AFRL-RI-RS-TR-2018-297

COMPOSITIONAL RESOURCE-ADAPTIVE CERTIFIED SYSTEM SOFTWARE

YALE UNNIVERSITY

DECEMBER 2018

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2018-297 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
STEVEN DRAGER
Work Unit Manager

/ S /
JOHN D. MATYJAS
Tech Advisor, Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
DECEMBER 2018	FINAL TECHNICAL REPORT	AUG 2016 – AUG 2018

4. TITLE AND SUBTITLE

COMPOSITIONAL RESOURCE-ADAPTIVE CERTIFIED SYSTEM SOFTWARE

5a. CONTRACT NUMBER
FA8750-16-2-0274

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Zhong Shao

5d. PROJECT NUMBER
BRAS

5e. TASK NUMBER
YA

5f. WORK UNIT NUMBER
LE

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Yale University
New Haven, CT 06520

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

DARPA
675 North Randolph Street
Arlington, VA 22203-2114

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2018-297

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The BRASS program aims to build resource adaptive systems that can operate under widely differing environments. This seedling project addressed several important technical challenges for building long-lived resource adaptive system software. CertiKOS layers were extended with formal resource models. New thread objects were added as basic building blocks and used to model the hardware and virtual device layers. A general mechanism for managing available CPU resources and support compositional layered refinement for concurrent programs on both single core and multicore machines was provided. A fully verified preemptive OS kernel with temporal and spatial isolation was developed.

15. SUBJECT TERMS
Program verification, Formal methods, Resource-Aware Abstraction Layers, Operating Systems, Concurrency, Temporal and Spatial Isolation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**STEVEN DRAGER**
U	U	U	UU	38	19b. TELEPHONE NUMBER (*Include area code*) **NA**

Contents

List of Figures

1 Summary

The DARPA Building Resource Adaptive Software Systems (BRASS) program aims to build resource adaptive software systems that can operate under widely differing environments and can last more than 100 years. We believe that the BRASS vision would greatly benefit from having a *high-assurance* and *resource-adaptive* hypervisor Operating System (OS) kernel. Because any bug in the OS kernel can compromise the resource guarantee of the entire system, it is important that the kernel is not only adaptive but also high-assurance.

In this seedling project, we tackle two major technical challenges for building long-lived resource adaptive system software:

- *Formal resource and environment model.* A formal resource and environment model is a prerequisite for reasoning about the resource usage of a program and its adaptation to changes in its ecosystem. Unfortunately, due to the low-level nature of hardware resources, such a model often does not exist, and even if it does, it is informal and too low level; and it has a huge gap with the high-level notions of resources in today's programming languages.

- *Compositional specification for communicating threads and hardware devices.* Modern system software often rely on a collection of communicating threads so it can best adapt to the available CPU resources on multicore machines. It is unclear how to specify the behaviors and resource usage of these threads in a modular way using existing technology.

In the last two years, we have developed a general formal framework for reasoning about resources and environment contexts. An OS kernel manages and multiplexes a large variety of physical or virtual resources at different abstraction levels, but these resource managers are often ad hoc and intertwined with other kernel components. We aggressively decompose an otherwise complex software stack into a large number of simple but carefully designed *resource-aware abstraction layers*. All these layers share a unified abstract notion of resource objects with proper cost models and resource-safety properties.

We have applied our formal framework to turn Certified Kit Operating System (CertiKOS) into a *resource-aware* certified OS kernel. The CertiKOS kernel [Gu et al., 2016] and its device drivers are decomposed into many certified abstraction layers. It can also support various forms of kernel- and user-level concurrency on multicore machines. We support *resource-aware concurrency* by providing resource-aware synchronization primitives and real-time schedulers. We have also developed a fully verified preemptive OS kernel with temporal and spatial isolation.

We believe that our work contributes significantly to the BRASS vision and complements the current BRASS-funded efforts. Our formal resource framework and automated tools can be applied to facilitate building resource-adaptive software systems. The Concurrent and Realtime CertiKOS hypervisor kernels could be used to build a stronger assurance case for end-to-end resource guarantees.

2 Introduction

Our research is built on top of the clean-slate CertiKOS hypervisor kernel [Gu et al., 2016] developed by PI Shao and his team at Yale. In contrast to traditional OS kernels which use the hardware-enforced "red line" to define a single system call Application Programmer Interface (API), CertiKOS is a high-assurance, extensible kernel that provides a large number of abstraction layers enforced via formal specification and proofs. Programming with certified abstraction layers enables a disciplined way of composing a large number of components in a complex system. Without using layers, we might have to consider arbitrary interactions between the current module and its environment: an invariant held in one function can be easily broken when it calls a function defined in another module. A layered approach aims to sort and isolate all components based on a carefully designed set of abstraction levels so we can reason about one small abstraction step at a time. This can dramatically simplify the environment model that needs to be considered at each layer. The compositional layered architecture also allows CertiKOS to support rich kernel extensions and certified ring-0 processes, and it is the main reason that made the CertiKOS effort scale.

The goals of our seedling project are to develop a general formal resource framework and then apply it to build new resource-aware high-assurance OS kernels. More specifically, we extend CertiKOS's abstraction layers with a detailed resource specification that also specifies adaptive variants for anticipated changes in the underlying layer implementation. To overcome a current limitation of CertiKOS, we also develop a new class of layered thread objects that model physical devices, interrupt controllers, device drivers, or kernel daemon threads.

We use *machine-checked formal verification:* we prove mathematical theorems about programs and their behavior, in formal logic. Of course, to a mathematician, most of these theorems would look like "engineering:" they're very "applied" properties of specific software components. And indeed, this is engineering—it's what software engineering should look like in the 21st century, especially for high-assurance systems. The formal logic that we use is called the Calculus of Inductive Constructions (CiC), and we use the Coq proof assistant to build and check our proofs. Within Coq, we use CiC to build application-specific logics—such as a logic for proving correctness of C programs. We have primarily focused on building resource-aware OS kernels. All kernel components must be written in some programming language, typically a low-level language such as C or assembly language; and must be translated to machine-language by a *verified compiler* such as CompCert [Leroy, 2005–2014].

In our research for this seedling project, we have successfully carried out the following three lines of work. First, we extended CertiKOS layers with formal resource models and we added new thread objects as basic building blocks and use them to model the hardware and virtual device layers. Second, we developed a general mechanism for managing available CPU resources and support compositional layered refinement for concurrent programs on both single core and multicore machines. Third, we developed a fully verified preemptive OS kernel with temporal and spatial isolation.

Resource-Aware CertiKOS We extended each CertiKOS abstraction layer with a detailed resource specification. The specification not only defines the cost models of all abstract states and primitives (in each layer object), but also specify adaptive variants for anticipated changes in the underlying layer implementation. Resource specifications can range from simple resource counters (e.g., memory or stack usages) to more sophisticated potential functions (where tokens are spread across different parts of an abstract state).

One current limitation of CertiKOS is that it does not model any physical devices, interrupt controllers, device drivers, or kernel daemon threads. We developed *thread objects*, a new class of layered object, to model these kernel features. A thread object has its own abstract state, resource model, and adaptive invariants, but its behaviors follow a particular communication protocol. Each thread object is given a formal specification using technologies based on session types. We then used these thread objects to build accurate "layered" models for all kinds of hardware or virtual devices and interrupt controllers.

We developed a new extensible architecture for building certified OS kernels with device drivers. Instead of mixing the device drivers with the rest of the kernel (since they both run on the same physical CPU), we treat the device drivers for each device as if they were running on a "logical" CPU dedicated to that device. This novel idea allows us to build up a certified hierarchy of extended abstract devices over the raw hardware devices, meanwhile, systematically enforcing the isolation among different "devices" and the rest of the kernel. All device layers were then connected to existing CertiKOS layers so they can be programmed and accessed at higher abstraction layers.

We introduced a notion of container, inspired by container objects in HiStar [Zeldovich et al., 2006]. Whenever a new agent (which will be a thread object) is created, a container is created for the agent that dynamically keeps track of its resource usage (e.g., memory, time slices). An agent's usage may increase for a few reasons, including a direct request for dynamically-allocated resources, or a successfully-handled page fault. Each container object is initialized with some maximum *quota*; any attempt for an agent to increase its usage beyond this quota will be denied by the kernel. Furthermore, the kernel maintains a mapping of agent ids to containers using a hierarchical tree structure. Whenever an agent's process makes a request to spawn a new process, the new container is added as a child to the requesting agent's container, and the new container's quota is taken from the requester's. Using this notion of container, we proved a theorem that agents' requests for additional resources will always be fulfilled as long as their quota is not exceeded.

Certified Concurrent Abstraction Layers Despite the importance of concurrent objects and a large body of recent work on shared-memory concurrency verification, there are no certified programming tools that can specify, compose, and compile concurrent layers to form a whole system [Chong et al., 2016]. Formal reasoning across multiple concurrent layers is challenging because different layers often exhibit different interleaving semantics and have a different set of observable events. Reasoning across these different abstraction levels requires a general, unified compositional semantic model that can cover all of these concurrent layers. It must also support a general "parallel layer composition rule" that can handle explicit thread control primitives (e.g., sleep and wakeup). It must also support vertical composition [Anderson and Dahlin, 2011] of these

concurrent layer objects [Herlihy and Shavit, 2008] while preserving both the linearizability and progress (e.g., starvation-freedom) properties.

Under the seedling project, we developed Certified Concurrent Absyraction Layers (CCAL)—a fully mechanized programming toolkit implemented in Coq [Barras et al., 1998] for building certified *concurrent* abstraction l ayers. CCAL consists of a novel compositional semantic model for concurrency, a collection of C and assembly program verifiers, a library for building layered refinement proofs, a thread-safe verified C compiler based on CompCertX [Gu et al., 2015], and a set of certified linking tools for composing multithreaded or multicore layers.

We define a certified concurrent abstraction layer as a triple $(L_1[A], M, L_2[A])$ plus a mechanized proof object showing that the layer implementation M, running on behalf of a thread set A over the interface L_1, indeed faithfully implements the desirable interface L_2 above.

