

# Silent Zero

Bare minimum. Verified. Real-time. An OS for the AI era.

Tobias von Dewitz · Susann Mürling · Stefan Probst · Daniel Strecker

26 October 2026

Text and figures: [CC BY 4.0](#)

**Abstract.** This paper proposes Silent Zero, an operating system reduced to what cannot be removed. Silent Zero does not try to make AI trustworthy. It is built to stay safe although the agent is not, so the agent never holds more authority than a person granted and no motion leaves a verified loop. Silent Zero consists of a bootloader, a formally verified microkernel, and the bare minimum needed to run AI agents safely and in real time. Nothing at its base is new. The seL4 microkernel with its mixed-criticality extensions, its system framework and drivers, capability-based security, open roots of trust and open agent protocols all exist as open-source work. What has to be built is the layer between agents and the world: a broker that turns human intent into bounded authority, and a safety path that keeps every motion under a verified loop. Whatever is not needed to run agents is left out, and for every part that remains we state whether a guarantee depends on it and whether it is proven, tested or assumed.

The history of 1991 frames the case. Three shifts visible in 2026 carry it, together with the physical turn of agents into robots, which makes real time a first principle. Silent Zero therefore starts where agents act physically, with one robot and one task. We also state where the design can fail, and we look only five years ahead, to 2031.

**Keywords:** microkernel, seL4, formal verification, real time, mixed criticality, object capabilities, AI agents, robotics, open source

## In brief

AI agents no longer just give answers; they deliver work. They read email, hold credentials, call tools and, increasingly, move arms or wheels in the physical world. Run as an ordinary program, an agent can inherit broad access to its user's files, credentials and network; sandboxes narrow that as a layer added on top. And an agent can be misled by anything it reads. In robots, the agent's software and the real-time control of the motors mostly live in separate worlds, joined by glue.

**We do not trust the agent, so authority and safety move beneath it.** Silent Zero treats every agent as untrusted, whatever model drives it. An agent holds only the authority a person granted it, limited in scope and time, and revocable. It may propose any motion; a small, deterministic loop has

the last word. Processor time is capability-controlled like memory, so an agent can be prevented from monopolising the processor and starving the control path. Verified isolation, explicit capabilities and hard real time form one architecture, not three layers. Every part a guarantee depends on is labelled proven, tested or assumed: proof where feasible, explicit assumptions everywhere else.

**The building blocks exist.** seL4, whose mixed-criticality (MCS) configuration on 64-bit RISC-V now has a machine-checked proof of functional correctness; a verified real-world Ethernet driver; OpenTitan roots of trust, shipping in production Chromebooks since March 2026. The hard part is composing them, with an end-to-end assurance argument that holds across every seam between proven and unproven parts. Known gaps include verified

multicore configurations and timing bounds for modern processors. This paper starts that argument and says where it does not know yet.

**Where it can fail.** It does not stop prompt injection; it limits how far a deceived agent can reach. Firmware and GPUs remain trusted, not proven. And hardened Linux beside a separate motor controller may prove good enough.

**What we need now.** People who know firmware, verification or robots, to test this argument before the architecture hardens around it.

## Prologue: Imagine it's 1991

*Never trust an OS you can't verify.*

After the Macintosh at its launch in January 1984, which said of IBM's mainframes: "Never trust a computer you can't lift" (Mangrum 2025).

Imagine it's 1991. We are about to release a few lines of code into a world that will be very different from today.

On 6 August 1991, a researcher at CERN posts a note about a project called World-WideWeb to the newsgroup alt.hypertext and invites others to join (W3C 2016). Nineteen days later, on 25 August, a student at the University of Helsinki writes to comp.os.minix: *"I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones"* (Torvalds 1991). The first code follows in September. Neither author knows what he has started. The web will connect the planet; the hobby will end up in servers, phones and much of what lies in between.

Several things came together. Hardware had become cheap and belonged to no single vendor. A network let strangers work on the same code, a licence kept their work open, and the idea was small enough to start that evening.

Now imagine it's 2026. The machines that matter are no longer only desks and servers.

More and more of them are agents, programs that read, decide and act on their own, increasingly with arms, wheels and rotors. They all run on operating systems built for a person at a keyboard. Finding flaws in software has become cheap because machines now do it, and for the same reason proving software correct is getting cheaper. Almost everything one would need for a new foundation already exists, scattered across open-source projects and waiting to be assembled.

We are, once again, about to release a few lines of code into a world that will be very different from today. This paper asks what those lines should be, and we think they should be as few as possible.

## 1 The idea: zero as a design rule

### **Don't trust the agent. Verify the system.**

Silent Zero assumes nothing about the agent's good behaviour. An agent may simply be wrong, it may have been manipulated by something it read, or it may work against its user on purpose. Distributed systems have a name for a component like that. Lamport called it Byzantine, after generals who must agree on a plan without knowing which of them are traitors (Lamport et al. 1982). Safety must then come from the parts that decide what the agent may do and what moves, and since trusting them would only move the problem, they have to be proven.

How large such a system may be was settled long before agents. In 1972 James Anderson set three conditions for what he called the reference monitor, the mechanism that checks every access. It must always be invoked and must be tamper-proof, and Silent Zero takes its size from the third, that it be small enough to be analysed and tested (Anderson 1972).

Both words of the name are design rules. Zero means that we start from nothing and add only what we cannot leave out. Silent means that nothing runs that is not needed. What remains comes to three parts.

1. **A bootloader** that brings up the machine, checks what it loads, shows what it found, and hands over to the kernel or stops.

2. **A microkernel** that isolates components, moves messages between them, and schedules time. Nothing else runs at its privilege level. Firmware beneath it runs higher still and is named as trusted.
3. **The bare minimum to run agents.** A way to reach a model and a way to reach tools, a broker that hands out authority, and real-time loops for anything that touches the physical world.

File systems, window managers and compatibility with existing software stay outside the core or do not exist. Legacy code that cannot be avoided runs as a guest in its own virtual machine.

For every part we ask two questions. Does a guarantee depend on it? Then it belongs to the trusted base, even if it runs isolated. A broker that grants the wrong authority makes the system unsafe, and the kernel will enforce every boundary correctly while that happens. And what do we know about it? Its behaviour is proven against a specification, tested, or simply assumed. We want the trusted base small and, over time, proven. Some of it, such as firmware, boot code and devices that write straight into memory, will stay assumed for a long while, and Section 6 names those parts.

