Multiprogramming a 64kB Computer Safely and Efficiently

Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, Philip Levis

Rebecca Kempe

Tock is:

- an **open source**, **security-focused** embedded operating system (OS)
- used in Google Ti50 Security Chip Firmware (Chromebooks), Google Open Security Keys, Microsoft Pluton Security Processor (Copilot+ Windows 11 PCs)
- written entirely in **Rust** (was a pioneer in this regard!)
- a project that originated in (and is still largely run by) academic research circles

A Networked Embedded System Platform for the Post-Mote Era (Sensys 2014)

Pat Pannuto, Michael P. Andersen, Tom Bauer, Bradford Campbell, **Amit Levy**, David Culler, Philip Levis, Prabal Dutta

Ownership is Theft: Experiences Building an Embedded OS in Rust (PLOS 2015)

Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, Pat Pannuto

The Case for Writing a Kernel in Rust (APSys 2017)

Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, Philip Levis

Multiprogramming a 64 kB Computer Safely and Efficiently (SOSP 2017)

Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, Philip Levis

Tiered Trust for Useful Embedded Systems Security (EuroSec 2022)

Hudson Ayers, Prabal Dutta, Philip Levis, **Amit Levy**, Pat Pannuto, Johnathan Van Why, Jean-Luc Watson

Tock: From Research to Securing 10 Million Computers (SOSP 2025)

Leon Schuermann, Brad Campbell, Branden Ghena, Philip Levis, **Amit Levy**, Pat Pannuto

A Networked Embedded System Platform for the Post-Mote Era (Sensys 2014)

Pat Pannuto, Michael P. Andersen, Tom Bauer, Bradford Campbell, **Amit Levy**, David Culler, Philip Levis, Prabal Dutta

Ownership is Theft: Experiences Building an Embedded OS in Rust (PLOS 2015)

Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, Pat Pannuto

The Case for Writing a Kernel in Rust (APSys 2017)

Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, Philip Levis

Multiprogramming a 64 kB Computer Safely and Efficiently (SOSP 2017)

Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, Philip Levis

Tiered Trust for Useful Embedded Systems Security (EuroSec 2022)

Hudson Ayers, Prabal Dutta, Philip Levis, **Amit Levy**, Pat Pannuto, Johnathan Van Why, Jean-Luc Watson

Tock: From Research to Securing 10 Million Computers (SOSP 2025)

Leon Schuermann, Brad Campbell, Branden Ghena, Philip Levis, **Amit Levy**, Pat Pannuto

Questions

1. What problems was Tock trying to solve?
2. What changes enabled Tock to be different from its predecessors?
3. What does Tock do that previous embedded OSes couldn't do?
4. What design choices does Tock make to achieve its functional goals?
5. How do Tock's design choices help it achieve its security goals?
6. What else can we learn from Tock?

Questions

1. What problems was Tock trying to solve? 2. What changes enabled Tock to be different from its predecessors?	**Context: Why?**
3. What does Tock do that previous embedded OSes couldn't do? 4. What design choices does Tock make to achieve its functional goals? 5. How do Tock's design choices help it achieve its security goals?	**Design Choices: How?**
6. What else can we learn from Tock?	**Conclusion: So what?**

Context:

What problems was Tock trying to solve?

“Urban sensing”

- Tock was originally designed to support “low power urban sensing systems” and wearable electronics
- Urban sensing involves using wireless sensor networks to collect data
- Many constraints:
  - Power, size, and costs are limited
  - Hardware and software used must be relatively low-maintenance
  - Sensors may be left unattended for extended periods of time
  - Hardware may be installed in difficult to reach locations
  - Much of the available hardware was single purpose

Signpost

- Idea: How do we easily collect multiple types of data at city scale?
- Goals: solar-powered, modular, easy to install, “one system”
- This came with many hardware challenges… and by extension, software challenges…

Images are from the Signpost GitHub repository: https://github.com/lab11/signpost?tab=readme-ov-file

Challenges

- Multiple applications need to run on the same hardware
- OS needs to be able to respond to events from multiple sensors
- Many different developers could be writing applications
- Different modules may require new drivers to be created
- Should not have to recompile the OS each time a new application is added
- Cannot afford to crash or reboot the entire system if something bad happens to an application during operation

