HYDRA: HYbrid Design for Remote Attestation (Using a Formally Verified Microkernel)

***Abstract*—Remote Attestation (RA) is the process involving a trusted entity (*verifier*) that securely measures internal state of a remote untrusted hardware platform (*prover*). RA can be used to establish a static or dynamic root of trust in embedded and cyber-physical systems. It can also be used as a building block for other security services and primitives, such as software updates/-patches, verifiable deletion and memory resetting. There are three major classes of RA designs: *hardware-based*, *software-based*, and *hybrid*, each with its own set of benefits and drawbacks.**

This paper presents the first hybrid RA design – called HYDRA – that builds upon formally verified software components that ensure memory isolation and protection guarantees, as well as enforce access control to memory and other resources. HYDRA obtains these properties by using the formally verified seL4 microkernel. (Until now, this was only attainable with purely hardware-based designs.) Using seL4 requires fewer hardware modifications to the underlying microprocessor. Building upon a formally verified software component increases confidence in security of the overall design of HYDRA and its implementation. We instantiate HYDRA on two commodity hardware platforms and assess the performance and overhead of performing RA on such platforms via experimental results; we show that HYDRA can attest 10MB of memory in less than 500msec when using Speck-based message authentication code (MAC) to compute a cryptographic checksum over attested memory.

I. Introduction

In recent years, embedded systems (ES), cyber-physical systems (CPS) and internet-of-things (IoT) devices, have percolated into many aspects of people's daily lives, such as: households, offices, buildings, factories and vehicles. This trend of "smart-ification" of everything that was previously analog (or at least not connected) brings many obvious benefits. However, it also expands the attack surface and makes these newly computerized gadgets into natural and attractive targets.

Remote Attestation (RA) is the process whereby a trusted entity called "verifier" securely probes internal state of a remote and untrusted hardware platform, called "prover." RA can be used to establish a static or dynamic root of trust in ES, CPS and IoT devices. Also, RA can be used as a foundation for constructing more specialized security services, e.g., software updates, verifiable deletion and memory resetting. There are three main classes of RA designs: *hardware-based*, *software-based*, and *hybrid* (blending hardware and software). Each class its own advantages and shortcomings or limitations.

This paper introduces *the first* hybrid RA design – called HYDRA – based upon formally verified components to provide memory isolation and protection guarantees. Our main rationale is that designing RA techniques based upon such components increases confidence in security of such designs and their implementations. Of course, ideally, one would formally prove security of the entirety of an RA system, as opposed to demonstrating it separately for each component and then proving that its composition is secure. However, we believe that this is not yet possible given the state-of-the-art in (automated) formal verification and synthesis of hardware and software.

One recent prominent example illustrating the difficulty of correctly designing and implementing security primitives (especially, those blending software and hardware) is the recent TrustZone kernel vulnerability and exploit reported in CVE-2015-6639 [25]. ARM TrustZone [4] is a popular System-on-Chip (SoC) and a CPU system-wide approach to security. According to ARM, it is adopted in billions of processors on various platforms, including: smartphones, tablets, personal computers, wearables and enterprise systems. TrustZone is a hardware-based security architecture and can be built into SoCs by semiconductor designers and manufacturers to provide secure end-points and establish root of trust. TrustZone can be integrated into most ARM-based systems, ranging from small micro-controller units (MCU) to high-end application processors. The CVE-2015-6639 vulnerability was found in the TrustZone-based Qualcomm Secure Execution Environment (QSEE). It enables privilege escalation and allows execution of code in the TrustZone kernel which can then be used to achieve undesired outcomes and expose keying material. This vulnerability was used to effectively break Android's Full Disk Encryption (FDE) scheme by recovering the master keys [1].

We use the above example to demonstrate the difficulty of getting both the design and the implementation right, as well as a motivation to use formally verified building blocks, which (we hope) will yield more secure RA techniques. To this end, our RA design uses the formally verified seL4 microkernel to obtain memory isolation and access control. Such features have been previously attained with hardware in designs such as SMART [11] and TrustLite [18]. Using seL4 requires fewer hardware modifications to the underlying microprocessor and provides an automated formal proof of isolation guarantees of the actual implementation of the microkernel. *To the best of our knowledge, this is the first attempt to design and implement RA using a formally verified microkernel.*

The main goal and motivation of this paper is to investigate a previously unexplored segment of the design space in hybrid RA schemes, specifically, techniques that incorporate formally verified and proven[1] components, such as the seL4 microker-

[1]Using automated proof and synthesis methods.

nel. Beyond using seL4 in our design, our implementation is also based on the formally verified executable of seL4; that executable is guaranteed to adhere to the formally verified and proven design.

Another important goal, motivation and feature of our design is the expanded scope of efficient hybrid RA techniques. While applicability of prominent prior results (particularly, SMART [11] and TrustLite [18]) is limited to very simple single-process low-end devices, we target more capable devices that can run multiple processes and threads.

We believe that this paper represents an important and necessary step towards building efficient hybrid RA techniques upon solid and verified foundations. Admittedly, we do not verify our entire design and prove its security using automated formal methods. However, we achieve the next best thing by taking advantage of already-verified components and carefully arguing security of the overall design, considering recent results on systematic analysis of exact features required for hybrid RA [12].

To achieve the goals above this paper makes two main contributions: (1) design of HYDRA – the first hybrid RA technique based on the formally verified seL4 microkernel which provides memory isolation and access control guarantees, (2) implementations of HYDRA on two commercially available development boards (Sabre Lite and ODROID-XU4) and their extensive analysis via experiments to demonstrate practicality of the proposed design. We show that HYDRA can attest 10MB of memory in less than 500 milliseconds when using Speck as the underlying block-cipher to compute a cryptographic checksum over the attested memory.

Organization: Section II overviews related work, followed by Section III which presents our goals and assumptions. The design of HYDRA is presented in Section IV and its security analysis appears in Section VI. Implementation issues and performance assessment are discussed in Section V and VII; Section VIII concludes the paper.

II. Related Work

Prior work in remote attestation (RA) can be divided into three approaches: software-based, hardware-based, and hybrid.

a) Hardware-Based Remote Attestation: The hardware-based approach for RA typically relies on the security provided by a Trusted Platform Module (TPM) [34]. A TPM is a secure co-processor designed to protect cryptographic keys, and utilize them to encrypt or digitally sign data. A TPM can also produce a summary (e.g., hash) of hardware and software configurations in the system. A typical TPM also contains Platform Configuration Registers (PCR) that can be used as a secure storage of such a configuration summary. The values in PCRs can then be used as an evidence of attestation by accumulating an unforgeable chain of values of the system's state since the last reset. A TPM eventually signs these values with an attestation key along with a random challenge, provided by a verifier, and submits the computed result to the verifier. Gasmi et al. [14] presents how to link this evidence to secure channel end-points.

In 2015, Intel introduced a set of new instructions, Software Guard Extensions (SGX), mainly for providing a hardware-enforced isolated execution environment (*enclave*) for specific software. An enclave contains only private data and code executing that data [10], which allows an isolation of that data inside an enclave from other processes on the same platform. Thus, RA for software inside an enclave can be performed locally without interference from other processes. Similar to TPM, attestation evidence in SGX can be a hash of code, signed by the CPU.

b) Software-Based Remote Attestation: Despite resisting all but physical attacks, a hardware-based RA approach is not suitable for embedded devices for various reasons. First, such a trust anchor generally consists of several hardware components and thus is considered to be too expensive and unpractical for low-end embedded devices [2]. Moreover, additional firmware (e.g., drivers and/or libraries) is required to operate a TPM, which can increase the complexity of the whole system. Therefore, many software-only RA approaches have been proposed, specifically for embedded devices. Pioneer [30] is among the first to study RA without relying on any secure co-processor or CPU-architecture extensions. The main idea behind Pioneer is to create a special checksum function with run-time side-effects (e.g., status registers) for attestation. Any malicious emulation of said checksum function can be detected through additional timing overhead incurred from the absence of those side-effects. [29] adapted this time based software checksum approach to embedded devices. Security of this approach became questionable after several attacks on such schemes (i.e., [9]) were demonstrated.

c) Hybrid Remote Attestation: The main limitation of the software-based RA approach is that it makes strong assumptions about adversarial capabilities, which may be unsatisfied in practical networked settings [2]. Thus, several hybrid software-hardware co-designs have been proposed to overcome this limitation. The primary goal of such hybrid approach is to minimize the amount of secure hardware while maintaining strong resilience against all adversarial attacks.

SMART [11] presents a hybrid approach for RA with minimal hardware modifications to existing MCUs. In addition to having uninterruptable attestation code and attestation keys resided in ROM, this architecture utilizes a hard-wired memory protection unit (MPU) to restrict access to secret keys to only SMART code. The attestation is performed inside ROM-resident attestation code by computing a cryptographic checksum over a memory region and returning the value to the verifier. Prototypes on two low-end devices are implemented to show the viability of SMART in practice. Moreover, Lugou et al. [20] presents a conceptual guideline for formal verification of SMART.

