1)` indicates that

[3]It is defined either `uint32` or `uint64` depending on the underlying architecture, and the example code is shown with `uint64 seL4_Word`.

```
1  #define _seL4_int64_type    long long int
2  typedef unsigned _seL4_int64_type   seL4_Uint64;
3  typedef seL4_Uint64 seL4_Word;
4  typedef struct seL4_IPCBuffer_ {
5      seL4_Word msg[seL4_MsgMaxLength]; // seL4_MsgMaxLength = 120
6  } seL4_IPCBuffer __attribute__((__aligned__(sizeof(struct
         seL4_IPCBuffer_))));
7  extern __thread seL4_IPCBuffer *__sel4_ipc_buffer;
8  __thread __attribute__((weak)) seL4_IPCBuffer *__sel4_ipc_buffer;
9  LIBSEL4_INLINE_FUNC seL4_IPCBuffer *seL4_GetIPCBuffer(void)
10 {
11     return __sel4_ipc_buffer;
12 }
13 LIBSEL4_INLINE_FUNC seL4_Word seL4_GetMR(int i)
14 {
15    return seL4_GetIPCBuffer()->msg[i];
16 }
```

```
1  type seL4_Word = uint64
2  noeq type seL4_IPCBuffer = {
3    msg : mbuffer seL4_Word 120;
4  }
5  noeq type state = {
6    ipc_buffer: ipc:seL4_IPCBuffer;
7  }
8  let st_var: state =
9    let msg = B.gcmalloc HS.root (I.u64 0) 120ul in
10   let ipc_buffer = {
11     msg = msg;
12   } in
13   {
14     ipc_buffer = ipc_buffer;
15   }
16 val st (_:unit):state
17 let st _ = st_var
18 assume val seL4_GetMR
19 ( i : size_t )
20 : Stack seL4_Word
21 ( requires fun h0 -> (size_v i < 120) /\ (size_v i >= 0) )
22 ( ensures fun h0 a h1 -> B.(modifies loc_none h0 h1) /\ a == B.get h1 (
       st ()).ipc_buffer.msg (v i))
```

```
1  typedef uint64_t seL4_Word;
2  typedef uint64_t *seL4_IPCBuffer;
3  typedef struct state_s
4  {
5    uint64_t *ipc_buffer;
6  } state;
7  state st_var;
8  state st()
9  {
10   return st_var;
11 }
12 extern uint64_t seL4_GetMR(uint32_t i);
```

Fig. 5: Simplified example seL4 API in original seL4 library (top), axiom in F^* (middle), and generated header file in C (bottom)

no locations are modified from `seL4_GetMR` function call (line 22).

Next, we insert an assertion detailed in Fig. 6 after the *Sign* phase to ensure the functional correctness of the signing function, i.e., it correctly computes the signature according to Equation (1).

```
1  // h0 is the initial memory state and h1 is the state right after the
         signing function call, using ST.get ()
2  assert ( B.as_seq h1 sign_result_u8 ==
3              Spec.Ed25519.sign (B.as_seq h0 s.sign_key)
4                  (Spec.Agile.Hash.hash alg
5                      (Lib.Sequence.concat #uint8 #64 #32
6                          (Lib.Sequence.concat #uint8 #32 #32
7                          (B.as_seq h chal) (B.as_seq h pk))
8                      (B.as_seq h measurement_process))
9                  )
10         );
```

Fig. 6: Assertion for Functional Correctness of *Sign*, equation (1)

Finally, we check the invariance of $\mathcal{K}$ and `mmap` throughout the SP execution via intermediate assertions and the post-condition of the runtime SP function. Similar to the assertion above, it compares the $\mathcal{K}$ and `mmap` contents in the memory (`h`) after executing each function call with the ones in the initial memory (`h0`), specified in Fig. 7. This invariance along with the post-conditions of seL4 APIs and the assertion in Fig. 6 implies the functional correctness of runtime SP.