The key novelty of our work is to build a formal compositional semantics model based upon ideas from game semantics [Murawski and Tzevelekos, 2016]. The observable behavior of each run of a program can be viewed as playing a game involving members of multiple threads: each participant contributes its play by appending an event into the global log; its semantics (a.k.a. strategy) is a deterministic partial function from the current log to its next move whenever the control is transferred back to the current thread. In other words, the semantics of a specific thread is defined o ver t hose t hreads i n i ts e nvironment. O ur s emantics i ntroduces f ormal environment models that specify how a thread would behave relative to the behavior of its environment.

Following Gu et al. [2015], certified concurrent layers enforce *termination-sensitive* contextual correctness property. In the concurrent setting, this means that every certified c oncurrent object satisfies not only a safety property (e.g., *linearizability*) [Herlihy and Wing, 1990, Filipovic et al., 2010] but also a progress property (e.g., *starvation-freedom*) [Liang et al., 2013].

Real-Time CertiKOS with Temporal and Spatial Isolation Real-time systems are computer systems that interact with their environment at a speed given by the environment, not the system itself. In particular, they must exhibit a predictable behavior in their reaction to environment events.

We consider *hard real-time*, namely all real-time tasks must meet their deadlines, given their periods and budgets. In a hard real-time system, there are usually many different components interacting with various parts of the environment at different rates. Ensuring that *all* interactions are done at the correct pace cannot be reduced to independent checks as various interactions may compete for shared resources, the first of which is the processor.

Under this seedling grant, we studied hard real-time operating system kernels from the perspective of formal methods. Our first goal i s to s pecify temporal i solation b etween t asks of a real-time OS kernel, which means that each task is given its full amount of budget within each period. This is essential for the proper operation of critical subsystems such as auto braking, flight control, etc. Despite the rich literature in formal reasoning of real-time scheduling or temporal properties of OS kernels [Andronick et al., 2016, Xu et al., 2016, Dutertre, 2000, Cerqueira et al., 2016], none of these works tackle the connection between the high-level temporal property and the low-level assembly code that is actually running on the hardware. Unlike them, we aim to bridge the gap. Our second goal is to prove spatial isolation between processes, ensuring that there is no information leak. Combining both goals together, we achieve a fully verified p reemptive OS

kernel with both temporal and spatial isolation, and all properties we prove on the abstract model will carry down to the assembly code level, which is connected with an extended machine model based on CompCert [Leroy, 2005–2014].

To this end, we build upon CertiKOS [Gu et al., 2016], an OS kernel formally verified in the Coq proof assistant [Barras et al., 1998], which provides formal proofs of correctness and non-interference between user processes [Costanzo et al., 2016]. Our contributions are the following: (1) an extended machine model that supports interrupts and enables us to reason about preemptions in user mode; (2) a fully verified OS kernel with fixed-priority real-time scheduling; (3) a novel application of supply functions [Liu, 2000] to reason about each task in isolation and connect temporal properties with the actual code; (4) formal proofs that our schedulability test (the Critical Instant Theorem [Liu and Layland, 1973] expressed in a different setting) indeed ensures that all tasks meet all deadlines; (5) a proof of non-interference between processes (both user and kernel ones); and (6) the implementation of a system call that provides resource measurement in a non-interfering way.

3 Methods, Assumptions, and Procedures

3.1 Basic Methodology and Assumptions

Modern computer systems consist of a multitude of abstraction layers (e.g., OS kernels, hypervisors, device drivers, network protocols), each of which defines an interface that hides the implementation details of a particular set of functionality. Client programs built on top of each layer can be understood solely based on the interface, independent of the layer implementation. Despite their obvious importance, abstraction layers have mostly been treated as a system concept; they have almost never been formally specified or verified. This makes it difficult to establish strong correctness properties, and to scale program verification across multiple layers.

Under the DARPA High Assurance Cyber Millitary Systems (HACMS) program, the CertiKOS team at Yale developed a novel language-based account of abstraction layers and show that they correspond to a strong form of abstraction over a particularly rich class of specifications which we call *deep specifications*. Just as *data abstraction* in typed functional languages leads to the important *representation independence* property, abstraction over deep specification is characterized by an important *implementation independence* property: any two implementations of the same deep specification must have *contextually equivalent* behaviors.

We take compositionality (especially semantic compositionality) as the most important structuring principle. Existing programming languages contain many features that are not compositional. Existing software development practices do not structure their software components based on their semantic dependencies. We insist on using compositional programming language features, compositional semantics, compositional linking mechanisms, compositional verified compilers, compositional concurrency constructs, and compositional program logics.

Our key principle is to decompose the specifications, the semantics, and the proofs of any large complex (software and hardware) system using certified abstraction layers. These layers correspond to traditional notions of components, but they are *certified*, meaning that they come with formal deep specifications and certified implementation. They can be composed horizontally and vertically; they also support concurrency (i.e., general parallel composition); and they can be compiled from one implementation language (e.g., C) into another (e.g., assembly). We use contextual refinement as a unifying mechanism to support certified linking and prove end-to-end functional correctness properties. We also use *fully abstract* deep specifications so security properties proved at higher abstraction levels can be propagated to the actual low-level implementation.

In the following, we give an overview of our abstraction-layer-based approach on verifying system software, first introduced in [Gu et al., 2015]. As in any other system verification, we associate every code module (a piece of code) a specification, and prove that the code meets its specification, or more formally, there is a *forward simulation* [Lynch and Vaandrager, 1995] from the module implementation to its specification. A specification of a module is a logical abstract representation of the module's behavior with the concrete implementation details hidden. For example, to specify operations on a doubly linked list stored in memory, we may logically interpret the complex in-memory data structure as a simple logical list and specify its *push* and *pop* operations as a simple list *append* and *remove* operations. To support this, the framework needs to provide a systematic way to hide the private memory state from its client, and replace them with *abstract states*

to specify the full functionality of each operation in the interface in terms of the *abstract states*. Furthermore, a complex system like a kernel module is normally implemented in a combination of the C and assembly language. Thus, the framework ought to be instantiated in both languages and provide a way to certifiably compile the C-based framework into the assembly-based one. The *certified abstraction layers* provide exactly such support.

C and Assembly Languages Used Our framework supports both a C-like language and an x86 assembly language called Clight and LAsm, respectively.

Clight [Blazy and Leroy, 2009] is a subset of C and is formalized in Coq as part of the CompCert project [Leroy, 2005–2014]. Its formal semantics relies on a memory model [Leroy and Blazy, 2008] that is not only realistic enough to specify C pointer operations, but also designed to simplify reasoning about non-aliasing of different variables. From the programmer's point of view, Clight avoids most pitfalls and peculiarities of C such as nondeterminism in expressions with side effects. On the other hand, Clight allows for pointer arithmetic and is a true subset of C: valid Clight programs are valid C programs with the same semantics. Such simplicity and practicality turn Clight into a solid choice for certified programming. Furthermore, the CompCert verified compiler provides strong guarantees on code obtained by compilation of Clight programs. However, Clight provides little support for abstraction, and proving properties about a Clight program requires intricate reasoning about data structures. This issue is addressed by our layer infrastructure. Our Clight code is automatically generated from standard C code through a tool called clightgen provided by CompCert. We directly verify the generated Clight code. Thus, correctness of the clightgen does not affect the correctness of the verified code.

LAsm is a superset of the CompCert x86 assembly language with more machine-dependent registers and instructions needed for implementing low level system software.

Layer Interface A layer interface L consists of the *abstract states*, *primitives*, a set of *invariants* on the abstract states, and *proofs* that all the primitives in the layer interface preserves the layer invariants. An *abstract state* could be a logical state that does not correspond to any physical state in the machine, but in most cases, it is a logical state that is abstracted from a concrete state in the registers or memory. Each *primitive* operates on the abstract states and is associated with an atomic specification. It is abstracted from a concrete, verified piece of the actual code. The *invariants* are preserved by all the primitives, the abstract states can only be accessed through calling one of the primitives, and the primitives execute atomically; therefore, the invariants hold at any moment during the system execution.

Code Module A code module M corresponds to a concrete piece of code in Clight or LAsm assembly. Note that a module M implemented on top of a layer interface L may call any of the primitives defined in L. However, the standard Clight semantics is unaware of either the abstract states or the abstract primitives defined in the layer interface. While we would like to support the new abstract states and primitives, we seek to minimize the impact on the existing proof infrastructure for program and compiler verification. Thus, we do not modify the semantics of basic

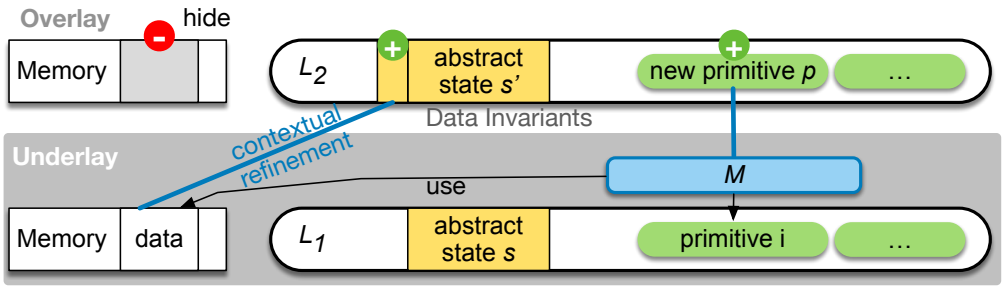

Figure 1: Layer-based contextual refinement

operations of Clight, but access the abstract states exclusively through the Clight's external function mechanism provided in CompCert. In addition, the external function mechanism is also used to model the interaction with the devices, such as input/output. Indeed, CompCert models compiler correctness through traces of events which can be generated only by external functions. CompCert axiomatizes the behaviors of external functions without specifying them, and only assumes they do not behave in a manner that violates compiler correctness. We use the external function mechanism to extend Clight with our primitive operations, and supply their specifications to make the semantics of external functions more precise. The semantics of LAsm is also instrumented accordingly to support the primitive calls in the assembly code. The verified Clight source code can be compiled by our extended CompCertX compiler [Gu et al., 2015] to the corresponding LAsm assembly in such a way that all proofs at the Clight level are preserved at the LAsm level. Then, the compiled LAsm modules and their proofs can be linked with the ones directly developed in LAsm.