TrustLite [18], a security architecture for low-end embedded devices, provides isolation for specific software modules (*trustlets*). TrustLite can be seen as a generalized version of SMART. TrustLite's *Execution-Aware Memory Protection*

Unit (EA-MPU), similar to SMART's MPU, can be utilized to control access to specific hardware components. One of the distinctions of both MPUs is that EA-MPU can be programmed at run-time by trustlets whereas SMART's MPU is always static. Additionally, TrustLite supports an interrupt handling in trustlets. The concept of EA-MPU has been also recently adapted in the TyTAN system [7].

A common drawback of the hybrid approach is that it is not immune to physically intrusive adversaries. With physical control over a device, adversaries can extract secret keys and thus impersonate and/or clone that device. Thus, Physically Unclonable Functions (PUF) have been explored to mitigate this kind of attack. A PUF is a physical structure that can produce a challenge-response output that is unique to each individual device by exploiting unpredictable side-effects introduced during manufacturing. Secret keys in the hybrid design can be generated by PUFs at run-time. Since an evidence of attestation is bound to each hardware device, PUF-based RA can detect hardware attacks due to tamper-evidence of PUFs [5]. [28] and [19] presents a PUF-based design for RA.

In addition to the above work designing RA schemes, the work in [12] provides a systematic treatment of RA by presenting a precise definition of the desired service and proceeding to its systematic deconstruction into necessary and sufficient (security) properties. These properties are then mapped into a minimal collection of hardware and software components that results in secure RA. We build on the work of [12] and utilize these properties and components (which are described in Section III) and show how to instantiate them in new ways to develop our new hybrid RA design, HYDRA.

III. Goals and Assumptions

This section presents a high-level overview of HYDRA and the rationale behind its design, discusses common remote attestation (RA) security objective and features, and outlines the considered adversary model.

A. Design Rationale

As mentioned above, our main objective is to explore a new segment of the design space for RA schemes. Our hybrid RA design – HYDRA – requires very little in terms of secure hardware and builds upon the formally verified seL4 microkernel. As shown in Section V, hardware support needed by HYDRA is readily available on commercial off-the-shelf development boards and processors, e.g., Sabre Lite and ODROID-XU4 boards. The rationale behind our design is that seL4 offers certain guarantees (mainly process isolation and access control to memory and resources) that provide RA features that were previously feasible only using hardware components.[2] In particular, what was earlier attained using additional MCU controls and Read-Only Memory (ROM) in the SMART [11] and TrustLite [18] architectures can now be instantiated using capability controls in seL4.

[2]Indeed, previous hybrid designs pushed many RA requirements into hardware.

To motivate and justify the design of HYDRA, we start from the work by Francillon, et al. [12]. It provides a systematic treatment of RA by developing a semi-formal definition of RA as a distinct security service, and systematically deconstructing it into necessary and sufficient security objective, from which specific properties are derived. These properties are then mapped into a collection of hardware and software components that results in an overall secure RA design. Below, we summarize the security objective in RA and its derived security properties. In Sections IV and V we show how the security objective and each property are satisfied in HYDRA design and instantiated in two concrete prototypes based on the Sabre Lite and ODROID-XU4 boards.

B. Hybrid RA Security Objective and Derived Properties

According to [12], the RA security objective is to allow a (remote) prover ($\mathcal{PRV}$) to create an unforgeable authentication token, that convinces a verifier ($\mathcal{VRF}$) that the former is in some well-defined (expected) state. Whereas, if $\mathcal{PRV}$ has been compromised and its state has been modified, the authentication token must reflect this. [12] also describes a combination of platform properties that achieve aforementioned security objective. The goal of the analysis in [12] is to obtain a set of properties that are both necessary and sufficient for secure RA. The security objective is examined and the properties needed to attain it are identified. The conclusion is that the following properties collectively represent the minimal and necessary set to achieve secure RA on any platform.

- *Exclusive Access to $\mathcal{K}$:* the attestation process (P_{Attest}) must be the only process with access to the attestation key ($\mathcal{K}$). This is the most difficult property to impose on embedded (especially, low-end and mid-range) devices. As argued in [12], there is no way to achieve this property without some hardware support on low-end devices. If the underlying processor supports multiple privilege modes and a full-blown separation of memory for each process, one could use a privileged process to handle all computations that involve the key. However, low-end and medium-end processors generally do not offer such "luxury" features.
- *No Leaks:* no information related to (or derived from) $\mathcal{K}$ must be accessible after the execution of P_{Attest} completes. To achieve this, all intermediate values that depend on the key – except the final attestation token to be returned to $\mathcal{VRF}$ – must be securely erased, when they are no longer needed. This is again applicable to very low-end devices, with none or minimal Operating System (OS) support and assuming that memory is shared between processes. However, if the underlying hardware and/or software (i.e., OS) guarantees that each process' memory is inaccessible by any other process, then this property is trivially satisfied.
- *Immutability:* To ensure that the attestation executable (Attest) cannot be modified, SMART [11] and the design in [12] place it in ROM, which is available on most, even low-end, platforms. ROM is certainly one relatively

inexpensive way to enforce Attest's code immutability. However, owing to seL4's features, Attest in HYDRA is resident in, and executed from, RAM.

- *Uninterruptability:* Execution of Attest must be uninterruptible. This is necessary to ensure that malware does not obtain the key (or some function thereof) by interrupting Attest while any key-related values remain in registers or other accessible locations. As discussed later, HYDRA achieves this property by assuring that it runs with the highest possible priority under seL4.
- *Controlled Invocation (aka Atomicity):* Attest must only be invocable from its very first instruction and must exit only at one of its legitimate last (exit) instruction. This is mainly motivated by the need to prevent code-reuse attacks.

Note that [12] has one additional property of *Secure Reset* when someone attempts to start the attestation executable (Attest) from the middle. We argue that a complete secure reset is not needed if controlled invocation is actually enforced. All needed in that case is to raise an exception as long as the memory space of Attest is protected and integrity of the executable is guaranteed.

C. Adversarial Model & Other Assumptions

Based on the recent taxonomy in [2], adversaries in the RA context can be categorized as follows:

- *Remote Adversaries:* exploit vulnerabilities in a prover's software to inject (over the network) malware onto it.
- *Local Adversaries:* locate sufficiently near in order to eavesdrop on, and manipulate, the prover's communication channel(s).
- *Physical Adversaries:* have full (local) physical access to the prover and its hardware and can perform physical attacks, e.g., side-channel attacks to obtain keys, physically extract memory values, and modify the states of various hardware components.

Similar to prior hybrid RA designs, HYDRA aims to protect against remote and local adversaries, while physical adversaries are out-of-scope. However, we note that, at least in a single-prover setting[3], protection against physical attacks can be attained by encasing the CPU in tamper-resistant coating and employing standard techniques to prevent side-channel key leakage. These include: anomaly detection, internal power regulators and additional metal layers or meshes for tamper detection.

We consider $\mathcal{PRV}$ to be a (possibly) unattended remote hardware platform running multiple processes on top of the seL4 microkernel. Once $\mathcal{PRV}$ boots up and runs in steady state, the adversary ($\mathcal{ADV}$) might exercise complete control over all application software (including code and data) before and after execution of P_{Attest}. In particular, $\mathcal{ADV}$ can modify any code executing as an application on the $\mathcal{PRV}$ and learn any secret belonging to application software (excluding, of course, Attest) running on $\mathcal{PRV}$. Furthermore, $\mathcal{ADV}$ has complete control over the communication channel and can use multiple colluding malicious devices in order to try to pass or subvert attestation. $\mathcal{ADV}$ can also engage in asymmetric denial-of-service (DoS) attacks by causing $\mathcal{PRV}$ to gratuitously perform attestation [8].

As mentioned above, physical attacks are out of scope in HYDRA: $\mathcal{ADV}$ cannot violate the read-only nature of ROM, induce hardware faults or retrieve a stored $\mathcal{K}$ using external side channels. Similarly, $\mathcal{ADV}$ has no means of interrupting execution of seL4 or the P_{Attest} code when invoked (how this is ensured will be demonstrated later).

Finally, recall that $\mathcal{PRV}$ and $\mathcal{VRF}$ must share at least one secret (attestation) key $\mathcal{K}$. This key can be pre-loaded onto $\mathcal{PRV}$ at production or installation time; it is stored as part of P_{Attest} binaries. We do not address the details of this procedure.

IV. HYDRA Design

We first overview the seL4 microkernel and discuss how it can be used in HYDRA to realize previously identified RA properties. We then describe the sequence of operations in HYDRA.

A. seL4 Overview

seL4 is a member of the L4 microkernel family, specifically designed for high-assurance applications by providing isolation and memory protection between different processes. These properties are mathematically guaranteed by a full-code level functional correctness proof, using automated tools. A further correctness proof of the C code translation is presented in [32], thus extending functional correctness properties to the binary level without needing a trusted compiler. Therefore, behavior of the seL4 binary strictly adheres to, and is fully captured by, the abstract specifications.