Time matters as much as authority. A robot arm, a drone or a vehicle can do real damage when it misses a deadline. Such machines need real time, computing in which a result counts only if it arrives before its deadline. So the kernel protects processor time the way it protects memory, from the first line of code. Whether the timing can also be proven is open; Section 4 says how far that work has come.

Nothing at the base has to be invented. Section 5 goes through what already exists as open-source work. What we have to build is the layer that connects agents to authority and to motion, and the argument that the assumptions of all these parts fit together.

And we start narrow. Minimal only means something relative to a task, so Silent Zero begins with one robot doing one defined job under known conditions (Section 7). A gen-

eral operating system for agents may come later.

## 2 Why now

### 2.1 What 1991 teaches

On 29 January 1992, Andrew Tanenbaum declared Linux obsolete in the same newsgroup, because it was monolithic and tied to one processor family (DiBona et al. 1999). Many in operating-systems research found his design argument the stronger one. Linux won anyway. It ran on cheap hardware, came under the GPL, took contributions pragmatically and ran the software people already had. Silent Zero gives up that compatibility in its core and buys it back through Linux guests (Section 3), at a price.

The microkernel did not lose on its own, though. Mach, the flagship of the 1980s, was slow. Jochen Liedtke rebuilt the idea from scratch and made inter-process communication up to 22 times faster than Mach for short messages (Liedtke 1993); against contemporary microkernels the gap was roughly a factor of 10 to 20 (Heiser and Elphinstone 2016). From this L4 line came seL4, the first general-purpose kernel with a machine-checked proof of functional correctness (Klein et al. 2009). The proof shows that the kernel's C code always does exactly what its specification says. Mach survives mainly inside Apple's hybrid XNU kernel (Apple n.d.).

1991 teaches two things. A good design is not enough on its own, because timing, licence, drivers and distribution decide as much. And a better design needs a moment in which its cost falls. History cannot prove that 2026 is such a moment, but the three shifts below suggest it may be.

### 2.2 Finding flaws has become cheap

In the final of DARPA's AI Cyber Challenge in August 2025, autonomous systems found 54 of 63 planted vulnerabilities, patched 43 and discovered 18 real ones (DARPA 2025). In April 2026, Anthropic reported that an unreleased model had found thousands of high-severity vulnerabilities, including some in every major operating system and web browser,

among them a 27-year-old flaw in OpenBSD (Anthropic 2026a). A month later the company and around 50 partners reported more than 10,000 findings rated high or critical; of the 1,752 assessed further, 62 per cent were confirmed at that severity. These are the company's own figures; the bottleneck, it reported, had moved from finding to verifying, disclosing and patching (Anthropic 2026b).

When finding flaws is cheap, every line of unproven privileged code becomes a liability. The Linux kernel source approached 40 million lines in early 2025, about 25 million of them drivers (Leemhuis 2025). An installation runs only part of that tree, but all of it with full privilege. The same tools also help harden existing systems (Section 6); for systems that must not fail, we argue for less privileged code as well as more patches.

### 2.3 Proving correctness is becoming cheaper

seL4's first proof covered about 8,700 lines of C and took, together with design and implementation, roughly 25 person-years, much of it new research and tools; the authors put a repeat at under ten (Klein et al. 2009; Trustworthy Systems n.d.-a). Two developments now push that cost down. Verification tools for Rust such as Verus check systems code 3 to 61 times faster than earlier tools (Lattuada et al. 2024). And language models, paired with a proof checker that rejects every wrong step, generate proofs. A neuro-symbolic method, for instance, proves 77.6 per cent of the seL4 theorems in the FVEL benchmark, though only about 20 per cent of those whose proofs exceed ten lines (He et al. 2026). So short proofs are getting cheap, while long ones and the invariants behind them remain expensive.

This is also why we are attempting it now. Until recently, a project of this size would have been unreasonable for a small group like ours. Since the models of 2026, AI contributes code, proofs and reviews at a level that brings it within reach. Whether that is enough is one of the doubts in Section 6.

### 2.4 Agents act, and increasingly in the physical world

Classical operating systems grant a process the full authority of its user. Norm Hardy described in 1988 why this fails: a program acting on behalf of one party misuses authority granted by another, a *confused deputy* (Hardy 1988). Indirect prompt injection follows the same pattern, with instructions hidden in data steering an agent that acts with its user's rights (Greshake et al. 2023). Heuristic defences have not held up. Adaptive attacks broke 12 published defences, most of them with success rates above 90 per cent (Nasr et al. 2026). The structural answer is old. Authority must be passed explicitly, as capabilities (Dennis and Van Horn 1966; Miller 2006). A capability is an unforgeable token that names a resource and grants a right to it, and a program that does not hold one cannot use the resource.

At the same time, agents are acquiring bodies. Robot stacks now separate a reasoning model that plans and calls tools from an action model that drives the motors (Google 2025). Action models emit motor commands in chunks so that a robot can hold a 50 Hz control rate while the slower model thinks (Black et al. 2025). And a project for US Army unmanned ground vehicles has built proofs of concept that run ROS 2, the common robot middleware, on seL4 (Lide 2026). In such systems a missed deadline can mean a dropped load or a collision.

## 3 Anatomy of Silent Zero

Silent Zero has six layers. Read from the bottom, each one exists only because the one above cannot work without it.