Certified Layer A *certified layer* is a new language-based module that consists of a triple (L_1, M, L_2) plus a mechanized proof object showing that the layer implementation M that is built on top of the layer interface L_1 (the *underlay interface*), denoted $[\![M]\!]L_1$, is a *contextual refinement* of the desirable layer interface L_2 above (the *overlay interface*), as shown in Figure 1. A *deep specification* of L_2 captures everything *contextually observable* about running M over its underlay L_1. Once a certified layer (L_1, M, L_2) with its deep specification is built, there is no need to ever look at M again, since any property about M can be proven using L_2 alone.

The contextual refinement is proven by showing a forward simulation from L_2 to $[\![M]\!]L_1$ over a refinement relation. Thus, for every contextual refinement, we need to find a refinement relation R that can relate the system's states (including the abstract states) between the layer interface L_1 and L_2. In the above doubly linked list example, R needs to relate the in-memory doubly linked list of L_1 to the abstract logical list in L_2. In the case when there is no data abstraction between L_1 and L_2, R is simply an identity relation. To prove forward simulation, we need to prove that for every state (s_1, s_2) in R, and for every primitive p in L_2, if p takes the state from s_2 to s_2', then there exists zero or more steps in M which can take the state s_1 to s_1', where (s_1', s_2') is also in R.

In addition, the contextual refinement also needs to guarantee that the context code running on the overlay interface does not accidentally damage the underlay in-memory data by directly accessing the relevant memory. As shown in Figure 1, we achieve this by utilizing the CompCert

memory permissions [Leroy and Blazy, 2008] to hide the relevant memory region at overlay, which prevents the context code from accessing the relevant memory. These logical permissions do not correspond to any physical protection mechanism, but are used to ensure that the abstract machine at overlay gets stuck if any code tries to directly access this portion of memory. The safety proof of our entire system (the system never gets stuck) guarantees that such a situation never happens. Thus, the only way to affect the abstracted memory by any context code running over the overlay interface L_2 is to explicitly call relevant primitives in L_2.

Verification of Clight and LAsm functions Given that the majority of the system software are developed in C (Clight in our case), we need a good framework-level automation support to verify that C modules meet their specifications. In our framework, this proof is achieved semi-automatically through Coq tactic libraries implemented in the Coq's tactical language L_{tac}. Our primary proof tactic *cauto* consists of many components.

One main component is a verification condition generator that decomposes all the Clight expressions and statements, and produces conditions as subgoals for the further expression evaluation and statement execution based on the big-step semantics defined in CompCert. The only exception is the loop, whose verification conditions cannot be generated by simply applying the semantic rules. We developed a separate logic for loops, which requires the user to provide the loop invariants that are preserved on every iteration of the loop execution. Our proof is termination sensitive. Thus, the logic also requires a termination metric, a well-founded order of the type of the provided metric, and a proof that the metric decreases at every iteration of the loop according to the provided well-founded order.

The language Clight strictly follows the C standard and disallows the undefined behaviors described in the standard C semantics. Thus, these all become the preconditions in the semantic rules of the Clight language. For a reasonably realistic C module, *many* verification conditions are generated. Discharging the conditions after they are fully generated would be very inefficient. Instead, our *cauto* tactic integrates many of the theory solvers to discharge the subgoals on the fly as soon as they become provable.

First, to prevent the integer overflow, the Clight semantics requires every intermediate value in the middle of expression evaluations to be within the range regarding its type. In this way, most of the Clight code generates a huge set of arithmetic subgoals for checking value ranges. However, the standard *omega* tactic is too weak to prove most of the goals. We have incorporated the *cauto* tactic a powerful arithmetic solver that can handle division, modular operations, bitwise operations, machine finite precision integers, *et cetera*.

Clight semantics also uses partial maps and Coq lists to represent the local variable environments and arguments. We also use partial maps and Coq lists in the abstract states to abstract many of the concrete data structures in memory. To support those, the tactic contains theory solvers to discharge proof goals for properties related to partial maps and Coq lists. The tactic also contains a number of domain specific libraries which handle items such as device transitions and logs.

The automation library is easy to learn and use, and is exercised extensively by many students and researchers in our group to prove thousands of lines of C code in our verified OS kernel.

We have also developed an automation library to semi-automatically prove the modules directly

```
struct ConsoleBuffer {
 char buffer[CB_SIZE];
 unsigned int rpos;
 unsigned int wpos;
};

// in-memory circular buffer
struct ConsoleBuffer cons_buf;

// console buffer module M
void cons_buf_init() {
 cons_buf.rpos = 0;
 cons_buf.wpos = 0;
}

char cons_buf_read () {
 unsigned int rv = CB_EMPTY;
 if (cons_buf.rpos != cons_buf.wpos) {
  rv = cons_buf.buffer[cons_buf.rpos];
  cons_buf.rpos = (cons_buf.rpos + 1) %
    CB_SIZE;
 }
 return rv;
}

void cons_buf_write (char c) {
 cons_buf.buffer[cons_buf.wpos] = c;
 cons_buf.wpos = (cons_buf.wpos + 1) %
    CB_SIZE;
 if (cons_buf.rpos == cons_buf.wpos) {
  cons_buf.rpos = (cons_buf.rpos + 1) %
    CB_SIZE;
 }
}
```

Figure 2: Console circular buffer implementation

implemented in LAsm. The automation support for LAsm is not as mature and powerful as the support for Clight, as the assembly code is much less structured in nature compared to the Clight programs. In practice, it is not a big issue since the part of system code directly implemented in assembly is relatively small.

Example: Verification of Console Circular Buffer To better illustrate how the certified abstraction layers work, we demonstrate how we can utilize the techniques to verify a console circular buffer implementation used in our verified device drivers. As shown in Figure 2, in memory, the circular buffer is implemented as a circular array (to store the received input characters) with two additional fields to mark the head (`rpos`) and the tail (`wpos`) of the circular buffer. Since the console buffer module M in Figure 2 does not use any layer primitives, we can view M as running on a layer interface L_{low} with empty abstract state and primitives.

Now we define a new layer interface L_{high} with an abstract state d representing the circular buffer, and a list of abstract primitives cb_init, cb_read, and cb_write that operate on the abstract buffer. First, we define $d = (\mathsf{cons\_buf} : \mathsf{list}\ \mathbb{Z})$, i.e., we simply abstract the in-memory circular buffer as a simple logical list. Next, we give specifications to the set of primitives as shown in Figure 3. Here, we use the notations $[\cdot]$ and ++ to represent a singleton list and list concatenation, respectively. The specification shown in Figure 3 is much cleaner and simpler than the actual implementation; this simplifies future reasoning about code modules that use the data structure.

Next, we can define a refinement relation R to relate the concrete circular buffer in the memory

$$\frac{d' = d[\mathsf{cons\_buf} \quad \mathsf{nil}]}{\mathsf{cb\_init}(d) = d'} \qquad \text{(cb\_init)}$$

$$\frac{c :: tl = d.\mathsf{cons\_buf} \qquad d' = d[\mathsf{cons\_buf} \quad tl]}{\mathsf{cb\_read}(d) = (d', c)} \qquad \text{(cb\_read\_char)}$$

$$\frac{\mathsf{nil} = d.\mathsf{cons\_buf}}{\mathsf{cb\_read}(d) = (d, CB\_EMPTY)} \qquad \text{(cb\_read\_empty)}$$

$$\frac{l = d.\mathsf{cons\_buf} \qquad \mathsf{length}\ l < CB\_SIZE \qquad d' = d[\mathsf{cons\_buf} \quad l{+}{+}[c]]}{\mathsf{cb\_write}(d, c) = d'} \qquad \text{(cb\_write\_char)}$$

$$\frac{c :: tl = d.\mathsf{cons\_buf} \qquad \mathsf{length}\ l = CB\_SIZE \qquad d' = d[\mathsf{cons\_buf} \quad tl \mathbin{+}{+} [c]]}{\mathsf{cb\_write}(d, c) = d'} \qquad \text{(cb\_write\_overflow)}$$

Figure 3: Specifications of abstract console buffer primitives

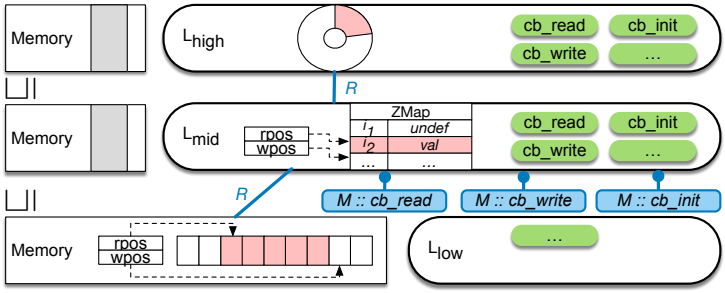

Figure 4: The layer hierarchy of circular console buffer

and the abstract list, then prove the contextual refinement between $[\![M]\!]L_{low}$ and L_{high} over the simulation relation R. We complete forward simulation proof, one primitive at a time. This proof is also complex, because R is nontrivial.

The complexity may further explode when this logical complexity gets mixed with the complexity of handling the accesses to the memory. CompCert's memory model is an axiomatized model where the properties are defined through a big list of axioms without a specific implementation. Any concrete implementation of this memory model needs to satisfy all the axioms. Thus, one cannot perform any simple evaluations on the memory, but needs to keep applying appropriate axioms to derive any desired properties. This severely limits the room for proof automation and significantly increases the proof size and memory consumption for proof compilation as the proof gets more complex. To separate the complexity that comes from the CompCert memory model from the actual complexity of the proof, in our layered approach, we always make the gap between

the underlay and overlay interface as small as possible when it comes to data abstraction, i.e., when a piece of memory gets abstracted into an abstract state at the overlay interface.