Similar to other operating systems, seL4 divides the virtual memory into two separated address spaces: *kernel-space* and *user-space*. The kernel-space is reserved for the execution of the seL4 microkernel while the application software is run in user-space. By design, and adhering to the nature of microkernels, the seL4 microkernel provides minimal functionalities to user-space applications: thread, inter-process communication (IPC), virtual memory, capability-based access control and interrupt control. The seL4 microkernel leaves the implementations of other traditional operating system functions – such as device drivers and file systems – to user-space.

Figure 1 (borrowed from [31]) shows an example of seL4 instantiation with two threads – sender A and receiver B – that communicate via an *EndPoint* EP. Each thread has a *Thread Control Block* (TCB) that stores its context, including: stack pointer, program counter, register values, as well as pointers to *Virtual-address Space* (VSpace) and *Capability Space* (CSpace). VSpace represents available memory regions that the seL4 microkernel allocated to each thread. The root of VSpace represents a *Page Directory* (PD), which contains *Page Table* (PT) objects. *Frame* object representing a region

[3] See [16] for recent results on physical attack resilience in groups or swarms of provers.

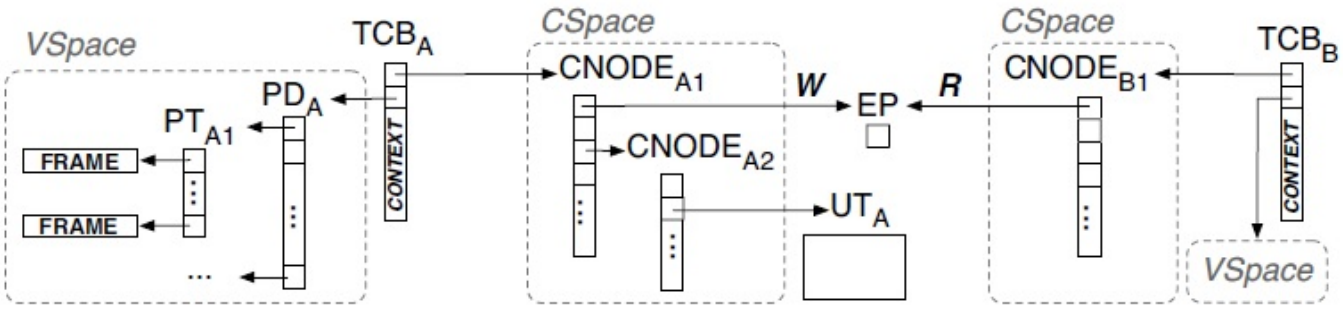


Fig. 1. Sample seL4 instantiation from [31].

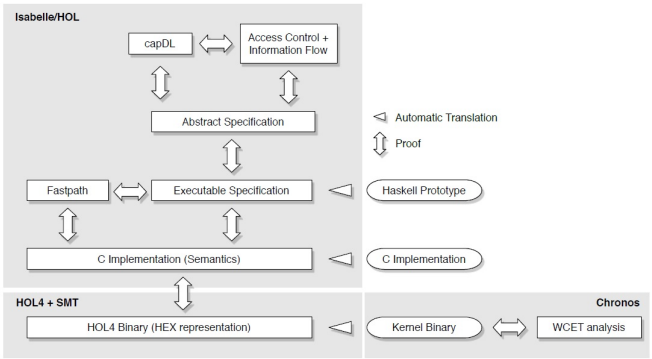


Fig. 2. Proof Layers in seL4 (adapted from [17]).

of physical memory resides in a PT. Each thread also has its own kernel managed CSpace used to store a *Capability Node* (CNode) and *capabilities*. CNode is a table of slots, where each slot represents either a capability or another CNode.

A capability is an unforgeable token representing an access control authorization of each kernel object or component. A thread cannot directly access or modify a capability since CSpace is managed by, and stored inside, the kernel. Instead, a thread can invoke an operation on a kernel object by providing a pointer to a capability that has sufficient authority for that object to the kernel. For example, sender A in Figure 1 needs a write capability of EP for sending a message, while receiver B needs a read capability to EP to receive a message. Besides read and write, *grant* is another access right in seL4, available only for an endpoint object. Given possession of a grant capability for an endpoint, any capability from the possessor can be transferred or issued across that endpoint. For instance, if A in Figure 1 has grant access to EP, it can transfer one of its capabilities, say a frame, to B via EP. Also, capabilities can be statically issued during a thread's initialization by the *initial process*.

The initial process is the first executable user-space process loaded into working memory (i.e., RAM) after the seL4 microkernel is loaded. This special process then forks all other processes. Section IV-D describes the role, the details and the capabilities of the initial process in HYDRA design.

seL4's main "claim to fame" is in being the first formally verified general-purpose operating system. Formal verification of the seL4 microkernel is performed by interactive, machine-assisted and machine-checked proof using a theorem prover Isabelle/HOL. Overall functional correctness is obtained through a *refinement* proof technique, which demonstrates that the binary of seL4 refines an abstract specification through three layers of refinement. Figure 2 summarizes the proof layers in seL4's formal verification. Consequently (under some reasonable assumptions listed in Appendix A) the seL4 binary is fully captured by the abstract specifications. In particular, one important feature derived from seL4's abstract specifications, is that: **the kernel never crashes**. Another one is that: **every kernel API call always terminates and returns to user-space**. Comprehensive details of seL4's formal verification can be found in [17].

Another seL4 feature very relevant to our work is: **correctness of access control enforcement** derived from functional correctness proof of seL4. [31] and [21] introduce formal definitions of the access control model and information flow in seL4 at the abstract specifications, as shown in the top layer of Figure 2. They demonstrate the refinement proof from these modified abstract specifications to the C implementation using Isabelle/HOL theorem prover, which is later linked to the binary level (by the same theorem prover). As a result, three properties are guaranteed by the access control enforcement proof: (1) *Authority Confinement*, (2) *Integrity* and (3) *Confidentiality*. Authority confinement means that authority propagates correctly with respect to its capability. For example, a thread with a read-only capability for an object can only read, and not write to, that object. Integrity implies that system state cannot be modified without explicit authorization. For instance, a read capability should not modify internal system state, while write capability should only modify an object associated with that capability. Finally, confidentiality means that an object cannot be read or inferred without a read capability. Thus, the proof indicates that access control in seL4, once specified at the binary level, is correctly enforced as long as the seL4 kernel is active.

We now show how seL4's access control enforcement property satisfies required RA features.

TABLE I
SECURITY PROPERTIES IN HYBRID RA DESIGNS

Security Property	SMART [11]	TrustLite [18]	HYDRA
Exclusive Access to $\mathcal{K}$	HW (Mod. Data Bus)	HW (MPU in ROM)	SW (seL4)
No Leaks	SW (CQUAL and Deputy)	SW (CPU Exception Engine)	SW (seL4)
Immutability	HW (ROM)	HW (Boot and MPU in ROM)	HW (Boot in ROM) and SW (seL4)
Uninterruptability	SW (Interrupt Disabled)	SW (CPU Exception Engine)	SW (seL4)
Controlled Invocation	HW (MPU in ROM)	HW (MPU in ROM)	SW (seL4)

B. Deriving seL4 Access Controls from Properties

We now describe access control configuration of seL4 user-space that achieves most required properties for secure RA, as described in section III. Unlike previous hybrid designs, HYDRA pushes almost all of these required features into software, as long as the seL4 microkernel boots correctly. (A comparison with SMART and TrustLite is in Table I.) We now derive access control configurations necessary and sufficient for secure RA running in seL4 user-space. We examine each feature and identify the corresponding access control configuration.

a) Exclusive Access to $\mathcal{K}$: directly translated to an access control configuration. Similar to previous hybrid approaches, $\mathcal{K}$ can be hard-coded into the Attest at production or installation time. There are two physical places where $\mathcal{K}$ can reside when the system is active: software executable stored in secondary storage and software executable loaded in RAM. Those memory locations needs to be configured to be accessible only to P_{Attest}.

b) No Leaks: achieved by the separation of virtual address space. Specifically, the virtual memory used for $\mathcal{K}$-related computation needs to be configured to be accessible to only P_{Attest}.

c) Immutability: achieved by a combination of verifiable boot and runtime isolation guarantee from seL4. At runtime, Attest needs to be immutable, which can be guaranteed by restricting the access control to the executable to only P_{Attest}. However, this is not enough to assure immutability of Attest executable because Attest can be modified after loaded into RAM but before executed. Hence, a verifiable boot of Attest is required.

d) Uninterruptability: satisfied by setting the scheduling priority of P_{Attest} higher than other processes since the formal proof of seL4 scheduling mechanism guarantees that a lower priority process cannot preempt the execution of a higher priority process. In addition, seL4 guarantees that, once set, the scheduling priority in seL4 cannot be increased during runtime (but possible to decrease the priority value).

Note that this feature implies P_{Attest} needs to be the initial user-space process since the seL4 microkernel always assigns the highest priority to the initial user-space process. For access control, this feature does not require any additional configuration.

e) Controlled Invocation: achieved by the isolation of process' execution. In particular, TCB of P_{Attest} cannot be accessed or manipulated by other processes.

