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

Karim Eldefrawy*
Information and Systems Sciences Lab
HRL Laboratories
keldefrawy@hrl.com

Norrathep Rattanavipanon
Computer Science Department
University of California, Irvine
nrattana@uci.edu

Gene Tsudik
Computer Science Department
University of California, Irvine
gene.tsudik@uci.edu

## ABSTRACT

Remote Attestation (RA) allows a trusted entity (*verifier*) to securely measure 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 and patches, verifiable deletion and memory resetting. There are three major types 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, 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 imposes fewer hardware requirements on the underlying microprocessor. Also, 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 experimentation; we show that HYDRA can attest 10MB of memory in less than 250msec when using a Speck-based cryptographic checksum.

**ACM Reference format:**
Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. 2017. HYDRA: HYbrid Design for Remote Attestation (Using a Formally Verified Microkernel). In *Proceedings of WiSec '17 , Boston, MA, USA, July 18-20, 2017,* 12 pages.
https://doi.org/10.1145/3098243.3098261

## 1 INTRODUCTION

In recent years, embedded systems (ES), cyber-physical systems (CPS) and internet-of-things (IoT) devices, have percolated into many aspects of daily life, such as: households, offices, buildings, factories and vehicles. This trend of "smart-ification" of devices that were previously analog (or at least not connected) brings many obvious benefits. However, it also expands the attack surface and turns these newly computerized gadgets into natural and attractive attack 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 has its own advantages and 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 proving security separately for each component and then proving that its composition is secure. However, we believe that this is not yet possible given the current state of developments and capabilities in (automated) formal verification and synthesis of hardware and software.

One recent prominent example illustrating difficulty of correctly designing and implementing security primitives (especially, those blending software and hardware) is the TrustZone-based Qualcomm Secure Execution Environment (QSEE) kernel vulnerability and exploit reported in CVE-2015-6639 [6]. ARM TrustZone [18] is a popular System-on-Chip (SoC) and a CPU system-wide approach to security; it is adopted in billions of processors on various platforms. CVE-2015-6639 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 break Android's Full Disk Encryption (FDE) scheme by recovering the master keys [17]. This example demonstrates the difficulty of getting both the design and the implementation right; it also motivates the use of formally verified building blocks, which can 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 [8] and [16]. Using seL4 requires fewer hardware modifications to the underlying microprocessor and provides an automated formal proof of isolation guarantees of the 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.*

*Currently at the Computer Science Laboratory, SRI International. karim@csl.sri.com


*WiSec '17 , July 18-20, 2017, Boston, MA, USA*

ACM ISBN 978-1-4503-5084-6/17/07…$15.00
https://doi.org/10.1145/3098243.3098261

The main goal of this paper is to investigate a previously unexplored segment of the design space of hybrid RA schemes, specifically, techniques that incorporate formally verified and proven (using automated methods) components, such as the seL4 microkernel. 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 [8] and TrustLite [16]) 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 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 results on systematic analysis of features required for securely realizing hybrid RA [9]. To achieve our goals we make 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 analysis via experiments to demonstrate practicality of the proposed design. We show that HYDRA can attest 10MB of memory in less than 250ms when using Speck [23] as the underlying block-cipher to compute a cryptographic checksum (MAC).

**Organization**: Section 2 overviews related work, followed by Section 3 which presents our goals and assumptions. The design of HYDRA is presented in Section 4 and its security analysis in Section 6. Implementation issues and performance assessment are discussed in Sections 5 and 7.

## 2 RELATED WORK

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

### 2.1 Hardware-Based Remote Attestation

The hardware-based approach typically relies on the security provided by a Trusted Platform Module (TPM) [12]. 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. [11] presents how to link this evidence to secure channel end-points.

In 2015, Intel introduced a new set of instructions, termed Software Guard Extensions (SGX), that enable a hardware-enforced isolated execution environment (*enclave*) for specific software. An enclave contains only private data and code executing computations using such data [5], this enables isolation of such 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 the code (or memory) to be attested, signed by the CPU.

### 2.2 Software-Based Remote Attestation

Despite resisting all but physical attacks, the hardware-based approach is not suitable for embedded devices due to its additional hardware and software complexity and expense. Therefore, many software-only RA approaches have been proposed, specifically for embedded devices. Pioneer [25] 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. Security of this approach became questionable after several attacks on such schemes (i.e., [3]) were demonstrated.

### 2.3 Hybrid Remote Attestation

The main shortcoming of the software-based approach is that it makes strong assumptions about adversarial capabilities, which may not hold in practical networked settings. Thus, several hybrid software-hardware co-designs have been proposed to overcome this limitation. SMART [8] presents a hybrid approach for RA with minimal hardware modifications to existing MCUs. In addition to having uninterruptable attestation code and attestation keys residing in ROM, this architecture utilizes a hardware-based MCU access control 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. TrustLite [16] extends [8] to enable RA while supporting an interrupt handling in a secure place.

In addition to the above work designing hybrid RA schemes, [9] 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 upon the analysis in [9] and utilize these properties and components (which are described in Section 3) and show how to instantiate them in new ways to develop the new hybrid RA design – HYDRA.

## 3 GOALS AND ASSUMPTIONS

This section overviews HYDRA and its design rationale, discusses security objectives and features, as well as the adversarial model. Our notation is summarized below.

### 3.1 Design Rationale

Our main objective is to explore a new segment of the overall RA design space. The proposed hybrid RA design – HYDRA – requires

| | |
|---|---|
| $\mathcal{A}dv$ | Adversary |
| $\mathcal{P}rv$ | Prover |
| $\mathcal{V}rf$ | Verifier |
| $PR_{Att}$ | Attestation Process on $\mathcal{P}rv$ |
| $BC_{Att}$ | Attestation Code/Binary on $\mathcal{P}rv$ |
| $\mathcal{K}$ | Symmetric secret key shared by $\mathcal{P}rv$ and $\mathcal{V}rf$ |

**Table 1: Notation**

very little in terms of secure hardware and builds upon the formally verified seL4 microkernel. As shown in Section 5, the only hardware support needed by HYDRA is a hardware-enforced secure boot feature, which is readily available on commercial off-the-shelf development boards and processors, e.g., Sabre Lite 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. In particular, what was earlier attained using additional MCU controls and Read-Only Memory (ROM) in the SMART [8] and TrustLite [16] architectures can now be instantiated using capability controls in seL4.

