Tiered Trust for Useful Embedded Systems Security

Hudson Ayers
hayers@stanford.edu
Stanford University

Prabal Dutta
prabal@berkeley.edu
UC Berkeley

Philip Levis
pal@cs.stanford.edu
Stanford University

Amit Levy
aalevy@cs.princeton.edu
Princeton University

Pat Pannuto
ppannuto@ucsd.edu
UC San Diego

Johnathan Van Why
jrvanwhy@google.com
Google

Jean-Luc Watson
jlw@berkeley.edu
UC Berkeley

ABSTRACT

Traditional embedded systems rely on custom C code deployed in a monolithic firmware image. In these systems, all code must be trusted completely, as any code can directly modify memory or hardware registers. More recently, some embedded OSes have improved security by separating userspace applications from the kernel, using strong hardware isolation in the form of a memory protection unit (MPU). Unfortunately, this design requires either a large trusted computing base (TCB) containing all OS services, or moving many OS services into userspace. The large TCB approach offers no protection against seemingly-correct backdoored code, discouraging the use of kernel code produced by others and complicating security audits. OS services in userspace come at a cost to usability and efficiency. We posit that a model enabling two tiers of trust for kernel code is better suited to modern embedded software practices. In this paper, we present the threat model of the Tock Operating System, which is based on this idea. We compare this threat model to existing security approaches, and show how it provides useful guarantees to different stakeholders.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**.

KEYWORDS

embedded systems, operating systems, security, IoT

ACM Reference Format:
Hudson Ayers, Prabal Dutta, Philip Levis, Amit Levy, Pat Pannuto, Johnathan Van Why, and Jean-Luc Watson. 2022. Tiered Trust for Useful Embedded Systems Security. In *15th European Workshop on Systems Security (EUROSEC '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3517208.3523752

1 INTRODUCTION

Embedded system security is notoriously lacking compared to the security of more traditional computer systems [13]. We believe this is partially caused by typical embedded system design not providing proper security abstractions to the different stakeholders in this space, often requiring non-experts to make security-affecting decisions without the proper knowledge. We posit that embedded operating systems (OSes) should provide a threat model that maintains the goal of a minimal *trusted computing base* (TCB) without forcing common OS functionality to be treated equivalently to application code. In particular we believe that there should be two tiers of trust *within* the kernel, to encourage code reuse between deployments without compromising on security, usability, or efficiency.

We have previously argued the premise that embedded systems are often treated more like software platforms than single purpose devices [19]. Based on this premise, our discussion of security focuses on three stakeholders: application developers, kernel developers, and board integrators. We believe an OS threat model with three tiers of trust can serve all of these stakeholders better than prior approaches. For application developers, this means providing isolation from other applications and independence from (although with trust of) the kernel. This enables the audit of stand-alone applications for security without knowledge of other code on the system. For kernel developers this means limiting the scope of the TCB, modularizing non-core kernel functionality, and reducing the level of trust that must be placed in non-core kernel functionality. This means that in the common case kernel extensions will not affect the TCB, which eases auditing burden. For board integrators—platform developers who are responsible for bringing up new hardware—this means making it apparent which kernel components have additional privileges and what those privileges are. Often, these privileges may present security/usability trade-offs, for example questions of how processes may be started, stopped, or introspected. As board integration creates the TCB, this must balance flexibility and auditability.

Here, we describe the threat model for Tock, an embedded OS for low power microcontrollers, and how we think this threat model represents a useful improvement over the status quo. This threat model treats the majority of OS functionality (such as network stacks, drivers, and hardware virtualisation layers) as more trusted than application code but significantly less trusted than the TCB.

2 BACKGROUND

2.1 Embedded development practices

Embedded applications are extremely heterogeneous, ranging from large-scale sensor deployments [1] to personal hardware cryptocurrency wallets [6]. As a result, the embedded software ecosystem contains large amounts of bespoke code targeted at different constrained design points. This is not necessarily a bad thing for application developers: they have the freedom to choose components that respect performance, real-time deadlines, or platform-specific constraints. They might require, for example, a networking stack

EUROSEC '22, April 5–8, 2022, RENNES, France
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9255-6/22/04.
https://doi.org/10.1145/3517208.3523752

that supports TCP [8] or IPv6 [11], while others may need a protocol designed specifically for constrained settings [2].