With these features, we conclude that the access control configuration of seL4 user-space needs to (at least) include the following:

- C1 P_{Attest} has exclusive access to Attest; this also includes $\mathcal{K}$ residing in Attest. (Recall that P_{Attest} is the attestation **process**, while Attest is the executable that actually performs attestation.)
- C2 P_{Attest} has exclusive access to its TCB.
- C3 P_{Attest} has exclusive access to its VSpace.

Even though this access control configuration can be enforced at the binary code level, this assumption is based on that seL4 is loaded into RAM correctly. However, this can be exploited by an adversary by tricking the boot-loader to boot his malicious seL4 microkernel instead of the formally verified version and insert a new configuration violating above access controls. Thus, the hardware signature check of the seL4 microkernel code is required at boot time. The similar argument can also be made for P_{Attest}' code. As a result, additional integrity check of P_{Attest}' code needs to be performed by seL4 before executing.

C. Building Blocks

In order to achieve all security properties described above, HYDRA requires the following four components.

a) Read-Only Memory: region primarily storing immutable data (e.g. hash of public keys or signature of software) required for secure boot of the seL4 microkernel.

b) MCU Access Control Emulation: high-assurance software framework capable of emulating MCU access controls to attestation key $\mathcal{K}$ and the system clock. At present, seL4 is the only formally verified and mathematically proven microkernel capable of this task.

c) Attestation Algorithm: attestation algorithm residing in P_{Attest} and serving two main purposes: authenticating an attestation request, and performing attestation on memory regions.

d) Real-Time Clock: loosely synchronized (with $\mathcal{PRV}$) real-time clock for detecting replayed, reordered and delayed requests. If $\mathcal{VRF}$ does not have a clock, a secure counter can replace a real-time clock with a sacrifice of the delayed

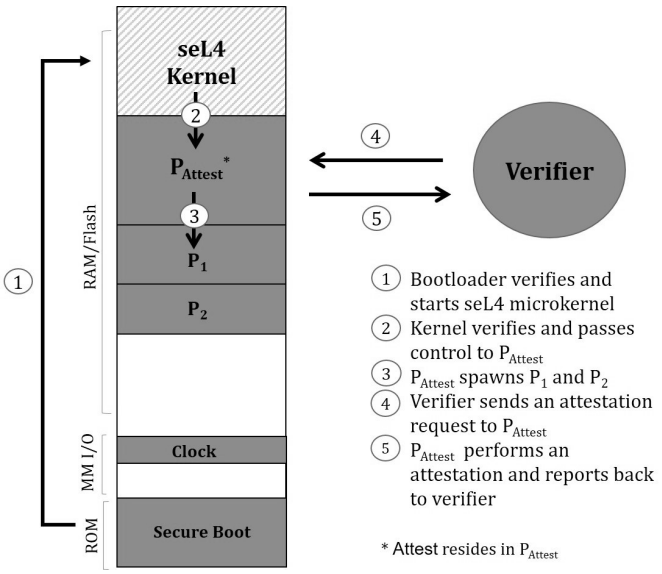


Fig. 3. Sequence of Operation in HYDRA

message detection. We also note that this component needs not be write-protected at hardware level due to a capability-based access control guarantee from seL4.

D. Sequence of Operation

The sequence of operations in HYDRA, shown in Figure 3, has three steps: boot, setup, and attestation.

a) Step 1: Boot Process: Upon a boot, $\mathcal{VRF}$ first executes a ROM-resident boot-loader. The boot-loader verifies authenticity and integrity of the seL4 microkernel binary. Assuming this verification succeeds, the boot-loader loads all executables, including kernel and user-space, into RAM and hands over control to the seL4 microkernel. Further details of secure boot in our prototype can be found in Section V.

b) Step 2: seL4 *Setup:* The first task in this step is to have the seL4 microkernel setting up the user-space and then starting P_{Attest} as the initial user-space process. Once the initialization inside the kernel is over, the seL4 microkernel gathers capabilities for all memory-mapped locations and assigns them to P_{Attest}. The seL4 kernel also performs an authenticity and integrity check of P_{Attest} to make sure that it has not been modified. After successful authentication, the seL4 microkernel passes control to P_{Attest}.

With full control over the system, P_{Attest} starts the rest of user-space with a lower scheduling priority and distributes capabilities that do not violate the configuration specified earlier. After completing configuration of memory capabilities and starting the rest of the user-space, P_{Attest} initializes the network interface and waits for an attestation request.

c) Step 3: Attestation: An attestation request, sent by a verifier, consists of 4 parameters: (1) T_R reflecting $\mathcal{PRV}$'s time when the request was generated, (2) target process p, (3) its memory range $[a, b]$ that needs to be attested, and (4) cryptographic checksum C_R of the entire attestation request.

Similar to SMART [11], the cryptographic checksum function used in attestation is implemented as a MAC, to ensure authenticity and integrity of attestation protocol messages.

Upon receiving an attestation request P_{Attest} checks whether T_R is within an acceptable range of the $\mathcal{PRV}$'s real-time clock before performing any cryptographic operation; this is in order to mitigate potential DoS attacks. If T_R is not fresh, P_{Attest} ignores the request and returns to the waiting state. Otherwise, it verifies C_R. If this fails, P_{Attest} also abandons the request and returns to the waiting state.

Once the attestation request is authenticated, P_{Attest} computes a cryptographic checksum of the memory region $[a, b]$ of process p. Finally, P_{Attest} returns the output to $\mathcal{VRF}$. The pseudo-code of this process is shown in Algorithm 1.

Algorithm 1: P_{Attest} Pseudo-Code

Input : T_R timestamp of request
p target process for attestation
a, b start/end memory region of target process
C_R cryptographic checksum of request
Output: Attestation Report

```
1  begin
2      /* Check freshness of timestamp */
3      if CheckFreshness(T_R) = False then
4          exit();
5      end
6      /* Verify an attestation request */
7      if VerifyRequest(C_R, K_Auth, T_R||p||a||b) = False
         then
8          exit();
9      end
10     /* Retrieve address space of process p */
11     Mem ← RetrieveMemory(p);
12     /* Start computing attestation report */
13     MacInit(K_Attest);
14     /* Start MAC computation on attestation parameters
         */
15     MacUpdate(T_R||p||a||b);
16     /* Continue MAC computation on memory ranging
         from a to b */
17     for i ∈ [a, b] do
18         MacUpdate(Mem[i]);
19     end
20     out ← MacFinal();
21     return out
22 end
```

V. HYDRA Implementation

To demonstrate feasibility and assess the practicality of HYDRA, we implement two prototypes of it on two commercially available hardware platforms: ODROID-XU4 [15] and Sabre Lite [6]. In this section, we focus more on the Sabre Lite implementation due to the lack of an seL4 compatible networking (e.g., Ethernet) driver and a programmable ROM

TABLE II
COMPLEXITY OF HYDRA IMPLEMENTATION ON SABRE LITE AND ODROID-XU4

Complexity	HYDRA with net. and libs	HYDRA w/o net. stack	HYDRA w/o net. and libs	seL4 Kernel Only
LoC	105,360	68,490	11,938	9,142
Exec Size	817KB	721KB	N/A	215KB

in current ODROID-XU4 boards. We provide details of, and insight gained from, our implementation, mostly concerning secure initialization and the configuration of the access control in HYDRA; we then describe the implementation of key storage and timestamp generation. Section VII presents a detailed performance evaluation of the HYDRA implementation.

A. seL4 *User-space Implementation*

Our prototype is implemented on top of version 1.3 of the seL4 microkernel [23]. The complete implementation, including helper libraries and the networking stack, consists of 105,360 lines of C code (see Table II for a more detailed breakdown). The overall size of executable is 817KB whereas the base seL4 microkernel size is 215KB. However, excluding all helper libraries, the implementation of HYDRA is just 600 lines of C code. In the user-space, we base our C code on following libraries: seL4utils, seL4vka and seL4vspace; these libraries provide the abstraction of processes, memory management and virtual space respectively. In our prototypes, P_{Attest} is the initial process loaded in the user-space and receives capabilities to all memory locations not used by the seL4 microkernel. Other processes in user-space are spawned by this P_{Attest}. We also ensure that the access control of those processes does not conflict with what we specified in Section IV. The details of this access control implementation are described below in this section.

The basic C function calls are implemented in muslc library. seL4bench library is used to evaluate timing and performance of our HYDRA implementation. For a timer driver, we rely on its implementation in seL4platsupport. All source code for these helper libraries can be found in [22] and these libraries contribute around 50% of the code base in our implementation. The formal verification of these libraries are left for future work. And lastly, we use an open-source implementation of a network stack and an Ethernet driver in the user-space [24]. We argue that this component, even though not formally verified, should not affect security objective of our design as long as an IO-MMU is used to restrict Direct Memory Access (DMA) of an Ethernet driver. Thus, the worst case that can happen from not formally verified network stack is symmetrical denial-of-service, which is out of scope of HYDRA.

