Can We Put the “S” Into IoT?

Gernot Heiser
UNSW Sydney, Australia
0000-0002-7069-0831

Lucy Parker
UNSW Sydney, Australia
lucy.parker@student.unsw.edu.au

Peter Chubb
UNSW Sydney, Australia
peter.chubb@unsw.edu.au

Ivan Velickovic
UNSW Sydney, Australia
i.velickovic@unsw.edu.au

Ben Leslie
Breakaway Pty Ltd, Australia
benno@brkawy.com

***Abstract*—Security of IoT systems is often weak or absent, resulting in systems being compromised. We present the seL4 Core Platform, an operating-system framework that leverages the formally verified security enforcement of the seL4 microkernel to enable the construction of secure-by-design IoT systems, and even enables an incremental cyber retrofit of existing systems. The framework is designed to make its formal verification tractable. An initial evaluation shows that for performance-sensitive high-throughput networking, the platform significantly outperforms Linux.**

***Index Terms*—operating systems; verification; performance; internet of things; microkernel; hypervisor.**

I. Introduction

“In IoT, the ’S’ stands for security” is a meme that has been around since at least early 2017,[1] yet seems as fitting as ever. The reasons for the widespread lack of security in IoT are multiple, including lack of care or awareness of the developers, the classical market failure which rewards security shortcuts that reduce time-to-market, and lack of availability of security-enabling development frameworks that are easy enough to use.

While it is unrealistic to think that there is an easy solution to this combination of challenges, it seems that the last reason in the above list is particularly worthwhile to target: If we can provide a development environment that is easy use for building deployable products, but at the same time provides a good degree of security-by-construction, then it might be possible to significantly raise the bar in IoT security.

While a complete IoT development framework will require significant development effort in itself, once a security-oriented core exists, much of the remaining effort is engineering, based on well-established techniques and re-using established protocols and components (many of them open source).

Here we present such a core framework, based on a highly secure operating system kernel, that can scale to meet the requirements of present and near-future IoT systems. We present this *seL4 IoT framework* in the following sections, specifically:

- the seL4 microkernel, Section II;
- the seL4 core platform (seL4CP), Section III;
- the seL4 device driver framework (sDDF), Section IV
- virtualisation for legacy reuse, Section V.

[1]We would love to credit the person who coined the phrase but were unable to find the original quote.

We also provide a threat assessment for the framework and how this will be impacted by verification activities that are in progress. A preliminary evaluation demonstrates that the security enabled by the framework does not have a detrimental effect on performance, and in fact outperforms Linux. We finally present the state of the implementation and open-source availability of the framework.

II. Background: The seL4 Microkernel

seL4 is a highly secure, high-performance, open-source operating system (OS) microkernel. It is the world’s first OS kernel with a formal proof of implementation correctness [1]. It is also the first OS kernel with formal proof of enforcement of the core security properties of *confidentiality*, *integrity* and *availability*, with the proof extending to the kernel’s executable binary, and the first protected-mode OS with a sound and complete *worst-case execution-time* (WCET) analysis [2]. It furthermore has support for mixed-criticality real-time systems [3]. seL4’s performance is unbeaten and within 25% of the limits imposed by hardware [4].[2] Furthermore, while there now exist other verified OS kernels, seL4 remains the only one using fine-grained access control based on capabilities [5], a key enabler of reasoning about overall system security.

As a microkernel, seL4 only provides fundamental mechanisms, just enough to securely multiplex the hardware between mutually-distrusting components. In particular, it provides none of the OS services that applications depend on, such as file systems, network services and resource mamagement. These must be provided by unprivileged service components (usually called an *OS personality*). seL4 can also serve as a hypervisor to support virtual machines (VMs) that can run a legacy OS.

The advantage of the seL4-based approach is that system services are subject to the kernel-enforced security policies, just as applications (and VMs) are. In fact, the kernel makes no distinction between applications and OS services, resulting in the horizontal system structure shown in Figure 1.

[2]Note that while [4] demonstrates even better performance by utilising virtualisation-support features of the Intel hardware, this approach hijacks hypervisor mode and is therefore unsuitable for a system which needs that mode to support virtual machines.