To motivate and justify the design of HYDRA, we start with the result of Francillon, et al. [9]. It provides a systematic treatment of RA by developing a semi-formal definition of RA as a distinct security service, and systematically de-constructing it into a 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. Sections 4 and 5 show how the security objective and properties are satisfied in HYDRA and instantiated in two concrete prototypes based on Sabre Lite and ODROID-XU4 boards.

## 3.2 Hybrid RA Objective and Properties

According to [9], the RA security objective is to allow a (remote) prover ($\mathcal{P}rv$) to create an unforgeable authentication token, that convinces a verifier ($\mathcal{V}rf$) that the former is in some well-defined (expected) state. Whereas, if $\mathcal{P}rv$ has been compromised (i.e., malware is present), the authentication token must reflect this. [9] describes a combination of platform features that achieve aforementioned security objective and derives a set of properties both necessary and sufficient for secure RA. The conclusion of [9] is that the following properties collectively represent the minimal requirements to achieve secure RA on any platform.

- *Exclusive Access to Attestation Key ($\mathcal{K}$):* the attestation process ($PR_{Att}$) must have exclusive access to $\mathcal{K}$. This is the most difficult requirement for (especially, low-end and mid-range) embedded devices. As argued in [9], this property is unachievable without some hardware support on low-end devices. If the underlying processor supports multiple privilege modes and a full-blown memory separation for each process, one could use a privileged process to handle any computation that involves $\mathcal{K}$. However, low-end and mid-range processors generally do not offer such "luxury" features.
- *No Leaks:* no information related to (or derived from) $\mathcal{K}$ must be accessible after execution of $PR_{Att}$. To achieve this, all intermediate values that depend on $\mathcal{K}$ – except the final attestation token to be returned to $\mathcal{V}rf$ – must be securely erased. This is applicable to very low-end devices, with none (or minimal) OS support and assuming that memory is shared between processes. However, if the underlying hardware and/or software guarantees strict memory separation among processes, this property is trivially satisfied.
- *Immutability:* To ensure that the attestation executable ($BC_{Att}$) cannot be modified, [8] and [9] place it in ROM, which is available on most, even low-end, platforms. ROM is a relatively inexpensive way to enforce $BC_{Att}$'s code immutability. Whereas, if the OS guarantees: (1) run-time process memory separation, and (2) immutability of $BC_{Att}$ code (e.g., by checking its integrity/authenticity prior to execution), then $BC_{Att}$ can reside, and be executed, in RAM.
- *Uninterruptability:* Execution of $BC_{Att}$ must be uninterruptible. This is necessary to ensure that malware does not obtain the key (or some function thereof) by interrupting $BC_{Att}$ while any key-related values remain in registers or other locations. SMART achieves this property via MCU controls. However, if $BC_{Att}$ runs with the highest possible priority, the OS can ensure uninterruptibility.
- *Controlled Invocation (aka Atomicity):* $BC_{Att}$ must only be invocable from its first instruction and must exit only at one of its legitimate last (exit) instruction. This is motivated by the need to prevent code-reuse attacks. As before, enforcing this property via MCU access controls can be replaced by OS support.

[9] stipulates one extra property: *Secure Reset*, initiated whenever an attempt is detected to execute $BC_{Att}$ from the middle of its code. We argue that this is not needed if controlled invocation is enforced. It suffices to raise an exception, as long as the memory space of $BC_{Att}$ is protected and integrity of executable is guaranteed.

Another important RA security feature identified in [2] is to protect $\mathcal{P}rv$ from $\mathcal{V}rf$ impersonation as well as denial-of-service (DoS) attacks that attempt to forge, replay, reorder or delay attestation requests. All such attacks aim to maliciously invoke RA functionality on $\mathcal{P}rv$ and thus deplete $\mathcal{P}rv$'s resources or take them away from its main tasks. According to [2], the following additional property is required:

- *$\mathcal{V}rf$ Authentication:* $PR_{Att}$ on $\mathcal{P}rv$ must: (1) authenticate $\mathcal{V}rf$ and (2) detect replayed, re-ordered and delayed requests. To achieve (1), the very same $\mathcal{K}$ can used to generate (by $\mathcal{V}rf$) and verify (by $\mathcal{P}rv$) all attestation requests. To satisfy (2), [2] requires an additional hardware component: a reliable real-time clock. This clock must be loosely synchronized with $\mathcal{V}rf$'s clock and must be write-protected.

## 3.3 Adversarial Model & Other Assumptions

Based on the recent taxonomy in [1], RA adversary ($\mathcal{A}dv$) can be categorized as follows:

- *Remote:* exploits vulnerabilities in $\mathcal{P}rv$'s software to inject malware, over the network.

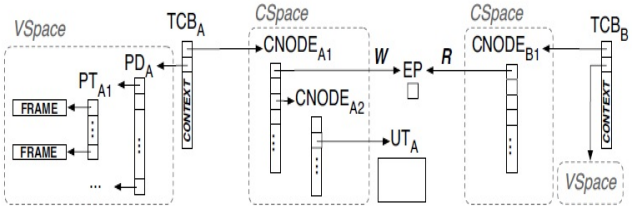



**Figure 1: Sample seL4 instantiation from [26].**

- *Local:* located sufficiently near $\mathcal{P}rv$ in order to eavesdrop on, and manipulate, $\mathcal{P}rv$'s communication channel(s).
- *Physical:* has full (local) physical access to $\mathcal{P}rv$ and its hardware; can perform physical attacks, e.g., use side channels to derive keys, physically extract memory values, and modify various hardware components.