B. Secure Boot Implementation

Here, we describe how we integrate an existing secure boot feature (in Sabre Lite) with our HYDRA implementation.

a) Secure Boot in Sabre Lite: NXP provides a secure boot feature for Sabre Lite boards, called High Assurance Boot (HAB) [13]. HAB is implemented based on a digital signature scheme with public- and private-keys. A private-key is required to generate a signature of the software image during manufacturing whereas a public-key is used by ROM APIs for decrypting and verifying the software signature at boot time. A public-key and a signature are attached to the end of software image, which is pre-installed into a flash medium during manufacturing. The digest of a public-key is fused into a one-time programmable ROM in order to ensure the authenticity of the public-key and the booting software image. At boot time, the ROM boot-loader first verifies an attached public-key by comparing it with the reference hash value in ROM. It then authenticates the software image through the attached signature and the verified public-key and then starts loading the software into RAM. Without a private-key, an adversary cannot forge a legitimate digital signature and thus is unable to insert and boot his malicious image. In addition, HAB will be able to detect such a malicious software image at an early stage and prevent it from taking control of a device later on.

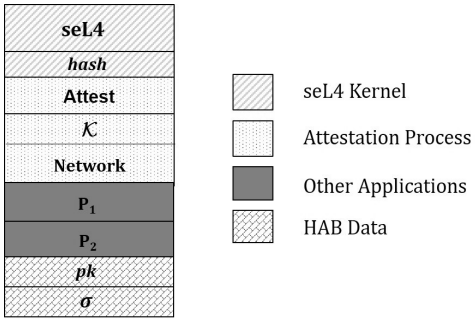


Fig. 4. Image Layout in Flash

b) Secure Boot of HYDRA*:* HAB can be utilized to ensure that the seL4 microkernel is the first program initialized after the ROM boot-loader. Therefore, the entire seL4 microkernel binary code can be covered when computing the digital signature by HAB during manufacturing. Moreover, the seL4 microkernel needs to be assured that it gives control of the user-space to the verified P_{Attest}. Thus, the seL4 microkernel has to perform an authenticity check of P_{Attest} before launching it. As a result, a hash of Attest needs to be computed and included in the seL4 microkernel's memory regions during production time and be validated upon starting the initial process.

With this procedure, a chain of trust is established in the remote attestation system in our design. This implies that no other programs, except the seL4 microkernel can be started by the ROM boot-loader and consequently only P_{Attest} is the certified initial process in the user-space, which achieve the goal of secure boot of remote attestation system. Figure 5 illustrates the secure boot of HYDRA in Sabre Lite prototype.

C. Access Control Implementation

Here we describe how the access control configuration specified in section IV is implemented in our HYDRA prototype.

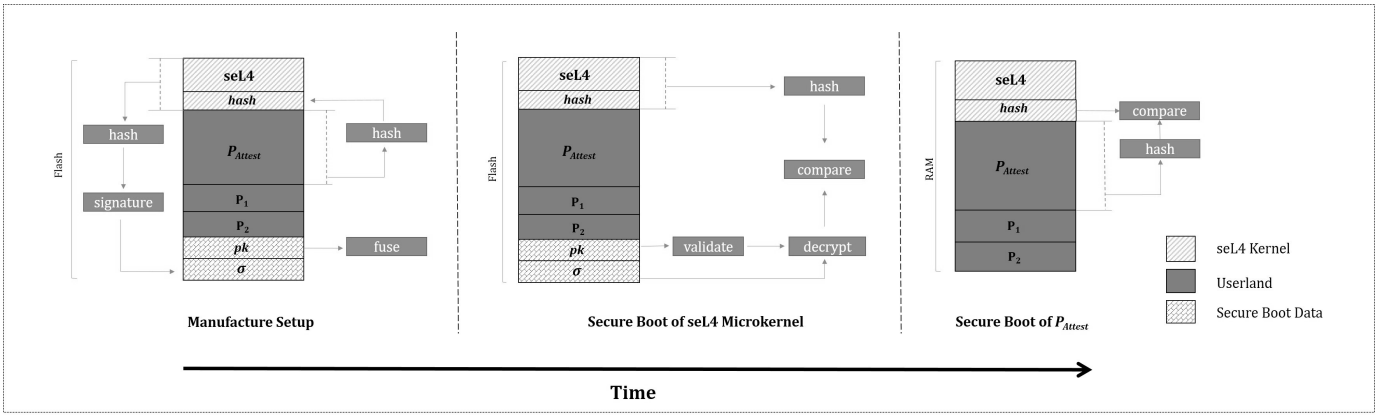


Fig. 5. Secure Boot Sequence in Sabre Lite Prototype

Our goal is to show that in the implementation of HYDRA no other user-space processes, except P_{Attest}, can have any kind of access to: (1) the binary executable code (including $\mathcal{K}$), (2) the virtual address space of P_{Attest}, and (3) the TCB of P_{Attest}.

To provide those access restrictions in the user-space, we make sure that we do not assign capabilities associated to those memory regions to other user-space processes. As mentioned earlier, P_{Attest} as the initial process contains all capabilities to every memory location not used by the seL4 microkernel. And there are two ways for P_{Attest} to issue capabilities across applications: dynamically transfer via endpoint with grant access right or statically assign during bootstrapping a new process.

In our implementation, P_{Attest} does not create any endpoint with grant access, which disallows any capability of P_{Attest} to transfer to a new process after created. Thus, the only way that capabilities can be assigned to a new process is before that process is spawned. When creating a new process, P_{Attest} assigns only minimal amount of capabilities required to operate that process, e.g. in our prototype, only the CSpace root node and fault endpoint (used for receiving IPCs when this thread faults) capabilities are assigned to any newly created process. Limited to only those capabilities, any other process cannot access the binary executable code as well as existing virtual memory and TCB of P_{Attest}.

Moreover, during bootstrapping the new process, P_{Attest} creates a new Page Directory object, which serves as the root of VSpace in the new process. This is to ensure that the new process' virtual address space is initially empty and does not overlap with the existing virtual memory of P_{Attest}. Without any further dynamic capability distribution, this guarantees that other processes cannot access any virtual memory page being used by P_{Attest} anytime afterwards. Sample code for configuring a new process in our prototype is provided in Appendix B.

In addition, we also add a sanity check before executing a new process by invoking a $revoke$ system call on the TCB and binary executable capabilities. With the revoke system call, the seL4 microkernel deletes all children of requested capability except the original one assuring that P_{Attest} does not accidentally issue any copy of those critical capabilities to other processes.

D. Key Storage

Traditionally, in previous hybrid designs, a prover device requires a special hardware-controlled memory location for securely storing $\mathcal{K}$ and protecting it from software attacks. However, in our design, it is possible to store $\mathcal{K}$ in a normal memory location (e.g. flash) due to the formally verified access control and isolation properties of seL4. Moreover, since $\mathcal{K}$ is stored in a writable memory, its update can easily happen without any secure hardware involvement. Thus, in our prototypes, $\mathcal{K}$ is hard-coded at production time and stored in the same region as P_{Attest}'s code. In order to ensure the read/write-protected access control of $\mathcal{K}$, P_{Attest} is coded such that it does not assign an access control capability of $\mathcal{K}$ to other processes except itself.

Alternatively, a separated process can also be created for storing and managing $\mathcal{K}$. P_{Attest} can then retrieve $\mathcal{K}$ from the key process through an IPC call. This approach, however, requires three configurations of the key process to ensure proper access controls of $\mathcal{K}$. First, the key process needs to be validated for correctness by P_{Attest} upon created. Thus, P_{Attest} must perform an integrity check of the key process on its binary before spawning it. Similar to P_{Attest} an access control to the address space and the binary data of the key process should not be shared by other processes to prevent them from directly accessing the $\mathcal{K}$ value. And lastly, an IPC endpoint of the key process should be accessible to only P_{Attest}. This is to assure that only P_{Attest} can retrieve $\mathcal{K}$ from the key process.

Besides $\mathcal{K}$, our design contains another key, K_{Auth}, used for verifying an authenticity of an attestation request. K_{Auth} can be a separate key and pre-stored next to K_{Attest} during a production time; or, K_{Auth} can be derived from $\mathcal{K}$ through a key derivation function (KDF) at runtime as well.

E. Timestamp Generation

Recall from section IV, a timestamp generated by a loosely synchronous real-time clock is required for detecting and ensuring freshness of an attestation request. However, at the present, there is no implementation of drivers for real-time clock in seL4.

In our prototype, a pseudo-timestamp can be generated by a counter, whose driver is provided by seL4platsupport, and a timestamp of the first validated request, as follows: When a device first wakes up and securely start P_{Attest}. P_{Attest} promptly loads a timestamp, T_{save}, that was previously saved in a separated location of flash medium before the last reset. When the first attestation request arrives, P_{Attest} checks its attached timestamp, T_{first}, whether it is greater than T_{save} and proceeds to $VerifyRequest$ if that is the case. After the assurance of the validated request, P_{Attest} keeps track of T_{first} and start a counter. At any time afterwards, a timestamp can be constructed by combining the current counter value with T_{first}. In addition, P_{Attest} periodically generates and saves this timestamp value into flash medium for the next reboot's usage.

VI. Security Analysis

In this section, we (informally) demonstrate how HYDRA satisfies the minimal set of requirements to realize secure RA (described in Section III). The following are key features ensured in the design and implementation of HYDRA:

(1) the seL4 microkernel is the first executable that is loaded in a HYDRA-based system upon boot/initialization. Correctness of this step is guaranteed by a ROM integrity check at boot time, e.g., HAB in the Sabre Lite case.