Embedded applications are increasingly the targets of security analysis [22], where large amounts of third-party software presents a burden to reviewers. To avoid this, system management is commonly delegated to a packaged set of functionality in the form of an embedded OS, of which some portion operates as a TCB. The TCB is assumed to be correct and performs security-critical functions, so highly secure systems minimize the size and scope of their TCB [14] to limit the potential attack surface.

2.2 Threat modeling

The specific security needs of an embedded application will vary widely by use case, but in general, developing an embedded threat model requires defining three items: a grounded set of attacks against which the system must be resilient, a specification of each of the system stakeholders (if more than one), and for each stakeholder, a set of security guarantees that must be maintained under the expected threats. We discuss each facet in more detail below.

The security of an embedded system depends on its threat environment. A threat model enumerates the types of adversaries that are expected to attack the system, and alternatively, which attacks are deemed to be out of scope. For example, embedded systems may consider attacks from network-based adversaries [3] or malicious co-located applications [19], but assume that attackers will not have physical device access [20] or the capability to selectively influence MEMS sensor readings [21]. Some threat models may even choose not to defend against attacks, assuming no substantial risk exists.

In this paper, we discuss additional potential attacks that may arise as the result of co-locating the execution of multiple, mutually-distrusting tasks. The idea of mutually-distrusting tasks is valid both in the case of a multi-programmed platform with multiple users, and in the case where there is potential for certain tasks to be compromised (e.g. network-based trojans). A malicious task, even while barred by the system from directly observing another targeted task, may instead observe various *side-channels* that leak information indirectly about the target. For example, with no access to a task performing a critical cryptographic operation, an attacker with the ability to measure its execution time might nonetheless be able to retrieve the secret keys used [16]. Similarly, a task may use a shared hardware bus for communication, and infer the communication behavior of other tasks based on frequency and duration of its availability.

2.2.1 Stakeholders. The components of an embedded system are split among one or more parties we denote as *stakeholders.* For the same high-level threat model, each stakeholder may have different security requirements and trust assumptions.

An embedded system will likely have at least two stakeholders: a *user*, or application developer, who is using the OS's interface to interact with device resources and executes custom tasks for a particular use case, and the OS *kernel developers* who implement kernel functionality that application developers use. Application developers benefit from reliable, secure kernel services. Kernel developers benefit when they can reuse code written by others, and when they can rely on isolation from the behavior of unrelated parts of the system. Often, the kernel may be entirely distrustful of the application, while the user must completely trust the kernel.

We discuss a third stakeholder, the board integrator, separately from an application or an embedded OS kernel developer. Board integrators configure and add platform-specific code to an embedded OS so that it can use the board's capabilities. Often, board integrators are responsible for determining which portions of kernel code to include/use for a given platform. For example, the board integrators for the OpenTitan root-of-trust chip implement the platform specific code to give the kernel access to hardware peripherals like UART and cryptographic accelerators. Different board integrators may make different security assumptions, and as such, other stakeholders should, when possible, limit the scope of their own security assumptions and make them clear to board integrators.

2.2.2 Security guarantees. A stakeholder's defense against a set of potential threats cannot be evaluated without a sense of what "security" actually means. To do so, system components can provide or require specific *security guarantees* that must be maintained. System security is then ensured when all security guarantees are met.

Security guarantees are often (but not always) expressed using the "CIA triad": systems may provide confidentiality (protection from unauthorized access to private, or "secret" data), integrity (protection from unauthorized data modification), availability (access to data when required and protection from denial-of-service attacks), or any combination thereof. For example, an embedded OS may provide a *memory safety* guarantee to mutually-distrustful applications, using a hardware Memory Protection Unit (MPU) to ensure that a user cannot read or modify another user's memory space or any of the kernel's memory. In this case, the OS is making confidentiality and integrity guarantees to each user.