[9] and [2] show that any RA that satisfies all properties described in 3.2 always yields correct attestation tokens (i.e., no false positives and no false negatives) while achieving resilience to DoS attacks even in the presence of remote and local $\mathcal{A}dv$-s. HYDRA builds on top of these properties and similarly considers remote and local $\mathcal{A}dv$-s; physical $\mathcal{A}dv$ is considered to be out-of-scope.

We note that, at least in a single-prover setting[1], 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 for tamper detection. We consider $\mathcal{P}rv$ to be a (possibly) unattended remote hardware platform running multiple processes on top of seL4. Once $\mathcal{P}rv$ boots up and runs in steady state, $\mathcal{A}dv$ might be in complete control of all application software (including code and data) before and after execution of $PR_{Att}$. Since physical attacks are out of scope, $\mathcal{A}dv$ can not induce hardware faults or retrieve $\mathcal{K}$ using side channels. $\mathcal{A}dv$ also has no means of interrupting execution of seL4 or $PR_{Att}$ code (details discussed later in the paper). Finally, recall that $\mathcal{P}rv$ and $\mathcal{V}rf$ must share at least one secret key $\mathcal{K}$. This key can be pre-loaded onto $\mathcal{P}rv$ at installation time and stored as part of $PR_{Att}$ binaries. We do not address the details of this procedure.

## 4 HYDRA DESIGN

This section overviews seL4 and discusses its use in HYDRA. It then describes the sequence of operations in HYDRA.

### 4.1 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 [27], 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.

[1] See [13] for physical attack resilience in groups of provers.

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 [26]) 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 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 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 across that endpoint. For instance, if A in Figure 1 has grant access to EP, it can issue 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 4.4 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. Consequently (under some reasonable assumptions listed in Appendix B) the seL4 binary is fully captured by the abstract specifications. In particular, two important feature derived from seL4's abstract specifications,

are 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 [15].

Another seL4 feature very relevant to our work is: **correctness of access control enforcement** derived from functional correctness proof of seL4. [26] and [19] introduce formal definitions of the access control model and information flow in seL4 at the abstract specifications. 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.

## 4.2 Deriving seL4 Access Controls

We now describe access control configuration of seL4 user-space that achieves most required properties for secure RA, as described in section 3. We examine each feature and identify the corresponding access control configuration. Unlike prior 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 2.)

- *Exclusive Access to $\mathcal{K}$:* is directly translated to an access control configuration. Similar to previous hybrid approaches, $\mathcal{K}$ can be hard-coded into the $BC_{Att}$ at production time. Thus, $BC_{Att}$ needs to be configured to be accessible only to $PR_{Att}$.
- *No Leaks:* is 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 $PR_{Att}$.
- *Immutability:* is achieved using combination of verifiable boot and runtime isolation guarantee from seL4. At runtime, $BC_{Att}$ must be immutable, which can be guaranteed by restricting the access control to the executable to only $PR_{Att}$. However, this is not enough to assure immutability of $BC_{Att}$ executable because $BC_{Att}$ can be modified after loaded into RAM but before executed. Hence, a verifiable boot of $BC_{Att}$ is required.
- *Uninterruptability:* is ensured by setting the scheduling priority of $PR_{Att}$ 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 of any process can not be increased at runtime.
  Note that this feature implies that $PR_{Att}$ needs to be the initial user-space process since the seL4 microkernel always assigns the highest priority to the initial process.
- *Controlled Invocation:* is achieved by the isolation of process' execution. In particular, TCB of $PR_{Att}$ cannot be accessed or manipulated by other processes.
- $\mathcal{V}rf$ *Authentication:* is achieved by configuring a capability of the real-time clock to be read-only for other processes.

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

**(C1):** $PR_{Att}$ has exclusive access to $BC_{Att}$; this also includes $\mathcal{K}$ residing in $BC_{Att}$. (Recall that $PR_{Att}$ is the attestation **process**, while $BC_{Att}$ is the executable that actually performs attestation.)
**(C2):** $PR_{Att}$ has exclusive access to its TCB.
**(C3):** $PR_{Att}$ has exclusive access to its VSpace.
**(C4):** $PR_{Att}$ has exclusive write-access to the real-time clock.

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 $PR_{Att}$ code. As a result, additional integrity check of $PR_{Att}$ code needs to be performed by seL4 before executing.

## 4.3 Building Blocks

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

- **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.
- **MCU Access Control Emulation:** high-assurance software framework capable of emulating MCU access controls to attestation key $\mathcal{K}$. At present, seL4 is the only formally verified and mathematically proven microkernel capable of this task.
- **Attestation Algorithm:** software residing in $PR_{Att}$ and serving two main purposes: authenticating an attestation request, and performing attestation on memory regions.
- **Reliable Real-Time Clock:** loosely synchronized (with $\mathcal{V}rf$) real-time clock. This component is required for mitigating denial-of-service attacks that involve $\mathcal{V}rf$ impersonation (via replay, reorder and delay)[2]. If $\mathcal{P}rv$ does not have a clock, a secure counter can replace a real-time clock with the downside of delayed message detection.

## 4.4 Sequence of Operation

The sequence of operations in HYDRA, shown in Figure 2, has three steps: boot, setup, and attestation.