In the case of the circular console buffer, instead of directly jumping from the in-memory implementation to a logical list, we introduce an intermediate layer interface where the representation of the circular buffer in the abstract state is very similar to the one in the memory. We define the intermediate layer interface L_{mid} with the abstract state d and the primitive specifications as shown in Figure 5. Here, for any type T, ZMap.t T is the type of partial map from integer keys to the values of type T. One can easily observe that the representations shown in Figure 5 are extremely similar to the actual implementations shown in Figure 2. Given the similarity, one can easily come up with a refinement relation R which maps the concrete values in the memory to their appropriate logical values in d. The simulation proof over R is also relatively easy and there is no other complex factors interfering with the ones from handling the CompCert memory.

Once the contextual refinement between $[\![M]\!]L_{low}$ and L_{mid} is proven, the contextual refinement between L_{mid} and L_{high} can be proven with no code module involved. Thus, this part of proof is completely logical and the refinement relation R_{cons_buf} (shown in Figure 6) in this case only needs to relate two sets of abstract states. In Figure 6, Abs is the type of the abstract states in each layer interface. The overall layer hierarchy of entire console buffer is shown in Figure 4. This kind of two-stage proof strategy significantly reduces the complexity of the proof and lifts the main complex proof effort to pure logical level.

3.2 Resource-Aware CertiKOS

Under this seedling project, we extended each CertiKOS abstraction layer with a detailed resource specification. Figure 7 shows the device hierarchy of CertiKOS. Here the white boxes represent raw hardware devices; the green boxes denote the device drivers, and the gray boxes are the data structures used by the drivers. The purple/black lines show how these device and driver components are related. Note that the drivers in CertiKOS are not verified; they are implemented in about 1,600 lines of C and assembly code, and would be considered as part of the trusted computing base (if they are kept inside the kernel).

We take CertiKOS's lowest level machine model, *LAsm*, and extend it with device models. We model devices as finite state transition systems interacting with the processor and the external environments. Since devices run concurrently with the processor, parts of the device state change without the processor explicitly modifying them. Though these "volatile" device states can change nondeterministically, the processor itself only ever observes a "current" state when it reads the device data via an explicit I/O operation. The processor does not, and in fact *cannot*, care about any states that the device may enter between these observed states. Therefore, instead of designing fine-grained small-step transition systems that model all possible interleaved executions amongst the processor and devices, our devices simply perform an atomic big-step transition whenever they are observed, i.e., when there is a device read/write operation from the CPU.

Next, the machine model needs to be extended with the hardware interrupt model. The processor responds to an interrupt by temporarily suspending the current execution and then jumping to another routine (i.e., an interrupt handler). Interrupts can be triggered by both hardware and software. Software interrupts (e.g., exceptions, system calls) are relatively easy to reason about, since

$$d = (\quad \begin{array}{ll} \mathsf{cons\_buf\_concrete} : \mathsf{ZMap}.t\ \mathbb{Z}, & \triangleright \textit{ Concrete console buffer} \\ \mathsf{rpos} : \mathbb{Z}, & \triangleright \textit{ The head of the buffer} \\ \mathsf{wpos} : \mathbb{Z}). & \triangleright \textit{ The tail of the buffer} \end{array}$$

$$\frac{d' = d[\mathsf{rpos} \leftarrow 0][\mathsf{wpos} \leftarrow 0]}{\mathsf{cb\_init}(d) = d'} \quad \text{(cb\_init)}$$

$$\frac{\begin{array}{c} i = d.\mathsf{rpos} \qquad i \neq d.\mathsf{wpos} \\ c = d.\mathsf{cons\_buf\_concrete}[i] \qquad d' = d[\mathsf{rpos} \leftarrow (i+1) \bmod CB\_SIZE] \end{array}}{\mathsf{cb\_read}(d) = (d', c)} \quad \text{(cb\_read\_char)}$$

$$\frac{d.\mathsf{wpos} = d.\mathsf{rpos}}{\mathsf{cb\_read}(d) = (d, CB\_EMPTY)} \quad \text{(cb\_read\_empty)}$$

$$\frac{\begin{array}{c} i = d.\mathsf{wpos} \qquad i' = (i+1) \bmod CB\_SIZE \qquad d.\mathsf{rpos} \neq i' \\ d' = d[\mathsf{cons\_buf\_concrete}[i \mapsto c]] \qquad d'' = d'[\mathsf{wpos} \leftarrow i'] \end{array}}{\mathsf{cb\_write}(d, c) = d'} \quad \text{(cb\_write\_char)}$$

$$\frac{\begin{array}{c} i = d.\mathsf{wpos} \qquad i' = (i+1) \bmod CB\_SIZE \qquad d.\mathsf{rpos} = i' \\ i'' = (i'+1) \bmod CB\_SIZE \qquad d' = d[\mathsf{cons\_buf\_concrete}[i \mapsto c]] \\ d'' = d'[\mathsf{wpos} \leftarrow i'][\mathsf{rpos} \leftarrow i''] \end{array}}{\mathsf{cb\_write}(d, c) = d''} \quad \text{(cb\_write\_overflow)}$$

Figure 5: Intermediate specifications of console buffer primitives

```
Fixpoint match_cons_buf (cons_buf: list Z) (cons_buf_concrete: ZMap.t Z) (rpos wpos
    : Z) : Prop :=
  match cons_buf with
   | nil => rpos = wpos
   | bv :: cons_buf' =>
       ZMap.get rpos cons_buf_concrete = bv /\
       match_cons_buf cons_buf' cons_buf_concrete ((rpos + 1) mod CB_SIZE) wpos
  end.

Inductive R_cons_buf: L_high.Abs -> L_mid.Abs -> Prop :=
| MATCH_CONS_BUF:
    forall d_high d_mid,
      match_cons_buf d_high.cons_buf d_mid.cons_buf_concrete d_mid.rpos d_mid.wpos
    ->
      R_cons_buf d_high d_mid.
```

Figure 6: The definition of refinement relation between L_{high} and L_{mid} in Coq

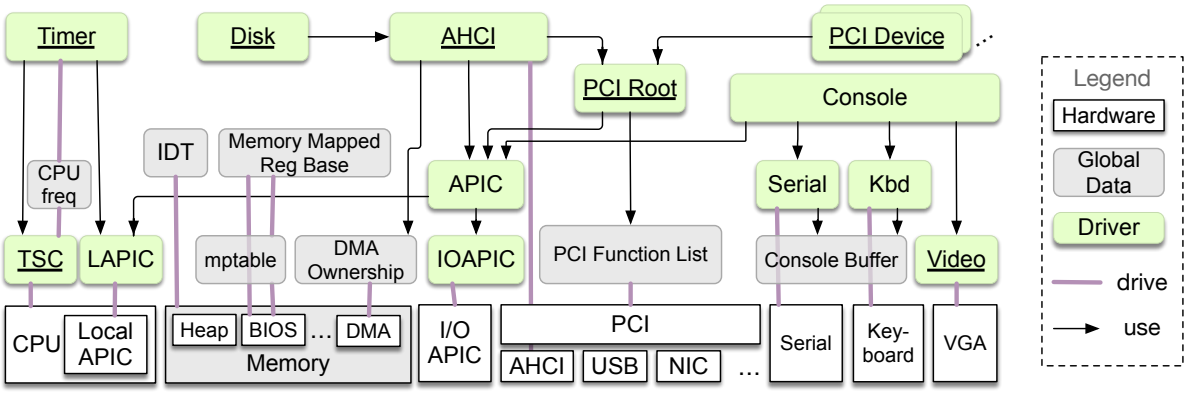

Figure 7: The device driver hierarchy of CertiKOS

their behaviors are always deterministic. For example, a page fault exception occurs whenever the accessed address belongs to an unmapped page or a page with wrong permission, and a system call is triggered by an explicit instruction. However, hardware interrupts (IRQs) are unpredictable; when we execute some code with interrupts turned on, at every fine-grained processor step, the machine state (e.g., registers and memory) may undergo significant changes. Recent work on verified operating systems (including CertiKOS) neglects this kind of reasoning, ignoring one of the largest kernel threat-surfaces [Gu et al., 2015, Klein et al., 2009, Alkassar et al., 2010]. Finally, modeling interrupts is important because it also opens the way toward enabling interrupts within the kernel.

On top of this lowest-level machine model, each kernel module can be related to either device drivers (denoted as *DD*) or the rest of the kernel (denoted as *K*, representing non-device-related kernel components). To introduce, verify, and abstract each such kernel module into an abstract object with atomic logical primitive transitions, we need to prove the following isolation properties:

- For each function in *K* or user space, which has interrupts turned on, the interrupt must not affect the behavior of the function. Although the code can be interrupted at any moment, and the control flow transferred to a place outside the function, it will eventually return with states (which the function relies upon) unchanged.
- Devices which directly change the memory through Direct Memory Access (DMA), do not change any memory that the execution of any function in *K* depends on.
- For each interruptible device driver function in *DD*, any interrupt not related to the current device must not change any state related to the current device.
- In case that all interrupts related to a device are masked out, no interrupts can affect the state of the interrupt handler for the device.

For a particular fixed set of functions, the proof of the above properties may not seem hard. However, they have to be proven repeatedly for all possible combinations of currently introduced

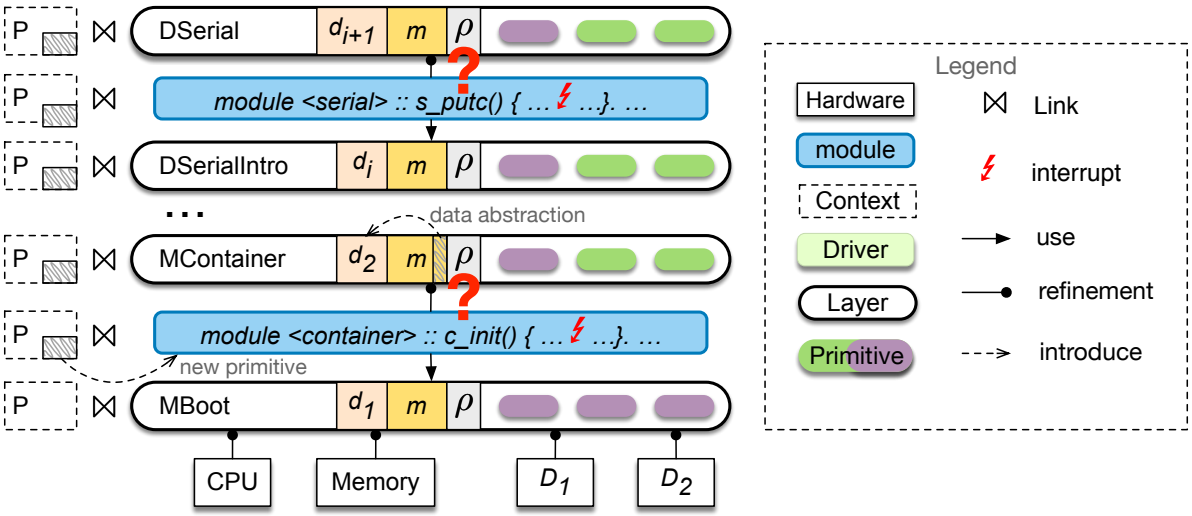

Figure 8: Abstraction layers w. interrupts: a failed attempt

sets of functions and devices. This immediately makes the verification of an interruptible operating system with device drivers unscalable.

