Advanced Development of Certified OS Kernels

Zhong Shao (PI) and Bryan Ford (Co-PI)

Department of Computer Science
Yale University
P.O.Box 208285
New Haven, CT 06520-8285, USA
{zhong.shao,bryan.ford}yale.edu

July 15, 2010

1 Innovative Claims

Operating System (OS) kernels form the bedrock of all system software—they can have the greatest impact on the resilience, extensibility, and security of today's computing hosts. A single kernel bug can easily wreck the entire system's integrity and protection. We propose to apply new advances in certified software [86] to the development of a novel OS kernel. Our certified kernel will offer safe and application-specific extensibility [8], provable security properties with information flow control, and accountability and recovery from hardware or application failures.

Our certified kernel builds on proof-carrying code concepts [74], where a binary executable includes a rigorous machine-checkable proof that the software is free of bugs with respect to specific requirements. Unlike traditional verification systems, our certified software approach uses an expressive general-purpose meta-logic and machine-checkable proofs to support modular reasoning about sophisticated invariants. The rich meta-logic enables us to verify all kinds of low-level assembly and C code [10, 28, 31, 44, 68, 77, 98] and to establish dependability claims ranging from simple safety properties to advanced security, correctness, and liveness properties.

We advocate a modular certification framework for kernel components, which mirrors and enhances the modularity of the kernel itself. Using this framework, we aim to create not just a "one-off" lump of verified kernel code, but a statically and dynamically extensible kernel that can be incrementally built and extended with individual certified modules, each of which will provably preserve the kernel's overall safety and security properties. In place of the rigid safety conditions used in traditional extension mechanisms (e.g., that kernel extensions must be type-safe), our approach will assure both the safety and semantic *correctness* of extensions with respect to appropriate specifications (e.g., that a file system extension behaves like a file system). Our certified kernel will use this flexibility, for example, to provide accountability and recovery mechanisms, formally guaranteeing that whenever an application fails, the system can always be rolled back to an earlier, consistent state. Our certified kernel will also provide information flow control [20, 100] not only enforcing policies on user applications, but also guaranteeing that the security monitor itself and other kernel modules manipulate all security labels correctly.

More specifically, we propose a new synergistic effort that combines novel advances in operating systems, programming languages, and formal methods to support advanced development of

certified crash-proof kernels. Our work is divided into the following three areas:

1. **Clean-slate design and development of crash-proof kernels.** With certified components as building blocks, we propose to design and develop new kernel structures that generalize and unify traditional OS abstractions in microkernels, recursive virtual machines [41], and hypervisors. By replacing the traditional "red line" (between the kernel and user code) with customized safety policies, we show how to support different isolation and kernel extension mechanisms (e.g., type-safe languages, software-fault isolation, or address space protection) in a single framework. We will also show how to provide built-in accountability and recovery mechanisms from the very beginning and how to combine them with information flow control to enforce the integrity of security labels and capabilities. Our new kernel will not only provide certified guarantee about the soundness of its innate immunity mechanisms but also offer solid support for new adaptive immunity mechanisms.

2. **Programming languages for building end-to-end certified kernels.** OS kernels must address a multitude of abstraction layers and programming concerns. We propose a new open framework for supporting certified low-level programming and cross-abstraction linking of heterogenous components. We will develop a set of domain-specific variants of assembly and C-like languages. Each such variant will be equipped with a specialized program logic or type system (i.e., DSL). We will apply them to certify different components at different abstraction layers (ranging from scheduler, interrupt handling, virtual memory manager, optimistic concurrency, file system, to information flow control), and then link everything together to build end-to-end certified systems. By imposing DSLs over familiar low-level constructs, we can program and verify kernel components at a higher abstraction level, yet without sacrificing code performance or precise control over hardware resources.