**Table 2: Security Properties in Hybrid RA**

| Security Property | SMART [8] | TrustLite [16] | HYDRA |
|---|---|---|---|
| Exclusive Access to $\mathcal{K}$ | HW (Mod. Data Bus) | SW (programmed MPU) | SW (seL4) |
| No Leaks | SW (CQUAL and Deputy) | HW (CPU Exception Engine) | SW (seL4) |
| Immutability | HW (ROM) | HW (ROM) and SW (programmed MPU) | HW (ROM) and SW (seL4) |
| Uninterruptability | SW (Interrupt Disabled) | HW (CPU Exception Engine) | SW (seL4) |
| Controlled Invocation | HW (ROM) | HW (ROM) | SW (seL4) |

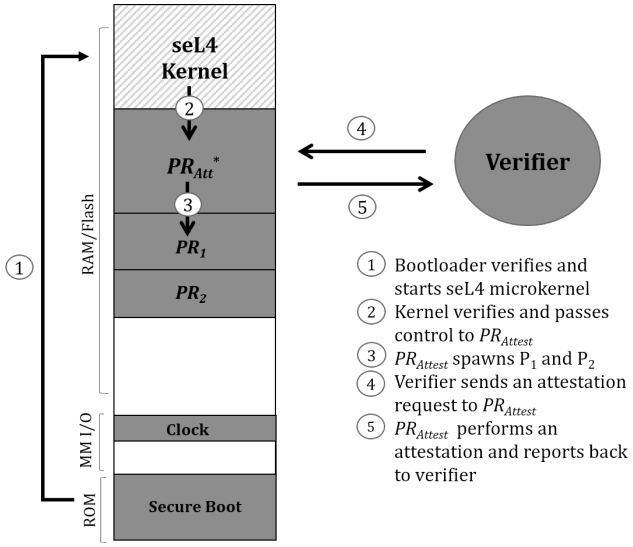



**Figure 2: Sequence of Operation in HYDRA**

*4.4.1 Boot Process.* Upon a boot, $\mathcal{P}rv$ 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 5.

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

With full control over the system, $PR_{Att}$ 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, $PR_{Att}$ initializes the network interface and waits for an attestation request.

*4.4.3 Attestation.* An attestation request, sent by a verifier, consists of 4 parameters: (1) $T_R$ reflecting $\mathcal{P}rv$'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 [8], the cryptographic checksum function used in attestation is implemented as a Message Authentication Code (MAC), to ensure authenticity and integrity of attestation protocol messages.

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

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

## 5 HYDRA IMPLEMENTATION

To demonstrate feasibility and practicality of HYDRA, we developed two prototypes on commercially available hardware platforms: ODROID-XU4 [4] and Sabre Lite [7]. We focus on the latter, because of lack of seL4 compatible network drivers and programmable ROM in current ODROID-XU4 boards. Section 7 presents a detailed performance evaluation of the implementation.

### 5.1 seL4 User-space Implementation

Our prototype is implemented on top of version 1.3 of the seL4 microkernel [22]. The complete implementation, including helper libraries and the networking stack, consists of 105,360 lines of C code (see Table 3 for a more detailed breakdown). The overall size of executable is 574KB whereas the base seL4 microkernel size is 215KB. Excluding all helper libraries, the implementation of HYDRA is just 2800 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, $PR_{Att}$ is the initial process in the user-space and receives capabilities to all memory locations not used by seL4. Other processes in user-space are spawned by this $PR_{Att}$. We also ensure that access control of those processes does not conflict with what we specified in Section