An embedded OS serves to simplify security analysis by specifying some black-box security guarantees on which applications can rely. However, ensuring that the entire system (application, platform, OS kernel) is secure requires an explicit understanding of each stakeholder's security needs and the presence of matching guarantees from the other stakeholders. If the OS does not guarantee a particular security property, or vaguely documents it, a time-consuming code audit is often necessary.

3 RELATED WORK

3.1 Embedded OSes and their threat models

3.1.1 "Sensor Mote" OSes. TinyOS [18], Contiki [9], and Riot [7] are all widely used embedded OSes, but focus on fairly benevolent threat models. As is the case with highly monitored or isolated sensor deployments, there is only one stakeholder, responsible for all application and OS code on the system. All code is implicitly trusted, and as such, no formal threat model is provided and security in general is not addressed. The presence of concurrent programming constructs in these OSes does not necessarily make a security guarantee: the Contiki kernel, for example, manages event handling and interprocess communication (IPC) but exposes direct hardware access to applications and provides no memory isolation. Similarly, some hardware resources in TinyOS are virtualized to support simpler application development [17], but also don't give any explicit guarantee of isolation, and Riot places many of its drivers in threads for fault isolation, but a malicious driver could easily modify the kernel directly.

3.1.2 Zephyr OS. In contrast, Zephyr OS has generated extensive security documentation [23, 24]. Zephyr considers two types of stakeholder: application developers whose threads may be mutually distrustful of each other, and the OS kernel development team that is assumed to be trusted. Zephyr's threat model allows for malicious application threads, but vulnerabilities in the kernel (e.g. compromised I/O from malicious network data or denial-of-service attacks from high-priority threads) are out of scope.

Zephyr supports hardware-based memory isolation using an MPU, with allows it to guarantee the integrity and confidentiality of each thread's memory space and kernel memory. Platform resources can be assigned to specific threads and access validated at the system call layer, guaranteeing that a device will only be used by threads explicitly granted access.

Finally, an effort to develop a secure/certified branch of Zephyr is still pending, but may require additional assumptions, namely, that only trusted applications are running on the device and that appropriate protections are implemented against storage and timing side-channels [24]. These added assumptions may make verification easier, but incur significant overhead [23] and make the applicable threat model brittle. For example, a remote code execution vulnerability on a secure system that assumes trusted applications may invalidate any verified security guarantees.

3.1.3 Arm MBED OS. MBED OS is a collection of system management libraries, including an RTOS scheduler, that can be included with an embedded application [5]. Like the mote OSes discussed above, the application code and the OS kernel are controlled by the same stakeholder and are not isolated from each other in any way. However, using additional architectural security features in ARMv8-M microprocessors [4], MBED introduces an additional stakeholder that develops code to execute in a "secure world" isolated from both the application and OS and accessible through special call gates. On those microprocessors, MBED's threat model includes the possibility of both malicious applications and a compromised OS, and considers attacks across the hardware security boundary. Given these capabilities, MBED can provide confidentiality and integrity for secrets and code operating on the secure side by preventing unauthorized code execution. It can also provide an availability guarantee, since the architecture allows secure side code to preempt the application or "non-secure" OS at any time.

Finally, a TCB on the secure side can implement software isolation between multiple secure execution contexts, effectively isolating secure components, such as cryptographic APIs or precious machine learning models, from each other if they are developed by individual stakeholders. While the defense in depth provided by this mechanism is excellent in the context of providing tiered security, it requires significant integration with the underlying architecture and leaves the non-secure operating system vulnerable to attack unless it builds its own security guarantees or moves the bulk of its functionality to the secure side.

4 TOCK THREAT MODEL

This section describes the intended threat model of Tock – not a snapshot of the currently supported threat model. The small differences between the intended and currently supported models are merely the result of unfinished development, as discussed in the Tock repository (https://github.com/tock/tock/tree/master/doc/threat_model).