Furthermore, it is not obvious how to apply techniques presented in Section 3.1 to handle hardware interrupts. Figure 8 shows one such attempt. Here, P denotes the kernel/user-level context code; MBoot, MContainer, DSerialIntro, and DSerial denote several kernel and driver layers. With interrupts turned on in the kernel, it is immediately unclear how to show contextual refinement among different layers. For a kernel function like c_init, it cannot be easily refined into an atomic specification as the code can be interrupted at any point during the execution by a device interrupt, unless all possible interleaving of interrupts are encoded into the specification itself. Similarly, for a device driver function like puts, the code can be interrupted at any moment by interrupts triggered from other devices or the device itself.

Under this seedling project, we developed a systematic way that strictly enforces isolation among different entities by construction. Our approach consists of the following two key ideas.

First, rather than viewing drivers as separate modules that interact with the CPU via in-memory shared-state, we instead view each driver as an extended device. We utilize abstraction layers and contextual refinement to gradually abstract the memory shared between a device and its driver into the internal abstract states of a more general device. Furthermore, we use the same technique to abstract those driver functions that manipulate these data into the abstract primitives of a higher level device. After this, our approach ensures that those abstract states can no longer be accessed by the other entities, through, e.g., memory reads and writes, but, rather, can only be manipulated via explicit calls to the device interface. We repeat these procedures so we can incrementally refine a raw device into more and more abstract devices by wrapping them with the relevant device drivers (see Figure 9). We call this extended abstract device a *device object*, to distinguish it from the raw hardware device. Note that in our model, device objects are indeed treated similarly to raw devices, and both have quite similar interfaces.

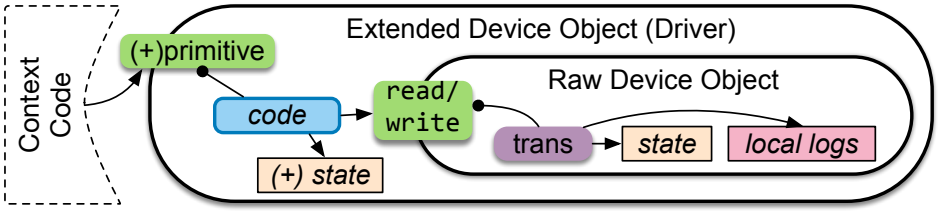

Figure 9: The driver as an extended device

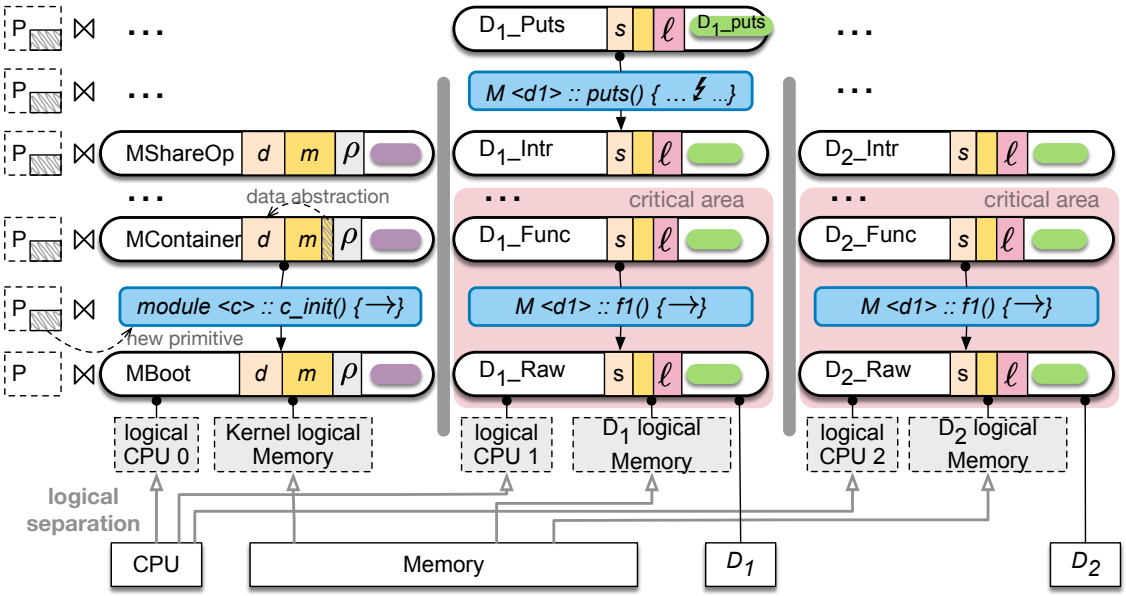

Figure 10: Building certified abstraction layers with hardware interrupts: our new approach

Second, we introduce and verify the interrupt handler for each device at the lowest machine model, which is not yet suitable for reasoning about interruptible code. This is possible because, for each device, we require that either the interrupt be disabled or its corresponding interrupt line be masked inside the interrupt handler of the device. Next, we introduce a new abstract machine with a more abstract interrupt model, that provides strong isolation properties amongst different device objects and the kernel, in which any future (context) code with interrupts turned on can be reasoned about naturally. We prove a strong contextual refinement property between these two abstract machines: any context code running on the machine with the abstract interrupt model (overlay) retains an equivalent behavior when it is running on top of the machine with the concrete hardware interrupt model (underlay).

Figure 10 shows the layer hierarchy of our interruptible kernel with device drivers. We treat the driver code as if it runs on its own device's "logical CPU," and each logical CPU operates on its own separate internal states. Thus, the approach provides a systematic way of assuring isolation

among different device objects (running on its own local logical CPUs) and the rest of the kernel.

On the kernel side (the layer hierarchy on the left hand side of Figure 10), the contextual refinement is achieved in the same way as shown in Section 3.1 since the hardware interrupts (from the other logical CPUs with separate states) no longer affect the execution of any kernel primitive (like c_init), i.e., the kernel is completely interrupt-unaware.

Similarly, the device driver functions are no longer affected by the hardware interrupts triggered from other devices. For each device D running on top of its own logical CPU, we first introduce and verify part of the driver in the *critical area*, i.e., the low-level device functions that should not be interrupted by the same device, and the interrupt handler of the device. Next, we use contextual refinement to introduce a new layer that has a more abstract interrupt model. On this layer, we can introduce and verify interruptible driver code (e.g., puts) while still enforcing strong isolation and providing clean interface to the kernel.

3.3 Certified Concurrent Abstraction Layers

We have successfully extended our methods and tools for certified abstraction layers to support concurrency. We parameterize each layer interface L with an "active" thread set A (where $A \subseteq D$ and D is the domain of all thread and CPU IDs). The layer machine based on a concurrent layer interface $L[A]$ specifies the execution of threads in A (with threads outside A considered as the *environment*).

A concurrent layer interface extends its sequential counterpart with a set of abstract *atomic* objects, a global log l, and a set of *valid* environment contexts. Unlike calls to thread-local objects which are not *observable* by other threads, each method call to an atomic object (together with its arguments) is recorded as an observable event appended to the end of the global log. Each environment context specifies the observable behaviors of the execution of those threads/CPUs in the environment under one possible interleaving.

To define the semantics of a concurrent program P in a *generic* way, we developed a novel compositional (operational) model based upon ideas from game semantics. Each run of P over $L[D]$ can be viewed as playing a game involving members of D (plus a scheduler): each participant $t \in D$ contributes its play by appending an event into the global log l; its *strategy* φ_t is a deterministic partial function from the current log l to its next move $\varphi_t(l)$ whenever the last event in l transfers control back to t.

The semantics of the (concurrent) layer machine based on an interface $L[A]$ can be defined over its set of valid environment contexts. Each environment context $\mathcal{E}$ provides a strategy for its "environment" (i.e., the union of the stategies by the scheduler plus those participants *not* in A).

For example, Figure 11 shows a system with two threads (t_1 and t_2) and a scheduler. On the left, it shows one execution of method *foo* over the layer machine $L'[t_1]$ under a specific environment context $\mathcal{E}'_1$. Here, $\mathcal{E}'_1$ is the union of the strategy φ'_0 for the scheduler and φ'_2 for thread t_2. In the middle, it shows the execution of *foo* (invoked by t_2) over $L'[t_2]$ under the environment context $\mathcal{E}'_2$. On the right, it shows the interleaved execution of two invocations to *foo* over $L'[\{t_1, t_2\}]$ where the environment context $\mathcal{E}'$ is just the scheduler strategy φ'_0.

Given an environment context $\mathcal{E}$ which also contains a specific scheduler strategy, the execution of P over $L[A]$ should be deterministic; the concurrent machine will run P when the control is

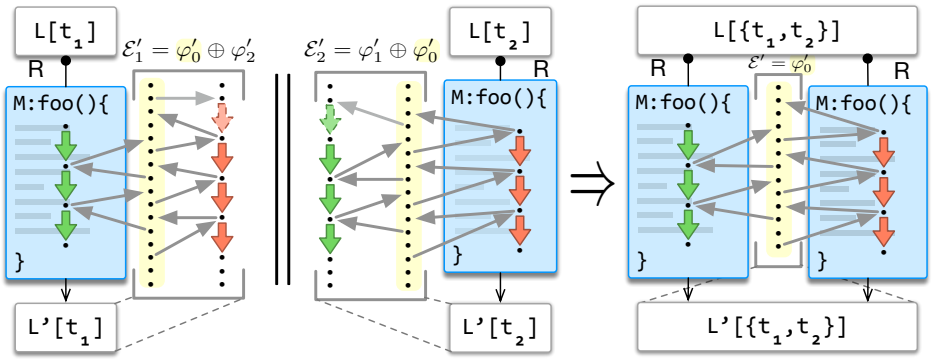

Figure 11: Environment contexts and parallel layer composition

transferred to any member of A, but will ask $\mathcal{E}$ for the next move when the control is transferred to the environment.