**Hardware and root of trust.** Silent Zero needs memory management, an IOMMU that limits which memory a device may read and write directly, and a root of trust that measures the first code to run. OpenTitan, an open-source silicon root of trust, has shipped in production Chromebooks since March 2026 (Google 2026). Where available, CHERI adds unforgeable pointers in hardware; it is being standardised for RISC-V (Moore 2026), the

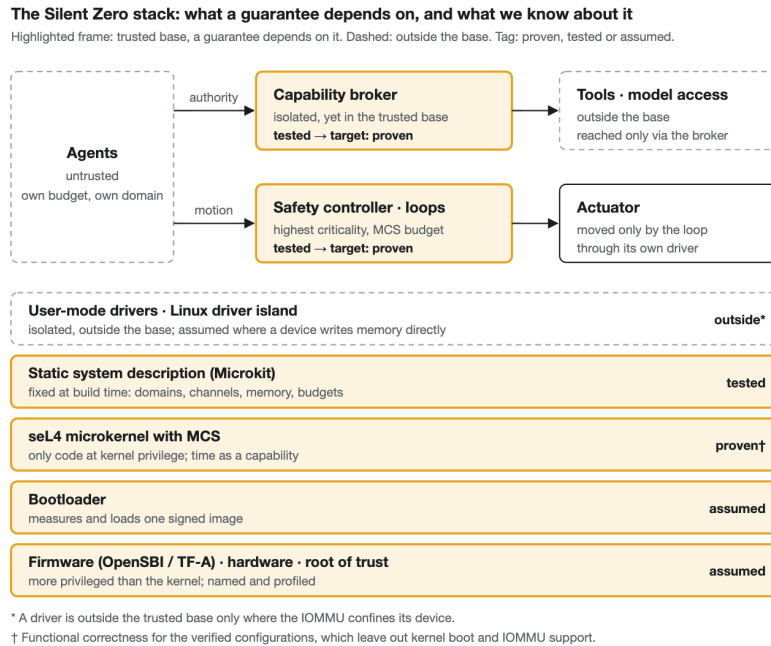


**Figure 1.** The Silent Zero stack. A gold frame marks the trusted base: every part a guarantee depends on, isolated or not. The tag says what we know about it. Authority runs from the agent through the broker, motion through the safety controller; there is no third path.

open instruction set that anyone may implement. Firmware runs here too, with more privilege than the kernel. seL4 Microkit is the tool that builds the system image. On RISC-V, it relies on SBI firmware such as OpenSBI in machine mode (seL4 Microkit n.d.); on Arm, trusted firmware at the highest exception level plays that role. This layer belongs to the trusted base and is *assumed*. Silent Zero states a hardware profile instead of assuming any hardware will do.

**Bootloader.** It checks one signed, static system image and starts it, or stops. There is no menu, no configuration and no fallback, and nothing of it stays resident. It also draws the first screen, because when the check fails nothing else will run to say why. The screen shows only compiled-in text and what the loader has measured itself, and marks as assumed what it cannot check, the loader included. Boot code lies outside seL4’s verified configurations (seL4 Foundation n.d.-a), so this layer, too, is trusted base and *assumed*, and its size counts. A first prototype for RISC-V, which so far checks a hash rather than a signature,

is just under 9 KB, about two thirds of it the renderer and its font data.

**Microkernel.** seL4 with its mixed-criticality (MCS) extensions. MCS treats time, like memory, as a resource protected by capabilities (Heiser 2026). Its C-level verification is complete for RISC-V; AArch64 is targeted for the third quarter of 2027 (seL4 Foundation n.d.-b). It is the only code at the kernel’s privilege level, and its functional correctness is *proven* for the verified configurations, which leave out kernel boot and IOMMU support (seL4 Foundation n.d.-a).

**Static system.** The whole system is declared in one description, which says what isolated programs exist (protection domains), which of them may talk (channels) and what memory they share. Microkit turns that description into the bootable image (seL4 Microkit n.d.). Nothing is loaded into the core at run time. A system whose structure is fixed at build time can be checked, scheduled and audited before it ever runs.

**Drivers.** Drivers run in user mode, each with access only to its own device registers

(Parker 2022). Legacy drivers, above all for graphics and AI accelerators, run inside a Linux virtual machine that serves as a driver island (Heiser et al. 2025). These parts are isolated as far as memory goes and stay outside the trusted base. Wherever a device can write to memory directly and the IOMMU configuration is not proven, the driver joins the trusted base as *assumed* and is listed as such. The island sits on the agent's side: neither the authority path nor the motion path runs through it. If it fails, the system becomes less useful and may leak the model's context, which local inference sends through the island, but physical safety is not affected.

**The agent minimum** consists of four components.

1. **Model access.** Local inference on an accelerator, reached through the driver island, or remote inference over the network.
2. **Tool access.** An open protocol for connecting agents to tools. The Model Context Protocol (MCP) has been governed by the Agentic AI Foundation under the Linux Foundation since December 2025 (Anthropic 2025).
3. **Capability broker.** The only component that hands out authority. It grants capabilities that are attenuated, time-limited and revocable, and it words every request a human sees, so the agent never does. It runs isolated and still belongs to the trusted base, because its decisions are the guarantee. It will be *tested* first, with *proven* as the target (Section 7).
4. **Agents.** Untrusted tenants, each in its own protection domain, with a CPU budget enforced by the kernel's scheduling contexts. No guarantee depends on them.

Next to the agents, and above them in criticality, sit the **real-time loops** for control, safety monitoring and actuation. Among them is the safety controller, which has the last word over every motion. Section 4 explains why.

A guarantee depends on every framed part: firmware and hardware, bootloader, kernel,

static system, broker and safety controller. Agents, model access and tool access stay outside that base. Two paths cross the figure. Authority runs from the agent through the broker, and motion runs through the safety controller. There is no third path.

#### 4 Real time throughout

Agents are slow and make mistakes, while control loops are fast and must not fail. Silent Zero is designed around that difference.

**Time is a protected resource.** In seL4 MCS, CPU time is handed out through capabilities, with budgets the kernel enforces (Heiser 2026). An agent that loops, stalls or is hijacked uses up its own budget and leaves the arm controller's untouched. Memory isolation keeps a compromised agent from writing into the loop, and time isolation keeps it from taking the loop's processor time. A CPU budget is still no end-to-end guarantee. Memory contention, interrupts, devices and communication also decide whether a loop meets its deadline, and each of them needs its own bound.

**The agent proposes, the verified loop disposes.** Robotics described the structural pattern twenty-five years ago. In the Simplex architecture, a complex high-performance controller runs the system as long as the plant stays inside a safe envelope; a simple, high-assurance controller takes over before it leaves the region from which it can still recover (Sha 2001). Silent Zero maps this onto its layers. The agent and its action model play the complex controller, powerful and unproven, isolated and lower in criticality. The safety controller and the switching logic play the simple one. They are small, run at the highest criticality and are candidates for proof. The agent may suggest any motion. Only the loop moves the motor. Regulation points the same way: from January 2027, EU machinery law lists safety components whose behaviour evolves through machine learning among the machinery that needs third-party assessment (EU 2023). A safety function that stays deterministic stays out of that class.