4.1 Tock OS Overview

Tock is an embedded OS for low-power constrained platforms with support for multi-programming with mutually distrusting applications. Tock prioritizes isolation between applications, the kernel, and partially untrusted kernel "capsules", such as device drivers. The Tock kernel itself is entirely written in Rust [15], a language with strong memory- and thread-safe guarantees, and as a result, capsules are isolated from the core kernel using Rust's type system with no runtime cost. Tock also supports hardware memory isolation mechanisms that are used to isolate each application from each other and the kernel. Tock supports ARM Cortex-M based and RISC-V based microcontrollers.

Tock is not merely a research artifact. For example, Tock has been deployed in city-scale sensing research [1]. Tock is currently in use in an open-source security key released by Google [10], and in the open source root-of-trust being developed by the OpenTitan project [12].

4.2 Trust and Isolation

Tock focuses on providing the mechanisms required to build secure systems. The Tock threat model defines three levels of trust which can be prescribed to different software components in the system. To make these divisions useful, the threat model focuses on defining isolation guarantees for software components at different trust levels.

There are 3 types of software components in Tock – userspace applications, kernel capsules, and the core kernel (see Figure 1). Tock apportions trust to these components as follows:

1. **Completely untrusted** (by kernel, each other): Applications
2. **Trusted for liveness, software enforcement of memory safety**: Kernel capsules (networking stack, console, sensor drivers, virtualization layers), chip specific code
3. **Completely trusted**: Core kernel (board/architecture specific code, scheduler, configuration)

Based on these levels, Tock provides specific isolation guarantees:

1. Application secrets may not be modified or accessed by other applications, unless explicitly shared (e.g. via IPC).
2. Capsules can only access application secrets when specifically allowed by the application, or when granted explicit capabilities by the kernel.
3. Kernel secrets may not be accessed by applications or by kernel capsules these secrets are not directly shared with.
   - This means, for example, that secrets held by a radio capsule will not be readable by another capsule unless the radio directly exports them (such as to the SPI virtualization capsule used to provide communication with the hardware radio).
   - Kernel data cannot be modified by applications or capsules except through APIs intentionally exposed by the owning code.
4. Applications cannot cause the system to fault.
5. Applications cannot perform denial-of-service attacks against each other or against the kernel.
   - With an exception for resources that cannot reasonably be shared, explored in Section 4.4.
6. Kernel components must tolerate arbitrary application restarts.

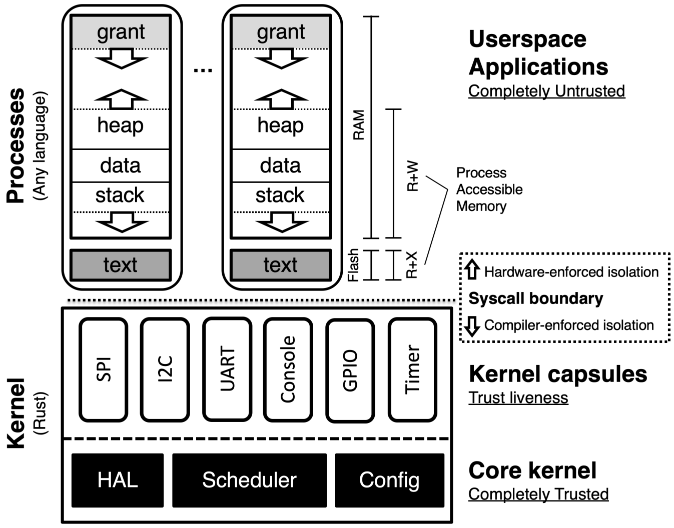

Figure 1: Tock Architecture

These isolation guarantees are met through a combination of memory safety and careful virtualization of shared resources. Sometimes, these virtualized interfaces are insufficient for certain use cases, and capsules or applications require more direct access to hardware. The method by which Tock safely allows such privileged access is discussed in Section 5.