Embedded hardware is limited

- RAM is limited, which makes heap fragmentation dangerous – often means that OSes will statically allocate memory
  - Leads to problems where memory allocation is overestimated and concurrency is impeded
- Many chips do not have memory management units (MMUs)
  - No virtual addressing
  - Encourages monolithic designs where the application and kernel share memory regions – this requires installing the app and kernel together
- Fault isolation capabilities are limited
  - Everything must be trusted – if one process crashes, so does everything else

Five desired properties

Dependability: Memory usage should be predictable at compile time, since embedded applications typically run without intervention for long periods of time.

Concurrency: Many embedded applications have tight energy budgets. Increasing concurrency improves energy efficiency, allowing the embedded device to spend more time in a sleep state.

Memory Efficiency: Embedded devices typically have very little RAM, so embedded OSes should minimize the amount of RAM allocated to exactly what the application needs.

Fault Isolation: Memory faults between applications should be isolated so that applications running on the same system cannot corrupt each other's state.

Loadable Applications: Wearable electronics (e.g. sports watches) and city sensing infrastructure may run applications written by different groups; installing new applications should not require recompiling the whole OS.

Table 1: Existing embedded OSes do not deliver all five of the desired properties.

System	Dependability	Concurrency	Memory Efficiency	Fault Isolation	Loadable Applications
Arduino			✓		
RIOT OS			✓		
Contiki		✓	✓		
FreeRTOS	✓	✓			
TinyOS	✓	✓	✓		
TOSThreads	✓	✓			✓
SOS		✓	✓		✓
Tock	✓	✓	✓	✓	✓

(Adapted from Table 2 in Levy et al., SOSP 2017)

What changed?

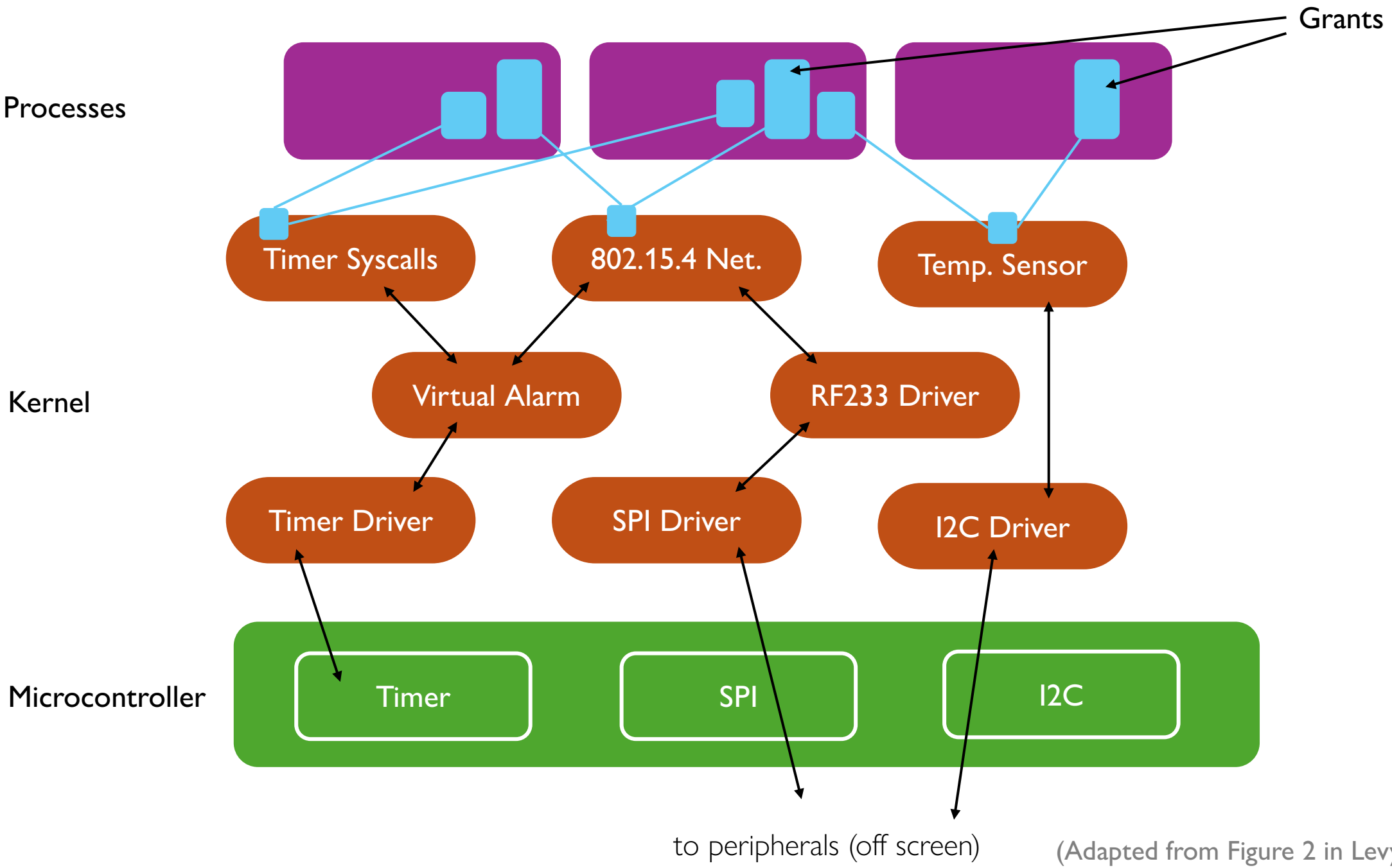
- Microcontrollers with memory protection units (MPUs) became available, which can be used for memory isolation in hardware
  - This enables process isolation in Tock
- Advances in language design: the Rust language was being developed
  - Rust was starting to mature: Rust 0.1 (first public release) came out in 2012, Rust 1.0 came out in 2015
  - Rust's type system enables kernel functionality isolation in Tock