**Three goals.** Security means the agent cannot exceed its authority. Safety means the system violates no safety condition under stated assumptions. Usefulness means the task still gets done. None of them follows from the others, and a controller that stops at every doubt is safe and useless. Most of the engineering lies in deciding what it can reliably detect, for one robot and one task.

**It has worked under attack.** In DARPA's HACMS programme, the mission computer of Boeing's Unmanned Little Bird helicopter was rebuilt on seL4. In flight tests in 2017 the helicopter came under cyberattack mid-flight; compromised subsystems could not affect safe operation, and a red team given root access to the camera's virtual machine stayed confined to it (Klein et al. 2018).

**Where real time is not yet provable.** A real-time guarantee needs a bound on the kernel's worst-case execution time (WCET). seL4 has had a sound and complete WCET analysis, but only for the ARM11 processor. The work is archived because Arm stopped publishing worst-case instruction latencies for its out-of-order cores; open RISC-V cores such as CVA6 are named as the way forward (Trustworthy Systems n.d.-b). That analysis is now being rebuilt for the current kernel on 64-bit RISC-V, so far with placeholder instruction latencies and loop bounds (Sillitoe et al. 2026). Open RISC-V cores also matter for timing side channels, through which one program learns another's secrets from how long its operations take. Closing them needs a hardware instruction that so far exists only as prototypes on such cores (Wistoff et al. 2022; Wistoff et al. 2024). Provable schedulability for MCS remains ongoing research (Heiser 2026).

The first version of Silent Zero should therefore run on an open RISC-V core, because only there is a proof of its timing within reach.

## 5 Nothing to invent at the base

The base of Silent Zero exists today as open-source work. Table 1 lists it with its state in 2026. The list mixes running software, hardware designs, protocols, tools and patterns,

and the difference matters. A proof tool does not make a proven broker, and a safety pattern does not make a safety controller for a given robot.

Integrating these parts safely is a substantial engineering and research task, and that is where the contribution of Silent Zero lies. The **broker** must bind each permitted action to recipients, data and parameters, enforce that at every tool call, map kernel capabilities onto external credentials, define revocation for actions under way, and word approval requests in a form the agent cannot imitate, which needs a trusted path to a screen and an input device that does not run through the driver island. The **tool adapters** must leave no path around it, including the context sent to remote models. The **safety path** must be specified for a concrete robot. All three need evidence that their assumptions match the parts beneath.

**Others are building this.** LionsOS, from the group behind seL4, already composes seL4, Microkit and user-mode drivers into a static, modular system for cyber-physical use; a first driver, for Ethernet, has been verified with automated proofs, and the others are under way (Trustworthy Systems 2026; Tyler et al. 2026). Silent Zero should build on it. Agent operating systems on seL4 are appearing too. agentOS aims to give agents capabilities for tools, models and memory and to let them replace system services at run time, though its booted image does not include these services yet (Hubbard 2026); Cohesix coordinates edge AI fleets (Bower 2026); Google's KataOS paired seL4 with Rust for machine-learning devices (Google 2022). Above the kernel, the first designs for the broker's job have appeared. AgenticOS lets agents declare intents instead of requesting resources and derives a least-capability environment from them, so far as an architecture without an implementation (Zhao et al. 2026); Agent libOS keeps an agent's authority fixed while its tools and skills grow, on top of a conventional host (Zhang 2026). None of them names real-time control as its goal, and services that agents replace at run time, schedul-

| Component                | Existing work                             | State in 2026                                                                                                                                                                 |
|--------------------------|-------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Root of trust            | OpenTitan                                 | In production Chromebooks since March 2026 (Google 2026)                                                                                                                      |
| Capability hardware      | CHERI, CHERIoT                            | Standardisation for RISC-V under way (Moore 2026); CHERIoT pairs CHERI with a clean-slate embedded OS (Amar et al. 2023)                                                      |
| Microkernel              | seL4 with MCS                             | Functional correctness proven (Klein et al. 2009); MCS verified at C level on RISC-V, AArch64 targeted for Q3 2027 (seL4 Foundation n.d.-b)                                   |
| System composition       | seL4 Microkit                             | Available for use, under active development (seL4 Microkit n.d.)                                                                                                              |
| Drivers                  | seL4 Device Driver Framework, LionsOS     | Modular OS on seL4 that outperforms Linux on context-switch-intensive loads and in networking; legacy drivers in VMs (Heiser et al. 2025)                                     |
| Verified components      | Verus for Rust                            | Verifies systems code 3 to 61 times faster than prior tools (Lattuada et al. 2024)                                                                                            |
| Proof automation         | AutoVerus;<br>neuro-symbolic proof search | Over 90 % of 150 Rust proof tasks (Yang et al. 2025); 77.6 % of seL4 theorems in FVEL (He et al. 2026)                                                                        |
| Real-time safety pattern | Simplex architecture                      | Established since 2001 (Sha 2001)                                                                                                                                             |
| Robot middleware         | ROS 2 on seL4                             | Proofs of concept for Army ground vehicles (Lide 2026)                                                                                                                        |
| Agent-tool protocol      | Model Context Protocol                    | Open, under the Linux Foundation since December 2025 (Anthropic 2025)                                                                                                         |
| Agent authority          | CaMeL, FIDES                              | Capability and information-flow enforcement for agents; CaMeL solves 77 % of tasks with provable security versus 84 % undefended (Debenedetti et al. 2025; Costa et al. 2025) |

**Table 1.** Nothing to invent: the open-source parts of Silent Zero and their state in 2026.

ing among them, rule out what a control loop needs. Silent Zero adds the authority layer from intent to actuator, on a static, real-time base.

Silent Zero builds entirely on open source, and we cannot say what rules open source will follow by 2031. Licences written for human authors now meet machine-written code, and maintainers face a flood of machine-found flaws. The licence will matter, as it did for MINIX inside Intel’s Management Engine (Tanenbaum 2017). Who maintains the integration and its proofs over years matters at least as much.