4.3 Memory Safety

The confidentiality and integrity guarantees described above are partially the result of the memory safety guarantees that Tock provides, enforced for applications by hardware such as an MPU. Kernel capsules, which are more trusted than applications, are still prevented from violating memory safety's confidentiality and integrity guarantees (described in 2.2.2) by language sandboxing. The use of the `unsafe` keyword is forbidden in capsule code as enforced by the lint `#![forbid(unsafe_code)]`. Of course, this means that compiler bugs, bugs in the Rust language, or bugs in libraries or the core kernel could all allow an attacker to bypass this guarantee. Finally, the core kernel is not bound by any of these restrictions, and manual auditing of all uses of `unsafe` in the core kernel is required to ensure these guarantees are met.

4.4 Virtualization

In addition to memory safety, the confidentiality and integrity described above is achieved by strict rules for kernel capsules that virtualize shared resources. These rules state that kernel capsules with multiple clients should not share data between their clients (except the IPC capsule), and that data from a client should not be shared with a capsule the client is unaware of. Accordingly, when application buffers are shared with capsules, capsules must only share the buffer with lower-level components necessary for completing some associated functionality. For example, a capsule providing virtualized access to a piece of hardware may pass the buffer to the driver for that hardware. If a capsule copies application data into a buffer, that buffer must be wiped before it is shared with another client. Virtualization capsules must be manually verified as providing these guarantees. Tock already provides vetted virtualization capsules for many common resources, such that this requirement does not significantly impact most uses of third-party code, which would sit on top of these already verified virtualization capsules.

The availability guarantees that Tock provides also depend on effective virtualization. Applications are prevented from starving other applications or capsules of the CPU thanks to a preemptive scheduler. For other limited hardware or software resources, capsules provide virtualized access to both applications and other kernel capsules. These virtualization capsules are expected to prevent starvation of resource requests when the semantics of the operation allow it. This often manifests as requiring round-robin scheduling of access to shared resources. When it is not possible to prevent starvation – such as shared resources that may need to be locked indefinitely for practical use – components have two options:

- Allow resource reservations on a first-come, first-served basis. For resources distributed in this manner, it is recommended that reservations are requested immediately after boot, such that failures due to contention surface immediately.
- Block access to the API behind a kernel capability, and require board integrators to distribute this capability to at most one capsule at a time. This precludes use of the API by applications.

Notably, Tock components do not need to guarantee fairness – a UART virtualization layer may allow capsules/apps using large buffers to see higher throughput than those using small buffers.

While virtualization does impose some overhead – both code size and performance – it is not required for components which are not shared by multiple apps/capsules. In practice, this overhead has not precluded Tock's use in real systems like those mentioned in section 4.1

One other feature supporting Tock's availability guarantees is the lack of dynamic allocation in the kernel – while the Tock threat model does allow for the possibility that kernel capsules may fault the kernel, it attempts to limit the likelihood of this happening unexpectedly by removing the possibility for RAM exhaustion causing a fault. Stack overflows are still possible, however, as no bounds on recursion or the size of parameters passed on the stack currently exist.

4.5 Isolation explicitly not provided

In the general case, no guarantees of side channel isolation are provided, and apps should not rely on them. In practice, many side channels are simply too expensive to mitigate, but specific components can still provide side-channel mitigation of their own. A constant-time guarantee might be implemented for a cryptography API, for instance. Outside of this, however, it is worth noting that Tock does not hide application CPU usage from others, and does not hide the size of application data regions – in both cases the cost of mitigating these channels was deemed too high given the low-sensitivity signals these channels represent.

4.6 External Trust

4.6.1 Third Party Dependencies. The Tock threat model currently forbids the use of third party dependencies in the Tock kernel except for the Rust libcore, a minimal, dependency free foundation of The

Rust Standard Library, which hasn't been audited by the Tock core team. Third party dependencies may be added to Tock in the future, but their code will likely be audited if this occurs. Applications may use any dependencies, as application code is untrusted.

4.6.2 Application Loader. An application loader is a mechanism for adding applications to a Tock system. This mechanism can be an application loader running on a host system that uses programming interfaces to manipulate applications in nonvolatile storage. It can also be a capsule that acts as a kernel-assisted installer that receives an application over USB or the network and writes it to flash, or a monolithic image that bundles the kernel with apps. The application loader must be trusted, as it must have the ability to read/write arbitrary memory for any application. Therefore, the loader must be trusted to not modify, erase, or exfiltrate application data. The loader does not have to be trusted to not modify the kernel, as other mechanisms may be used to protect the kernel. Tock relies on application headers stored with applications to verify application integrity and to store application metadata. These headers are treated as untrusted, and any application loader must verify them before installing applications.