(2) The attestation process (P_{Attest})[4] is the first user-space process that is loaded into memory and is bootstrapped by the formally verified seL4 microkernel. This is also guaranteed using a software integrity check step performed by the seL4microkernel before spawning the initial process.

(3) P_{Attest} then starts with the highest scheduling priority and never decreases its own priority value. This can be guaranteed by inspecting the code for P_{Attest} and ensuring that it does not contain any system calls to decrease its priority.

(4) Any other subsequent process that is loaded on top of the seL4microkernel is spawned by P_{Attest} and does not receive the highest scheduling priority. This can be ensured by inspecting the P_{Attest} code and ensuring that all invocations of other processes are with a lower priority value. Once a process is loaded with a certain priority, seL4 prevents it from increasing its priority value; this is formally verified and guaranteed in the seL4 implementation.

(5) The software executable and $\mathcal{K}$ can only be mapped into the address space of P_{Attest}. This is guaranteed by inspecting the P_{Attest} code and ensuring that no other process on initialization (performed in P_{Attest}) receives the capabilities to access said memory ranges.

[4]Note that P_{Attest} is different from Attest as shown in Figure 4. P_{Attest} is what is called "initial process" in Figure 4 and it contains Attest executable as a component.

(6) Virtual memory used by P_{Attest} cannot be used by any other process; this includes any memory used for any computation touching the key, or related to other values computed using the key. This is formally verified and guaranteed in the seL4 implementation.

(7) Other processes cannot control or infer the execution of P_{Attest}. This is protected by the exclusive capability to TCB's P_{Attest}.

(8) The access control properties, i.e., authority confinement, integrity and confidentiality, in seL4 are mathematically guaranteed in the binary code level by its formal verification.

Given the above features, the security properties in Section III are satisfied because:

Exclusive Access to $\mathcal{K}$: (5), (6) and (8) guarantee that only P_{Attest} can have access to $\mathcal{K}$.

No Leaks: (6) ensures that intermediate values created by key-related computation inside P_{Attest} cannot be leaked to or learned by other processes.

Immutability: (1) and (2) implies that HYDRA is initialized into the correct expected (and known) initial states and that the correct binary executable is securely loaded into RAM. (5) also prevents other processes from modifying that executable.

Uninterruptability: (3) and (4) guarantees that other processes, always having a lower priority value compared to P_{Attest}, cannot interrupt the execution of P_{Attest}.

Controlled Invocation: (7) ensures that the execution of P_{Attest} cannot be manipulated by other applications.

In the systematic analysis of RA and its security in [12], the authors argue that any RA design satisfying the minimal set of features above is secure. We argued (informally) above how this is also the case for HYDRA. Formalizing the above and proving the security claims using automated methods, or other formal means, are natural next steps but out of the scope of this work which mainly focuses of developing the HYDRA design and assessing its feasibility.

VII. Experimental Evaluation

In this section, we present performance evaluation of HYDRA using the Sabre Lite prototype. We present the results of HYDRA on ODROID-XU4 in Appendix C. In particular, we conduct experiments to assess the speed of, and overhead involved in, performing attestation using different types of keyed Message Authentication Code (MAC) functions, on various numbers of user-space processes and sizes of memory regions to be attested. We first provide a detailed breakdown of the attestation runtime to identify the most computationally intensive steps, we then investigate in depth the time P_{Attest} takes to map the attested memory regions to its own virtual address space. We conclude our assessment with measuring the performance of HYDRA using five different MAC functions. We obtain the fastest performance using the Speck MAC; HYDRA can attest 10MB in less than 500msec in that case.

TABLE III
PERFORMANCE BREAKDOWN OF ALGORITHM 1 ON I.MX-SL @ 1GHZ

Operations	1 MB of Memory		10 KB of Memory	
	Time in cycle	Proportion	Time in cycle	Proportion
$VerifyRequest$	3,992	<0.01%	3,961	0.75%
$RetrieveMem$	8,731,858	15.11%	47,136	8.93%
$MacMem$[5]	49,021,846	84.89%	476,857	90.32%
$Overall$	57,757,696	100%	527,954	100%

Our prototypes are implemented in C and compiled with an O2 flag.

a) Breakdown of Attestation Runtime: Recall from Section IV, that the attestation algorithm (Algorithm 1) is composed of three operations. $VerifyRequest$ (lines 3 to 9) is responsible for verifying an attestation request and whether it has been recently generated by an authorized verifier. $RetrieveMem$ (line 11) maps memory regions from a target process to P_{Attest}'s address space and returns a pointer to the mapped memory data. And, lastly, $MacMem$ (lines 13 to 20) computes a cryptographic checksum (using $\mathcal{K}$) on the memory regions.

As shown in Table III, the runtime of $MacMem$ contributes the highest amount of the overall Attest runtime: 84% of total time for attesting 1MB of memory and 90% for attesting 10 KB of memory on Sabre Lite; whereas $RetrieveMem$ and $VerifyRequest$ together require less than 20% of the overall time. We will examine later in this section how other factors (e.g., number of processes and size of attested memory) influence the performance of the MAC computation.

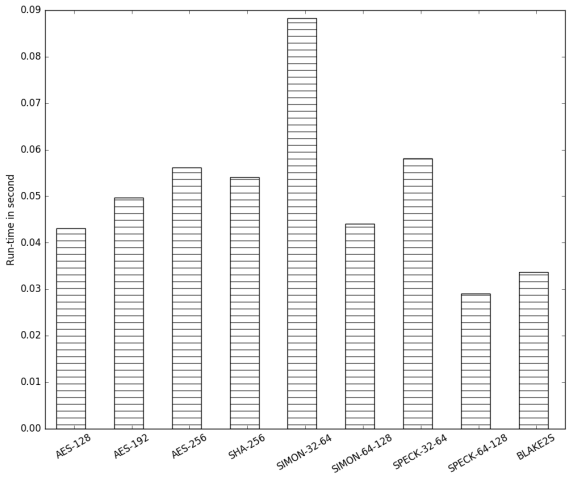


Fig. 6. Performance of Various MAC Implementations (avg over 1000 runs)

b) Performance of $RetrieveMem$ in seL4*:* Another important factor affecting the performance of HYDRA is the runtime of $RetrieveMem$: the time P_{Attest} takes to map the attested memory regions to its own virtual address space. As expected, Figure 7 illustrates the memory mapping runtime in

[5]Speck-64-128 based MAC covers line 13 to 18 in Algorithm 1

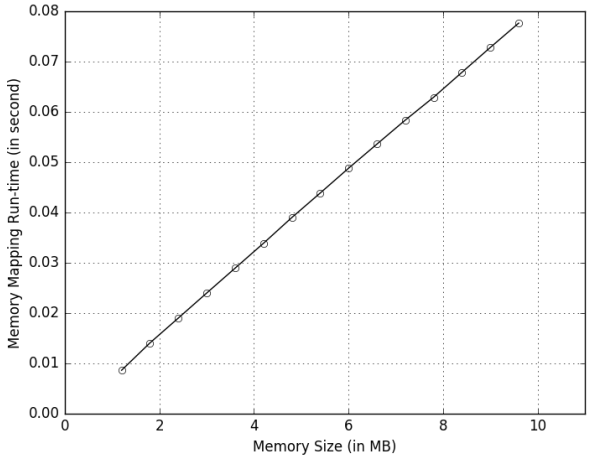


Fig. 7. Performance of Memory Mapping in seL4 (avg over 1000 runs)

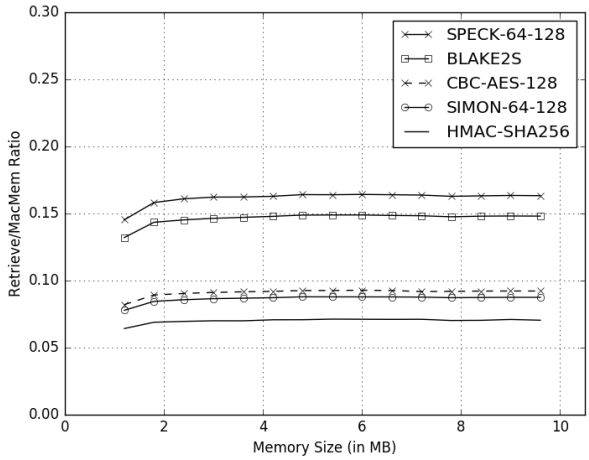


Fig. 8. Comparing $MacMem$ and $RetrieveMem$ (avg over 1000 runs)

seL4 is linear in terms of mapped memory size. In addition, we compare the runtime of $RetrieveMem$ and $MacMem$ on larger memory sizes. Figure 8 illustrates that the runtime ratio of $RetrieveMem$ to various implementations of $MacMem$ is always less than 20%. This confirms that retrieving memory and mapping it to the address space account for only a small fraction of the total attestation time in HYDRA. This illustrates that whatever overhead seL4 introduces when enforcing access control on memory is not significant and does not render HYDRA impractical.

c) Performance of $MacMem$ in seL4*:* Since $MacMem$ is the biggest contributor to the runtime of our implementations, we explore various types of (keyed) cryptographic checksums and their performance on top of seL4. We compare the performance of five different MAC functions, namely, CBC-AES [33], HMAC-SHA-256 [3], Simon and Speck [26], and BLAKE2S [27], on 1MB of data in the user-space of seL4. The performance results in Figure 6 illustrate that the

runtime of Speck-64-128[6] and BLAKE2S in seL4 are similar; and they are at least 33% faster than other MAC functions when running on Sabre Lite.