## 6 What Silent Zero is not, and where it can fail

An earlier draft of this design was attacked by seven AI-assisted red-team reviews, each told to tear apart one aspect: formal methods, hardware, exploitation, agent security, supply

chain, systems and facts. Their findings are [published with this paper](#). The limitations below are what survived.

**Silent Zero does not solve prompt injection.** Capabilities limit how far a deceived agent can reach, and that is all they do. An injected agent can simply ask for more authority, and humans make poor gatekeepers. In one study only about 17 per cent of Android users paid attention to permissions at install time, and only 3 per cent of survey respondents answered all three comprehension questions correctly (Felt et al. 2012). Authority is also not information flow. Data can leak through channels an agent is allowed to use. Capability systems for agents therefore add information-flow control, which tracks where data comes from and where it may go (Costa et al. 2025). Silent Zero shrinks the blast radius and leaves the agent exactly as trustworthy as it was.

**The trusted class is larger than one**



**Figure 2.** The path from 2026 to 2031. Each station names its evidence and whether it is a demonstration, a test or a proof; a failed check stops the path.

**would like.** Firmware below the kernel, from management engines to baseboard controllers, has full access to memory and cannot be confined by any kernel. Devices that write to memory directly have defeated IOMMU protection in Linux, macOS and FreeBSD (Markettos et al. 2019). And GPUs, the hardware agents depend on most, have leaked memory between processes, and one flaw exposed about 181 MB per query of a running language model (Trail of Bits 2024). Silent Zero lists these parts as trusted and demands a hardware profile, but it cannot prove them.

**A proof is only as good as its seams and its specification.** In three formally verified distributed systems, 11 of 16 bugs found lay in the layer between verified and unverified code, two in the specifications (Fonseca et al. 2017). Silent Zero, built from many isolated parts, consists largely of such seams. Silent Zero draws a rule from this. The two critical paths get as few seams as possible, and each seam on them is specified as an interface that is checked when the system image is built. Verification of seL4’s multikernel configuration, one kernel instance per core, is scheduled for the third quarter of 2028 (seL4

Foundation n.d.-b)), so a proven Silent Zero starts on a single core per kernel instance.

**Hardening may win instead.** Writing new code in a memory-safe language, without rewriting the old, works. In Android, memory-safety flaws fell below 20 per cent of all vulnerabilities in 2025 for the first time (Vander Stoep 2025). Linux won in 1991 the same way, by being good enough, improving step by step and staying invisible to users. Linux restricts processes with Landlock (Linux kernel n.d.-a) and supports real time with PREEMPT\_RT (Linux kernel n.d.-b); the ELISA project works on making Linux usable in safety applications (ELISA 2026). The obvious alternative to Silent Zero is an agent on hardened Linux plus a separate safety controller for the motors. For the actuators that split is sound, and Silent Zero adopts it. It leaves out the authority the agent holds over data, credentials and tools, and what it sends out, including the context shipped to remote models, because a safety controller only guards motion. Silent Zero makes sense where hardening cannot reach, which means explicit authority for agents, provable timing for machines that move, and a small trusted base for

systems that must not fail.

**We may be wrong about the glue.** The case for Silent Zero rests on the claim that the parts exist and that what remains is putting them together. We do not know how much of the vision is really in place, how much connecting work lies between the parts, or whether AI can already do that work to the standard a proof demands. The progress of the last two years makes the plan look feasible, and that impression may be an illusion. There is a plainer risk as well. A project like this needs people who stay with it for years while everything else competes for their time, and it needs money to pay for at least part of the work; so far we have secured neither.

**Silent Zero can make weapons more reliable, too.** An agent that cannot exceed its authority and a control loop that keeps its deadline under attack are what a care robot needs, and they are just as useful in a weapon. Two of the projects Silent Zero builds on show it. seL4's best-known field test was a DARPA programme for an unmanned helicopter, and ROS 2 on seL4 is being developed for US Army ground vehicles (Klein et al. 2018; Lide 2026). We cannot prevent that kind of use. An open-source licence may not exclude any field of use (Open Source Initiative n.d.), and an open project has no way of checking who builds on it. We can say where we stand. The broker is there to carry out what people decided, so the reference design has no configuration in which an agent picks a target and engages it, or acts against a person, unless a human decision is bound to that very act. The broker logs every grant of authority, so that afterwards one can trace who allowed what. And we, the authors, do not work on deployments meant to harm people. No proof or test enforces any of this. It is a declaration, and we list it as one, next to the parts that are proven, tested or assumed.

**Small does not mean frugal.** A small static system needs less memory and storage, carries no layers it does not use, and runs on boards such as the Raspberry Pi 4 (seL4 Microkit n.d.). A verified base also does not have

to grow with every patch cycle, which could keep devices in service for longer; from December 2027 the EU's Cyber Resilience Act generally requires security updates for at least five years (EU 2024). Most of an agent's energy, though, goes into its model, on a local accelerator or in a data centre. Google puts the median text prompt to Gemini at 0.24 watt-hours (Elsworth et al. 2025), and the kernel underneath changes nothing about that figure. So we claim no saving we have not measured. From 2027 on, each station of the path in Section 7 is to report what the device draws at rest and at work, how that compares with the same task on Linux, and how long the hardware is supported. If minimality turns out to save nothing, we will report that as well.

**The Hurd trap.** GNU Hurd, a multi-server system on Mach, never reached broad use. Its history has more than one cause, but one lesson holds for Silent Zero. It has to start smaller than the vision, with one board, one kernel, one agent and one loop.

## 7 Five years back from 2031

The path is read backwards from a deliberately modest goal. By 2031, forty years after Linux, one kind of robot doing one defined task should run on Silent Zero from boot to actuator. Each station below is speculative, except the starting point and the verification dates the seL4 Foundation has announced. Each names its evidence and what kind of evidence it is. A **demonstration** shows that something works once, a **test** that it holds under defined load or attack, and a **proof** that it holds for every case its specification covers. A red team that fails to break something gives evidence and nothing more. A station that fails its check stops the path, so the plan can be shown to be wrong.

Read from the bottom up, each station makes the next one possible. The later stations depend on dates the seL4 Foundation has announced; if those slip, the path slips. Silent Zero 0.01 does not wait for them: on RISC-V, with one core, the MCS proof it needs exists today. Most depends on the first station,

where Silent Zero 0.01 is as small as Linux 0.01 was and just as easy to dismiss.