**Algorithm 1:** $BC_{Att}$ 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 and verify request */
3      if ¬ CheckFreshness(T_R) then
4          exit();
5      end
6      if ¬ VerifyRequest(C_R, K, T_R||p||a||b) then
7          exit();
8      end
9      /* Retrieve address space of process p */
10     Mem ← RetrieveMemory(p);
11     /* Compute attestation report */
12     MacInit(K);
13     MacUpdate(T_R||p||a||b);
14     for i ∈ [a, b] do
15         MacUpdate(Mem[i]);
16     end
17     out ← MacFinal();
18     return out
19 end
```

**Table 3: Complexity of HYDRA Impl. on Our Prototype**

| 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 | 574KB | 476KB | N/A | 215KB |

4. 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 [21] and these libraries contribute around 50% of the code base in our implementation. We use an open-source implementation of a network stack and an Ethernet driver in the user-space [20]. We argue that this component, even though not formally verified, should not affect security objective of HYDRA as long as an IO-MMU is used to restrict Direct Memory Access (DMA) of an Ethernet driver. The worst case that can happen from not formally verified network stack is symmetrical denial-of-service, which is out of scope of HYDRA.

## 5.2 Secure Boot Implementation

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

*5.2.1 Secure Boot in Sabre Lite.* NXP provides a secure boot feature for Sabre Lite boards, called High Assurance Boot (HAB) [10]. HAB is implemented based on a digital signature scheme

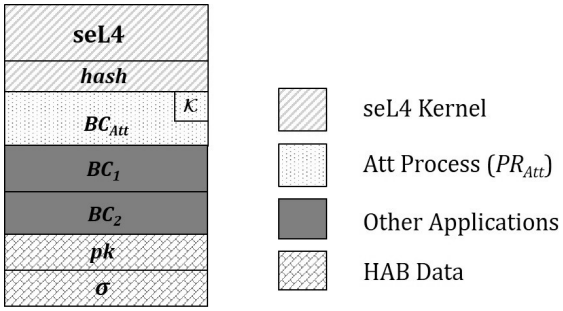



**Figure 3: Image Layout in Flash**

with public and private keys. A private key is needed 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 software image, which is pre-installed in a flash during manufacturing. The digest of a public key is fused into a one-time programmable ROM in order to ensure authenticity of the public key and the booting software image. At boot time, the ROM boot-loader first loads the software image into RAM and then verifies the 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. Execution of this image is allowed only if signature verification succeeds. Without a private key, an adversary cannot forge a legitimate digital signature and thus is unable to insert and boot his malicious image.

*5.2.2 Secure Boot of* HYDRA. HAB ensures that the seL4 microkernel is the first program initialized after the ROM boot-loader. This way, the entire seL4 microkernel binary code can be covered when computing the digital signature during manufacturing. Moreover, seL4 needs to be assured that it gives control of the user-space to the verified $PR_{Att}$, which means that seL4 has to perform an integrity check of $PR_{Att}$ before launching it. Consequently, a hash of $BC_{Att}$ needs to be included in the seL4 microkernel's binaries 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 HYDRA. This implies that no other programs, except the seL4 microkernel can be started by the ROM boot-loader and consequently only $PR_{Att}$ is the certified initial process in the user-space, which achieve the goal of secure boot of remote attestation system. Figure 4 illustrates the secure boot of HYDRA in Sabre Lite prototype.

## 5.3 Access Control Implementation

Here we describe how the access control configuration specified in section 4 is implemented in our HYDRA prototype. Our goal is to show that in the implementation of HYDRA no other user-space processes, except $PR_{Att}$, can have any kind of access to: (1) the binary executable code (including $\mathcal{K}$), (2) the virtual address space of $PR_{Att}$, and (3) the TCB of $PR_{Att}$. 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

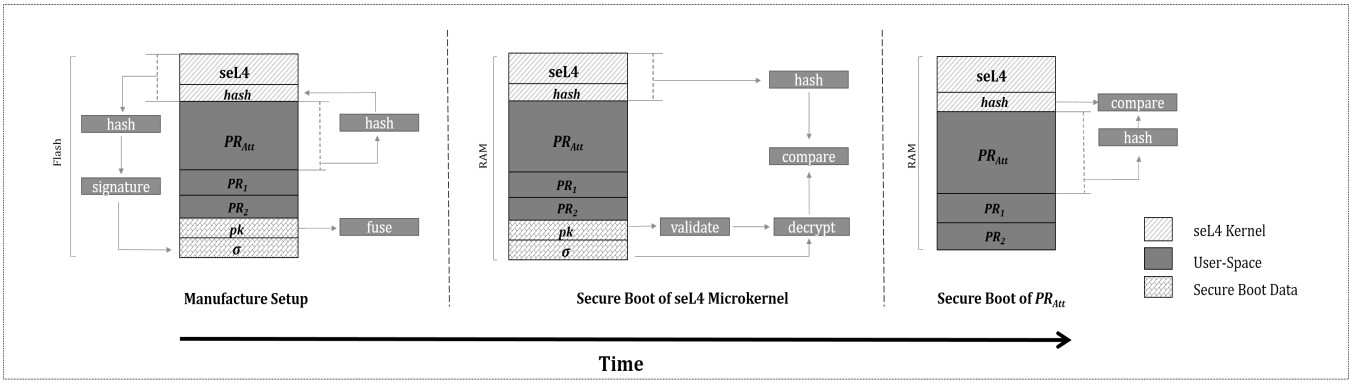



**Figure 4: Secure Boot Sequence in Sabre Lite Prototype**

processes. Recall that $PR_{Att}$ as the initial process contains all capabilities to every memory location not used by the seL4 microkernel. And there are two ways for $PR_{Att}$ to issue capabilities: dynamically transfer via endpoint with grant access right or statically assign during bootstrapping a new process.

In our implementation, $PR_{Att}$ does not create any endpoint with grant access, which disallows any capability of $PR_{Att}$ 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, $PR_{Att}$ 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 $PR_{Att}$.

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

## 5.4 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 HYDRA, 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 $BC_{Att}$.

## 5.5 Mitigating Denial-of-Service Attacks

Our HYDRA prototype uses the same $\mathcal{K}$ for two purposes: (1) $\mathcal{P}rv$ computing the attestation token, and (2) authenticating $\mathcal{V}rf$ attestation requests. (Recall that $\mathcal{K}$ can be accessed only by $PR_{Att}$.) Alternatively, $PR_{Att}$ can derive two separate keys from $\mathcal{K}$, one for each purpose, through a key derivation function (KDF).

[2] also shows that authenticating attestation requests is insufficient to mitigate DoS attacks since $\mathcal{A}dv$ can eavesdrop on genuine attestation requests and then delay or replay them. [2] concludes that timestamps, obtained from a reliable real-time clock (synchronized with $\mathcal{V}rf$'s clock), are required in order to handle replay, reorder and delay attacks.