For *memory safety*, we first implement all SP components with `Stack` effect, which prevents any memory leakage due to deallocated heap regions. We also check the "liveness" and "disjointness" of

```
1   assert (B.as_seq h0 s.mmap == B.as_seq h s.mmap);
2   assert (B.as_seq h0 s.sign_key == B.as_seq h s.sign_key);
```

Fig. 7: Assertion for $\mathcal{K}$ and `mmap` invariance

all buffers before they are referenced (via `live` and `disjoint` clauses), which prevents stack-based memory corruption. The former guarantees that a buffer must be properly initialized and not de-allocated (so "live") before it is used, whereas the latter ensures that all buffers used in SP are located in separate memory regions without any overlap. Lastly, we specify a post-condition for every function in SP to ensure that it modifies only the intended memory region. This can be done through the `modifies` clause with the form of `modifies s h0 h1`, which ensures that the memory `h1` after the function call may differ from the initial memory `h0` (before the function call) *at most* regions in `s`, i.e., no regions outside of `s` are modified by the function execution. For example, in Fig. 5, `seL4_GetMR` function ensures not to modify any memory location (with '`loc_none`') in its post-condition (line 22).

Finally, for the *secret independence*, we use the same technique employed by HACL*. We use the secret machine integers for private values (i.e., $\mathcal{K}$), including all intermediate values, and do not use any branch on those secret integers. This ensures that the execution time or the accessing memory addresses are independent of the secret values so that the implementation is timing side-channel resistant.

[Generating `C` code using *KaRaMeL* **]** Finally, we carefully write a build system and generate readable `C` code from our verified *Low** code using *KaRaMeL*. It takes an F^* program, erases all the proofs, and rewrites the program from an expression language to a statement language, performing optimizations. If the resulting code contains only *Low** code with no closures, recursive data types, or implicit allocations, then *KaRaMeL* proceeds with a translation to `C`.

KaRaMeL generates a readable `C` library, preserving names so that one not familiar with F^* can review the generated code before integrating it into a larger codebase. For example, the refinement type (`b: B.buffer uint32 B.length b = n`) in *Low** is compiled to a `C` declaration (`uint32_t b[n]`), while referred to via (`uint32_t *`) as `C` pointer.

C. Secure Boot of seL4 and PARseL TCB

Similar to HYDRA, PARseL relies on a secure boot feature to protect against a physical $\mathcal{A}$dv attempting to re-program seL4 and PARseL TCB when $\mathcal{P}$rv is offline. In HYDRA, this feature works by having a ROM boot-loader validate seL4 authenticity before loading it. Once seL4 is running, it authenticates the user-space TCB by comparing it to a benign hash value, hard-coded within the seL4 binary. Since HYDRA TCB is user-dependent, updating a user application implies a software update not only to the TCB but also to the seL4 binary that stores the TCB referenced hash value, which can be inconvenient in practice. Conversely, PARseL TCB is user-independent, allowing user applications to be updated directly without the need to modify PARseL TCB or seL4 binary.

D. Evaluation

Our source code including verification proofs is available at [23].

1) Evaluation Setup: To demonstrate the practicality of PARseL, we developed our prototype on a commercially available hardware platform: SabreLite [22] – on which seL4 is fully verified [33] including all proofs for functional correctness, integrity, and information flow. SabreLite features an ARM Cortex-A9 CPU core (running at 1 GHz), with RAM of size 1 GB, and a microSD card slot (which we use to boot and load PARseL image). PARseL is implemented on seL4 version 12.0.1 (latest at the time of writing). Besides seL4 IPC kernel APIs, RP uses seL4 Runtime, seL4 Utils, and seL4 Bench user-space libraries (offered by seL4 Foundation) to implement PSMT process spawning procedure.

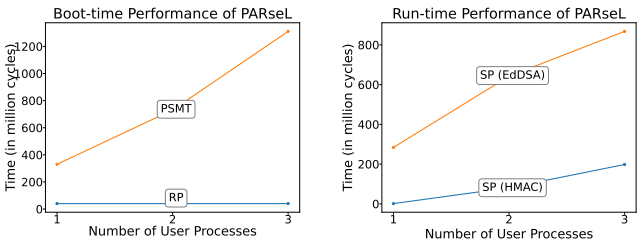


Fig. 8: PARseL Performance while varying the number of spawned user processes (excluding SP)

2) PARseL Performance: The left sub-figure of Fig. 8 shows the boot-time performance of RP and PSMT, and the right one shows the run-time performance of SP (using either HMAC or EdDSA). Reported results are averaged over 50 iterations. The size of each spawned process is ≈ 0.4 MB.