Design Choices:

Capsules, processes, and grants

Tock's Design

- Tock has a hybrid kernel design: it's not quite a monolithic kernel, but it isn't a microkernel either
- Goals: isolate things from each other, minimize trust, be efficient with memory usage
- Capsules are kernel abstractions that isolate kernel functionality (e.g. drivers) from each other
- Processes isolate different applications from each other.
- Grants eliminate the need for a unified kernel heap by allocating kernel memory directly in the process for which it is needed



(Adapted from Figure 2 in Levy et al., SOSP 2017)

What are capsules?

- Capsules are kernel components and are written in Rust
- A capsule is a Rust struct, including its fields, methods, and static variables
- Capsules are isolated from each other using the Rust language type and module systems
  - Capsules cannot access data in other capsules
  - Capsules cannot access process memory
- Capsules prevent direct memory access (DMA) from occurring
  - This is a common source of kernel memory violations

Types of capsules

- Core kernel capsules, such as the process scheduler (semi-trusted)
- Capsules that interact directly with hardware (semi-trusted)
- System call capsules, which translate between application requests and multiplexing capsules (untrusted*)
- Multiplexing capsules, such as hardware-agnostic peripheral drivers (untrusted*)

* In this context, untrusted means that they are not allowed to contain unsafe code.

But we do actually need to trust capsules… sort of

- Capsules share a single stack with each other and the core kernel, and are isolated from each other using the Rust type and module system
- Capsules are scheduled cooperatively, so they need to be trusted for liveness
  - Cooperative scheduling is when the scheduler starts the process, but **the process has control over when/if it yields back to the kernel**
  - A liveness property is an **assurance that something good will happen** within a program (e.g. the program will always terminate)
  - Poorly designed Tock capsules can exhaust CPU resources
- Capsules provide memory efficiency, dependability, concurrency, and fault isolation, but **not** loadable applications

What are processes?

- Similar to processes on other OSes
- Each process has its own memory space
- Processes communicate with each other through inter-process communication (IPC) and with the kernel through system calls
- Processes are pre-emptively scheduled – this means the scheduler starts and stops each process according to some plan
- Different from Linux processes in that they do not use virtual memory and that the system calls are non-blocking
  - Memory isolation is done in hardware using the MPU
  - Processes don't necessarily stop while waiting for system calls to return

Process advantages

- Hardware isolation (as opposed to language isolation) means applications can be written in any language
- Pre-emptive scheduling prevents applications from starving the CPU
- Processes provide all five properties: memory efficiency, dependability, concurrency, and fault isolation, and loadable applications

What are grants?