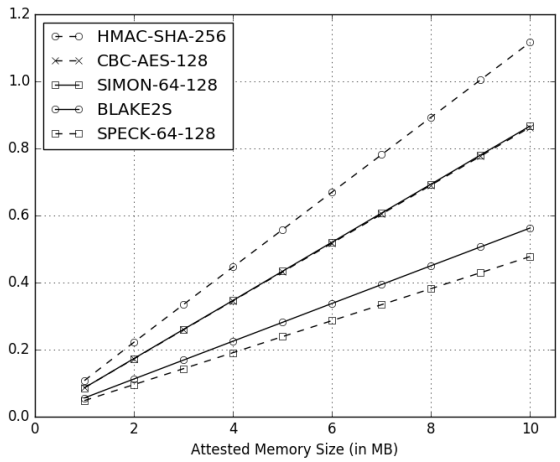


Fig. 9. Performance of $MacMem$ with Different Memory Sizes (avg over 1000 runs)

d) Performance of $MacMem$ vs Memory Sizes: Another factor that affects $MacMem$'s performance is the size of memory regions to be attested. We experiment by creating another process in the user-space and perform attestation on various sizes (ranging from 1MB to 10MB) of memory regions inside that process. As expected, the results of this experiment, illustrated in Figure 9, indicate that $MacMem$ performance is linear as a function of the attested memory sizes. This experiment also illustrates feasibility of performing attestation of 10MB of memory on top of seL4 in HYDRA using a Speck-based MAC in less than half a second.

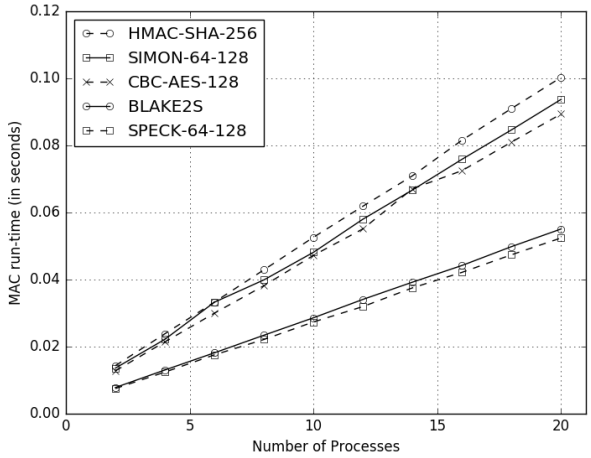


Fig. 10. Performance of $MacMem$ with Different Number of Processes (avg over 1000 runs)

[6]Speck with 64-bit block size and 128-bit key size

e) Performance on $MacMem$ vs Numbers of Processes: The goal of this experiment is to answer the following question: How would an increase in number of processes affect the performance of HYDRA? To answer this question, we have the initial process spawn additional user-space processes (from 2 to 20 extra processes) and, then, perform $MacMem$ on 100 KB memory data in one of those processes. To ensure fair scheduling of every process, we set priority of all processes (including the initial process) to the maximum priority. As shown in Figure 10, the performance of $MacMem$ is reasonably linear as a function of the number of processes on Sabre Lite.

VIII. Conclusions

This paper introduces the first hybrid Remote Attestation (RA) design, HYDRA, that leverages the formally verified seL4 microkernel to instantiate the memory and process isolation, and to enforce access control to memory and other resources; such isolation and access control have been ensured through hardware in previous hybrid designs. Using seL4 has the advantage of requiring fewer hardware modifications to the underlying microprocessor, or micro-controller, and provides an (automated) formal proof of the isolation guarantees of the actual implementation of the microkernel. We implement our design on two commodity hardware platforms and conduct experiments to assess its practicality and performance; we show that HYDRA can attest 10MB of memory in less than 500msec when using Speck as the underlying block-cipher to compute a cryptographic checksum over the attested memory. Our prototypes demonstrate feasibility of designing and implementing hybrid RA schemes that significantly improve security in contemporary embedded and cyber-physical systems, and that can work on commodity hardware and require no modification to it while providing security guarantees that can be assured using automated formal methods.

References

[1] "Extracting qualcomm's keymaster keys!" [Online]. Available: https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html

[2] 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 *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 121.

[3] Apple Computer, Inc, "LibOrange," 2006. [Online]. Available: https://github.com/unixpickle/LibOrange/blob/master/LibOrange/hmac-sha256.c

[4] ARM Limited, "Arm security technology - building a secure system using trustzone technology," 2009.

[5] F. Armknecht, R. Maes, A.-R. Sadeghi, B. Sunar, and P. Tuyls, "Pufprfs: a new tamper-resilient cryptographic primitive," in *Advances in Cryptology–EUROCRYPT*, vol. 2009, 2009, pp. 96–102.

[6] Boundary Devices, "BD-SL-I.MX6." [Online]. Available: https://boundarydevices.com/product/sabre-lite-imx6-sbc/

[7] F. Brasser, P. Koeberl, B. E. Mahjoub, A.-R. Sadeghi, and C. Wachsmann, "Tytan: Tiny trust anchor for tiny devices," in *52nd Design Automation Conference (DAC) 2015*, Jun. 2015.

[8] F. Brasser, K. B. Rasmussen, A.-R. Sadeghi, G. Tsudik, I. Martinovic, K. B. Rasmussen, M. Roeschlin, G. Tsudik, G. Revadigar, C. Javali *et al.*, "Remote attestation for low-end embedded devices: the provers perspective," in *Design Automation Conference (DAC)*, 2016.

[9] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 400–409.
[10] V. Costan and S. Devadas, "Intel sgx explained," Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint. iacr. org/2016/086, Tech. Rep.
[11] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and minimal architecture for (establishing a dynamic) root of trust," in *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2012.
[12] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "Systematic treatment of remote attestation." *IACR Cryptology ePrint Archive*, vol. 2012, p. 713, 2012.
[13] Freescale Semiconductor, Inc., "i.MX 6 Linux High Assurance Boot (HAB) User's Guide," Tech. Rep., 2013.
[14] Y. Gasmi, A.-R. Sadeghi, P. Stewin, M. Unger, and N. Asokan, "Beyond secure channels," in *Proceedings of the 2007 ACM workshop on Scalable trusted computing*. ACM, 2007, pp. 30–40.
[15] Hardkernel co., Ltd. [Online]. Available: http://goo.gl/mKgcy4
[16] A. Ibrahim, A.-R. Sadeghi, G. Tsudik, and S. Zeitouni, "Darpa: Device attestation resilient to physical attacks."
[17] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an os microkernel," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 1, p. 2, 2014.
[18] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *European Conference on Computer Systems*, 2014.
[19] J. Kong, F. Koushanfar, P. K. Pendyala, A.-R. Sadeghi, and C. Wachsmann, "Pufatt: Embedded platform attestation based on novel processor-based pufs," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 109:1–109:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2593192
[20] F. Lugou, L. Apvrille, and A. Francillon, "Toward a methodology for unified verification of hardware/software co-designs."
[21] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "sel4: from general purpose to a proof of information flow enforcement," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 415–429.
[22] National ICT Australia, "seL4 Libraries," 2014. [Online]. Available: https://github.com/seL4/seL4_libs
[23] ——, "The seL4 Repository," 2014. [Online]. Available: https://github.com/seL4/seL4
[24] ——, "UNSW Advanced Operating Systems," 2014. [Online]. Available: https://bitbucket.org/kevinelp/unsw-advanced-operating-systems
[25] National Vulnerability Database, "Vulnerability summary for cve-2015-6639." [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-6639
[26] B. Ray, S. Douglas, S. Jason, T. Stefan, W. Bryan, and W. Louis, "The simon and speck families of lightweight block ciphers," Cryptology ePrint Archive, Report./404, Tech. Rep., 2013.
[27] M. Saarinen and J. Aumasson, "The blake2 cryptographic hash and message authentication code (mac)," Tech. Rep., 2015.
[28] S. Schulz, C. Wachsmann, and A. Sadeghis, "Lightweight remote attestation using physical functions, technische universitat darmstadt, darmstadt," Germany, Technical Report, Tech. Rep., 2011.
[29] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT: Software-based attestation for embedded devices," in *IEEE Symposium on Security and Privacy*, 2004, pp. 272–282.
[30] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems," in *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2005, pp. 1–16.
[31] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein, "sel4 enforces integrity," in *International Conference on Interactive Theorem Proving*. Springer, 2011, pp. 325–340.
[32] T. A. L. Sewell, M. O. Myreen, and G. Klein, "Translation validation for a verified os kernel," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 471–482.
[33] The OpenSSL Project, "Openssl 1.1.0-pre7-dev," 2016. [Online]. Available: https://github.com/openssl/openssl/
[34] Trusted Computing Group (TCG), "Website," http://www.trustedcomputinggroup.org, 2015.