3. **Formal methods for automated analysis and proofs.** To make our DSL-centric approach scalable, we propose to build a new integrated environment named VeriML [92] for combining automated provers and decision procedures (tailored to each DSL) with certified programming and proof development. Existing automated verifiers often depend on a rather restricted logic. Proof assistants with a richer meta logic (e.g., Coq [54]), however, provide poor support for automation. VeriML extends ML-style general-purpose programming with support for type-safe manipulation of arbitrary logical terms. Typed VeriML programs serve as excellent proof witnesses, since they are much smaller than proof objects (in the meta logic). We will build certifying static analysis and rewriting tools to synthesize program invariants automatically and to serve as generalized proof tactics.

Because future computing hosts will almost certainly be multicore machines, as an option to our base effort, we propose to extend all three lines of our work to support the development of certified multicore kernels. Multicore machines will likely require multiple kernels running on different cores that still share memory—this creates new challenges for recovery and security. Certified kernels should offer a significant advantage over traditional ones because they have spelled out all of its formal invariants and abstraction layers, making it easier to identify orthogonal concerns and reason about sophisticated cross-core interaction. Under this option, we will also extend our formal specification and proof efforts to certify larger and more realistic kernels.

We are well-qualified for this work. Zhong Shao has 20 years experience in the design and implementation of high-assurance languages, compilers, and runtime systems. He was a coauthor of the SML/NJ compiler, the main architect of the FLINT certifying infrastructure, and a PI on the DARPA OASIS PCC project (in 1999-2004). During the last 10 years, his FLINT group has pioneered and led an extensive body of work on certified low-level programming and formal methods (available at `http://flint.cs.yale.edu`). Bryan Ford is a leading expert on operating systems and has 15 years experience in the design and implementation of extensible OS kernels and virtual machine monitors. He was a key member of the Utah Flux group, an architect and the main developer of the OSKit infrastructure, the Fluke kernel, the Unmanaged Internet Architecture (UIA) project at MIT, and the Vx32 virtual sandboxing environment.

2 Technical Approach

OS kernels are critical software components that can have the greatest impact on the resilience, extensibility, and security of the underlying computer systems. They often have to address a multitude of potentially interfering concerns ranging from boot loading, virtual memory, interrupt handling, thread preemption, optimistic concurrency, protection and isolation, information flow control, file system, device drivers, interprocess communication, persistence, recovery, to all kinds of application-specific kernel extensions. Getting all of these work properly with each other is a difficult task. As the hardware community moves deep into new multicore or cyber-physical platforms, the complexity of OS kernels could only get much worse even though the demand for crash-proof kernels will become much more urgent.

We believe that the most promising approach to deal with such complexity is to take a clean slate approach to reexamine these different programming concerns and abstraction layers, spell out their formal specifications and invariants, and then design and develop new kernel structures that minimize unwanted interferences and maximize modularity and extensibility.

To this end, we propose to apply recent advances on certified software [86] to the development of a novel OS kernel. Certified software, which originates from Proof-Carrying Code (PCC) [74], consists of a binary machine executable plus a rigorous machine-checkable proof that the software is free of bugs with respect to specific requirements. A certified OS kernel is a library-based kernel [21] but with formal specifications and proofs about all of its abstraction layers and system libraries. It formally guarantees that proper recovery and protection mechanisms are in place, so that the kernel itself will not behave unexpectedly, and so that crashes by user applications will not damage the consistency of kernel state or the integrity of security labels.

Our proposed research will attack the following three important questions:

- Under a clean-slate approach, what are the right OS kernel structures that can offer the best support for resilience, extensibility, and security?
- What are the best programming languages and developing environments for implementing such certified crash-proof kernels?
- What new formal methods we need to develop in order to support the above new languages and make certified kernels both practical and scalable?