- Sometimes, the kernel needs dynamic resources to respond to unanticipated requests from a process
  - For example, a timer driver may need to allocate space for metadata every time an application creates a new timer
- Grants are a special section of kernel heap located within each individual process's memory space
- Grant allocations for one process do not affect other processes
- However, this does mean that capsules need to iterate across multiple data structures to get the state across all processes

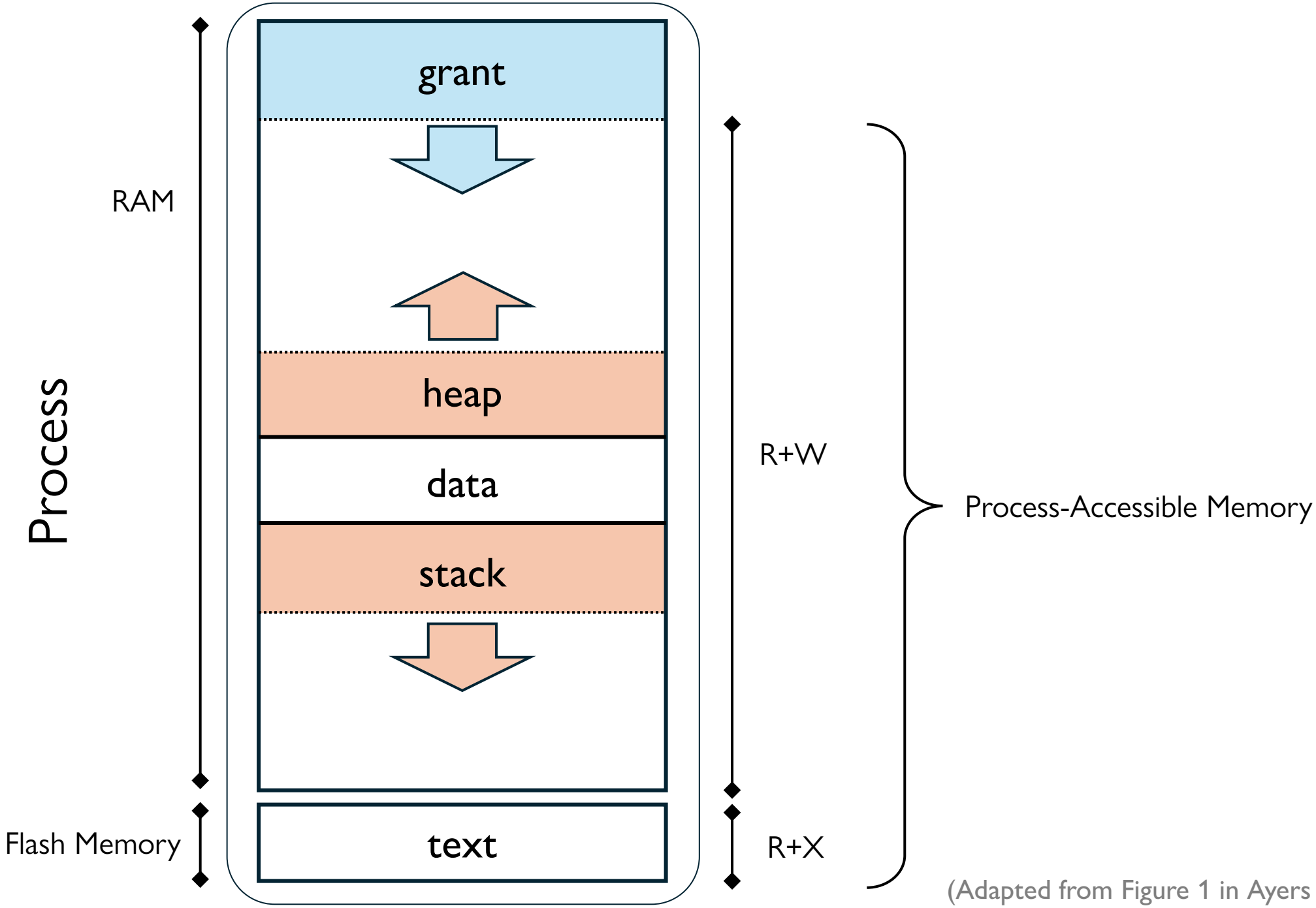
How grants work

- When a capsule needs extra memory, it calls an API that checks if the process is valid and allocates memory from the top of the process block
- The locations of the memory breaks in each process are stored in the process table
- These allocations are isolated using the type system; references created using a grant cannot escape
- Grants are deallocated when an owned value goes out of scope or when the process dies

Table 2: Grant API table