5 PRIVILEGED OPERATIONS

Maintaining isolation between kernel components and applications means sensitive operations should be marked as privileged, such that they can only be called within the TCB. Some examples include:

- Direct memory access (raw pointer reads/writes)
- Direct Bus access (SPI, I2C, etc.)
- Indefinite use of limited resource (indefinite ADC sampling)
- Start/stop processes or read process metadata
- Modify hardware registers
- Open network connections
- Invoking kernel main loop

However, certain kernel components (such as those that virtualize these interfaces) will inevitably require access to some of these privileged operations, and maintaining a small trusted computing base means these components should still be isolated from the core kernel. In such cases, it is important that an OS provide safe, easily auditable mechanisms for less trusted kernel components to perform privileged operations. It should be clear to board integrators when these mechanisms are used, and should be easy to audit the portions of kernel components that perform privileged operations. In Tock, privileged operations are exposed to less trusted kernel components (capsules) in two ways:

(1) **Privileged Object** By hiding privileged functionality in non-public methods on objects, references to which are only given to kernel components allowed this functionality. These privileged objects are either not visible outside the core kernel crate, or their construction relies on the `unsafe` keyword, and thus cannot be instantiated outside the TCB (see Listing 2).

(2) **Capabilities** By passing references to capabilities, 0-size types required as input to otherwise globally-callable functions or methods on widely shared object (such as the core Kernel struct). Capabilities can only be created by using `unsafe`, ensuring that any capabilities used are created by the core kernel. An example can be seen in Listing 1. Notably, while the capabilities are 0-size, references to these capabilities are still pointer-size and allocated on the stack during function calls. The 0-size feature of capabilities ensures objects that store these capabilities do not require extra space in flash or RAM.

```
//! main.rs
pub struct PMCap;
unsafe impl ProcessManagementCapability for PMCap {}
let console = ProcessConsole::new(console_uart, PMCap);

//! kernel/src/scheduler.rs
// Fault all apps for debugging purposes
pub fn hardfault_all_apps<C: ProcessManagementCapability>
                     (&self, _c: &C) {
  for p in self.processes.iter() {
    p.map(|proc| proc.set_fault_state());
  }
}
```

Listing 1: Capability Example. Most capsules should not be able to stop/modify processes. Tock enforces only capsules with the `ProcessManagementCapability` can do so. Board integrators may audit or exclude capsules with this capability.

```
//! main.rs
// Create a capsule to virtualize lone hardware alarm.
let mux_alarm = static_init!(
    MuxAlarm<'static, rv32i::machine_timer::MachineTimer>,
    MuxAlarm::new(&arty_e21::timer::MACHINETIMER)
);
Alarm::set_client(&arty_e21::timer::MACHINETIMER, mux_alarm);

//! chips/arty_e21/src/timer.rs
// Exposes safe methods for modifying machine timer
// Code in chips/ is not visible to capsules
pub static mut MACHINETIMER: MachineTimer =
  MachineTimer::new(MTIME_BASE);

const MTIME_BASE: StaticRef<MTimerRegisters> = unsafe {
  StaticRef::new(0x0200_0000 as *const MTimerRegisters)
}; // Direct memory access/casts require unsafe
```

Listing 2: Privileged Object Example. To prevent interrupt stealing, Tock enforces that only a capsule given a reference to the hardware timer object is able to control the timer. Board integrators verify this reference is only given to a single MuxAlarm capsule which virtualizes access to the hardware timer for use by other capsules.

We use the "privileged object" method for exposing safe interfaces to hardware objects (GPIO pins, buses, hardware timer, radio, etc.). Typically this involves constructing a safe interface for reading/writing hardware registers that control these peripherals. This safe interface is implemented as methods on the privileged object, thus access to the interface requires a reference to that object.