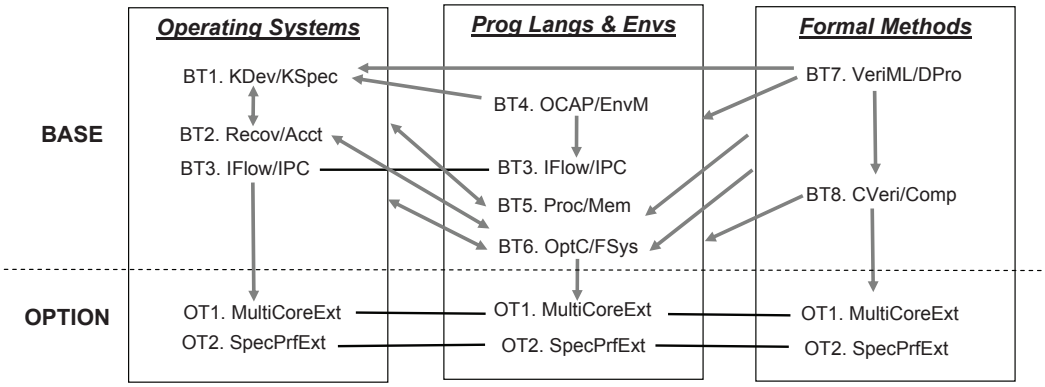


Figure 1: Subprojects, tasks, and their relationships.

We will develop new technologies in operating systems, programming languages, and formal methods, and integrate them to build certified crash-proof kernels. The individual technologies will also be useful as scientific advances and as practical tools for constructing other certifiably dependable software for crash-proof computing hosts.

Figure 1 is an overview of the component parts of this research project. There are eight tasks (BT1-8) in the base and two tasks in the option. Many tasks require co-designs involving multiple technical areas; the edges between subprojects and tasks illustrate their dependency and also the synergistic nature of the proposed work. KDev/KSpec is the new kernel design that generalizes and unifies traditional OS abstractions, with formal specifications. Recov/Acct is a new technique for supporting the accountability and recovery mechanism. IFlow/IPC is a new language-based technique for enforcing information-flow control and for supporting efficient secure interprocess communication. OCAP/EnvM is our new open framework for supporting certified low-level programming, environment modeling, and certified linking. Proc/Mem consists of domain-specific program logics (DSLs) and linking mechanisms for building certified thread/process management modules, interrupt handling, and virtual memory managers. OptC/FSys consists of DSLs for reasoning about optimistic concurrency, persistence and rollbacks, and file systems. VeriML/DPro is our new integrated environment for combining automated provers and decision procedures with certified programming and proof development. CVeri/Comp consists of an automated program verifier and analysis tool and a simple certifying compiler for a subset of C implemented in VeriML. MultiCoreExt consists of subtasks that extend our work to support multicore kernels. SpecPrfExt consists of formal specification and proof efforts to certify larger and more realistic kernels.

2.1 Clean-Slate Design and Development of Crash-Proof Kernels

Our kernel design approach is broken into three main areas, corresponding to proposal tasks BT1–3. First, we will design and build a kernel from the ground up around the use of certified plug-ins, which will provide provably correct but extensible and replaceable implementations of key kernel

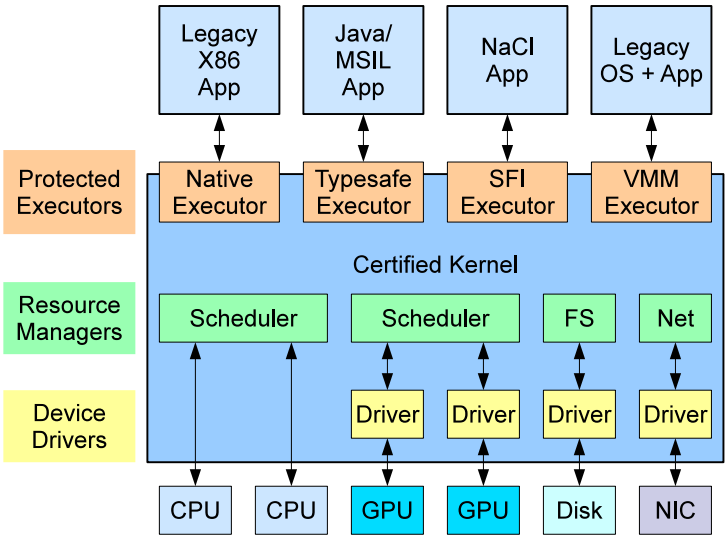


Figure 2: A secure but flexible kernel design built from certified plug-ins.

modules such as protected code executors, resource managers, and device drivers. Second, we will design the kernel's storage and computation mechanisms to incorporate pervasive support for recovery and accountability, enabling automatic identification and precise tracing of application-level crashes or compromises, corruption due to hardware errors, etc. Third, the kernel will support explicit information flow control (IFC) on all forms of inter-process communication (IPC): not only on explicit communication mechanisms such as messages or shared memory, but also via a novel *temporal information flow control* (TIFC) mechanism to control leaks due to implicit timing channels. The following sections describe these aspects of the design in more detail.