Grant<T>	Provides access to grant memory of a particular type
create()	Reserves an ID for a new grant, which is used to allocate space for it in process memory
enter(proc_id, closure)	Yields the **`Owned`** value from the specified process to the given closure; allocates new grant memory if necessary
each(closure)	Iteratively yields the **`Owned`** value from each process if already allocated; does not allocate new grant memory
Owned<T>	A reference to allocated grant memory for a process
deref()	Dereferences the value (syntactic sugar for this in Rust is the * operator)
drop()	Frees allocated space; this is automatically called when the **`Owned`** value goes out of scope

(Adapted from Table 4 in Levy et al., SOSP 2017)



(Adapted from Figure 1 in Ayers et al., EUROSEC 2022)

Wait a sec, how does any of this have anything to do with security?

Three-tiered trust and mutually distrustful applications

Stakeholders

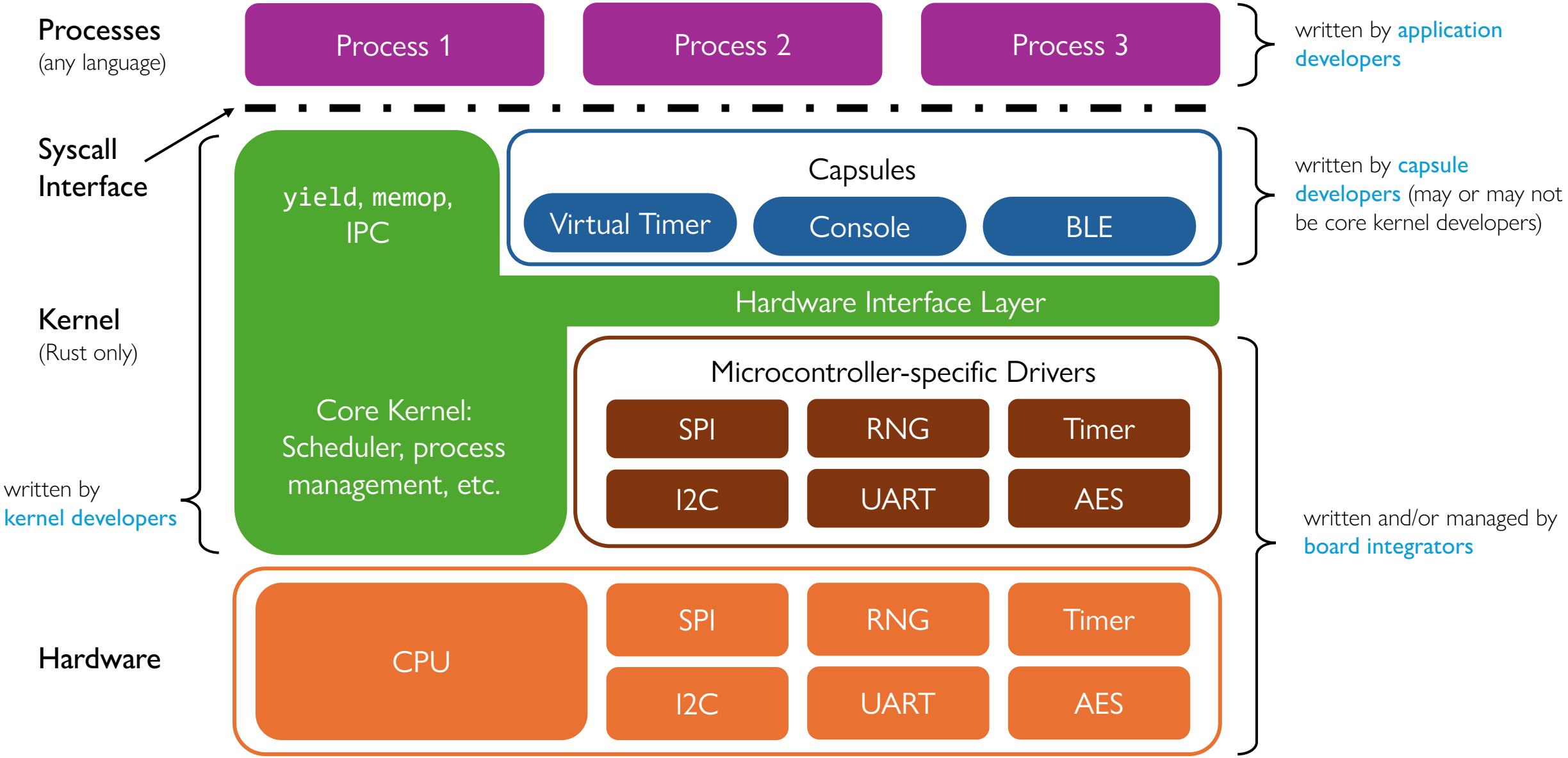
- Users
- Application developers
- Capsule developers
- Core kernel developers
- Board integrators