We use the "capabilities" method when it does not make sense to represent the restricted functionality by limiting visibility to an object. For example, if the method must be implemented on an otherwise broadly shared object, like the `Kernel` object, restricting visibility does not work, so we use capabilities as a 0-overhead mechanism for restricting access to certain methods. Alternatively, for instances where a globally callable function is required (such as a `new()` function used to create an object, capabilities can be used to restrict access to this global function (example: `IPC::new()`).

In both cases, the passing of ether a privileged object or a capability to a kernel component serves as a clear signal to a board integrator that this kernel component has elevated privileges. Further, in both cases the core kernel is still responsible for determining exactly how these privileges can be used, as privileged capsules are still limited by whatever API is exposed by the core kernel.

6 DISCUSSION

This section discusses pros and cons of Tock's threat model from the perspectives of security and usability. This analysis is based on the assumption that security typically means confidentiality and integrity of application data, combined with high availability of application services. It is also based on the assumption that board integrators will be the party ultimately responsible for determining the threat model faced by a given application, but that board integrators may not be intimately familiar with the kernel.

6.1 Advantages of the Tock Threat Model

6.1.1 Auditability. The Tock threat model simplifies the process of security audits. The two tiers of trust in the kernel reduce the size of the TCB to around 12k LoC out of the total 45k LoC in the kernel. These tiers enable safe import of service and driver code without imposing the overhead of process isolation. Additionally, Rust is uniquely auditable among popular, well supported languages that do not require garbage collection. Rust identifies potentially dangerous code via the `unsafe` keyword, which enables more targeted auditing. Rust's combination of type and memory safety with aggressive linting further helps minimize the potential for security issues from mistakes in non-malicious code. Further, the Tock threat model allows audits of applications independent of the underlying platform or co-located applications. Then, memory safety between most kernel components greatly reduces the need to verify additional kernel components will steal secrets, which allows auditors to focus on capsules that can perform privileged operations. Finally, Tock's modular kernel components (capsules) allow auditors to focus on auditing only the specific components of the kernel that are used in a given setting.

6.1.2 Isolation and Usability. Tock's approach to trust and isolation also improves usability for all stakeholders. Isolation of capsules allows kernel developers to extend the kernel at much lower risk to integrity and confidentiality, speeding development. Isolation of applications makes adding new applications to a given deployment straightforward, as it removes concerns that additional applications will break the existing deployment. The use of language-based isolation within the kernel removes the overhead of reconfiguring hardware isolation and the risk of misconfiguration breaking memory safety. The low impact methods used to enable privileged actions within the kernel make it easy for board integrators to understand the impact of including certain kernel components without forcing integrators to choose between performance and security. Type-based zero-size capabilities move almost all memory/CPU overhead of security checks to compile-time, valuable on low resource devices (unforgeable capabilities like those used in SeL4 rely on MMU/rings for protection, and thus impose a runtime cost). Finally, separating applications from the kernel means that updating or replacing applications cannot render systems inaccessible and unrecoverable. This encourages frequent updates and reduces the binary sizes that must be delivered to replace functionality.

6.1.3 Availability and Fault Tolerance. Application isolation also allows for much simpler failure isolation, improving availability, a core component of security. Additionally, errors in kernel capsules should only effect applications that rely on those capsules. For example, if the onboard ADC fails, leading to an error in the ADC capsule, a separate application not using the ADC should continue functioning correctly.

6.2 Disadvantages of the Tock Threat Model

6.2.1 Reliance on Rust. While Rust is responsible for many of the strengths of the Tock threat model, this does come with a couple downsides. It means that all kernel extensions must be written in Rust, which precludes the use of many well tested libraries targeted at C-based systems. Additionally, the inability to use `unsafe` in capsule code is inconvenient at times, precluding certain optimizations based on raw pointer manipulation or requiring trial-and-error approaches to hit compiler optimization paths.