A Kernel Design for Certified Heterogeneous Computation (BT1,OT1) Today's operating systems must supervise a wide variety of new and legacy applications running in multiple types of execution environments (e.g., 32-bit versus 64-bit x86, conventional CPU versus GPU architectures, typesafe versus unsafe languages), and manage many different types of physical resources (memory, disk, flash, network, etc.) each implemented by many possible devices requiring different device drivers. While there are many approaches to adding support for "safe" extensions to an existing OS kernel [8, 12, 46, 76, 90, 91], such mechanisms fail to meet our needs in two main ways. First, they address only low-level notions of safety (e.g., can the extension corrupt arbitrary memory?), while ingoring higher-level notions of safety specific to the purpose of a particular module (e.g., does a file system plug-in actually behave like a file system?). Second, classic extension mechanisms assume there is a more or less full-featured "base kernel" to extend, and do not address safety within that base kernel. Classic microkernels [65] attempt to minimize the size of that base kernel, and systems like Singularity [22] implement as much of the base kernel as possible in a typesafe language, but the semantic correctness of the base kernel remains merely an assumption.

Our approach, in contrast, will be to design the "base kernel" itself so as to be composed purely of modular, replaceable, and individually certifiable plug-ins that interact to implement *all* the ker-

nel's functions. In our design, there is no longer a "base kernel" at all *per se*, because the "base kernel" is nothing but interacting plug-ins. The kernel will support a number of different module *classes*, each supporting a different kernel function, and each class embodying different safety and correctness models for certification purposes. As illustrated in Figure 2, some of the important certified module classes will include device drivers for specific types of hardware; physical resource managers such as schedulers, memory managers, and file systems; and *protected executors* that implement secure "sandboxes" for different types of application code.

Our design's class of protected executors, in particular, addresses the increasingly important need to support a wide variety of application code execution mechanisms both securely and efficiently. Today's complex applications and services are often composed of many interacting processes implemented in different languages, sometimes using non-traditional processing hardware such as GPUs for improved performance. Further, many of these applications depend on the APIs of a particular legacy OS, and thus need to be run within a virtual machine (VM) supporting both the application and the legacy OS it requires.

Our kernel therefore replaces the traditional "red line" with an extensible suite of certified executor plug-ins, each of which supervises the execution of a particular class of untrusted process. For example, one executor will run x86 code natively in user mode ("ring 3") [56] as in traditional process models; a second might run typesafe code in kernel mode as in SPIN [8] or Singularity [22]; a third executor might run native x86 code in the kernel via dynamic translation [39] or SFI [93] to enable more efficient fine-grained IPC or instruction-level instrumentation; a fourth executor might serve as a full-system virtual machine monitor (VMM) or hypervisor, running an entire legacy *guest OS* and its many *guest processes* within one host process. The kernel design will allow any of these forms of "processes" to interact via IPC, and to create child processes or child VMs of any type, thereby supporting efficient nested virtualization [41]. In this model, for example, a web server process running within a virtualized legacy guest OS could invoke a web server plug-in running under a different, *non-legacy* executor for greater security and/or efficiency.

History-based Accountability and Recovery (BT2,OT1) Even with a certified kernel, the correct functioning of an overall system still depends on that of the underlying hardware and the applications running atop the kernel. But even correctly-designed hardware is prone to failures due to slight manufacturing defects, cosmic rays, etc., which may corrupt both stored data and computed results. Further, since in most cases it will be infeasible to certify all application code, applications will still manifest bugs and security vulnerabilities that, while hopefully not compromising the kernel's protection boundaries, still need to be contained, traced, analyzed, and debugged.