The semantics $[\![P]\!]_{L[A]}$ is then just the set of global logs generated from running P over $L[A]$ under all *valid* environment contexts. The definition of validity here can be customized by each layer interface. It corresponds to a generalized version of the "rely" (or "assume") condition in rely-guarantee-based reasoning [Feng et al., 2007, Vafeiadis and Parkinson, 2007, Feng, 2009, Fu et al., 2010, Sergey et al., 2015]. Each layer interface can also provide its own "guarantee" condition which is simply expressed as invariants over its atomic objects and global log.

Using these ideas, we developed a fully mechanized programming toolkit (called CCAL) for specifying, composing, compiling, and linking certified concurrent abstraction layers. CCAL consists of three technical novelties: a new game-theoretical, strategy-based compositional semantic model for concurrency (and its associated program verifiers), a set of formal linking theorems for composing multithreaded and multicore concurrent layers, and a new CompCertX compiler that supports certified thread-safe compilation and linking. The CCAL toolkit is implemented in Coq and supports layered concurrent programming in both C and assembly. It has been successfully applied to build our fully certified concurrent OS kernel, CertiKOS, with fine-grained locking.

3.4 Real-Time CertiKOS

Preemptive Priority-Based Scheduling A real-time system associates a deadline to each real-time task (and a period in the case of periodic ones) and requires that its execution finishes by the deadline. We use *task* or *process* interchangeably to denote the basic scheduling unit of a real-time system. From the OS kernel's point of view, a task corresponds to a process. Scheduling is done with a basic time unit called *time slot* or *time quantum*, corresponding to the interval between two timer interrupts.

In our seedling research, we consider real-time requirements for periodic tasks and assume that the deadline is the start of the next period (called *implicit deadlines*). We also do not allow the creation of new real-time tasks at runtime. In other words, the number of real-time tasks and their parameters are fixed before the system starts. This is the most common case in control-loop systems, which represents the majority of real-time systems.

In our setting, each real-time task is defined by a period, a budget of execution time within each period, and a priority level. Higher priority levels correspond to smaller numbers, in particular, priority level 0 is the highest one. When there are multiple pending tasks, the one with the highest priority will be scheduled. Each priority level contains at most one task, so that we can identify tasks with their priority level. In particular, we abbreviate "task at priority level p" into "task p".

The OS kernel is responsible for cutting off the execution of a task when it has used up its budget in its current period. A real-time task meeting its deadline means that it will be allocated the exact number of execution time slots it has declared by the end of its period. If the budget it has initially is high enough, it will meet its deadline; otherwise it will be prevented from overrunning, thus not jeopardizing other real-time tasks. Finally, a system is schedulable if all tasks are given their exact budget during each period. While no real-time task is pending, we allow batch tasks to run without any sort of real-time guarantee.

To do so, we extend the machine model to support reasoning about interrupts, in particular timer interrupts that can preempt the current user process and schedule a new one. This is a substantial change compared to the verification of a cooperative kernel, where a user process cannot be interrupted and has to explicitly invoke a system call to suspend itself: in a preemptive setting, a user process can be preempted after the execution of any assembly instruction.

Schedulability Analysis Schedulability analysis tests statically whether a group of tasks can be scheduled on a given system without missing any deadline (or equivalently, are given the right amount of execution time per period). A sound schedulabiltiy analysis usually performs checking on certain task parameters, such as priorities, periods and budgets, and concludes that the task set is schedulable only when each task can be scheduled for its full budget in all periods.

Reusing an existing schedulability analysis, we focus on connecting the analysis (which is usually done within an abstract model) with the concrete code that is running on the machine. In particular, we interpret schedulabiltiy as a safety property: whenever the kernel decides to refill the budget of a task, it must have been depleted, meaning that the task was scheduled during this past period for its budget worth. We insert such an assertion in the specification of the OS kernel, and prove that it never fails if task parameters pass the schedulability test. In this way, the schedulability analysis is faithfully connected to the state transition of the verified OS kernel, and is carried down all the way to generated assembly code.

Yet, proving schedulability alone is not the end of the verification. We need a way to encapsulate schedulabiltiy into the specification, so that it can in turn be used to deduce other temporal properties, such as liveness for all real-time processes. Furthermore, we want to minimize the proof effort by doing it in a compositional way, despite the fact that scheduling itself is a whole system functionality.

To achieve the above reasoning, we reuse the time supply function to isolate different tasks from each other. Supply functions, as presented in [Liu, 2000, Lehoczky et al., 1989, Audsley et al., 1993], compute the response time of a task (the duration from the start of a period to the end of execution in the current period) by finding the intersection of its time demand and time supply. Time demand counts the execution time of this particular task and all higher priority ones, which may span multiple periods. Time supply represents the total available time that can be allocated to

them, and is usually expressed as the identity map for fixed-priority scheduling (except for instance in hierarchical scheduling).

We reuse and adapt this idea, which allows us to easily deduce liveness for all real-time tasks. Their adoption also helps in the proof of schedulability analysis, which suggests that it is indeed a suitable abstraction of fixed-priority scheduling, and is most suitable to serve as the connection between schedulability analysis and the verified OS kernel.

Information-Flow Non-Interference Information-flow non-interference is a crucial property for mixed-criticality systems, ensuring that non-critical tasks cannot affect critical ones. It is also very useful in shared systems, avoiding eavesdropping between users and preventing certain types of denial of service.

In practice, system calls may reveal information about the internal state of the machine, such as the amount of allocated memory through malloc or the number of processes created through process id (pid) assignment. To solve these problems, we can use, for instance, memory quotas and random or fixed pid assignment, respectively. These techniques have been used to prove that the single-core CertiKOS kernel is indeed information-flow secure [Costanzo et al., 2016]. However, this previous work assumes a cooperative OS.

Preemption requires special treatment. In the previous work, non-interference is guaranteed by assuming that two executions of the same process are always synchronized. They execute the same sequence of instructions, and yield to another process at the same instant w.r.t. the number of instructions already executed (not the physical time). This is no longer true with a preemptive scheduler where yielding might happen non-deterministically.

To demonstrate the expressiveness of our reasoning framework, we extend the non-interference guarantee to the presence of a `get_usage` system call for resource accounting. The specific type of resource is irrelevant, we only assume a cost model for each instruction, which must be deterministic in order to preserve non-interference. Valid examples of resources include time, I/O, money, or fuel (ethereum for instance). We implement two examples: one with time, the other with the instruction count.

The time version works in a similar way as the `clock_gettime` system call in Linux with the option `CLOCK_THREAD_CPUTIME_ID` enabled. The caveat is that the determinism assumption on the cost model is not very realistic for time, particularly so for the x86 platform. It makes more sense on ARM platforms that exhibit a more deterministic timing behavior, such as the ARM Cortex R or M series.

We implement the instruction count version of `get_usage` on x86 using the `INST_RETIRED` performance monitoring counter. It counts the number of retired instructions (instructions that have completed and whose effects are valid in contrast to incorrect speculative guesses) executed in rings 1 through 3. In this case, the cost model is the constant function 1.

Evaluation We developed a fully verified preemptive OS kernel with both spatial and temporal isolation. The extracted assembly code is further compiled by gcc, and runs on an Intel Sandy Bridge based x86 machine, with timer interrupts configured to arrive every 1 ms. Although our focus is not currently on experimental evaluation, our experiments report that the `get_usage` sys-

Feature	LoC	Files Touched
Interrupts/Preemption	10,533	161
Real Time Scheduler	26,142	22
Schedulability Analysis	1,398	4
`get_usage` System Call	4,401	28
Non-Interference	28,458	28
Total	**70,932**	**243**

Figure 12: Changes with respect to the non-interference version of CertiKOS.

tem call is indeed non-interfering: invocations from the observer process always return the same sequence of values regardless of how we change other processes in the system.

We developed the following Coq proofs:

- Extension of CertiKOS with a model of a cost register and a generic interrupt mechanism. This involves changing the machine model to parametrize it over operations that increase the resource usage, trigger and handle interrupts, as well as instantiations of these operations at each layer.
- Formalization of the schedulability analysis.
- Functional correctness of the new priority based scheduler, and the successive refinement between the concrete implementation, the virtual time based scheduler, and the dedicated scheduler for any particular priority level.
- Non-interference of user processes. We extend the framework of previous work by Costanzo et al. [2016] to a preemptive scheduler and fix an existing bug in the kernel.
- Functional correctness proof of the `get_usage` system call, while preserving non-interference.

The summary of the extent of these changes is given in Figure 12. The lines represent the modification as given by the `git diff` tool. For the sake of simplicity, we attribute each file only to one category, even though in some cases the modifications clearly belong to several ones. We think these figures still give a good estimate of the amount of work involved.

From this table, one can notice several things. First, although the introduction of interrupts only account for about 15% of the changes, it represents 2/3 of the modified files. Indeed, the modification of the machine model for the x86 assembly entails small changes at most layers of CertiKOS. Switching from a cooperative round-robin scheduler to a preemptive priority-base one changes some invariants of the kernel and introduces 9 new abstraction layers, hence the high number of line count although there is a relatively small number of files. Similarly, the high-level non-interference proof had to be redesigned completely because of preemption, thus the high number of changes. Taking advantage of the framework developed for the real-time scheduler, the work required for the `get_usage` system call is much smaller but spans a fair amount of files as its existence must be propagated through all upper layers. Finally, the schedulability analysis part is

completely self-contained and covers only the formalization of the analysis of the rate monotonic scheduling policy, the connection with CertiKOS being accounted for under the scheduler.