There are currently no real-time clock drivers available for seL4. Instead, we generate a pseudo-timestamp by a timer, the driver for which is provided by seL4platsupport, and a timestamp of the first validated request, as follows:

When a device first wakes up and securely starts $PR_{Att}$ as the initial process, $PR_{Att}$ loads a timestamp, $T_0$, that was previously saved (in a separated location in flash) before the last reset. When the first attestation request arrives, $PR_{Att}$ checks whether its timestamp, $T_1 > T_0$ and, if so, proceeds to $VerifyRequest$. (Else, the request is discarded). Once the request is validated, $PR_{Att}$ keeps track of $T_1$ and starts a counter. At any later time, a timestamp can be constructed by combining the current counter value with $T_1$. Also, $PR_{Att}$ periodically generates and saves this timestamp value on flash, to be used after the next reboot. The prototype also ensures that the timestamp is write-protected by not assigning write capabilities for a memory region (storing $T_0$ and a timer device driver) to any other processes.

# 6 SECURITY ANALYSIS

We now informally show that HYDRA satisfies the minimal set of requirements to realize secure RA (described in Section 3). HYDRA's key features are:

(1) seL4 is the first executable 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) $PR_{Att}$ [2] is the initial user-space process loaded into memory and executed by seL4. This is also supported via a software integrity check performed by seL4 before spawning the initial process.

(3) $PR_{Att}$ starts with the highest scheduling priority and never decreases its own priority value. This can be guaranteed by checking

[2] $PR_{Att}$ is different from $BC_{Att}$ per Figure 3. $PR_{Att}$ is what is called "initial process" in Figure 3 and it contains $BC_{Att}$ executable as a component.

that $PR_{Att}$ code does not contain any system calls to decrease its priority.

(4) Any subsequent process executed by seL4 is spawned by $P_{Attest}$ and does not get the highest scheduling priority. This can be ensured by inspecting $PR_{Att}$ code to check 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 by seL4 implementation.

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

(6) Virtual memory used by $PR_{Att}$ cannot be used by any other process; this includes any memory used for any computation involving 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 execution of $PR_{Att}$ (protected by exclusive capability to TCB's $PR_{Att}$).

(8) Access control properties, i.e., authority confinement, integrity and confidentiality, in seL4's binary are mathematically guaranteed by its formal verification.

(9) Other processes cannot modify or reset the real-time clock. This can be guaranteed by verifying that $PR_{Att}$ code does not give away a write capability of the clock to other processes.

Given the above features, the security properties in Section 3 are satisfied because:

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

**No Leaks:** (6) and (8) ensures that intermediate values created by key-related computation inside $PR_{Att}$ cannot be leaked to or learned by other processes.

**Immutability:** (1) and (2) implies that HYDRA is initialized into the correct expected 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 $PR_{Att}$, cannot interrupt the execution of $PR_{Att}$.

**Controlled Invocation:** (7) ensures that the execution of $PR_{Att}$ cannot be manipulated by other applications.

$\mathcal{V}rf$ **Authentication:** (5), (6) and (8) ensures that $\mathcal{K}$ cannot be accessed and/or inferred by other processes. (8) and (9) ensures that no other process can modify and influence a timestamp value.

# 7 EXPERIMENTAL EVALUATION

Ideally, we would have liked to compare the performance of HYDRA with that of previous hybrid designs such as SMART and TrustLite on the same hardware platform. However, this is not feasible because SMART and TrustLite are designed for low-end micro-controllers and development platforms based on such micro-controllers (currently) cannot run seL4. In addition, SMART and TrusLite require some modifications to the micro-controller's hardware and are thus not available on off-the-shelf development platforms. We instead present performance evaluation of HYDRA using the commercially available Sabre Lite development platform. (Results of HYDRA on ODROID-XU4 are in Appendix A). We conduct experiments to assess 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 obtain the fastest performance using the Speck MAC; HYDRA can attest 10MB in less than 250msec in that case.

## 7.1 Breakdown of Attestation Runtime

Recall from Section 4, 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 $PR_{Att}$'s address space and returns a pointer to the mapped memory. $MacMem$ (lines 13 to 20) computes a cryptographic checksum (using $\mathcal{K}$) on the memory regions.