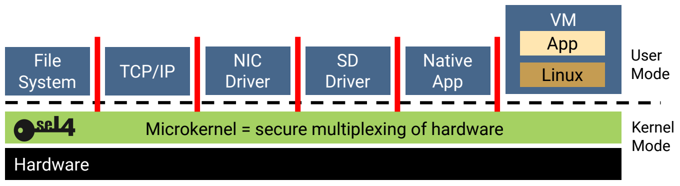

Fig. 1. Structure of an seL4-based system.

As the kernel provides bullet-proof isolation between components, failure of a component can be contained, no matter whether it is a system service or an app. This includes VMs: Virtualization is enabled by a user-mode *virtual-machine monitor* (VMM) component. Each VM has its own, private VMM [6], which is subject to the same isolation as normal usermode components. Hence, the VMM cannot break isolation between VMs, and each VM only needs to trust its own VMM. The kernel does provide communication channels, and the capability-based access control ensures that two components can only communicate if an explicit channel is set up between them, limiting interference between components.

seL4 is designed for real-world use, and has been successfully deployed on real-world embedded and cyberphysical systems [7]. Devices based on seL4 are in day-to-day use in several defence forces, and the kernel is being designed into critical infrastructure and autonomous vehicles.

In summary, while seL4 on its own is not a sufficient platform for IoT development, it provides the secure base on which to build such a platform.

III. The seL4 Core Platform

seL4's API is not only extremely low-level (and thus difficult to use), it is also very general, aiming to support the construction of arbitrary systems on top. This generality is not needed for IoT. By narrowing the application domain, an OS personality can provide simpler abstractions that are easy to understand and use.

The seL4 Core Platform [8] is an OS personality designed to support use cases in the IoT and cyberphysical domains. The core property that leads to a dramatically simpler model is that of a *static architecture*, meaning that the software architecture of the system is fixed at build time. Specifically there is a fixed number of components, each providing some functionality to other components (or the external world).

The seL4CP's abstractions are: *protection domain* (PD), *communication channel* (CC), *memory region* (MR), *notification* and *protected procedure call* (PPC). A *virtual machine* (VM) is a special case of a PD with extra, virtualisation-related attributes.

The PD represents a very simple process abstraction that provides an event-driven programming style, where events are triggered by notifications sent along a CC. A PD can optionally provide a *protected procedure* that can be invoked through a channel by performing a PPC, resulting in a synchronous execution of the protected procedure in its PD. MRs can be combined with CCs to support shared memory, with notifications providing semaphore-like synchronisation.

The original version of the seL4CP was fully static, in that all code had to be fixed at system build time, and PDs could not be restarted. The addition of dynamic features are in progress [9], specifically re-initialising PDs and dynamically loading code into PDs (which suports live code upgrades).

The seL4CP is designed to be itself formally verifiable. Work on its verification is in progress, exploring the use of push-button verification techniques [10].

IV. The seL4 Device Driver Framework

Device drivers are the components of an OS that interface with peripheral hardware, especially I/O devices. As they deal with low-level hardware, they tend to be complex, and, as a result, exhibit high defect densities, more than other OS code [11]. As a result, drivers present a major attack vector: of about 1200 vulnerabilities reported on Linux over the past five years [12], 39% are due to drivers.

In an seL4-based system, a driver, like any other OS service, is a usermode program, with no special privileges other than the ability to access the control registers of the device it drives. This in itself significantly reduces the attack surface.

The seL4 device driver framework [13] further reduces the likelihood of driver bugs by presenting a, compared to Linux or Windows, much simplified programming model for drivers: sDDF drivers implement a simple event loop that eliminates all concurrency control from the driver itself. It also presents a simple, yet low-overhead interface between the driver and the rest of the OS, using shared memory and asynchronous communication based on simple, lock-free, single-producer, single-consumer bounded queues; the interfaces are designed to avoid copying data. Finally, we keep sDDF drivers simple with a strict separation of concerns: A driver serves the single purpose of abstracting over device hardware. Other operations, such as setting up DMA mappings in the IOMMU, that do not depend on the specific device (but possibly on the device class), are kept in separate modules.