Subtleties Found in the Kernel During the proof development, we found certain subtleties in the implementation of the kernel, which either warns us of limitations of our semantics models, or helps us prevent bugs that could invalidate the whole system.

The kernel uses a timer counter to keep track of the number of timer interrupts that have occurred, providing a time measure for the scheduler. However, since the system can run indefinitely, a fixed-sized counter (which we call a bounded counter) will eventually overflow, leading to an inconsistent state and invalidating the schedulability guarantee.

We handle this problem by observing that the scheduling result is identical after each hyper-period, that is, the least common multiple of periods of all processes. Indeed, assuming the system is schedulable, after each hyper-period, the system can be seen as starting from scratch since every process has fulfilled its current budget and all periods are about to end.

We then introduce an unbounded counter in the system, which resides only in the abstract state. The unbounded counter keeps increasing, while the bounded counter will be reset every time it hits the hyper-period. We prove that as long as those two counters are equal modulo the hyper-period, the result of the scheduler is the same.

Finally, since we do not allow dynamic allocation of real-time tasks, the value of the hyper-period can be computed statically. Therefore, given the value of the hyper-period, we can always find a bounded counter with the proper size to store it and thus, the kernel is functionally correct. Since the reasoning of temporal properties only relies on the unbounded version, it is not affected by such low level details.

Another bug we found is related to the context switch primitive. The context switch primitive can satisfy non-interference in a cooperative setting without being correct at all. And this is actually exactly what happened: the existing context switch primitive of CertiKOS is incorrect because it does not save enough registers although it does satisfy non-interference in a cooperative setting. The missing registers are the CR register (where the test flags are set for instance) and the floating point ones.

On the one hand, not saving floating point registers by default is a common optimization in OS as most programs do not use them. When the first floating point instruction is executed, it traps into the kernel to enable the floating point operations for this process and also set the saving/restoring of floating point registers on context switches. Nevertheless, this optimization is not currently implemented in CertiKOS so their context switch routine was incorrect.

On the other hand, being able to observe (part of) the CR register is required as conditional jumps access it so it is already indirectly observable.

The fact that non-interference with preemption requires to prove that the saved and restored contexts are identical prevents this kind of mistakes.

Limitations Our current implementation only allows for preemption at the user level. This means that interrupts can be disabled for a long time if system calls are expensive, which can be detrimental to the responsiveness of the system.

A solution is to adopt the deferred post method used in μC/OS-III, where the interrupt handler is only responsible to record the event in a request queue, and the actual handling is performed by a user process which iterates over the queue. The benefit of this design is to minimize the interrupt disable time, thus achiving better responsiveness while still only requiring preemption in user mode, so that the current functional correctness proof would still be applicable.

In our current machine model, the resource consumed by kernel primitives is not taken into account. This allows us to avoid modifying the existing correctness proofs but is clearly unrealistic, in particular when the kernel executes a costly operation on behalf of a user process. The problem is then that our resource accounting will drift each time a system call happens, so that it does not track accurately what each user process is (even indirectly) using.

The solution is quite simple: bite the bullet and add a resource component to the specifications of each system call. Then we would remove the kernel-mode-only assembly semantic rule, thus enabling the resource accounting everywhere.

A lighter solution would be to simply determine a correct upper bound on the cost of each system call and assume its correctness. We would then use this bound to extend the cost model to system calls at runtime, thus giving us a more accurate resource accounting for user processes at the cost of using empirical values instead of certified ones for system calls.

Even more than a proper tracking of resources in kernel mode, not accurately tracking the time spent by the kernel can severely undermine our schedulability guarantees, although ignoring kernel time (especially for context switch and interrupt handlers) is a fairly common assumption for schedulability analysis in first approximation.

Indeed, the preemption of a running user process relies on the timer interrupt, which is triggered at a constant rate. However, if the interrupt happens in the middle of a system call, its handling will be delayed until control is given back to user mode. This means that the next process is given slightly less than one full time quantum to execute. Similarly, the interrupt handler itself also consumes time intended for running user processes. This small overhead may in turn jeopardize the proper operation of a task if it relies on executing for the full budget.

To account for this overhead, we could reserve a little bit of each time quantum for interrupts and system calls and ensure that the kernel cannot use more than this share.

4 Results and Discussion

We have achieved three kinds of results:

1. Scientific/engineering methods/techniques for security and correctness verification of resource-aware system software;
2. Tools that embody these techniques so they can be applied in the field; and
3. Demonstrations of these methods and tools in specific verifications of interest to the BRASS community.

Specific examples have been discussed in Section 3 and its subsections (pages 6–23). We summarize here:

1. Methods/techniques
   - Compositional specification and verification with certified abstraction layers
   - Game-semantics-based strategies and environment contexts for building and composing certified concurrent abstraction layers
   - Proof by refinement: certified concurrent abstraction layers with functional-programs-as-functional specs
   - Observation function and security-preserving simulation for specifying, propagating, and reasoning about end-to-end information-flow security properties such as spatial and temporal isolation.
2. Tools
   - CCAL, for modular refinement-based proofs of concurrent C and assembly programs
3. Verifications
   - Resource-Aware CertiKOS with verified device drivers and interrupt handlers
   - Concurrent CertiKOS hypervisor and OS kernel, verified using CCAL.
   - Real-Time CertiKOS with spatial and temporal isolation.

The success of these verification efforts demonstrates that it is now possible to build high-assurance resource-aware OS kernels.

4.1 Publications

The following publications resulted from our BRASS-supported research.

[Gu et al., 2018] *Certified Concurrent Abstraction Layers*

Concurrent abstraction layers are ubiquitous in modern computer systems because of the pervasiveness of multithreaded programming and multicore hardware. Abstraction layers are used to hide the implementation details (e.g., fine-grained synchronization) and reduce the complex dependencies among components at different levels of abstraction. Despite their obvious importance, concurrent abstraction layers have not been treated formally. This severely limits the applicability of layer-based techniques and makes it difficult to scale verification across multiple concurrent layers. In this paper, we present CCAL—a fully mechanized programming toolkit developed under the CertiKOS project—for specifying, composing, compiling, and linking certified concurrent abstraction layers. CCAL consists of three technical novelties: a new game-theoretical, strategy-based compositional semantic model for concurrency (and its associated program verifiers), a set of formal linking theorems for composing multithreaded and multicore concurrent layers, and a new CompCertX compiler that supports certified thread-safe compilation and linking. The CCAL toolkit is implemented in Coq and supports layered concurrent programming in both C and assembly. It has been successfully applied to build a fully certified concurrent OS kernel with fine-grained locking.

[Liu et al., 2018] *RT-CertiKOS: A Fully Verified Preemptive OS Kernel with Temporal and Spatial Isolation*

Mathematical proofs carry much stronger guarantees of correctness than testing. In the setting of critical systems, this is of paramount importance because the consequences of failure are dire (loss of lives or failure of the mission). From this perspective, the complete formal verification of a real-time operating system is an attractive solution. Nevertheless, it remains a challenge to this day. For instance, preemption alone requires careful modeling of the interrupt behavior. Furthermore, schedulability is a whole system property, making it difficult to compose and scale program verification on top of such real-time kernels. And indeed, even if these challenges have been separately addressed in the past, they have never been verified in tandem. We present here the, to our knowledge, first fully-verified preemptive kernel with formal proofs of both temporal and spatial isolation. Furthermore, our verification is not limited to the C code but carries down to the assembly code generated by the compiler, further strengthening the provided guarantees. More precisely, we build upon the fully-verified single-core non-preemptive kernel CertiKOS, and we extend it with a generic exception mechanism, user-level preemption, and real-time scheduling. We then formally prove in the Coq proof assistant that, on top of the existing functional correctness proof, this new kernel also enjoys spatial and temporal partitioning. For temporal partitioning, we use a variant of the supply bound function that we call a virtual timeline to connect the scheduling with concrete states of the system, and to isolate the reasoning of a process from interference by others. For spatial partitioning, we extend the existing non-interference proof of CertiKOS to handle the preemptive setting. As an illustration of the benefits of this

framework, we also introduce a secure implementation of a getusage system call for resource accounting.

[Gu et al., 2016] *CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels*

Complete formal verification of a non-trivial concurrent OS kernel is widely considered a grand challenge. We present a novel compositional approach for building certified concurrent OS kernels. Concurrency allows interleaved execution of kernel/user modules across different layers of abstraction. Each such layer can have a different set of observable events. We insist on formally specifying these layers and their observable events, and then verifying each kernel module at its proper abstraction level. To support certified linking with other CPUs or threads, we prove a strong contextual refinement property for every kernel function, which states that the implementation of each such function will behave like its specification under any kernel/user context with any valid interleaving. We have successfully developed a practical concurrent OS kernel and verified its (contextual) functional correctness in Coq. Our certified kernel is written in 6500 lines of C and x86 assembly and runs on stock x86 multicore machines. To our knowledge, this is the first proof of functional correctness of a complete, general-purpose concurrent OS kernel with fine-grained locking.

[Carbonneaux et al., 2017] *Automated Resource Analysis with Coq Proof Objects*

This paper addresses the problem of automatically performing resource-bound analysis, which can help programmers understand the performance characteristics of their programs. We introduce a method for resource-bound inference that (i) is compositional, (ii) produces machine-checkable certificates of the resource bounds obtained, and (iii) features a sound mechanism for user interaction if the inference fails. The technique handles recursive procedures and has the ability to exploit any known program invariants. An experimental evaluation with an implementation in the tool Pastis shows that the new analysis is competitive with state-of-the-art resource-bound tools while also creating Coq certificates.

[Kim et al., 2017] *Safety and Liveness of MCS Lock—Layer by Layer*

The Mellor-Crummey and Scott (MCS) Lock, a small but complex piece of low-level software, is a standard algorithm for providing inter-CPU locks with First In First Out (FIFO) ordering guarantee and scalability. It is an interesting target for verification—short and subtle, involving both liveness and safety properties. We implemented and verified the MCS Lock algorithm as part of the CertiKOS kernel, showing that the C/assembly implementation contextually refines atomic specifications of the acquire and release lock methods. Our development follows the methodology of certified concurrent abstraction layers. By splitting the proof into layers, we can modularize it into separate parts for the low-level machine model, data abstraction, and reasoning about concurrent interleavings. This separation of concerns makes the layered methodology suitable for verified programming in the large, and our MCS Lock can be composed with other shared objects in CertiKOS kernel.