RP takes constant 40 ms (40 million cycles @ 1 GHz), as it initiates the device and spawns PSMT, independent of the number of UP-s spawned. The time taken for PSMT increases linearly to the number of UP-s, as expected because PSMT loads, measures, and spawns each UP sequentially. Spawning each 0.4 MB UP takes ≈ 150 ms. Concretely, when there are 3 UP-s, the boot-time of PARseL is 1.3s.

Using HMAC requires significantly fewer cycles than using EdDSA, due to its relatively expensive operations in the latter. The attestation time for one UP using EdDSA is 282 ms while the one using HMAC is 1.2 ms. As the number of UP-s increase, the time taken for SP also increases. This is due to frequent kernel context switching, as seL4 (fully verified implementation) uses only one core.

3) PARseL TCB size: PARseL TCB contains 3.9K lines of `C` code, including 0.6K lines for RP + PSMT (excluding the seL4 user-space libraries), and 3.3K lines for SP. Out of 3.3K lines of SP, 3.2K lines are verified, including 3K lines from HACL* EdDSA and 0.2K lines from SP run-time attestation function.

VI. PARseL Security Analysis

To argue PARseL security with respect to the adversary model in Section III-B, we start by formulating PARseL security goal.

> **Security Definition:** *Let $\mathcal{B}$ be an arbitrary software binary selected by $\mathcal{V}$rf. In the context of a static root of trust for measurement of user-level processes, an $\mathcal{RA}$ scheme is considered secure if and only if $\mathcal{V}$rf is able to use the $\mathcal{RA}$ scheme to establish a secure channel with program $\mathcal{P}$, where:*
> * *$\mathcal{P}$ is an isolated user-level process running on the correct $\mathcal{P}$rv;*
> * *At boot time, $\mathcal{P}$ was loaded with the $\mathcal{V}$rf-selected binary $\mathcal{B}$;*

Security Argument: Assuming that $\mathcal{V}$rf uses pk, included in σ (recall Equation 1), to establish the secure channel, $\mathcal{A}$dv can attempt to circumvent PARseL security by:

(1) **Loading the Right Software on the Wrong Device.** $\mathcal{A}$dv can load process $P_{\mathcal{A}\text{dv}}$ with the expected binary $\mathcal{B}$ on a different device ($\mathcal{P}\text{rv}_{\mathcal{A}\text{dv}}$), also equipped with an instance of PARseL. Then, $\mathcal{A}$dv forwards $\mathcal{V}$rf's request (intended to the original $\mathcal{P}$rv) to $\mathcal{P}\text{rv}_{\mathcal{A}\text{dv}}$. $\mathcal{P}\text{rv}_{\mathcal{A}\text{dv}}$ inadvertently issues a PARseL attestation response that

matches software $\mathcal{B}$ (loaded on $P_{\mathcal{A}dv}$). However, as the secret key $\mathcal{K}$ is unique to each $\mathcal{P}$rv, $\mathcal{V}$rf would not accept the received σ, thereby refusing to establish the secure channel.

(2) **Loading the Wrong Software on the Right Device.** $\mathcal{A}$dv can load a user-space process on the correct $\mathcal{P}$rv but with an incorrect/malicious binary $\mathcal{B}_{\mathcal{A}dv}$. This can be accomplished with physical access to $\mathcal{P}$rv or by exploiting a vulnerability on a user-space process to perform persistent code injection, re-booting $\mathcal{P}$rv thereafter. In either case, σ would be signed with the expected secret key $\mathcal{K}$. However, mmap would be updated at boot to reflect $\mathcal{B}_{\mathcal{A}dv}$, i.e., the hash result $\mathtt{m}_{\mathtt{UP}_{\mathcal{A}dv}}$. Consequently, $\mathcal{V}$rf would refuse to establish a secure channel with a process on $\mathcal{P}$rv loaded with $\mathcal{B}_{\mathcal{A}dv} \neq \mathcal{B}$.

(3) **Loading the Wrong Software on the Wrong Device.** It follows from both arguments above that this option is infeasible to $\mathcal{A}$dv due to the mismatches on both secret key $\mathcal{K}$ and measurement $\mathtt{m}_{\mathtt{UP}_{\mathcal{A}dv}}$.