To support the analysis of and recovery from software and hardware failures of all kinds, our kernel design will incorporate *history-based accountability and recovery* as a basic kernel primitive. In a pure, "ideal" (but unrealistic) implementation of this idea, the OS kernel would keep in some internal, "write-once" memory, a complete history log of everything it has ever computed, or has ever read from or written to an I/O device. If the system ever produces output manifesting unexpected or suspcious behavior, e.g., in application-level output, the system can use replay techniques [18,45,48] to backtrack the cause of the unexpected output precisely to the application bug, intrusion, or hardware failure that caused it. Further, by *proactively* replaying selected portions of past computations on different machines, it is possible to detect hardware failures or other incorrect

behavior even before this behavior detectably manifests as bad output [48].

Maintaining a full record of a system's execution is costly, however, especially on multicore machines [19]. We plan to develop techniques to reduce history storage and processing costs to levels tolerable during everyday use. One such technique is to leverage our ongoing work on deterministic parallel execution [4–6] to create "naturally deterministic" application-level environments. In such an environment, the results of a computation depend *only* on its explicit inputs, and not by the timings of interrupts, messages, accesses to shared memory, etc. Thus, determinism facilitates history-based accountability and recovery by eliminating the need to log nondeterministic events, which in current replay systems often represents the vast majority of logging costs.

Ideas from our prior work on system call atomicity in the Fluke kernel [40] will also play an important role in supporting efficient recovery and accountability in the proposed kernel. This design atomicity ensures that all user and kernel state can be consistently checkpointed at any time during kernel execution, without incurring unbounded delays that might reduce performance or responsiveness, or even create denial-of-service (DoS) attack vulnerabilities, as is possible with blocking system calls in current monolithic kernels.

Even with deterministic execution and system call atomicity, however, we will often need to limit the logging of *true* inputs as well to provide a balance between accountability/recoverability and practical costs. If a small computation produces large intermediate output files used as inputs to another computation, for example, we can maintain exact repeatability by saving only the original inputs and final outputs, knowing that the large intermediate results can be reproduced if needed. One approach to this "history optimization" problem is to treat the system's history as a weighted dependency graph and use algorithms to find min-cuts on that graph: i.e., state snapshots carefully chosen to minimize the amount that must be stored. Another approach we will explore is to combine history "compression" and processing with garbage collection techniques, yielding a kernel memory manager that synergistically combines garbage collection, computation history compression, periodic checking and scrubbing of data stored in memory and on disk, and periodic spot-checking of past results computed by all CPU cores.

Information Flow Control and IPC (BT3,OT1) To provide strong mechanisms for controlling information propagation via inter-process communication (IPC) and other interaction mechanisms, our kernel will incorporate and build on recent work on Information Flow Control (IFC) [20, 72, 100], which enable the explicit labelling of information and control over its propagation through and out of a system. Our design will advance beyond prior work in two important ways, however.

First, through the common certification framework we will develop to certify all kernel executors, we will make it possible to enforce IFC using a common set of labels and rules across multiple heterogeneous execution environments: e.g., to track information flows precisely as they cross between typesafe language domains, legacy native code domains, and even guest virtual machines. This will require addressing a number of technical challenges, such as the differences in labeling schemes and control rules between traditional language-based IFC systems [72] and OS-based ones [20, 100], as well as the pragmatic differences in the granularity of flow tracking that different schemes provide (e.g., variable-granularity versus file and process granularity). We are confident, however, that the detailed formal understanding of inter-process communication and interaction mechanisms that will necessarily arise out of our kernel certification effort will also lead

to practical solutions to these challenges.

The second unique aspect of our kernel design is that it will address not just conventional, explicit interactions between processes, but also covert timing channels [57, 95], which have been largely avoided in previous IFC work but are becoming increasingly critical to real-world security [1, 2, 9, 82, 84, 94]. Further leveraging our work in deterministic execution and combining them with classic IFC techniques, we will design the kernel to provide pervasive controls over how and when potentially sensitive timinig information can enter or affect the results of any untrusted application computation. We describe these ideas in more detail elsewhere [4].