[Chen et al., 2018] *Toward compositional verification of interruptible OS Kernels and device drivers*

An OS kernel forms the lowest level of any system software stack. The correctness of the OS kernel is the basis for the correctness of the entire system. Recent efforts have demonstrated the feasibility of building formally verified general-purpose kernels, but it is unclear how to extend their work to verify the functional correctness of device drivers, due to the non-local effects of interrupts. In this paper, we present a novel compositional framework for building certified interruptible OS kernels with device drivers. We provide a general device model that can be instantiated with various hardware devices, and a realistic formal model of interrupts, which can be used to reason about interruptible code. We have realized this framework in the Coq proof assistant. To demonstrate the effectiveness of our new approach, we have successfully extended an existing verified non-interruptible kernel with our framework and turned it into an interruptible kernel with verified device drivers. To the best of our knowledge, this is the first verified interruptible operating system with device drivers.

5 Conclusion

In our BRASS seedling project we set out to tackle two major technical challenges for building long-lived resource adaptive system software:

- *Formal resource and environment model.* A formal resource and environment model is a prerequisite for reasoning about the resource usage of a program and its adaptation to changes in its ecosystem. Unfortunately, due to the low-level nature of hardware resources, such a model often does not exist, and even if it does, it is informal and too low level; and it has a huge gap with the high-level notions of resources in today's programming languages.

- *Compositional specification for communicating threads and hardware devices.* Modern system software often relies on a collection of communicating threads so it can best adapt to the available CPU resources on multicore machines. It is unclear how to specify the behaviors and resource usage of these threads in a modular way using existing technology.

We demonstrated methods and verification-engineering tools that accomplished a variety of application verifications useful to the BRASS vision. We have obtained the following important results.

We extended each CertiKOS abstraction layer with a detailed resource specification. We developed a novel compositional framework for reasoning about the end-to-end functional correctness of device drivers in a certified interruptible kernel. Our formalization of interrupts follows the abstraction-layer-based approach and includes a realistic hardware interrupt model and an abstract model of interrupts (which is suitable for reasoning about interruptible code). We have proven that the two interrupt models are contextually equivalent. We have successfully extended an existing verified non-interruptible kernel with our framework and turned it into an interruptible kernel with verified device drivers. The implementation, specification, and proofs are all performed in a unified framework (realized in the Coq proof assistant), yet the mechanized proofs verify the correctness of the assembly code that can run on the actual hardware.

We have also developed CCAL—a novel programming toolkit developed under the CertiKOS project for building certified concurrent abstraction layers. We have developed a new compositional model for concurrency, program verifiers for concurrent C and assembly, certified linking tools, and a thread-safe verified C compiler.

Finally, we developed a fully verified preemptive OS kernel with temporal and spatial isolation. Our extended machine model features an interrupt mechanism as well as a generic cost model for assembly instructions. We extended the existing work with a real-time scheduler (fixed-priority scheduling), proved functional correctness of the system, and also connected a schedulability analysis with the system. In doing so, we introduce a notion of virtual time to reason about each task in isolation while encapsulating interference from other processes in the virtual time map. We prove that all services provided by our kernel preserve the integrity and confidentiality of user processes, by showing that non-interference holds for this kernel.

References

Eyad Alkassar, Mark A. Hillebrand, Wolfgang J. Paul, and Elena Petrova. Automated verification of a small hypervisor. In *Proc. 3rd International Conference on Verified Software: Theories, Tools, Experiments*, pages 40–54, 2010.

Thomas Anderson and Michael Dahlin. *Operating Systems Principles and Practice*. Recursive Books, 2011.

June Andronick, Corey Lewis, Daniel Matichuk, Carroll Morgan, and Christine Rizkallah. *Proof of OS Scheduling Behavior in the Presence of Interrupt-Induced Concurrency*, pages 52–68. Springer International Publishing, Cham, 2016. ISBN 978-3-319-43144-4. doi: 10.1007/978-3-319-43144-4_4. URL `https://doi.org/10.1007/978-3-319-43144-4_4`.

N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, Sept 1993. ISSN 0268-6961. doi: 10.1049/sej.1993.0034.

Bruno Barras et al. The Coq Proof Assistant reference manual. Technical report, INRIA, 1998.

Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *J. Automated Reasoning*, 43(3):263–288, 2009.

Quentin Carbonneaux, Jan Hoffmann, Tom Reps, and Zhong Shao. Automated resource analysis with coq proof objects. In *Proc. 29th International Conference on Computer-Aided Verification (CAV), Part II*, pages 64–85, 2017.

F. Cerqueira, F. Stutz, and B. B. Brandenburg. Prosa: A case for readable mechanized schedulability analysis. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 273–284, July 2016. doi: 10.1109/ECRTS.2016.28.

Hao Chen, Xiongnan Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible os kernels and device drivers. *Journal of Automated Reasoning*, 61:141–189, 2018.

Stephen Chong, Joshua Guttman, Anupam Datta, Andrew Myers, Benjamin Pierce, Patrick Schaumont, Tim Sherwood, and Nickolai Zeldovich. Report on the NSF workshop on formal methods for security. `people.csail.mit.edu/nickolai/papers/chong-nsf-sfm.pdf`, 2016.

David Costanzo, Zhong Shao, and Ronghui Gu. End-to-end verification of information-flow security for c and assembly programs. In *PLDI '16: 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 648–664, 2016.

B. Dutertre. Formal analysis of the priority ceiling protocol. In *Proceedings 21st IEEE Real-Time Systems Symposium*, pages 151–160, 2000. doi: 10.1109/REAL.2000.896005.

Xinyu Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327, 2009.

Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume guarantee reasoning. In *Proc. 2007 European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 173–188. Springer-Verlag, April 2007.

Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. Reasoning about optimistic concurrency using a program logic for history. In *Proc. 21st International Conference on Concurrency Theory (CONCUR'10)*. Springer-Verlag, 2010. `flint.cs.yale.edu/publications/roch.html`.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan(Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proc. 42nd ACM Symposium on Principles of Programming Languages*, pages 595–608, 2015.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, GA, 2016. USENIX Association. ISBN 978-1-931971-33-1.

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jeremie Koenig, Vilhelm Sjober, Hao Chen, David Costanzo, and Tahnia Ramananandro. Certified concurrent abstraction layers. In *PLDI '18: 2018 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 646–661, 2018.

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, April 2008.

Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

Jieung Kim, Vilhelm Sjoberg, Ronhui Gu, and Zhong Shao. Safety and liveness of mcs lock—layer by layer. In *Proc. 15th Asian Symposium on Programming Languages and Systems (APLAS 2017)*, page (to appear), November 2017.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *[1989] Proceedings. Real-Time Systems Symposium*, pages 166–171, Dec 1989. doi: 10.1109/REAL.1989.63567.

Xavier Leroy. The CompCert verified compiler. http://compcert.inria.fr/, 2005–2014.

Xavier Leroy and Sandrine Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

Hongjin Liang, Jan Hoffmann, Xinyu Feng, and Zhong Shao. Characterizing progress properties of concurrent objects via contextual refinements. In *CONCUR*, pages 227–241, 2013.

C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973. ISSN 0004-5411. doi: 10.1145/321738.321743. URL http://doi.acm.org/10.1145/321738.321743.

Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000. ISBN 0130996513.

Mengqi Liu, Lionel Rieg, Zhong Shao, David Costanzo, and Ronghui Gu. Rt-certikos: A fully verified preemptive os kernel with temporal and spatial isolation. Technical report, Dept. of Computer Science, Yale University, New Haven, CT, July 2018.

Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. Untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

Andrzej S. Murawski and Nikos Tzevelekos. An invitation to game semantics. *ACM SIGLOG News*, 3(2):56–67, 2016.

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *ESOP'15*, pages 333–358, 2015.

Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *18th International Conference on Concurrency Theory*, pages 256–271, September 2007.

Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. *A Practical Verification Framework for Preemptive OS Kernels*, pages 59–79. Springer International Publishing, Cham, 2016. ISBN 978-3-319-41540-6. doi: 10.1007/978-3-319-41540-6_4. URL https://doi.org/10.1007/978-3-319-41540-6_4.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *OSDI*, pages 263–278, 2006.

List of Symbols, Abbreviations, and Acronyms

ADT Abstract Data Type

API Application Programmer Interface

BRASS Building Resource Adaptive Software Systems (a DARPA program that ran from 2015-2019)

CCAL Certified Concurrent Abstraction Layers, a tool for developing certified concurrent system software built at Yale University

CertiKOS Certified Kit Operating System, a project at Yale University

CiC Calculus of Inductive Constructions, a dependently typed constructive logic

Clight a high-level intermediate language of CompCert

CompCert *Compilateur Certifié*, that is, Certified Compiler, a proved-correct optimizing C compiler developed at INRIA

Coq An interactive theorem prover developed at INRIA.

CPU Central Processing Unit

DARPA Defense Advanced Research Projects Agency

DMA Direct Memory Access

FIFO First In First Out

HACMS High-Assurance Cyber-Military Systems (a DARPA program that ran fro m 2012 to 2017)

HOL Higher-order logic

IA-32 Intel Architecture, 32 bits (the instruction set also known as "x86")

IDE Interactive Development Environment

IL Intermediate language

INRIA *Institut national de recherche en informatique et en automatique*, a French national research institution focusing on computer science and ap plied mathematics

IRQ Interrupt Request Line

ISA Instruction-Set Architecture

Isabelle/HOL A proof assistant, developed in England and Germany, based on higher-order logic

LAsm Lower-level Assembly Language used by the CertiKOS project at Yale University

L4 a family of second-generation microkernels

MCS a locking protocol invented by John Mellor-Crummey and Michael Scott at University of Rochester in 1991

ML a functional programming language (originally the MetaLanguage of the Edinburgh LCF proof assistant)

NIST National Institute of Standards and Technology

OS Operating system

pid process id

seL4 A verified-correct microkernel operating system ("security-enhanced L4"), developed in Australia

VM Virtual machine