

All watched over by machines of loving grace

Dominic P. Mulligan  

Automated Reasoning Group, Amazon Web Services¹

Abstract

Modern operating systems are typically built around a trusted system component called the *kernel* which amongst other things is charged with enforcing system-wide security policies. Crucially, this component must be kept isolated from untrusted software at all times, which is facilitated by exploiting machine-oriented notions of separation: private memories, privilege levels, and similar.

Modern proof-checkers are typically built around a trusted system component called the *kernel* which is charged with enforcing system-wide soundness. Crucially, this component must be kept isolated from untrusted automation at all times, which is facilitated by exploiting programming-language notions of separation: module-private data structures, type-abstraction, and similar.

Whilst markedly different in purpose, in some essential ways operating system and proof-checker kernels are tasked with the same job, namely enforcing system-wide invariants in the face of unbridled interaction with untrusted code, yet the mechanisms through which the two types of kernel protect themselves are significantly different. In this paper, we introduce *Supervisory*, a prototype programmable proof-checking system for Gordon’s HOL that is organized in a manner more reminiscent of operating systems than typical LCF-style proof-checkers. In particular, *Supervisory* implements a kernel that executes at a relative level of privilege compared to untrusted automation, with trusted and untrusted system components communicating across a limited system call boundary to indirectly manipulate kernel objects managed by the *Supervisory* kernel via handles.

Unusually, *Supervisory* has no “metalanguage” in the LCF sense, as the language used to implement the kernel, and the language used to implement automation, need not be the same. Indeed, *any* programming language can be used to implement automation for *Supervisory* providing the resulting binary respects the *Supervisory* kernel calling convention and binary interface, with no risk to system soundness. Moreover, we observe that *Supervisory* allows arbitrary programming languages to be endowed with facilities for proof-checking, not only dedicated languages like Idris and Agda. Indeed, the handles that *Supervisory* uses to reference kernel objects under its management may be thought of as a form of *capability*, in the hardware sense. However, unlike typical capabilities, *Supervisory*’s capabilities are extremely expressive, essentially capturing the full expressive power of HOL, and could be used to enforce fine-grained correctness and security properties at runtime.

2012 ACM Subject Classification

Keywords and phrases Proof assistant design, operating systems, LCF, *Supervisory*, capabilities

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

1.1 On operating systems

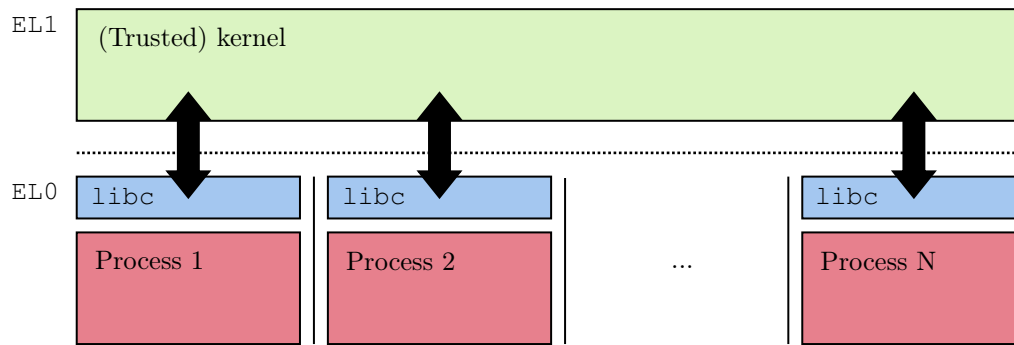
Most commodity operating systems—that is, Microsoft Windows and Unix-derivatives²—fit a common pattern and are architected around a relatively self-contained, trusted component typically called the system *kernel*.

At system initialisation, the kernel takes control of all system resources. Thereafter, untrusted user-space applications wishing to make use of a device, for example, must pass through the kernel in order to do so, which can either choose to gate or allow access. Moreover,

¹ All work done whilst at Arm Research

² To guard against quibbling over research operating systems like exokernels and similar which can be argued not to fit this pattern

XX:2 Supervisory system description



■ **Figure 1** A schematic of the typical system organization of a commodity operating system and its associated user-space. The kernel (in green) executes at a relative level of privilege compared to user-space (red) enforced by hardware—here, we follow the Arm convention and show the kernel executing at EL1 and user-space at EL0. The two communicate across a system call boundary (dashed line) using system calls (black arrows), with user-space typically making use of a library such as `libc` (blue) to abstract over this process.

the kernel introduces a process abstraction in user-space, and the kernel tasks itself with ensuring that processes are always isolated from each other. The kernel is therefore *the* key component responsible for enforcing system-wide policies. It is therefore imperative that the kernel is able to isolate itself sufficiently from untrusted user-space software at all times.

