Kirsch: It's 2025. Does your OS know what's on your SoC?

Roman Meier
Systems Group, ETH Zurich
Zurich, Switzerland

Zikai Liu
Systems Group, ETH Zurich
Zurich, Switzerland

Ben Fiedler
Systems Group, ETH Zurich
Zurich, Switzerland

Timothy Roscoe
Systems Group, ETH Zurich
Zurich, Switzerland

Modern operating systems are written to a model of machine hardware which is remarkably similar to that of the DEC PDP-11 of the 1960s and 1970: a set of homogeneous cores accessing (via MMUs) a common physical address space containing main memory, plus memory-mapped devices, some of which can perform DMA.

This is, of course, a fiction: modern SoCs and server platforms are really a complex network of heterogeneous cores and intelligent devices, many of which are running their own firmware and "operating systems". The *de facto* OS [2] of a modern computer – the thing managing the whole machine – is a motley collection of such cores and their system software, only a small fraction of which runs what is popularly termed "the operating system".

This is an impasse: modern OSes are designed for nothing more than a homogeneous set of cores with a common view of the address space, and extending them to a complete, real computer is impractical. Hardware designers recognize this near-impossibility, and so add semi-hidden cores running their own firmware to work around the problems. The result is a catastrophe of system design, including a plethora of security exploits like remote over-the-air compromises due to weaknesses in WiFi modem firmware [1]. Crucially, it is impossible to specify the correct behaviour of the *de facto* OS of a modern computer, let alone verify that it conforms to such a specification. This is because while the individual components of the *de facto* OS are designed with intention, in a modern SoC they are then just connected ad-hoc to complex bus topologies, with little regard for the presence and behaviour of the components at the other end of the bus. There is no overall design of the *de facto* OS.

We are building Kirsch, a new OS that solves this problem by embracing and formally capturing the heterogeneity and multiple trust domains of modern hardware. Rather than a clean-slate design, Kirsch acknowledges the need to build a secure OS for a complete machine out of both trusted *and untrusted* parts, accommodating the untrusted, proprietary firmware on some devices and inserting a new, trusted OS on cores we control. Instead of trying to extend an existing kernel to a platform totally unsuited to it, Kirsch assembles an OS for a real computer out of components which have *explicit, formally-specified trust relationships* based on the hardware.

The formal hardware model on which Kirsch is based is a decoding net representation[3] of the complete platform, which captures exactly what each physical core or device ("context" in Kirsch parlance) can access, and how its access can be restricted by MMUs or other protection units in the interconnect. This decoding net thus induces a trust relationship between contexts, which is the basis for reasoning about isolation, protection and authorization in the system.

The decoding net representation of a modern platform is highly complex, reflecting the complexity of the hardware itself. To make the OS design, and reasoning about it, tractable, we therefore define a *single, shared, logical address space* for the whole machine, in which every addressable resource is allocated a unique region. We call this the *global logical address space* or GLAS for short. Each context has access to a subset of this space. This factoring into privileged components which maintain this address space in each context, and the rest of the OS which runs within it, greatly simplifies the design and reasoning about it.

The Kirsch toolchain is responsible for taking the decoding net representation of a hardware platform together with a set of trust assumptions, validating the input trust assumptions and generating GLAS mappings. The GLAS mappings can further be used to generate page tables that instantiate the GLAS as a fixed, global virtual address space on targets with an MMU.

The Kirsch toolchain will also allow a trust development loop, in which a developer can iterate over a set of trust assumptions, given a certain platform decoding net. The toolchain can point out both exaggerated and insufficient trust assumptions, as well as clarifying which trust assumptions are required by the structure of the platform. This

allows a developer to either adapt the software on the platform to the trust assumptions the hardware requires, or to switch to another platform if the required trust assumptions are not acceptable.

Finally, the Kirsch toolchain will be able to use the decoding net description of the hardware and the developer trust assumptions to generate code and configurations for the Kirsch OS components, which can then in turn be used to make informed isolation decisions based on the actual structure and trust requirements of the hardware they run on.

So far, Kirsch is made up of the following heterogenous OS components:

- Cheriette, a custom CHERI[5] hardware capability kernel for both the CHERI Arm Morello and CHERI RISC-V targets. CHERI is a hybrid capability model that allows extending ISAs with hardware capability support, which Cheriette uses to pull the isolation Kirsch provides on a hardware level into the kernel and OS services.
- Verified microkernel seL4[4]. The seL4 kernel creates *virtual contexts* (processes or VMs) which benefit from the verified isolation properties of seL4 composed with the explicit underlying trust relations provided by Kirsch.

  In contrast with unverified kernels, the seL4 proofs make the axiomatic assumptions a kernel must make about kernel memory and device access explicit. In particular, seL4 assumes that kernel memory is protected from untrusted devices. Interated with Kirsch, an seL4 instance can run from, and manage, a region of RAM to which all possible access from other devices and cores in the system is explicitly considered and enabled.

Kirsch thus forms a complete OS which securely manages the entire machine, including heterogeneous cores and devices which run proprietary or otherwise untrusted software – assuming that such devices can be verifiably contained. If they can't, the Kirsch toolchain makes it clear why they cannot be isolated, and the developer is given an explicit choice between acquiring better hardware which can deliver the guarantees any OS needs, or making a leap of faith.

References

[1] Beniamini, G. Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 2). https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_11.html. [Accessed 05-02-2025].

[2] Fiedler, B., Meier, R., Schult, J., Schwyn, D., and Roscoe, T. Specifying the de-facto os of a production soc. In *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification* (New York, NY, USA, 2023), KISV '23, Association for Computing Machinery, p. 18–25.

[3] Fiedler, B., Schwyn, D., Gierczak-Galle, C., Cock, D., and Roscoe, T. Putting out the hardware dumpster fire. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (New York, NY, USA, 2023), HOTOS '23, Association for Computing Machinery, p. 46–52.

[4] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, Association for Computing Machinery, p. 207–220.

[5] Woodruff, J., Watson, R. N. M., Chisnall, D., Moore, S. W., Anderson, J., Davis, B., Laurie, B., Neumann, P. G., Norton, R., and Roe, M. The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)* (2014), pp. 457–468.