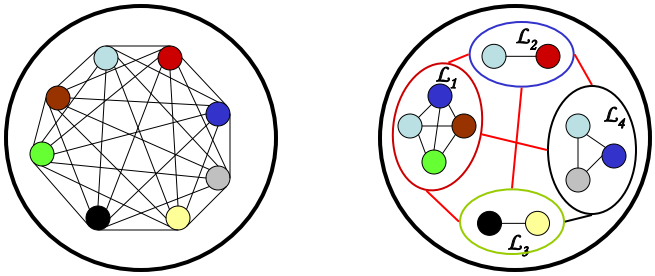
If we wish to timeshare a CPU core between two untrusted processes and prevent timing channels between them, for example, a classic approach would be to arrange a fixed timeslice for each process, not varying depending on either process's actual usage, and clear all caches and other state affecting timings on context switches. While this approach may be useful in some situations, it is undesirable due to the potential waste of resources it incurs, due to the flushing of potentially useful state and giving up the ability of one process to utilize fully any resources left underutilized by others. An alternative solution we will explore is to timeshare the processes without restriction, but run them deterministically and thus prevent them from being able to "tell the time" locally while running on the timeshared CPU core. If one process has a semantic need to tell the time, its "read time" request leads to an IFC "taint" fault, e.g., causing the process to be migrated to some other CPU core that is not timeshared at fine granularity between untrusted processes, and on which the system time is thus "untainted" by information from other processes.

Taking this approach further, suppose a process wishes to run on timeshared cores for performance, but also use fine-grained internal timers to make decisions for load-balancing parallel computations across cores or similar internal optimization purposes. In this case, instead of reading "tained" high-resolution timers directly, the process can fork off a parallel process to make dynamic load-balancing decisions on behalf of the original process. This new load-balancing process will become tainted by timing information from other processes sharing the same core. The kernel's determinism and IFC enforcement mechanisms, however, will allow the tainted process to affect only the *scheduling* (and hence execution performance) of the original process it was forked from, and not the actual *results* computed by that process; the original process will thus run (deterministically) without itself becoming tainted with potentially leaked timing information.

2.2 Programming Languages for Building End-to-End Certified Kernels

A key first step to make certified kernels practical is to actually show that it is possible to carry out end-to-end certification of a complete system. Low-level system software uses many different language features and span many different abstraction levels. Our recent work [28] on building a simple certified kernel exposes such challenges. The kernel has a simple bootloader, kernel-level threads and a scheduler, synchronization primitives, hardware interrupt handlers, and a simplified keyboard driver. Although it has only 1,300 lines of x86 assembly code, it already uses features such as dynamic code loading, thread scheduling, context switch, concurrency, hardware interrupts, and device drivers. How do we verify the safety or correctness properties of such a system?

Verifying the whole system in a single program logic or type system is impractical because, as Figure 3(a) shows, such a verification system needs to consider all possible interactions among



(a) The One-for-All Logic (b) Domain-Specific Logics

Figure 3: A DSL-centric view for building certified OS kernels

these different features (possibly at different abstraction levels). The resulting logic, if exists, would be very complex and difficult to use. Fortunately, in reality, software developers never seem to use all the features at the same time. Instead, only limited sets of features—at certain abstraction level—are used in specific program modules. It would be much simpler to design and use specialized "domain-specific" logics (DSL) to verify individual program modules, as shown in Figure 3(b). For example, for the simplified kernel, dynamic code loading is only used in the OS boot loader [10]; interrupts are always turned off during context switching; embedded code pointers are not needed if context switching can be implemented as stack-based control abstraction [31].

OCAP and Environment Modeling (BT4) Our approach for programming certified kernels follows this DSL-centric view. As shown in Figure 4, we first provide a detailed hardware model inside a mechanized meta logic [54, 92]). This formal model provides a detailed semantic description for all machine instructions, hardware registers, interrupts, devices, and timing behaviors, etc. We assign a trace-based semantics to the machine-level programs: the meaning of each binary is simply its set of execution traces; a certified program satisfies a particular specification if its set of execution traces does so. Each DSL provides a specific abstraction of the hardware model. Different kernel components can be certified using different DSLs; hardware details that are not relevant to the current property of interests are abstracted away. To build a complete system, we have also developed an open framework named OCAP [24] to support interoperability between different DSLs and certified linking of heterogeneous components.