The sDDF programming model maps well onto the seL4CP abstractions, and leads to drivers that are much less complex (and error-prone) than those of mainstream OSes. This is illustrated by the network driver we evaluate in Section VII: It consists of about 400 source lines of code (SLOC), compared to the Linux driver that weighs in at over 3,000 SLOC for comparable functionality.

At the time of writing, the sDDF targets primarily network drivers, extension to storage devices and USB is in progress.

A related activity is the Pancake project [14], which develops a new programming language specifically designed for implementing device drivers, with the aim of enabling their formal verification. Pancake complements the sDDF, and benefits from its simplified driver model that should ease verification. The project is at an early stage.

V. Virtualization

Virtualization serves two purposes in IoT: *legacy re-use* (Section V-A) and *incremental cyber retrofit* (Section V-B), the former being a deployment scenario while the latter is a development scenario. The deployment case needs to share devices, requiring *device virtualisation* (Section V-C).

A. *Legacy reuse*

Mainstream OSes provide a wealth of services, especially many file systems optimised for different use cases, and drivers for many devices. These represent an immense amount of development effort that is often not feasible to replicate for a new OS, such as the seL4CP. In addition, IoT devices may need to integrate with enterprise frameworks that require proprietary clients deployed on the IoT device. Furthermore, the IoT device may represent a step in de-centralising a previously centralised system and needs to incorporate existing components. Moreover, the specific circumstances of an IoT device may require the integration of components that are designed to work on different OSes (enterprise-style vs embedded OS).

These are all reasons why the integration of legacy services into IoT systems is required, not only as a transitory measure but for on-going use. Virtualization is a key enabler for such scenarios [15]: It supports the concurrent use of multiple OS environments, and allows running a (likely less trustworthy) legacy software stack besides critical components.

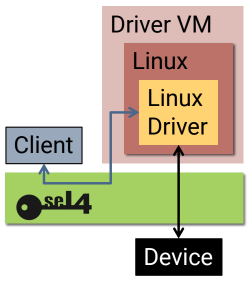

Fig. 2. A driver VM, wrapping a single Linux driver with its private, stripped-down Linux instance.

An attractive approach is wrapping each legacy component with its own (legacy) OS instance into its own, private VM. This is particularly attractive for legacy OS services, such as file systems and device drivers: Each such service runs in a separate VM, supported by a stripped-down copy of its "native" OS, reduced to the minimum functionality required to support the service [16], Figure 2 shows such a setup. Such a driver is encapsulated as any untrusted component in the system, protecting the rest of the system if the driver (or its OS) is misbehaving.

The main requirement on the OS personality is the need to support multiple VMs, and provide communication channels between them and the native components.

B. *Incremental cyber retrofit*

Given the deplorable state of security in contemporary IoT, a transition path to higher security is highly desirable. While "security must be designed in, not added later" is another popular meme, we have demonstrated in the DARPA

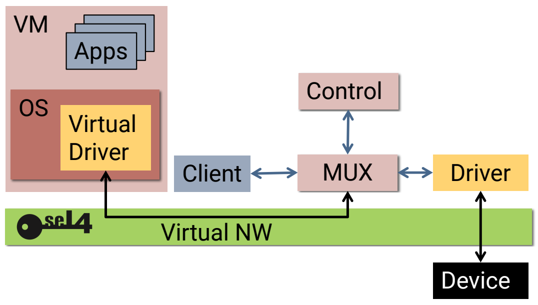

Fig. 3. Device virtualisation: A multiplexer (MUX) switches access to the device between clients as directed by the controller that implements the access policy.

HACMS program [17] that this is not necessarily true: Using a process dubbed "incremental cyber retrofit" (in analogy to the incremental seismic retrofit used to improve the resilience of buildings in earthquake-prone areas), an existing, highly vulnerable commercial autonomous aircraft was secured to the point where DARPA's professional penetration testers were unable to compromise it, despite being given root access to a VM running on the platform. DARPA even offered the design for a "steal this drone" challenge at DEFCON'21, which defeated all attackers [18].