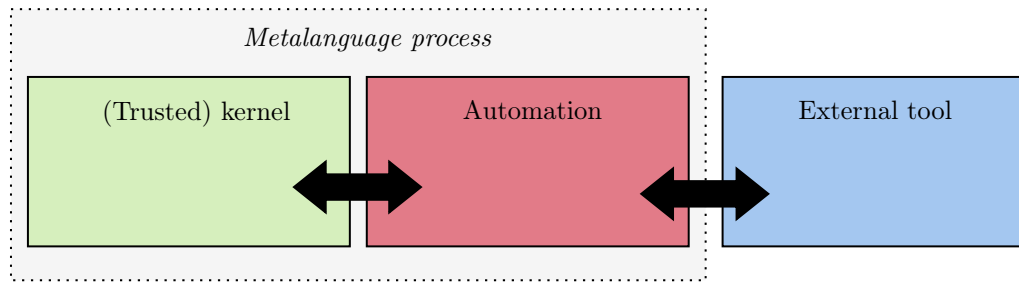
To help the kernel self-isolate, modern microprocessors are equipped with a number of features dedicated to this task. One key feature are *exception levels*³ which introduce a notion of *privilege* into the system. Here, software executing at a relative level of privilege—for example, an operating system kernel—is allowed to program sensitive system registers, controlling how the system operates, and “peer in” and potentially modify the runtime state of software executing at a lower-level of privilege. The kernel may read data from, or write data to, a buffer ostensibly within the memory space of an untrusted user-space process, for example. In this sense, a kernel can “supervise” or “watch over” untrusted user-space.

Moreover, modern microprocessors also provide a form of memory management built around page tables. These data structures are used for virtualisation of the system memory—granting user-space software the illusion that they own the entire physical address space—and also for memory access and permissions checks. By correctly initialising and managing these page tables the kernel’s own data structures and scratch space can be kept isolated from untrusted user-space, and user-space processes can be kept isolated from each other.

Note that for systems software, isolation is therefore enforced by low-level machine mechanisms: namely separate address spaces, private parts of memory, and machine-enforced privilege checks on executing software.

Finally the kernel, by necessity, must expose a limited interface to untrusted user-space, which can be used to request access to system resources, or make other similar requests for intercession by the kernel. To do this, operating system kernels commonly exposes a suite of *system calls* to untrusted user-space, which can be invoked with dedicated machine instructions. On Arm platforms—with which the author is most familiar—these instructions induce a processor exception, which causes a *context switch* into the kernel, which handles the system call appropriately, before exiting the exception and returning the flow of program

³ This is Arm-specific terminology. *Privilege rings* are the analogous feature on Intel platforms



■ **Figure 2** A schematic of the system organisation of a typical LCF-style proof assistant. The trusted kernel (green) is linked against untrusted automation (red) existing within the same metalanguage process (dotted line) and communicate with each other using the kernel’s API (black arrow). External tools existing as separate processes (blue, must communicate with a shim layer written in the proof assistant’s metalanguage to access the kernel (black arrow).

control back to the calling user-space program. In this respect, from user-space’s point-of-view, system calls have the appearance of very CISC-like machine instructions, and the operating system kernel essentially presents itself as “silicon by other means”.

For all of this to work, user-space and the kernel must work together, and adopt a common calling convention describing how arguments and results are passed back-and-forth across the system call interface, agree on a binary interface detailing how system calls are identified and errors are reported, and so on and so forth. However, crucially, it is *generally* not the case that the operating system kernel and untrusted user-space applications need be written in the same programming language for this all to work. Whilst most operating system kernels are written in some C-language derivative, user-space programs can be written in a variety of languages, and can all make use of system resources exposed by the kernel’s system call interface as long as they are capable of adhering to the calling convention and binary interface expected by the kernel. Whilst—on Linux at least⁴—this system call interface is abstracted over by the C library, `libc`, this is generally just a convenience, and user-space software can always invoke system calls directly if wanted by invoking the correct machine instruction and adhering to the appropriate calling convention. In this respect, the C-language may have prominence of the favoured language of system implementation, but by-and-large it is not *special* or given an unduly prominent status by the kernel itself.

1.2 On programmable proof-checkers

Most modern proof assistants—that is, systems in the wider HOL family, Coq, Matita, PRL, and similar—fit a common pattern and are architected around a relatively self-contained, trusted component typically called the system *kernel*.

The system kernel is the sole component that can authenticate claims as legitimate theorems of the implemented logic. Untrusted automation, residing outside of the kernel, must “drive” the kernel to derive a theorem on its behalf. The kernel is therefore *the* component responsible for ensuring system-wide soundness. It is therefore imperative that the kernel is able to isolate itself sufficiently from untrusted automation at all times. This method of system organisation is known as *the LCF approach* after the system which introduced it, and is the most common way of organising proof-checking systems today.