**A possible first robot.** *pib*, an open-source humanoid from Nuremberg with an education kit for schools, shows what this could look like; nothing of it is agreed or built. *pib* runs on a Raspberry Pi. Its servos are driven by small controllers with their own microcontrollers and open firmware, and a language-model agent can move its joints through MCP tools, which check the configured joint limits in software on the Linux side (*pib.rocks* 2026; Tinkerforge n.d.). A path would start without new hardware: limits, a speed bound and a heartbeat timeout move into the controller firmware, the loop's last word in miniature. That firmware then holds the last word and belongs to the trusted base as *assumed*. Next comes a broker in front of the agent's tools. The Raspberry Pi 4 can carry the step after that, seL4 with *pib*'s existing Linux as a driver island beside a native safety component (seL4 Microkit n.d.), but only as a demonstrator before Silent Zero 0.01, for three reasons that follow from Sections 3 and 4. Its kernel configuration is verified without MCS, whose proof for 64-bit Arm is not expected before 2027. Its Arm cores have no published instruction latencies, so no WCET bound is possible. And its public documentation describes no IOMMU, so the Linux island, which drives devices that write to memory directly, joins the trusted base. Silent Zero 0.01 itself runs on an open RISC-V board.

## Epilogue

In 1991, Linus Torvalds did not announce an ideal system. He announced a useful one, small enough to start and open enough to grow. Silent Zero asks for the same modesty in a different world, where the task is no longer a free Unix for a cheap PC but the smallest system on which an agent can act, and move, without being trusted.

None of its base is new. Capabilities date from the 1960s, the verified kernel from the 2000s and the Simplex pattern from 2001. What has changed is that the reasons that

kept them in niches are falling at the same time, just as the machines that need them are leaving the screen and entering the room.

The announcement of 1991 promised a system that “won’t be big”. Silent Zero makes the same promise. This time on purpose.

## Glossary

### Trust and proof

**Byzantine component.** A part that may fail in any way at all, including by lying or acting against the system on purpose. *In Silent Zero*: the assumption made about every agent (Section 1).

**Formal verification.** Proving with mathematics, checked by a computer, that a program does what its specification says, for every possible input. *In Silent Zero*: the standard for the parts a guarantee depends on; seL4's kernel already meets it (Sections 2.3, 3).

**Functional correctness.** The proven statement that an implementation behaves exactly as its specification describes. *In Silent Zero*: what seL4 has proven for its verified configurations (Section 3).

**Proven, tested, assumed.** Three levels of knowledge about a part: shown for every case its specification covers, shown to hold under defined load or attack, or simply trusted. *In Silent Zero*: every part of the trusted base carries one of these labels (Section 1).

**Reference monitor.** The mechanism that checks every access; it must always be invoked, be tamper-proof and be small enough to analyse and test. *In Silent Zero*: the reason the system must stay small (Section 1).

**Seam.** The place where a proven part meets an unproven one, and where the assumptions of one must match the guarantees of the other. *In Silent Zero*: kept to a minimum on the two critical paths, and each one is specified (Section 6).

**Trusted base.** The parts that must work correctly for a guarantee to hold; if one of them fails, the guarantee fails with it. *In Silent Zero*: kept small and listed part by part (Sections 1, 3).

## Kernel and time

**Microkernel.** A kernel that does only what must run with full privilege, such as isolation, messages and scheduling, and leaves everything else to ordinary programs. *In Silent Zero:* seL4 (Section 3).

**seL4.** A microkernel from the L4 family and the first general-purpose kernel with a machine-checked proof of functional correctness. *In Silent Zero:* the kernel (Sections 2.1, 3).

**Mixed criticality (MCS).** Running parts that matter to different degrees on one machine, so that a less important part can never take time from a more important one. *In Silent Zero:* seL4's MCS extensions make processor time a protected resource (Sections 3, 4).

**Scheduling context.** In seL4 MCS, a kernel object that grants a program a budget of processor time per period; the kernel enforces it. *In Silent Zero:* limits how much time an agent can use (Sections 3, 4).

**Real time.** Computing in which a result is only correct if it arrives before its deadline. *In Silent Zero:* required for every loop that moves something; provable timing is a goal, not yet a property (Sections 1, 4).

**Worst-case execution time (WCET).** The longest a piece of code can take to run; a safe upper bound on it is needed to prove that deadlines hold. *In Silent Zero:* exists for seL4 only on an old Arm core; being rebuilt for RISC-V (Section 4).

**Timing channel.** A leak in which one program learns secrets from how long another program's operations take. *In Silent Zero:* closed only with hardware support that exists as prototypes on open RISC-V cores (Section 4).

**Protection domain.** An isolated program in a static seL4 system, with its own memory and only the access it was given. *In Silent Zero:* every agent and every component runs in its own (Section 3).

**Static system.** A system whose structure, programs, channels and shared memory are fixed when the image is built and cannot change at run time. *In Silent Zero:* the whole

system is declared in one description (Section 3).

**Microkit.** seL4's framework for building static systems from protection domains, channels and shared memory. *In Silent Zero:* turns the system description into the bootable image (Section 3).

**LionsOS.** A modular, static operating system on seL4 and Microkit from the group behind seL4. *In Silent Zero:* the base to start from (Section 5).

**Multikernel.** A configuration that runs one separate kernel instance per processor core instead of one kernel for all cores. *In Silent Zero:* how a proven system will later use several cores (Section 6).

## Agents and authority

**Agent.** A program that reads, decides and acts on its own, usually driven by a language model. *In Silent Zero:* an untrusted tenant with a time budget (Sections 1, 3).

**Capability.** An unforgeable token that both names a resource and grants a specific right to it; whoever does not hold it cannot use the resource. *In Silent Zero:* the only way authority is passed (Sections 2.4, 3).

**Capability broker.** The single component that hands out authority: limited, time-bound and revocable. *In Silent Zero:* part of the trusted base; tested first, proven later (Sections 3, 5).

**Confused deputy.** A program that misuses authority given to it by one party while acting on behalf of another. *In Silent Zero:* the old name for what prompt injection does to agents (Section 2.4).