6.2.2 Downsides of high modularity. While the modularity of Tock has significant benefits for auditability and usability, it also makes cross capsule optimizations difficult (such as optimizing the interaction between a clock controller and a bus controller when the clock controller may not always be used). This degree of modularity also leads to long board initialization code, much of which is repeated across platforms. The high degree of isolation between applications can also be inconvenient when applications are intended to be used more as individual threads of a single application, as single-core context switching is expensive with MPU reconfiguration.

7 CONCLUSION

Achieving security requires providing useful security abstractions to developers. However, different components of embedded systems need different security abstractions. In particular, useful abstractions require more than two levels of trust, and require reliable isolation guarantees. We present the threat model for Tock, and show how multiple levels of trust enable useful, modular embedded security.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1931750 "Secure Smart Machining." This work was also supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by the National Science Foundation under grant CNS-1824277.

REFERENCES

[1] Joshua Adkins, Branden Ghena, Neal Jackson, Pat Pannuto, Samuel Rohrer, Bradford Campbell, and Prabal Dutta. 2018. The signpost platform for city-scale sensing. In *2018 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 188–199.

[2] Roger Alexander, Anders Brandt, JP Vasseur, Jonathan Hui, Kris Pister, Pascal Thubert, Philip Levis, Rene Struik, Richard Kelsey, and Tim Winter. 2012. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550.

[3] Manos Antonakakis, Tim April, Michael Bailey, Matthew Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, and et al. 2017. Understanding the Mirai Botnet. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC'17)*.

[4] Arm Holdings. Accessed: 2020-02-24. Platform Security Architecture. www.arm.com/why-arm/architecture/platform-security-architecture.

[5] Arm Limited. Accessed: 2020-02-24. Mbed. https://os.mbed.com/.

[6] Anish Athalye, Adam Belay, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2019. Notary: a device for secure transaction approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 97–113.

[7] Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, Matthias Wahlisch, and Thomas C Schmidt. 2013. RIOT OS: Towards an OS for the Internet of Things. In *2013 IEEE conference on computer communications workshops*. 79–80.

[8] Adam Dunkels. 2001. *Design and Implementation of the lwIP TCP/IP Stack*. Technical Report. Swedish Institute of Computer Science.

[9] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *29th annual IEEE international conference on local computer networks*. IEEE, 455–462.

[10] Google. Accessed: 2020-03-02. OpenSK. github.com/google/OpenSK.

[11] Google. Accessed: 2020-03-02. OpenThread. https://openthread.io/.

[12] Google. Accessed: 2020-03-02. Tock-on-Titan. github.com/google/tock-on-titan.

[13] M. M. Hossain, M. Fotouhi, and R. Hasan. 2015. Towards an Analysis of Security Issues, Challenges, and Open Problems in the Internet of Things. In *2015 IEEE World Congress on Services*. 21–28. https://doi.org/10.1109/SERVICES.2015.12

[14] Galen Hunt, George Letey, and Ed Nightingale. 2017. *The seven properties of highly secure devices*. Technical Report MSR-TR-2017-16. Microsoft Research.

[15] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*.

[16] Paul C Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*. 104–113.

[17] Philip Levis. 2012. Experiences from a Decade of TinyOS Development. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*.

[18] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. 2005. TinyOS: An operating system for sensor networks. In *Ambient intelligence*. 115–148.

[19] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. 234–251.

[20] Charlie Miller and Chris Valasek. 2013. Adventures in automotive networks and control units. *Def Con* 21 (2013), 260–264.

[21] Timothy Trippel, Ofir Weisse, Wenyuan Xu, Peter Honeyman, and Kevin Fu. 2017. WALNUT: Waging doubt on the integrity of MEMS accelerometers with acoustic injection attacks. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 3–18.

[22] Jacob Wurm, Khoa Hoang, Orlando Arias, Ahmad-Reza Sadeghi, and Yier Jin. 2016. Security analysis on consumer and industrial IoT devices. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 519–524.

[23] Zephyr Project. Accessed: 2020-02-24. Security Best Practices. https://github.com/zephyrproject-rtos/zephyr/wiki/Security-Best-Practices.

[24] Zephyr Project. Accessed: 2020-02-24. Zephyr Security Overview. https://docs.zephyrproject.org/latest/security/security-overview.html.