⁴ Note that this is not the case on some variants of BSD Unix, for example Apple MacOS, which generally consider the programming interface of the system C library as the interface of the kernel, proper

Most modern proof assistants tend to be written in a “metalanguage”, typically a strongly-typed functional programming language, for example an ML derivative such as OCaml or SML. These types of programming language offer strong modularity and abstraction features, which the kernel exploits to hide its own data structures from untrusted automation and expose a carefully limited API for proof-construction and manipulation. Notably, in an LCF-style system, the *only* mechanism automation has for constructing an authenticated theorem is by using this API, with the inference (or typing) rules of the logic exposed as “smart constructors” manipulating an abstract type of theorems or well-typed terms.

Untrusted automation and the system kernel are linked together, and reside side-by-side in the same process when the system is executed. As a result, system soundness ultimately rests on the soundness of the implementation metalanguage’s type-system—specifically its ability to correctly isolate module-private data structures, that is its ability to correctly enforce type abstraction. Moreover, the system metalanguage is, in a sense, unique amongst all programming languages, in that it is the only language capable of interfacing with the kernel, which is, after all, “just” a module written within that language like any other. Whilst external tools, and automation written in other languages, can interface with the kernel, it must do so indirectly, making use of a shim layer written in the system metalanguage.

1.3 The Supervisory system

In many respects, as the text above intimates, the role of the kernel in an operating system and the role of the kernel in a proof-checker is, abstractly, *essentially* the same. Namely, both components must enforce system-wide invariants in the face of—and correctly isolate themselves from—unbridled interaction with untrusted code. However, the two mechanisms through which these different types of kernel self-isolate are very different: for operating system kernels⁵ self-isolation is enforced using machine-oriented mechanisms; for LCF-style proof-checkers, self-isolation is enforced using programming language-oriented mechanisms.

In this paper we introduce *Supervisory*, a novel programmable proof-checker for Gordon’s HOL whose design will be further discussed in detail in Section 2. (Note that many of the ideas presented henceforth are logic-independent, and though we have chosen to use HOL in our prototype, can be applied to a wide array of other logics with minimal changes.) Of note, Supervisory’s system design has more in common with the typical system organisation of an operating system than comparable implementations of HOL. Specifically, the Supervisory kernel executes at a relative level of privilege when compared to untrusted automation, which can be thought of as executing in something akin to Supervisory’s version of “user space”. The trusted kernel, and untrusted user space, communicate across a system call boundary, which must be carefully designed in order to ensure system soundness.

One immediate consequence of this design is that the Supervisory kernel immediately takes on a very different character to the typical LCF kernel. Specifically, all of the typical paraphernalia of a HOL implementation—type formers, types, constants, terms, and theorems—are kept safely under the management of the Supervisory kernel itself, in private memory areas, and never exposed directly to user-space but rather referenced indirectly by handles. These handles may be thought of as somewhat akin to pointers into Supervisory’s private memories, or analogous to file handles in typical systems software, and are used by user-space software to denote a kernel object that the Supervisory kernel

⁵ Barring unikernels like Mirage, which are in some respects quite similar to LCF-style proof-checkers in this regard

should manipulate or query on user-space’s behalf.

Remarkably, Supervisory is also not implemented in a typed functional programming language, as is typical of most programmable proof-checkers, but rather in the *unsafe* systems programming language, Rust. Note that this implementation decision introduces no risk to system soundness which ultimately rests on the separation of kernel-private data from user-space, using privilege and private memories, not on programming language features. Moreover, as user-space and kernel communicate across a defined system call interface, untrusted user-space may also be written in *any* programming language capable of producing code that is binary-compatible with the Supervisory kernel. Supervisory therefore has no “metalanguage” in the sense understood by users of LCF-style provers, but rather an implementation language, with automation potentially written in multiple languages.

We implement Supervisory as a WebAssembly (or Wasm, henceforth) host. This allows us to experiment with the essential ideas behind Supervisory—namely isolating the kernel using private memory areas, which the “user-space” Wasm program executing under the host cannot access, and which exposes services via system call-like interface—without become bogged down in extraneous detail associated with booting a real machine, and interacting with real hardware. Moreover, we harness work on porting compiler and linker toolchains, allowing our “user space” to be written in any language capable of targeting Wasm.

2 Kernel design

2.1 Implemented logic

Supervisory implements a variant of Gordon’s HOL, a classical higher-order logic which can be intuitively understood as Church’s Simple Theory of Types extended with ML-style top-level polymorphism. We introduce the basics of this logic here, introducing just enough material so that the unfamiliar reader can follow the rest of the paper.