**Prompt injection.** Instructions hidden in data, such as a web page or an email, that steer an agent reading it. *In Silent Zero:* not solved, but limited in what it can reach (Sections 2.4, 6).

**Information-flow control.** Tracking where data comes from and where it may go, so that it cannot leave through a channel that is otherwise allowed. *In Silent Zero:* needed in addition to capabilities (Section 6).

**Model Context Protocol (MCP).** An open

protocol that connects agents to tools and data sources. *In Silent Zero*: the way agents reach tools, always through the broker (Section 3).

### Robots and safety

**Action model (VLA).** A model that turns what a robot sees and is told into motor commands; VLA stands for vision-language-action. *In Silent Zero*: plays the complex, unproven controller (Sections 2.4, 4).

**Action chunking.** Emitting several motor commands at once, so that the robot keeps moving while the model computes the next ones. *In Silent Zero*: why the motion path needs real time (Section 2.4).

**Simplex architecture.** A safety pattern in which a complex controller runs the system while a simple, trustworthy one takes over before the system leaves a safe region. *In Silent Zero*: the agent proposes, the verified loop disposes (Section 4).

**Safety controller.** The simple controller in the Simplex pattern, with the last word over every motion. *In Silent Zero*: part of the trusted base, at the highest criticality (Sections 3, 4).

**ROS 2.** The Robot Operating System, the most common middleware that connects the software parts of a robot. *In Silent Zero*: shown running on seL4 in proofs of concept (Section 2.4).

### Hardware

**RISC-V and CVA6.** RISC-V is an open instruction set anyone may implement; CVA6 is an open-source RISC-V core whose timing can be fully inspected. *In Silent Zero*: the first platform, because only there is a timing proof within reach (Sections 4, 7).

**Root of trust.** A small piece of hardware that is trusted to measure and check the first code a machine runs. *In Silent Zero*: the bottom of the trusted base; OpenTitan is an open one (Section 3).

**OpenTitan.** An open-source silicon root of trust, in production since 2026. *In Silent Zero*: the open option for the root of trust (Sections 3, 5).

**IOMMU.** Hardware that restricts which

memory a device may read and write directly (DMA, direct memory access). *In Silent Zero*: without a proven IOMMU configuration, a driver with such a device joins the trusted base (Sections 3, 7).

**Firmware.** Code that runs below or before the operating system, often with more privilege than the kernel. *In Silent Zero*: trusted and assumed; it cannot be confined by any kernel (Sections 3, 6).

**CHERI.** A hardware extension that makes pointers unforgeable, so that memory errors are caught by the processor. *In Silent Zero*: used where available (Sections 3, 5).

**Driver island.** A Linux virtual machine that runs drivers Silent Zero cannot replace, such as those for graphics and AI accelerators. *In Silent Zero*: on the agent's side, outside the trusted base where an IOMMU confines its devices (Section 3).

### References

- Amar, S. et al. (2023): [CHERIoT: Complete Memory Safety for Embedded Devices](#). Proc. MICRO '23, 641–653.
- Anderson, J. P. (1972): [Computer Security Technology Planning Study](#). Vol. II, ESD-TR-73-51, U.S. Air Force Electronic Systems Division.
- Anthropic (2025): [Donating the Model Context Protocol and establishing the Agentic AI Foundation](#). 9 December 2025.
- Anthropic (2026a): [Project Glasswing: Securing critical software for the AI era](#). 7 April 2026.
- Anthropic (2026b): [Project Glasswing: An initial update](#). 22 May 2026.
- Apple (n.d.): [XNU kernel source, README](#). Accessed 19 October 2026.
- Black, K.; Galliker, M. Y.; Levine, S. (2025): [Real-Time Execution of Action Chunking Flow Policies](#). arXiv:2506.07339.
- Bower, L. (2026): [Cohesix: a research operating system for edge AI](#). Project page, accessed 20 October 2026.
- Costa, M. et al. (2025): [Securing AI Agents with Information-Flow Control](#). arXiv:2505.23643.
- DARPA (2025): [AI Cyber Challenge marks pivotal inflection point for cyber defense](#). August 2025.
- DeBenedetti, E. et al. (2025): [Defeating Prompt Injections by Design](#). arXiv:2503.18813.
- Dennis, J. B.; Van Horn, E. C. (1966): Programming Semantics for Multiprogrammed Computations. Communications of the ACM 9(3).