APPENDIX A
seL4'S PROOF ASSUMPTIONS

seL4's functional correctness proof is based on the following assumptions:

- **Assembly** - it assumes the correctness of ARM assembly code mainly for entry to and exit from the kernel and direct hardware accesses.
- **Hardware** - it assumes hardware operates according to its specification and has not been tampered with.
- **Hardware Management** - it assumes the correctness of the underlying hardware management, including a translation look-aside buffer (TLB) and cache-flushing operations.
- **Boot Code** - it assumes the correctness of code that boots the seL4 microkernel into memory.
- **Direct Memory Access (DMA)** - it assumes DMA is disabled or trusted.
- **Side-channels** - it assumes there is no timing side-channels.

APPENDIX B
SAMPLE CODE FOR STARTING NEW PROCESS

In our implementation, P_{Attest} creates a new process with the configuration below:

```
int sel4utils_configure_process_custom(sel4utils_process_t *
     process, vka_t *vka, vspace_t *spawner_vspace,
     sel4utils_process_config_t config)
{
    int error;
    sel4utils_alloc_data_t * data = NULL;
    memset(process, 0, sizeof(sel4utils_process_t));
    seL4_CapData_t cspace_root_data = seL4_CapData_Guard_new(0,
          seL4_WordBits - config.one_level_cspace_size_bits);

    /* create a page directory */
    process->own_vspace = config.create_vspace;
    error = vka_alloc_vspace_root(vka, &process->pd);
    if (error) {
        goto error;
    }

    /* assign an asid pool */
    if (assign_asid_pool(config.asid_pool, process->pd.cptr) !=
          seL4_NoError) {
        goto error;
    }

    /* create a cspace */
    process->own_cspace = config.create_cspace;
    if (create_cspace(vka, config.one_level_cspace_size_bits,
         process, cspace_root_data) != 0) {
        goto error;
    }

    /* create a fault endpoint */
    if (create_fault_endpoint(vka, process) != 0) {
        goto error;
    }

    /* create a vspace */
    sel4utils_get_vspace(spawner_vspace, &process->vspace, &
         process->data, vka, process->pd.cptr,
         sel4utils_allocated_object, (void *) process);
```

```
    /* finally elf load */
    process->entry_point = sel4utils_elf_load(&process->vspace,
          spawner_vspace, vka, vka, config.image_name);

    if (process->entry_point == NULL) {
        goto error;
    }

    process->sysinfo = sel4utils_elf_get_vsyscall(config.
         image_name);

    /* create the thread, do this *after* elf-loading so that
         we don't clobber the required virtual memory*/
    error = sel4utils_configure_thread(vka, spawner_vspace, &
         process->vspace, SEL4UTILS_ENDPOINT_SLOT, config.
         priority, process->cspace.cptr, cspace_root_data, &
         process->thread);

    if (error) {
        goto error;
    }

    return 0;

error:
    /*  clean up */
    ...

    return -1;
}


int sel4utils_configure_thread_config(vka_t *vka, vspace_t *
     parent, vspace_t *alloc, sel4utils_thread_config_t config,
      sel4utils_thread_t *res)
{
    memset(res, 0, sizeof(sel4utils_thread_t));

    int error = vka_alloc_tcb(vka, &res->tcb);
    if (error == -1) {
        sel4utils_clean_up_thread(vka, alloc, res);
        return -1;
    }

    res->ipc_buffer_addr = (seL4_Word) vspace_new_ipc_buffer(
         alloc, &res->ipc_buffer);

    if (res->ipc_buffer_addr == 0) {
        return -1;
    }

    if (write_ipc_buffer_user_data(vka, parent, res->ipc_buffer
         , res->ipc_buffer_addr)) {
        return -1;
    }

    seL4_CapData_t null_cap_data = {{0}};
    error = seL4_TCB_Configure(res->tcb.cptr, config.
         fault_endpoint, config.priority, config.cspace, config
         .cspace_root_data, vspace_get_root(alloc),
         null_cap_data, res->ipc_buffer_addr, res->ipc_buffer);

    if (error != seL4_NoError) {
        sel4utils_clean_up_thread(vka, alloc, res);
        return -1;
    }

    res->stack_top = vspace_new_stack(alloc);

    if (res->stack_top == NULL) {
        sel4utils_clean_up_thread(vka, alloc, res);
        return -1;
    }

    return 0;
}
```

APPENDIX C
HYDRA'S PERFORMANCE IN ODROID-XU4

We also evaluate performance of HYDRA in ODROID-XU4 @ 2.1 GHz. Despite not have an Ethernet network, we evaluate the core component of HYDRA: $MacMem$. Unlike the results in Section VII, BLAKE2S-based MAC achieves the fastest performance for attesting 10MB on ODROID-XU4 platform.

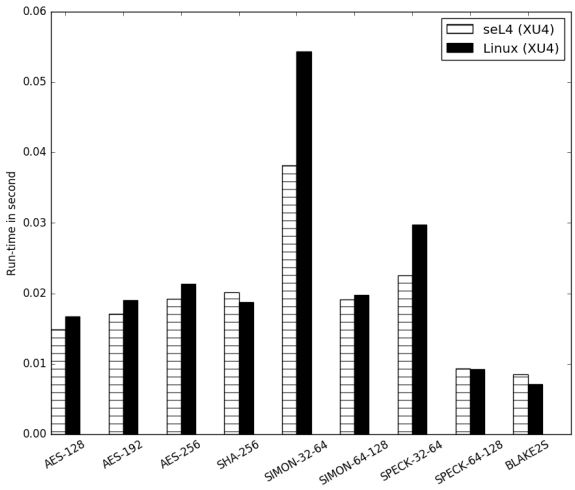


Fig. 11. Performance of Various MAC Functions on ODROID-XU4 (avg over 1000 runs)

a) MAC Performance in Linux vs in seL4*:* Figure 11 illustrates the performance comparison of various keyed MAC functions on ODROID-XU4 hardware platform when running on Ubuntu 15.10 and seL4. The result emphasizes the feasibility of RA in seL4 since the runtime of seL4-based RA can be as fast as that of RA running on top of widely prominent Linux operating system.

b) MAC Performance on ODROIX-XU4: The result in Section VII and the one from above demonstrate that Speck- and BLAKE2S-based MACs have the fastest attestation runtime in seL4. We conduct additional experiments on those two MAC functions in ODROID-XU4 platform. Figure 12 asserts the linear relationship between number of processes and $MacMem$ runtime. In addition, the MAC runtime, shown in Figure 13, is also linear in memory sizes to be attested. Lastly, runtime of the BLAKE2S-based MAC function achieves less than 200 milliseconds for attesting 10MB of memory regions in ODROID-XU4

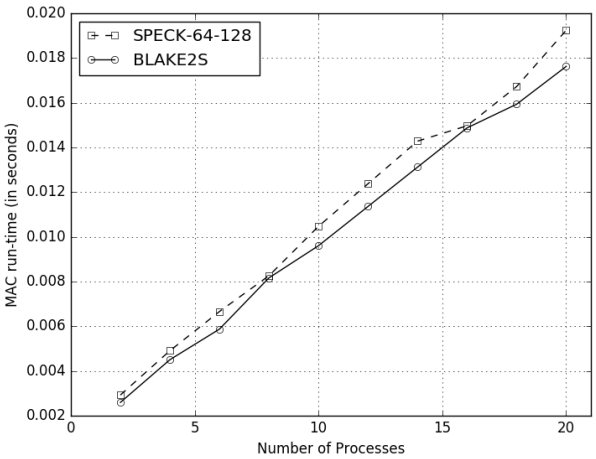


Fig. 12. Performance of $MacMem$ with Different Number of Processes on ODROID-XU4 (avg over 1000 runs)

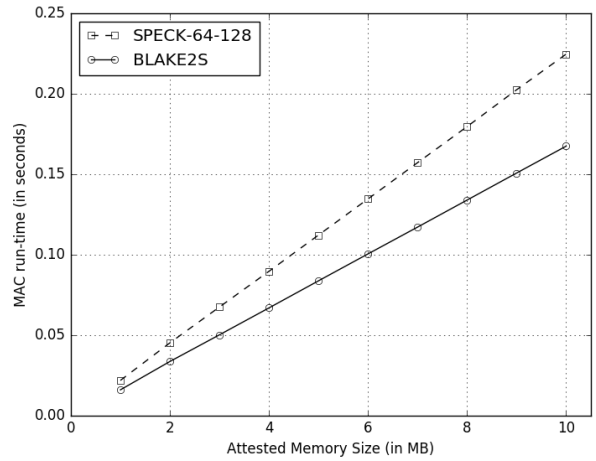


Fig. 13. Performance of $MacMem$ with Different Attested Memory Sizes on ODROID-XU4 (avg over 1000 runs)