The stepwise approach, described in detail in [7], first put the complete, Linux-based legacy system into a VM running on seL4. Then the developers broke this VM up into several VMs containing different parts of the legacy system, and ported some components to run natively on seL4. In further steps, they extracted more components from the VMs and either ported or re-implemented as native seL4 components. The final system comprised just one VM running a complex vision subsystem, and all other functionality was running as native, isolated components directly on seL4.

Virtualisation is a critical enabler of this approach.

C. *Device virtualisation*

Once there are multiple subsystems, be they native or legacy components hosted inside VMs, there inevitably arises the need to share devices between subsystems: Multiple subsystems need access to storage for saving/accessing persistent state, and multiple subsystems likely need access to networks, USB and display.

Such sharing of devices can be achieved by presenting each subsystem with the illusion of owning the device (and having a device driver for it). In effect, the device is virtualised.

Device virtualisation needs to be subject to a security policy, to prevent an untrusted (legacy) component from monopolising the device. This is achieved by a device virtualisation framework as indicated in Figure 3. Here, a multiplexer switches device access between multiple clients. The multiplexer is transparent to native clients, as it uses the same protocol as the (native) clients use to access an exclusively owned device.

The guest OS in a VM uses a virtual driver that translates the guest's native driver protocol to the sDDF protocol. Given that Linux uses virtIO [19] as a device virtualisation protocol, the multiplexer uses the virtIO protocol to interface to VMs. Among others, this has the advantage that existing Linux virtual drivers can be re-used.

The control component implements the (security or QoS) access policy, e.g. limiting network bandwidth or persistent storage usage.

The sDDF is presently being extended to support device virtualisation.

VI. Threats

The framework presented above mitigates threats by strong, seL4-enforced modularity and a design that aims to minimise the *trusted computing base* (TCB). The TCB contains the underlying processor, including its security mechanisms (such as an IOMMU), the seL4 microkernel, and the seL4CP, plus any trusted services that a specific system depends on.

seL4 is *trustworthy* as a result of its comprehensive verification [2], at least on a *verified hardware platform*. There exists only a small number of these at the moment; running seL4 on an platform not from this small set voids any verification guarantees. However, in practice, even on a platform for which seL4 is not specifically verified, the risk is dramatically reduced compared with any other OS, as most of the seL4 code is not platform-specific and can be expected to operate to specification. Also, seL4's verification still has limitations: The kernel's boot code is not (yet) verified, and some correctness-relevant hardware manipulations, including cache management and details of managing the MMU, are not verified to ISA level. But overall, the risk of compromising seL4 is tiny.

The largest risk of seL4 failure comes from multicore configurations: While seL4 runs and performs well on multicore platforms, nothing to do with multicore is verified at present, and this is likely the one significant source of critical vulnerabilities. Some early-stage work towards multicore verification of seL4 is in progress, but there is no timeline yet for its conclusion, other than that it will take years.

The biggest overall threat is an attack that compromises the system before seL4 gets control. We rely on the BIOS or U-Boot to establish initial conditions. This code can compromise the platform by changing the Arm monitor-mode code, re-routing interrupts or applying other mis-configurations that allow hijacking the system later. Obviously this is nothing that seL4 or its verification can protect against, and defences against this attack are platform-specific. Once possible protection is to burn a trusted setup code, including seL4, into ROM [20].

The seL4CP is presently unverified, but its implementation is extremely simple; the mapping to seL4 primitives consists of only about 200 SLOC, plus about 1 kSLOC of initialisation code. Verification of the seL4CP is in progress [10]. As this verification effort employs push-button techniques, and the code base is and will remain small, we are confident that it will also scale to the forthcoming extensions that add limited dynamic features. These features will grant extra powers to some privileged PDs; these are obviously part of the TCB and should be assured.

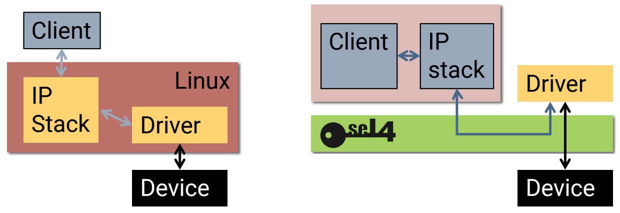

Fig. 4. Benchmark setup: Linux (left) vs seL4 (right).