Therefore, $\mathcal{P}$ARseL satisfies the security definition above. □

This argument assumes confidentiality of $\mathcal{K}$. In $\mathcal{P}$ARseL, this is supported through formal verification of SP functional correctness, secret independence, and memory safety. It also assumes that each process is appropriately measured at boot. In $\mathcal{P}$ARseL, this is implemented by PSMT when computing mmap. The association of pk with the correct $\mathtt{m}_{\mathtt{UP}}$ is guaranteed by seL4 badge assignments. Finally, the scheme relies on inter-process isolation for SP and any attested process $\mathcal{P}$, once the secure channel is established. The latter is inherited from seL4 provable isolation.

VII. Discussion

Limitations: Only $\mathcal{P}$ARseL runtime TCB is verified. The integrity of $\mathcal{P}$ARseL boot time TCB is ensured via secure boot, while the correct implementation of secure boot/boot TCB are assumed. Furthermore, $\mathcal{P}$ARseL measures processes at boot time. Thus, RP configures a write-xor-execution memory permission to prevent a user process from modifying its own code. By default, although seL4 guarantees strong inter-process isolation, it gives each process full control of its own code/data segments. Due to this write-restriction, $\mathcal{P}$ARseL does not support run-time updates to user-level processes. Currently, benign updates must be done physically and require rebooting the device (in order to measure the updated program on boot). However, we believe that any software update framework compatible with seL4 (e.g. [12]) can be used alongside with $\mathcal{P}$ARseL for remote updates. The only requirement then would be to reboot the device after the update, so that $\mathcal{P}$ARseL re-measures all UP-s including the new updated UP.

(Unexpected) Termination of UP does not cause any issues because no other user process can transfer the signature (from $\mathcal{V}$rf) on behalf of another process to SP. In $\mathcal{V}$rf's view, no response will arrive (in a certain amount of time) so it can deduce that UP or $\mathcal{P}$rv are no longer running. This is similar to any $\mathcal{R}$A protocol.

SP Stack Erasure is obviated in $\mathcal{P}$ARseL because SP is never terminated at run-time and seL4's inter-process isolation guarantees that only SP has access to its own stack.

VIII. Related Work

$\mathcal{R}$A: techniques can be classified into SW-based, HW-based, and hybrid (HW/SW co-design) architectures. Although SW-based methods such as [40], [41], [42], [43] require minimal overall costs, they rely on strong assumptions about precise time-based checksum, which is mostly unsuitable for the IoT ecosystem with the multi-hop network. HW-based methods [17], [44], [45], on the other hand, rely on some additional hardware support for $\mathcal{R}$A, e.g., some dedicated hardware components [44], or extension of existing instruction sets [19], which introduce cost and other barriers, especially for low-end and mid-range devices. Hybrid approach [14], [15], [16], [46] is considered to be more suitable for IoT ecosystems because it aims for minimal hardware changes while keeping the same security levels as HW-based $\mathcal{R}$A. Using the hybrid $\mathcal{R}$A as a building block, many security services have been also suggested, such as proof of execution [4], [5], control-flow and data-flow attestation [6], [7], [8], [47], [9], [10], [11], and secure software updates [12], [48], [13]. Since $\mathcal{P}$ARseL also provides a hybrid $\mathcal{R}$A, it can be also used for such security services. Several recent papers on hybrid $\mathcal{R}$A/$\mathcal{R}$A-based security services [4], [14], [48], [13], [46] provide formal verification of their suggested architectures/implementations. They use model checking with temporal logic to verify their implementations while they use theorem proving to show that their proved properties are sufficient for their security goal(s).

Verfied security applications in F^*: [49] lists papers that apply F^* in security, including HACL* [31]. DICE* [50] is a notable paper related to $\mathcal{P}$ARseL, which proposes a verified implementation of *Device Identifier Composition Engine* (DICE), an industry-standard *measured boot* protocol, for low-cost IoT devices. Similar to $\mathcal{P}$ARseL, it has layered architecture with static components whose implementations are verified over Low^*. The main difference is how to guarantee the $\mathcal{K}$ confidentiality. DICE enforces the access control to the master secret key by locating it in a read-only and latchable memory so that only a hardware reset can disable/restore access to it. The first hardware layer (called DICE engine) only has access to the secret, and it authenticates the next layer (L0) and derives the secret for L0 from its master secret and L0 measurement. This ensures the same derived secret only when L0 firmware is not compromised. Once received control, L0 derives a unique key pair from this secret and the next-layer firmware (L1); this key pair can then be used for L1 attestation and secure key exchange. Although $\mathcal{P}$ARseL assumes a secure boot for correct seL4 deployment, both $\mathcal{P}$ARseL and DICE* present verified implementations for the static root of trust for embedded devices, with different ways of guaranteeing the access control.