As shown in Table 4, the runtime of $MacMem$ contributes the highest amount of the overall $BC_{Att}$ runtime: 89% of total time for attesting 1MB of memory and 92% for attesting 20 KB of memory on Sabre Lite; whereas $RetrieveMem$ and $VerifyRequest$ together require less than 11% of the overall time.

## 7.2 Performance of RetrieveMem in seL4

Another important factor affecting the performance of HYDRA is the runtime of $RetrieveMem$: the time $PR_{Att}$ takes to map the attested memory regions to its own virtual address space. As expected, Figure 5b illustrates the memory mapping runtime in seL4 is linear in terms of mapped memory size. In addition, we compare the runtime of $RetrieveMem$ and $MacMem$ on larger memory sizes. Figure 5c 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.

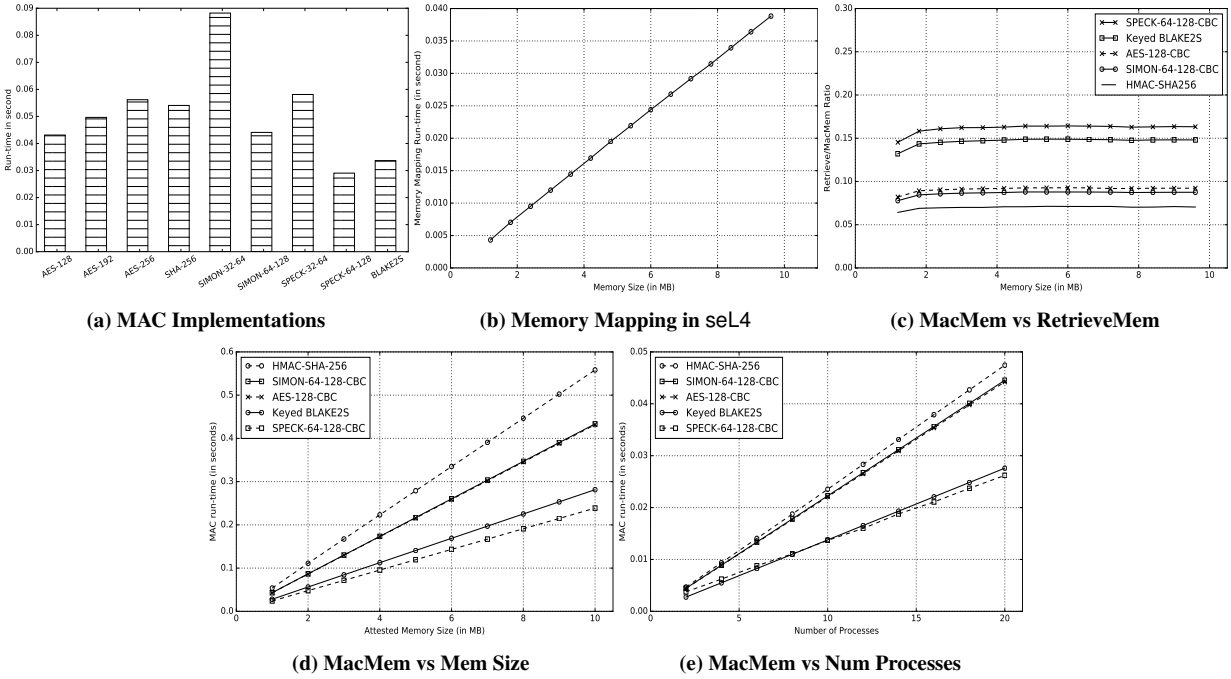
## 7.3 Performance of MacMem in seL4

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 [28], HMAC-SHA-256 [14], Simon and Speck [23], and BLAKE2S [24], on 1MB of data in the user-space of seL4. The performance results in Figure 5a illustrate that the runtime of MAC based on Speck-64-128 [3] and BLAKE2S in seL4 are similar; and they are at least 33% faster than other MAC functions when running on Sabre Lite.

## 7.4 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

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




**(a) MAC Implementations** **(b) Memory Mapping in seL4** **(c) MacMem vs RetrieveMem**

**(d) MacMem vs Mem Size** **(e) MacMem vs Num Processes**

**Figure 5: Evaluation of HYDRA in SabreLite prototype**

**Table 4: Performance Breakdown of Algorithm 1 on I.MX6-SL @ 1GHz**

| Operations | 1 MB of Memory | | 20 KB of Memory | |
|---|---|---|---|---|
| | Time in cycle | Proportion | Time in cycle | Proportion |
| $VerifyRequest$ | 1,604 | <0.01% | 1,604 | 0.29% |
| $RetrieveMem$ | 3,221,307 | 10.7% | 45,624 | 8.21% |
| $MacMem$ | 26,880,057 | 89.29% | 508,334 | 91.5% |
| $Overall$ | 30,102,968 | 100% | 555,562 | 100% |

(ranging from 1MB to 10MB) of memory regions inside that process. As expected, the results of this experiment, illustrated in Figure 5d, 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 250msec.

## 7.5 Performance on MacMem vs Numbers of Processes

This experiment answers the following question: How would an increase in number of processes affect the performance of HYDRA? To answer it, we have the initial process spawn additional user-space processes (from 2 to 20 extra processes) and, then, perform $MacMem$ on 100 KB memory in each process. The result from Figure 5e indicates that the performance of $MacMem$ is linear as a function of the number of processes on a Sabre Lite device.

# 8 CONCLUSIONS

This paper presented the first hybrid Remote Attestation (RA) design, HYDRA, that takes advantage of the formally verified seL4 microkernel to instantiate memory and process isolation and enforce access control to memory and other resources. HYDRA imposes minimal hardware requirements on the underlying microprocessor and provides an (automated) formal proof of isolation guarantees of the implementation of the microkernel. We implemented HYDRA on two commodity hardware platforms and demonstrated overall feasibility and practicality of hybrid RA schemes.

# ACKNOWLEDGMENTS

The authors are grateful to ACM WiSec'17 anonymous reviewers for their helpful comments and suggestions. UCI authors were supported, in part, by funding from: (1) the National Security Agency (NSA) under contract H98230-15-1-0276, (2) the Department of Homeland Security, under subcontract from the HRL Laboratories, (3) the Army Research Office (ARO) under contract W911NF-16-1-0536, and (4) the Australian Research Council (ARC) Discovery grant DP150100564.

# REFERENCES

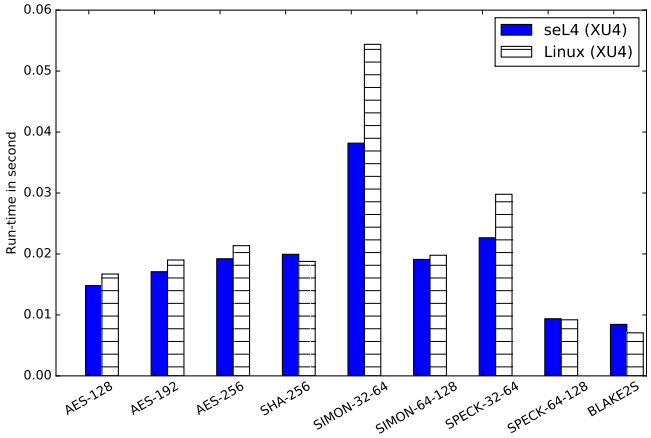
[1] Tigist Abera, N Asokan, Lucas Davi, Farinaz Koushanfar, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. Invited-things, trouble, trust: on building trust in iot systems. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 121.

[2] Ferdinand Brasser, Kasper B Rasmussen, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. Remote attestation for low-end embedded devices: the prover's perspective. In *Design Automation Conference (DAC)*. IEEE, 1–6.

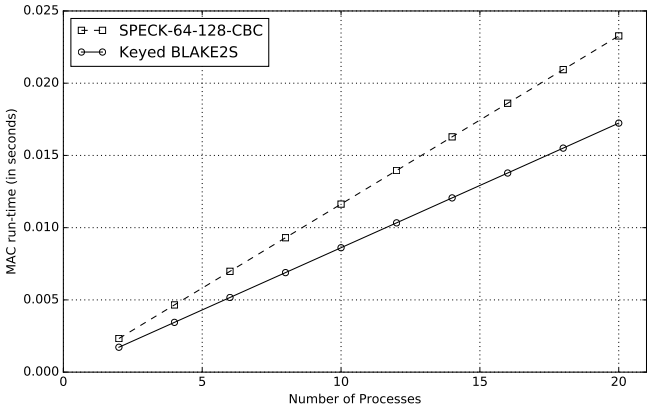
[3] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. 2009. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM conference on Computer and communications*

*security*. ACM, New York, NY, USA, 400–409.
[4] Hardkernel co. Ltd. 2013. ODROID-XU4. (2013). http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825
[5] Victor Costan and Srinivas Devadas. 2016. *Intel sgx explained*. Technical Report. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint. iacr. org/2016/086.
[6] National Vulnerability Database. 2015. Vulnerability Summary for CVE-2015-6639. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-6639. (2015).
[7] Boundary Devices. 2017. BD-SL-I.MX6. (2017). https://boundarydevices.com/product/sabre-lite-imx6-sbc/
[8] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. 2012. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society.
[9] Aurélien Francillon, Quan Nguyen, Kasper B Rasmussen, and Gene Tsudik. 2014. A minimalist approach to remote attestation. In *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 244.