Our proposed extension to OCAP will use an shallow-embedded assertion language capable of specifying arbitrary program traces. OCAP rules are expressive enough to embed most existing verification systems for low-level code. OCAP assertions can specify invariants enforced in most type systems and program logics, such as memory safety [98], well-formed stacks [31], non-interference between concurrent threads [28], and temporal invariants [44].

The soundness of OCAP ensures that these invariants are maintained when foreign systems are embedded in the framework. To embed a specialized verification system $\mathcal{L}$, we first define an interpretation $[\![\_]\!]_{\mathcal{L}}$ that maps specifications in $\mathcal{L}$ into OCAP assertions; then we prove system-specific rules/axioms as lemmas based on the interpretation and OCAP rules. Proofs constructed in each system can be incorporated as OCAP proofs and linked to compose the complete proof.

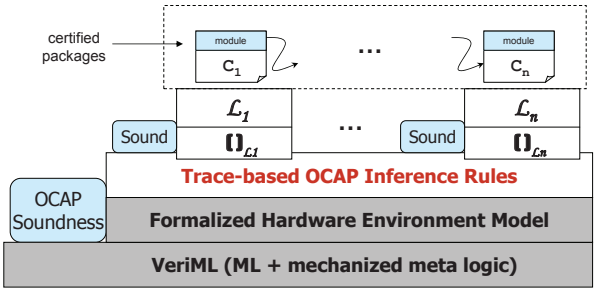


Figure 4: An open framework for building certified software

The OCAP approach to environment modeling makes it possible to lift much of the kernel development, validation, and verification to a higher abstraction layer, yet it still maintains the consistency with the raw hardware semantics. We plan to further extend OCAP to support modeling of various abstraction layers and executors based on the kernel design described in Section 2.1.

Certified Programming (BT3-6) We will develop a large set of domain-specific variants of assembly and C-like languages. Each such variant will be equipped with a specialized program logic or type system (i.e., DSL). Our OCAP/VeriML framework (also see Section 2.3) provides a great setup for implementing these variants, DSLs, and the corresponding certifying compilers. By imposing DSLs over familiar low-level constructs, we get the best of both worlds: we can program and verify kernel components at a higher abstraction level, yet without sacrificing the code performance and the firm control of hardware resources.

Certified Assembly Programming (CAP) [98] is a logic-based approach for carrying out general program verification inside a rich mechanized meta-logic. Subsequent work on CAP (by the PI and his students) developed new specialized program logics for reasoning about low-level constructs such as embedded code pointers [77], stack-based control abstractions [31], self-modifying code [10], garbage collectors [68], hardware interrupts [28], and nonblocking concurrency [44]. We plan to adapt these DSLs and also design new ones to support certified programming of different abstractions and plug-ins in our new kernel designs.

Virtual Memory Management (BT5,OT1-2) We propose to develop new DSLs for writing certified virtual memory managers. Figure 5 (left) shows decomposition of a kernel virtual memory manager. Here, the lowest layer is a model of the physical hardware, except that the OS is preloaded into memory. The next layer is the physical memory manager (PMM), which provides functions for marking physical memory pages as used or free. The next layer is the page table manager (PTABLE), which provides functions for updating a page table. The layer after that is the virtual memory manager (VMM), which provides functions for adding or removing virtual pages. The top layer is the rest of kernel, which only works in the presence of paged memory.