We fix a set of *type variables* and use α, β, γ , and so on, to range arbitrarily over them. With these, we work with a grammar of *types* generated by the following recursive grammar:

$$\tau, \tau', \tau'' ::= \alpha \mid f(\tau, \dots, \tau')$$

Here f is a *type-former* which has an associated *arity*—a natural number indicating the number of type arguments that it expects. If all type-formers within a type are applied to their expected number of types we call the type *well-formed*—that is, arities introduce a trivial or degenerate form of *kinding* for types. We will only ever work with well-formed types in Supervisory. We write $tv(\tau)$ for the *set of type-variables* appearing within a type, and write $\tau[\alpha := \tau']$ for the *type substitution* replacing all occurrences of α with τ' in the type τ . From the outset we assume two primitive type-formers—essentially built-in to the logic itself, and necessary to bootstrap the rest of the logic: **bool**, the type-former of the Boolean type (and in HOL, also propositions), with arity 0, and $- \Rightarrow -$, the type-former of the HOL function space, with arity 2. Note we will abuse syntax and also write **bool** for the *type* of Booleans and propositions, and also write $\tau \Rightarrow \tau'$ for the function space type.

For each well-formed type τ we assume a countably infinite set of *variables* and *constant symbols*. We use x_τ, y_τ, z_τ , and so on, to range over the variables associated with type τ , and use C_τ, D_τ, E_τ , and so on, to also range over the constants associated with type τ . With these, we recursively define *terms* of the explicitly-typed λ -calculus, as follows:

$$r, s, t ::= x_\tau \mid C_\tau \mid rs \mid \lambda x:\tau. r$$

■ **Figure 3** The Natural Deduction relation for Gordon’s HOL.

187 Note that there is an “obvious” typing relation on terms, which we omit here, and we call any
 188 term with a type *well-typed*. We will only ever work with well-typed terms in Supervisory,
 189 and call terms with type **bool** *formulae*. We will use ϕ, ψ, ξ , and so on, to suggestively
 190 range over terms that should be understood as being formulae. We work with terms up-to
 191 α -equivalence, write $fv(r)$ for the set of *free variables* appearing within the term r , write
 192 $r[x_\tau := t]$ for the usual *capture-avoiding substitution* on terms, and write $r[\alpha := \tau]$ for the
 193 recursive extension of the type substitution action to terms.

194 Like with type-formers, from the offset we assume a collection of typed constants needed
 195 to bootstrap the rest of the logic, summarised in the table below:

	$=$	$\alpha \Rightarrow \alpha \Rightarrow \mathbf{bool}$
	\top, \perp	\mathbf{bool}
	\neg	$\mathbf{bool} \Rightarrow \mathbf{bool}$
196	$\wedge, \vee, \longrightarrow$	$\mathbf{bool} \Rightarrow \mathbf{bool} \Rightarrow \mathbf{bool}$
	\forall, \exists	$(\alpha \Rightarrow \mathbf{bool}) \Rightarrow \mathbf{bool}$
	ϵ	$(\alpha \Rightarrow \mathbf{bool}) \Rightarrow \alpha$

197 The majority of the constants above correspond to the usual logical constants and connectives
 198 of first- or higher-order logic and are introduced without further explanation. Only the latter
 199 ϵ constant—Hilbert’s description operator, a form of choice—may be unfamiliar. In HOL,
 200 this can be used to “select”, or “choose” an element of a type according to some predicate,
 201 and is otherwise undefined if no such element exists. Note that all HOL types are inhabited
 202 by at least one element. We adopt usual mathematical conventions and precedence levels
 203 when writing terms making use of these constants, writing $\phi \longrightarrow \psi$ instead of $(\longrightarrow \phi)\psi$,
 204 for example. We also suppress explicit type substitutions required to make terms involving
 205 polymorphic types well-typed, writing $\forall x_\tau. \phi$ instead of $\forall[\alpha := \tau](\lambda x_\tau. \phi)$, for example.

206 We call a finite set of formulae a *context*, ranged arbitrarily over by $\Gamma, \Gamma', \Gamma''$, and so on,
 207 and introduce a two-place *Natural Deduction relation* between contexts and formula using
 208 the rules in Figure 3. We write $\Gamma \vdash \phi$ to assert that a derivation tree rooted at $\Gamma \vdash \phi$ and
 209 constructed according to the rules presented in Figure 3 exists.

210 2.2 The kernel state

211 The Supervisory kernel maintains several private *heaps* containing *kernel objects* of various
 212 kinds, corresponding to the various objects that constitute the HOL logic: type-formers,
 213 types, constants, terms, and theorems.

214 **2.3 The kernel system call interface**

215 **2.4 Programming the kernel**

216 **2.5 Specifying kernel functions**

217 **3 Capabilities on steroids**

218 **4 Conclusions**

219 **4.1 Related work**

220 **4.2 Future work**