Device drivers, and the sDDF, are only part of the TCB in so far as drivers need to be trusted. seL4-enforced driver encapsulation means that a compromised driver cannot compromise the rest of the system. A driver can be prevented from compromising confidentiality and integrity of the data it handles by encrypting all data, e.g. using TLS for network traffic. This leaves denial-of-service threats on the table, we hope to address this eventually by verifying critical drivers (through the Pancake project [14]).

A trusted driver will have to trust (parts of) the sDDF — this is, again, a fairly simple code base that poses a verification challenge similar to the seL4CP, and might also be amenable to push-button verification.

Short of actually verifying these components, their simplicity-by-design, enabled by strong separation of concerns, and complemented by seL4-enforced isolation, dramatically reduces the attack surface.

VII. Preliminary Evaluation

Our approach to secure IoT heavily relies on modularisation, with module boundaries enforced by the secure seL4 microkernel (Figure 1). There is a cost: every time a component boundary is crossed, the microkernel is involved. Instead of a simple function call (costing dozens of cycles), the crossing involves a mode switch for trapping into the kernel, a context switch, and another mode switch for returning to usermode, at a total cost of hundreds of cycles. Fortunately, typical execution times between such switches are tens to hundreds of thousands of cycles, so we can expect the overhead to be low.

We perform a preliminary evaluation to quantify the performance impact. We use an extreme case of high context-switching rates (and correspondingly little work done between context switches): high-throughput networking. Many of the typical IoT platforms barely have the CPU power to process Gigabit Ethernet traffic at wire speed, indicating that this is a tough test case.

A. Setup

We compare two configurations, as shown in Figure 4, both running on a single core. For Linux we use a standard setup, with an in-kernel network driver and the in-kernel IP stack, only the client running in usermode. The client simply echos back every received packet. Sending and receiving a packet requires at most one system call each; if more than one packet

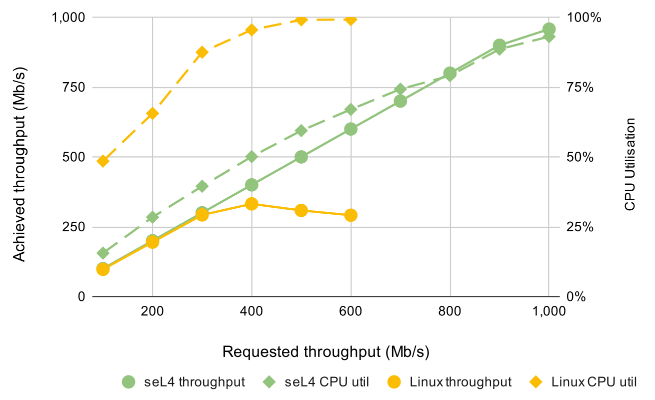

Fig. 5. Network performance of the seL4-based setup compared to Linux, showing throughput and CPU utilisation as a function of applied network load.

is available at a time, all will be processed with a single pair of system calls.

In contrast, the seL4 configuration runs the driver, IP stack (we use lwip [21]) and the client in usermode. We co-locate the IP stack with the client in a single protection domain. Compared to Figure 3 we omit the multiplexer, so this represents a reasonable configuration where the device is exclusively owned by the client. Still, sending and receiving a packet requires context switches. In total this configuration requires 9 kernel entries (and 6 context switches) for each packet round trip (i.e. driver receives a packet, hands it to the client via the IP stack, and the client echos it back via the IP stack to the driver). This number includes the driver receiving and acknowledging interrupts, the driver notifying the IP stack of data availability, the IP stack notifying the driver, a system call for each cache maintenance operation, plus the kernel scheduler switching to a background thread when the I/O system is idle. This compares to four kernel entries (each a context switch) for Linux (including switches to a background process).

We use a low-priority busy-looping usermode background thread in the seL4 system for measuring CPU utilisation of the I/O system: The time that thread manages to consume is the complement of the time used by the I/O system. In Linux we can use the `top` utility for measuring CPU utilisation.

While the sDDF I/O interface is asynchronous and zero-copy, we simulate Linux' copying read/write interface by copying each received packet twice before transmitting it back.