[10] Freescale Semiconductor, Inc. 2013. *i.MX 6 Linux High Assurance Boot (HAB) User's Guide*. Technical Report.
[11] Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, and N Asokan. 2007. Beyond secure channels. In *Proceedings of the 2007 ACM workshop on Scalable trusted computing*. ACM, New York, NY, USA, 30–40.
[12] Trusted Computing Group. 2017. Trusted Platform Module (TPM). (2017). http://www.trustedcomputinggroup.org/work-groups/trusted-platform-module/
[13] Ahmad Ibrahim, Ahmad-Reza Sadeghi, Gene Tsudik, and Shaza Zeitouni. 2016. DARPA: Device Attestation Resilient to Physical Attacks. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. ACM, New York, NY, USA.
[14] Apple Computer Inc. 2006. LibOrange. https://github.com/unixpickle/LibOrange/blob/master/LibOrange/hmac-sha256.c. (2006).
[15] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems (TOCS)* 32, 1 (2014), 2.
[16] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. 2014. TrustLite: A Security Architecture for Tiny Embedded Devices. In *European Conference on Computer Systems*.
[17] laginimaineb. 2016. Extracting Qualcomm's KeyMaster Keys! https://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html. (2016).
[18] ARM Limited. 2009. ARM Security Technology - building a secure system using trustzone technology. (2009).
[19] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: from general purpose to a proof of information flow enforcement. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 415–429.
[20] National ICT Australia. 2014. UNSW Advanced Operating Systems. (2014). https://bitbucket.org/kevinelp/unsw-advanced-operating-systems
[21] National ICT Australia and other contributors. 2014. seL4 Libraries. https://github.com/seL4/seL4_libs. (2014).
[22] National ICT Australia and other contributors. 2014. The seL4 Repository. https://github.com/seL4/seL4. (2014).
[23] B Ray, S Douglas, S Jason, TC Stefan, W Bryan, and W Louis. 2013. *The SIMON and SPECK Families of Lightweight Block Ciphers*. Technical Report. Cryptology ePrint Archive, Report./404.
[24] MJ Saarinen and JP Aumasson. 2015. *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*. Technical Report.
[25] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. 2005. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 1–16.
[26] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. 2011. seL4 enforces integrity. In *International Conference on Interactive Theorem Proving*. Springer, 325–340.
[27] Thomas Arthur Leck Sewell, Magnus O Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 471–482.
[28] The OpenSSL Project. 2016. OpenSSL 1.1.0-pre7-dev. https://github.com/openssl/openssl/. (2016).

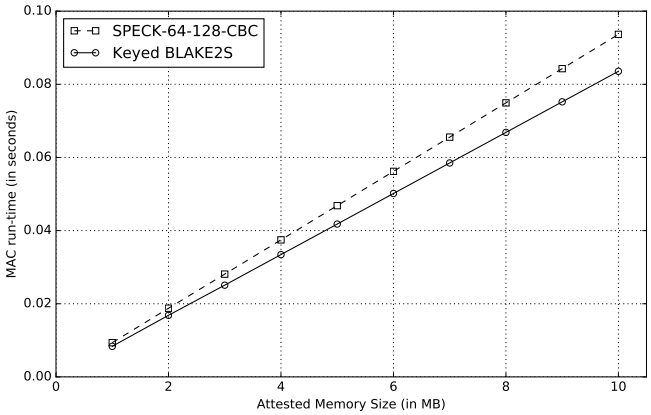
# APPENDIX




**(a) MAC Implementations**




**(b) $MacMem$ vs Num Processes**




**(c) $MacMem$ vs Mem Size**

**Figure 6: Evaluation of HYDRA in ODROID-XU4 prototype**

# A PERFORMANCE ON ODROID-XU4

We also evaluate performance of HYDRA on ODROID-XU4 @ 2.1 GHz. Despite lacking an Ethernet driver, we evaluate the core component of HYDRA: *MacMem*. Unlike results in Section 7, BLAKE2S-based MAC achieves the best performance for attesting 10MB on ODROID-XU4 platform.

## A.1 MAC Performance on Linux vs in seL4

Figure 6a illustrates the performance comparison of keyed MAC functions on ODROID-XU4 running on Ubuntu 15.10 and seL4. Results support feasibility of RA in seL4, since the runtime of seL4-based RA can be as fast as that of RA running on top of the popular Linux OS.

## A.2 MAC Performance on ODROIX-XU4

As follows from the results in Section 7 and above, Speck- and BLAKE2S-based MACs have the fastest attestation runtimes in seL4. We conducted additional experiments with these MAC functions on ODROID-XU4. Figure 6b shows the linear relationship between the number of processes and *MacMem* runtime. Also, MAC runtime in Figure 6c, is also linear in terms of the memory size to be attested. Finally, runtime of BLAKE2S-based MAC needs under 100 milliseconds to attest 10MB of memory.

# B SEL4 PROOF ASSUMPTIONS

seL4 functional correctness proof is based on the following assumptions:

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

# C SAMPLE CODE FOR STARTING NEW PROCESS

$PR_{Att}$ creates a new empty process with the default configuration as shown 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);
    process->own_vspace = config.create_vspace;
    error = vka_alloc_vspace_root(vka, &process->pd);
    if (error) {
        goto error;
    }
    if (assign_asid_pool(config.asid_pool, process->pd.cptr) != seL4_NoError) {
        goto error;
    }
    process->own_cspace = config.create_cspace;
    if (create_cspace(vka, config.one_level_cspace_size_bits, process,
            cspace_root_data) != 0) {
        goto error;
    }
    if (create_fault_endpoint(vka, process) != 0) {
        goto error;
    }
    sel4utils_get_vspace(spawner_vspace, &process->vspace, &process->data, vka,
            process->pd.cptr, sel4utils_allocated_object, (void *) process);
    process->entry_point = sel4utils_elf_load(&process->vspace, spawner_vspace,
            vka, vka, config.image_name);
    if (process->entry_point == NULL) {
        goto error;
    }
    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;
}
```