- DiBona, C.; Ockman, S.; Stone, M. (eds.) (1999): [Open Sources: Voices from the Open Source Revolution, Appendix A: The Tanenbaum–Torvalds Debate](#). O'Reilly.
- ELISA (2026): [Enabling Linux in Safety Applications project expands premier membership with NVIDIA](#). Linux Foundation, 26 February 2026.
- Elsworth, C. et al. (2025): [Measuring the environmental impact of delivering AI at Google Scale](#). arXiv:2508.15734.
- EU (2023): [Regulation \(EU\) 2023/1230 of the European Parliament and of the Council on machinery](#). Official Journal of the European Union; applies from 20 January 2027; Annex I, Part A, point 5.
- EU (2024): [Regulation \(EU\) 2024/2847 on horizontal cybersecurity requirements for products with digital elements \(Cyber Resilience Act\)](#). Official Journal of the European Union; main obligations apply from 11 December 2027.
- Felt, A. P.; Ha, E.; Egelman, S.; Haney, A.; Chin, E.; Wagner, D. (2012): [Android Permissions: User Attention, Comprehension, and Behavior](#). UC Berkeley, UCB/EECS-2012-26.
- Fonseca, P. et al. (2017): [An Empirical Study on the Correctness of Formally Verified Distributed Systems](#). Proc. EuroSys '17.
- Google (2022): [Announcing KataOS and Sparrow](#). Google Open Source Blog, October 2022.
- Google (2025): [Building the next generation of physical agents with Gemini Robotics-ER 1.5](#). 25 September 2025.
- Google (2026): [OpenTitan shipping in production](#). Google Open Source Blog, March 2026.
- Greshake, K. et al. (2023): Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. Proc. AISec '23, 79–90. [arXiv:2302.12173](#).
- Hardy, N. (1988): [The Confused Deputy \(or why capabilities might have been invented\)](#). ACM SIGOPS Operating Systems Review 22(4), 36–38.
- He, B.; Li, Z.; Sun, W.; Yao, Y.; Chen, T.; Ma, X.; Su, Z. (2026): [Neuro-Symbolic Proof Generation for Scaling Systems Software Verification](#). Proc. OSDI '26. Preprint arXiv:2603.19715.
- Heiser, G. (2026): [MCS safety – an OS perspective](#). Award talk, RTAS '26.
- Heiser, G.; Elphinstone, K. (2016): L4 Microkernels: The Lessons from 20 Years of Research and Deployment. ACM Transactions on Computer Systems 34(1).
- Heiser, G. et al. (2025): [Fast, Secure, Adaptable: LionsOS Design, Implementation and Performance](#). arXiv:2501.06234.
- Hubbard, J. (2026): [agentOS: an operating system designed from the core up for agents](#). Source repository, accessed 20 October 2026.
- Klein, G. et al. (2009): [seL4: Formal Verification of an OS Kernel](#). Proc. SOSP '09.
- Klein, G.; Andronick, J.; Kuz, I.; Murray, T.; Heiser, G.; Fernandez, M. (2018): [Formally Verified Software in the Real World](#). Communications of the ACM 61(10), 68–77.
- Lamport, L.; Shostak, R.; Pease, M. (1982): [The Byzantine Generals Problem](#). ACM Transactions on Programming Languages and Systems 4(3), 382–401.
- Lattuada, A. et al. (2024): Verus: A Practical Foundation for Systems Verification. Proc. SOSP '24.
- Leemhuis, T. (2025): [Zahlen, bitte! Linux – soon to weigh 40 million lines of code](#). heise online, January 2025.
- Lide, D. (2026): [ARSENAL: Assured Robotics Embedded Systems and Applications in Ground Vehicles](#). Talk, seL4 Summit 2026, Vancouver.
- Liedtke, J. (1993): [Improving IPC by Kernel Design](#). Proc. SOSP '93, 175–188.
- Linux kernel (n.d.-a): [Landlock: unprivileged access control](#). Kernel documentation, accessed 20 October 2026.
- Linux kernel (n.d.-b): [Real-time preemption \(PRE-EMPT\\_RT\)](#). Kernel documentation, accessed 20 October 2026.
- Mangrum, B. (2025): [The Comedy of Computation: Or, How I Learned to Stop Worrying and Love Obsolescence](#). Stanford University Press.
- Markettos, A. T. et al. (2019): [Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals](#). Proc. NDSS 2019.
- Miller, M. S. (2006): Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. PhD thesis, Johns Hopkins University.
- Moore, S. W. (2026): [CHERI is being standardised, so why do we still need to do research?](#) CHERI Blossoms 2026, Cambridge, 27 March 2026.
- Nasr, M. et al. (2026): [The Attacker Moves Second: Stronger Adaptive Attacks Bypass Defenses Against LLM Jailbreaks and Prompt Injections](#). Proc. USENIX Security '26.
- Open Source Initiative (n.d.): [The Open Source Definition](#), criterion 6: No Discrimination Against Fields of Endeavor.
- Parker, L. (2022): [The seL4 Device Driver Framework](#). Talk, 4th seL4 Summit, Munich.
- pib.rocks (2026): [pib-backend: software for the pib robot](#). Source repository, accessed 20 October 2026.
- seL4 Foundation (n.d.-a): [Verified Configurations](#). Accessed 19 October 2026.
- seL4 Foundation (n.d.-b): [seL4 Roadmap](#). Accessed 19 October 2026.

- seL4 Microkit (n.d.): [Microkit User Manual, v2.3.0](#). Accessed 19 October 2026.
- Sha, L. (2001): [Using Simplicity to Control Complexity](#). IEEE Software 18(4), 20–28. doi:10.1109/MS.2001.936213.
- Sillitoe, S.; Basit, A.; Giacometti, M.; Heiser, G. (2026): [Resurrecting seL4’s WCET Analysis](#). OSPERT 2026.
- Tanenbaum, A. S. (2017): [An Open Letter to Intel](#). Vrije Universiteit Amsterdam.
- Tinkerforge (n.d.): [Servo Bricklet 2.0 firmware](#). Source repository, accessed 20 October 2026.
- Torvalds, L. (1991): Posting to comp.os.minix, 25 August 1991. Reproduced in [iTWire](#).
- Trail of Bits (2024): [LeftoverLocals: Listening to LLM responses through leaked GPU local memory](#). 16 January 2024.
- Trustworthy Systems (2026): [LionsOS](#). Project page, accessed 20 October 2026.
- Trustworthy Systems (n.d.-a): [Proof Statistics](#). UNSW Sydney.
- Trustworthy Systems (n.d.-b): [WCET: Worst-Case Execution-Time Analysis of seL4](#). UNSW Sydney.
- Tyler, S.; Legnani, A.; Zhao, J.; Shen, Z.; Wu, R.; Tanaka, M.; Heiser, G. (2026): [SMT-based Deductive Verification of Device Drivers using the Pancake-to-Viper Transpiler](#). Proc. PLOS ’26, Prague.
- Vander Stoep, J. (2025): [Rust in Android: move fast and fix things](#). Google, 13 November 2025.
- W3C (2016): [25 years ago the world changed forever](#). W3C Blog, August 2016.
- Wistoff, N.; Schneider, M.; Gürkaynak, F. K.; Heiser, G.; Benini, L. (2022): [Systematic Prevention of On-Core Timing Channels by Full Temporal Partitioning](#). doi:10.1109/TC.2022.3212636.
- Wistoff, N.; Heiser, G.; Benini, L. (2024): [fence.t.s: Closing Timing Channels in High-Performance Out-of-Order Cores through ISA-Supported Temporal Partitioning](#). APPLEPIES 2024, Turin.
- Yang, C. et al. (2025): [AutoVerus: Automated Proof Generation for Rust Code](#). Proc. OOPSLA ’25.
- Zhang, Y. (2026): [Agent libOS: A Runtime Substrate for Capability-Controlled Self-Evolving LLM Agents](#). arXiv:2606.03895.
- Zhao, Z. et al. (2026): [AgenticOS: An Intent-Oriented Secure Operating System Architecture for Autonomous AI Agents](#). arXiv:2606.21129.