Architectures/applications over seL4: After being released in 2009 [51], seL4 has been actively implanted and used in both academia and industries in various domains, including automotive [52], aviation [53], and medical devices [54]. Apart from massive research from the Trustworthy Systems group in UNSW Sydney, many projects such as [1], [55] leverage their architecture atop seL4.

IX. Conclusions

This paper presented $\mathcal{P}$ARseL, a verifiable $\mathcal{R}$A root-of-trust over seL4. We implemented it on SabreLite and demonstrated its overall feasibility and practicality. We also formally verified its runtime component in terms of functional correctness, memory safety, and secret independence, using the Low^* tool-chain. All source code, including verification proofs, is available at [23].

Acknowledgements: We thank ICCAD'23 reviewers for constructive feedback. This work was supported by funding from NSF Awards SATC-1956393, SATC-2245531, and CICI-1840197, NSA Awards H98230-20-1-0345 and H98230-22-1-0308, as well as a subcontract from Peraton Labs.

REFERENCES

[1] K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "HYDRA: hybrid design for remote attestation (using a formally verified microkernel)," in *Wisec*, 2017.

[2] W. Englund, E. Nakashima, and T. Telford, "Colonial pipeline 'ransomware' attack shows cyber vulnerabilities of u.s. energy grid," https://www.washingtonpost.com/business/2021/05/10/colonial-pipeline-gas-oil-markets/, May 2021.

[3] K. Zetter, "Inside the cunning, unprecedented hack of ukraine's power grid," https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/, March 2016.

[4] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "APEX: A verified architecture for proofs of execution on remote devices under full software compromise," in *USENIX'20*.

[5] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "Litehax: lightweight hardware-assisted attestation of program execution," ser. ICCAD '18. IEEE.

[6] T. Abera et al., "C-flat: Control-flow attestation for embedded systems software," in *CCS '16*, 2016.

[7] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi, "Lo-fat: Low-overhead control flow attestation in hardware," in *DAC'17*.

[8] I. D. O. Nunes, S. Jakkamsetti, and G. Tsudik, "Dialed: Data integrity attestation for low-end embedded devices."

[9] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A.-R. Sadeghi, "Atrium: Runtime attestation resilient under memory attacks," in *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press, 2017, pp. 384–391.

[10] Z. Sun, B. Feng, L. Lu, and S. Jha, "Oat: Attesting operation integrity of embedded devices," in *2020 IEEE Symposium on Security and Privacy*.

[11] M. Geden and K. Rasmussen, "Hardware-assisted remote runtime attestation for critical embedded systems," in *2019 17th International Conference on Privacy, Security and Trust (PST)*. IEEE.

[12] N. Asokan, T. Nyman, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, "ASSURED: Architecture for secure software update of realistic embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[13] I. De Oliveira Nunes, S. Jakkamsetti, Y. Kim, and G. Tsudik, "Casu: Compromise avoidance via secure update for low-end embedded systems," ser. ICCAD '22.

[14] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified hardware/software co-design for remote attestation," in *USENIX Security*, 2019.

[15] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *EuroSys*, 2014.

[16] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "Tytan: Tiny trust anchor for tiny devices," in *DAC*, 2015.

[17] J. Noorman, J. V. Bulck, J. T. Mühlberg *et al.*, "Sancus 2.0: A low-cost security architecture for iot devices," *ACM Trans. Priv. Secur.*, 2017.

[18] M. Grisafi, M. Ammar, M. Roveri, and B. Crispo, "PISTIS: Trusted computing architecture for low-end embedded systems," in *31st USENIX Security Symposium*, Aug. 2022, pp. 3843–3860.

[19] Intel, "Intel Software Guard Extensions (Intel SGX)," https://software.intel.com/en-us/sgx.

[20] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium*, 2016.

[21] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "sel4: Formal verification of an os kernel," in *SIGOPS*. ACM, 2009.

[22] B. Devices. Boundary devices bd-sl-i.mx6. https://boundarydevices.com/product/bd-sl-i-mx6/.