Stakeholder responsibilities

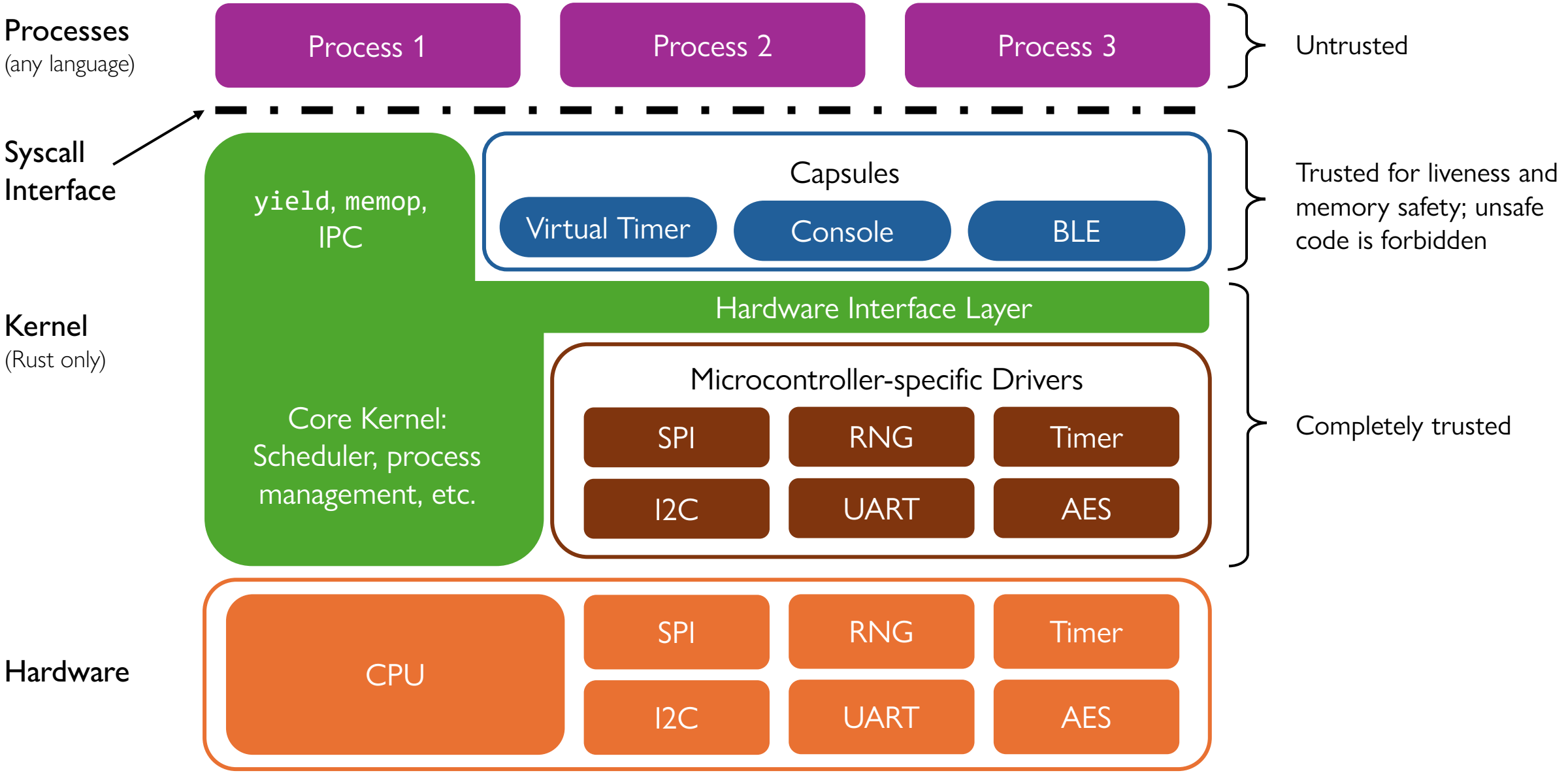
Application developers: Applications should be isolated from each other and from the kernel (though they can trust the kernel).