Our evaluation platform is a Freescale i.MX8MM Quad based on Arm Cortex A-53 cores, running at 1.2 GHz. The Ethernet adapter is the on-chip 10/100/1000 NIC connected to an Atheros AR8031 PHY capable of 1 Gb/s throughput. We use a local cluster running ipbench [22] for load generation.

B. Results

Figure 5 shows the results. Looking at throughput (solid lines, yellow for Linux and green for seL4) we see that seL4 keeps up with the applied, reaching 958 Mb/s. This represents 99.6% of the theoretical limit of 962 Mb/s (allowing for protocol overhead of 20 bytes per packet), meaning that our setup is able to fully keep up with traffic.

In contrast, Linux cannot handle more than 332 Mb/s. Comparing CPU utilisation (broken lines) we see the reason: Linux maxes out the processor at around 400 Mb/s and is therefore unable to handle more than that traffic. In contrast, seL4 can handle the maximum load with 90.5% CPU usage. In other words, the seL4-based setup can handle more throughput because its per-packet processing cost is 1/3 of that of Linux, despite the higher inherent mode and context-switching overhead.

These results clearly show that the seL4 I/O system is sufficiently lightweight to make up for the unavoidably higher number of kernel entries. In particular we observe that the sDDF's simplified driver model, aimed at enabling verification, does not hurt performance.

VIII. Implementation Status

At time of writing, seL4 runs on Arm, x86 and RISC-V processors. Its verification presently applies to 32-bit Arm and 64-bit x86 and RISC-V, although for x86 this applies only to implementation correctness (excluding proof of compilation correctness and the security-enforcement proofs). Verification for 64-bit Arm processors is in progress.

The seL4CP presently runs on 64-bit Arm and RISC-V processors, we do not expect difficulties in porting it to x86 or 32-bit mode. The total implementation presently comprises about 1,200 SLOC. The dynamic prototype adds only 53 SLOC. This will grow somewhat as not all the desired functionality is implemented yet, but the dynamic features are unlikely to have much impact on verifiability.

The sDDF is largely ISA-independent but presently only seriously tested on 64-bit Arm processors. The framework presently comprises about 1,500 SLOC, integrating with seL4CP adds another 200 SLOC. This will grow significantly as other device classes and essential services, such as device discovery and dynamic DRAM mapping are added. Drivers and servers are on top of this, eg. 400 SLOC for our Ethernet driver and some 2,000 SLOC for lwip.

Neither the seL4CP nor the sDDF are of mature production quality yet, although the seL4CP is already used in a product prototype.

All artefacts described here, including proofs (where available), are open source. Pointers to the code repositories can be found on the relevant project pages:

seL4: https://sel4.systems/contact/
seL4CP: https://trustworthy.systems/projects/TS/sel4cp/
sDDF: https://trustworthy.systems/projects/TS/drivers/

IX. Conclusions

The formally verified seL4 microkernel is a rock-solid base for military-grade security, and is applicable to IoT devices, but requires much expertise to use correctly. The seL4 Core Platform is a simple operating system that is suitable for IoT development, as it simplifies the seL4 model and enables security by design. Virtualization plays an important role as an enabler of legacy re-use: It allows utilising OS services not natively available on seL4, and enables reuse of complex application software that would be expensive to port. Virtualization also is a core enabler of the *incremental cyber retrofit* that can demonstrably secure legacy systems by migrating them to seL4.

The security benefits from running on seL4 stem from seL4 itself being practically impossible to compromise, due to its formal verification, but also from seL4 enforcing module boundaries in a componentised system. The latter has a cost: Crossing a module boundary requires a context switch, which bears a base cost that is an order of magnitude higher than that of a function call. Our initial evaluation, investigating high-throughput networking, which represents an extreme case in terms of context-switching rates, shows that seL4 can in fact outperform Linux I/O by a factor of three. In other words, the increased robustness and security, as well as the greatly simplified device driver model that aims to enable formal verification of drivers, does not come at a significant performance cost.