[23] Anonymous, "Parsel open-source code," https://anonymous.4open.science/r/parsel-submission-1EC5/.

[24] T. A. L. Sewell, M. O. Myreen, and G. Klein, "Translation validation for a verified os kernel," *ACM SIGPLAN Notices*, 2013.

[25] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," in *Advances in Cryptology — CRYPTO '96*.

[26] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin, "Dependent types and multi-monadic effects in f*," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[27] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, 1969.

[28] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy, "Verified low-level programming embedded in f*," *Proc. ACM Program. Lang.*, aug 2017.

[29] "The karamel compiler (2017)," https://github.com/FStarLang/karamel.

[30] S. Blazy and X. Leroy, "Mechanized Semantics for the Clight Subset of the C Language," *Journal of Automated Reasoning*, 2009.

[31] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *CCS*, 2017.

[32] SiFive. Hifive unleashed specifications. https://www.sifive.com/boards/hifive-unleashed.

[33] seL4 Team, "sel4 supported platforms with verification status," https://docs.sel4.systems/Hardware/.

[34] T. Abera, N. Asokan, L. Davi, F. Koushanfar, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "Invited: Things, trouble, trust: on building trust in IoT systems," in *DAC'16*.

[35] S. Ravi, A. Raghunathan, and S. Chakradhar, "Tamper resistance mechanisms for secure embedded systems," in *VLSI Design*, 2004.

[36] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and minimal architecture for (establishing dynamic) root of trust," in *NDSS*, 2012.

[37] S. H. Standard, "Fips pub 180-2," 2002.

[38] U. of Cambridge and T. Munich, "Isabelle," https://isabelle.in.tum.de/.

[39] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," in *Journal of Cryptographic Engineering*, 2011.

[40] R. Kennell and L. H. Jamieson, "Establishing the genuinity of remote computer systems," in *USENIX Security Symposium*, 2003.

[41] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "SWATT: Software-based attestation for embedded devices," in *IEEE Symposium on Research in Security and Privacy (S&P)*. Oakland, California, USA: IEEE, 2004, pp. 272–282.

[42] Y. Li, J. M. McCune, and A. Perrig, "Viper: Verifying the integrity of peripherals' firmware," in *CCS*. ACM, 2011.

[43] V. D. Gligor and S. L. M. Woo, "Establishing software root of trust unconditionally." in *NDSS*, 2019.

[44] Trusted Computing Group., "Trusted platform module (tpm)," 2017. [Online]. Available: http://www.trustedcomputinggroup.org/work-groups/trusted-platform-module/

[45] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth, "New results for timing-based attestation," in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 239–253.

[46] I. D. O. Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "On the TOCTOU problem in remote attestation," in *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds., 2021.

[47] I. De Oliveria Nunes, S. Jakkamsetti, and G. Tsudik, "Tiny-CFA: Minimalistic control-flow attestation using verified proofs of execution," in *Design, Automation and Test in Europe Conference (DATE)*, 2021.

[48] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "Pure: Using verified remote attestation to obtain proofs of update, reset and erasure in low-end embedded systems."

[49] "Project everest bibiliography," https://project-everest.github.io/papers/.

[50] Z.-Z. Tao, A. Rastogi, N. Gupta, K. Vaswani, and A. V. Thakur, "Dice*: A formally verified implementation of dice measured boot," in *USENIX Security Symposium*, 2021.

[51] G. Klein, K. Elphinstone, G. Heiser *et al.*, "seL4: Formal verification of an OS kernel," in *ACM SIGOPS*, 2009.

[52] "Darpa high assurance cyber military systems (hacms) heavy equipment transporter," https://www.youtube.com/watch?v=6cllzGGxRfE, 2017.

[53] D. Cofer, A. Gacek, J. Backes, M. W. Whalen, L. Pike, A. Foltzer, M. Podhradsky, G. Klein, I. Kuz, J. Andronick, G. Heiser, and D. Stuart, "A formal approach to constructing secure air vehicle software," *Computer*, 2018.

[54] M. Russell, "Enable the security potential and versatility of sel4 in medical device development," https://dornerworks.com/blog/sel4-medical-products/, 2020.

[55] A. Petz, G. Jurgensen, and P. Alexander, "Design and formal verification of a copland-based attestation protocol," in *ACM-IEEE International Conference on Formal Methods and Models for System Design*, 2021.