Capsule/kernel developers: Limit the scope of the trusted computing base (which contains unsafe code); modularize non-core kernel functionality (i.e. create separate capsules for these); limit trust in non-core kernel functionality.

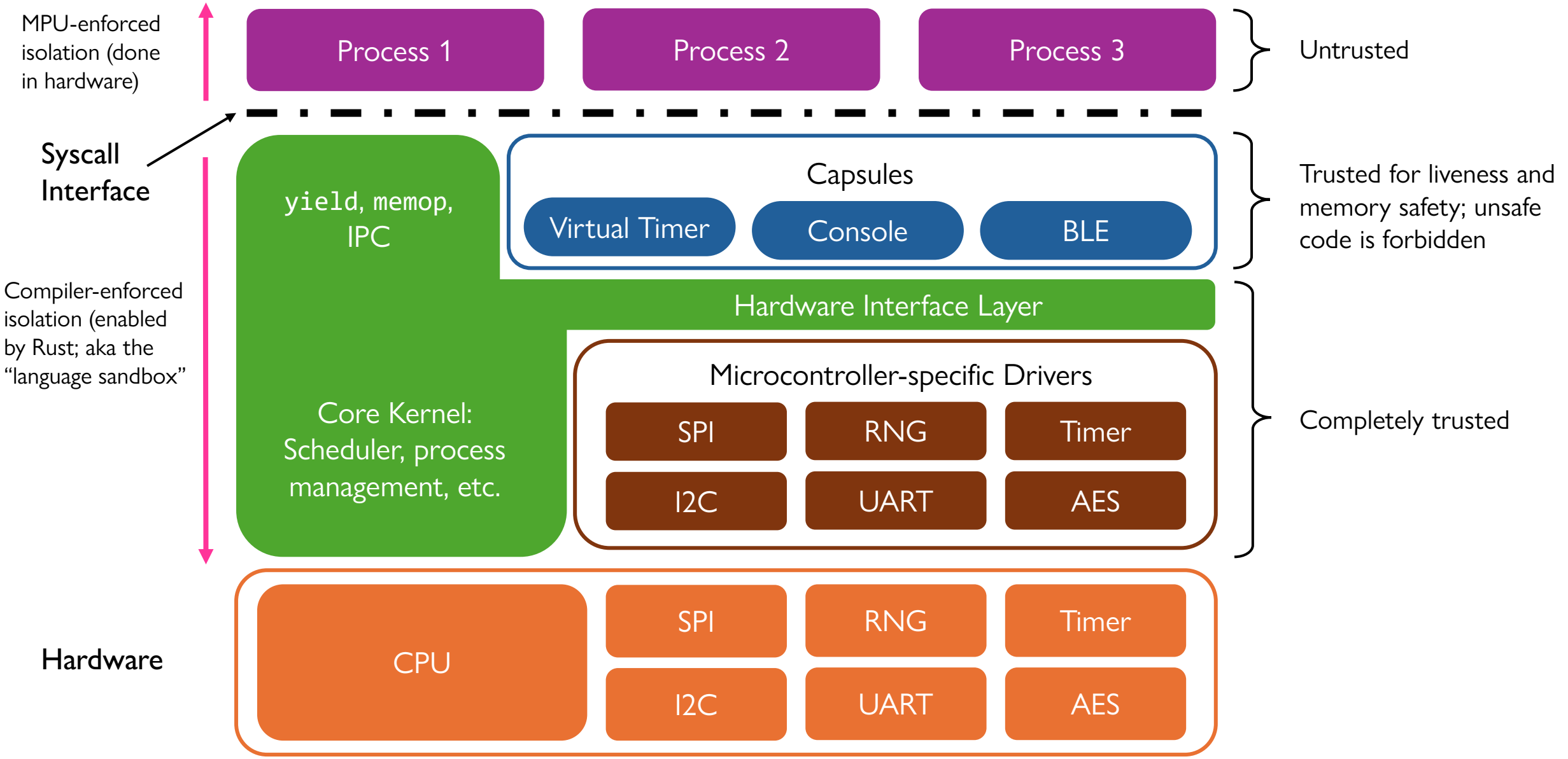
Board integrators: It should be clear which kernel components have additional privileges, and what those privileges are – board integration directly influences the trusted computing base.