Acknowledgments

We gratefully acknowledge the financial support provided by the UAE's Technology Innovation Institute (TII) for the work on the sDDF and device virtualisation. We are equally grateful for the support provided by the UK's National Cyber Security Centre (NCSC) for work on verifying the seL4CP. The original development of the seL4CP was supported by an ICERA grant from the Australian Defence Science and Technology Group. We thank the many members of the seL4 Foundation who are making invaluable contributions to the seL4 ecosystem, by contributing to the development and maintenance, and directly or through their membership fees funding some of the verification work. We also thank the anonymous reviewer whose questions triggered significant improvements to the paper.

References

[1] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *ACM Symposium on Operating Systems Principles*. Big Sky, MT, USA: ACM, Oct. 2009, pp. 207–220.

[2] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM Transactions on Computer Systems*, vol. 32, no. 1, pp. 2:1–2:70, Feb. 2014.

[3] A. Lyons, K. McLeod, H. Almatary, and G. Heiser, "Scheduling-context capabilities: A principled, light-weight OS mechanism for managing time," in *EuroSys Conference*. Porto, Portugal: ACM, Apr. 2018.

[4] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen, "SkyBridge: Fast and secure inter-process communication for microkernels," in *EuroSys Conference*, Mar. 2019.

[5] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, pp. 143–155, 1966.

[6] U. Steinberg and B. Kauer, "NOVA: A microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th EuroSys Conference*. Paris, FR: ACM, Apr. 2010, pp. 209–222.

[7] G. Klein, J. Andronick, I. Kuz, T. Murray, G. Heiser, and M. Fernandez, "Formally verified software in the real world," *Communications of the ACM*, vol. 61, pp. 68–77, Oct. 2018.

[8] B. Leslie and G. Heiser. (2020, Nov.) The sel4 core platform. [Online]. Available: https://trustworthy.systems/projects/TS/sel4cp/2011-draft-spec.pdf

[9] ——. (2022, Mar.) Evolving seL4CP into a dynamic OS. [Online]. Available: https://trustworthy.systems/projects/TS/sel4cp/2203-report-dynamic.pdf

[10] Trustworthy Systems Group. (2022) Verifying the seL4 core platform. Project web page. [Online]. Available: https://trustworthy.systems/projects/TS/sel4cp/verification

[11] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *ACM Symposium on Operating Systems Principles*, Lake Louise, Alta, CA, Oct. 2001, pp. 73–88.

[12] MITRE Corporation. (2022) Linux kernel: CVE security vulnerabilities, versions and detailed reports. Accessed: 2022-04-14. [Online]. Available: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor{_}id=33

[13] Trustworthy Systems. (2022) Research on device drivers. Project web page. [Online]. Available: https://trustworthy.systems/projects/TS/drivers/

[14] ——. (2022) Pancake: A language for formally verified device drivers. Project web page. [Online]. Available: https://trustworthy.systems/projects/TS/drivers/pancake

[15] G. Heiser, "The role of virtualization in embedded systems," in *Workshop on Isolation and Integration in Embedded Systems*, Glasgow, UK, Apr. 2008, pp. 11–16.

[16] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, "Unmodified device driver reuse and improved system dependability via virtual machines," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, US, Dec. 2004, pp. 17–30.

[17] K. Fisher, J. Launchbury, and R. Richards, "The HACMS program: using formal methods to eliminate exploitable bugs," *Philosophical Transactions of the Royal Society A*, vol. 375, no. 2104, 2017.

[18] Trustworthy Systems. (2021) seL4 protects world's most secure drone from DEFCON hackers. [Online]. Available: https://sel4.systems/news/2021#defcon

[19] R. Russell, "virtio: towards a de-facto standard for virtual I/O devices," *ACM Operating Systems Review*, vol. 42, pp. 95–103, Jul. 2008.

[20] HENSOLDT Cyber. (2022) MiG-V – made in Germany RISC-V. [Online]. Available: https://hensoldt-cyber.com/mig-v/

[21] A. Dunkels, "Minimal TCP/IP implementation with proxy support," SICS, Tech. Rep. T2001-20, Feb. 2001, http://www.sics.se/~adam/thesis.pdf.

[22] I. Wienand and L. Macpherson, "ipbench: A framework for distributed network benchmarking," in *Conference for Unix, Linux and Open Source Professionals (AUUG)*, Melbourne, Australia, Sep. 2004, pp. 163–170.