(Adapted from Figure 2 in Schuermann et al., SOSP 2025)



(Adapted from Figure 2 in Schuermann et al., SOSP 2025)



(Adapted from Figure 2 in Schuermann et al., SOSP 2025)

Tock Isolation Guarantees

1. Application secrets may not be modified or accessed by other applications, unless explicitly shared.
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.
4. Applications cannot cause the system to fault.
5. Applications cannot perform denial-of-service attacks against each other or against the kernel.
6. Kernel components must tolerate arbitrary application restarts.

(Ayers et al., EUROSEC 2022)

Conclusion:

What can we learn from Tock?

Lessons Learned

- Secure design depends both on hardware and on software capabilities – sometimes progress can't be made until the right technology exists
- Modularity is key when designing for security; it helps minimize what needs to be trusted
- Different research projects can cross-pollinate in interesting ways, and evolve in unexpected ways
- The grants idea was probably the most interesting in this paper – it was a very creative solution to an old problem
- Usable software can come out of academia; however, it requires a lot of collaboration (and time!)

References

[1] J. Adkins et al., "The signpost platform for city-scale sensing," in 2018 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), 2018, pp. 188–199. DOI: 10.1109/IPSN.2018.00047

[2] H. Ayers et al., "Tiered trust for useful embedded systems security," in Proceedings of the 15th European Workshop on Systems Security, ser. EuroSec '22, Rennes, France: Association for Computing Machinery, 2022, pp. 15–21, ISBN: 9781450392556. DOI: 10.1145/3517208.3523752 [Online]. Available: https://doi.org/10.1145/3517208.3523752

[3] A. Levy et al., "Multiprogramming a 64kb computer safely and efficiently," in Proceedings of the 26th Symposium on Operating Systems Principles, ser. SOSP '17, Shanghai, China: Association for Computing Machinery, 2017, pp. 234–251, ISBN: 9781450350853. DOI: 10.1145/3132747.3132786 [Online]. Available: https://doi.org/10.1145/3132747.3132786

[4] P. Pannuto et al., "A networked embedded system platform for the post-mote era," in Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems, ser. SenSys '14, Memphis, Tennessee: Association for Computing Machinery, 2014, pp. 354–355, ISBN: 9781450331432.DOI: 10.1145/2668332.2668364 [Online]. Available: https://doi.org/10.1145/2668332.2668364

[5] L. Schuermann, B. Campbell, B. Ghena, P. Levis, A. Levy, and P. Pannuto, "Tock: From research to securing 10 million computers," in Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, ser. SOSP '25, Lotte Hotel World, Seoul, Republic of Korea: Association for Computing Machinery, 2025, pp. 36–49, ISBN: 9798400718700. DOI: 10.1145/3731569.3764828 [Online]. Available: https://doi.org/10.1145/3731569.37648282