Each layer of code provides a layer of abstraction to the components above, enforced via a new DSL. Figure 5 (left) also gives a dotted line showing that the left side of the line is code running

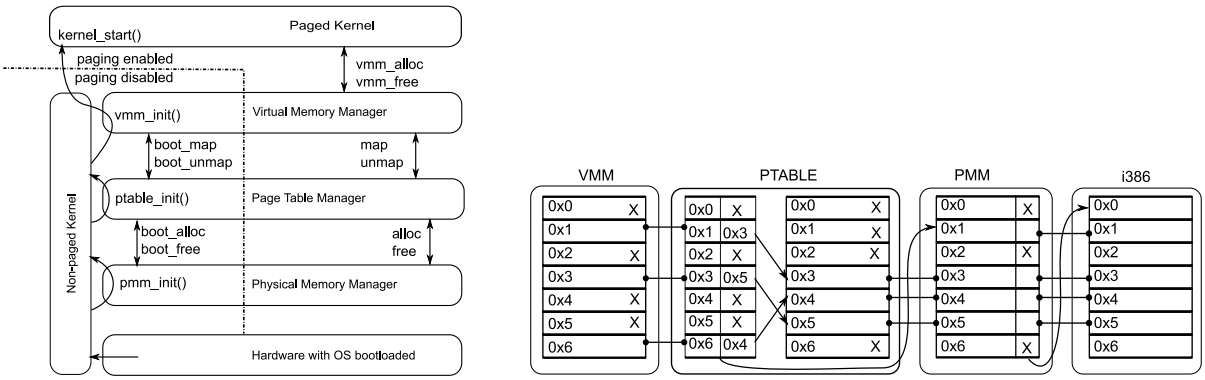


Figure 5: Decomposition of a virtual memory manager

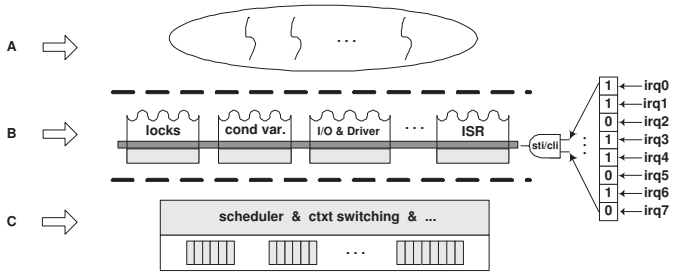


Figure 6: Decomposition of a preemptive thread implementation

with paging disabled and the right side is code running with paging enabled. Notice that at the top layer, the paged kernel is always running with the address translation turned on, but the bottom four layers must support both paged and non-paged code. During the bootstrap process, the non-paged kernel would invoke the initialization routines for the middle three layers before turning on the paging flag. If a routine might be called in the non-paged mode (e.g., during initialization) or the paged mode, it must be duplicated and compiled twice (e.g., `boot_alloc` and `alloc`); but they must satisfy the similar sets of invariants.

To put these components together, in Figure 5 (right), we show how each abstraction layer maps into the lower abstraction layer. For example, the VMM machine only sees the virtual address space (virtual pages marked with X's are inaccessible). The actual page tables (the left table in the PTABLE box) and the underlying physical memory (the right table in the PTABLE box) are only visible inside the PTABLE machine. Page tables in PTABLE are abstract — we don't know where it is stored in physical memory; they are stored in page 0x1 in the PMM machine. PMM also maintains a data structure which marks all the used physical pages; this information is abstract in PMM but actually stored in page 0x0 on the bare hardware. With all these mapping, we can create verified transformations layer by layer. This chain of abstractions allow us to guarantee that the entire code is safe to run on the actual hardware, while the verification took place over various abstract machine models using specialized DSLs. For the Option, we plan to extend and adapt this base framework to certify advanced virtual memory features and primitives.

Thread/Process Management (BT5,OT1-2) We plan to build certified thread/process management modules based on our recent work [28] in which we have successfully certified the partial correctness of a preemptive thread library. The thread library consists of thread context switching [78], scheduling, synchronizations, and hardware interrupt handlers. In Figure 6, we show how we decompose the implementation into different abstraction layers. At the highest level (Level A), we have preemptive threads following the standard concurrent programming model. The execution of a thread can interleave with or be preempted by other threads. Synchronization operations are treated as primitives. Hardware interrupts are abstracted away and handled at Level B. Code at Level B involves both hardware interrupts and threads; synchronization primitives, input/output operations, device drivers, and interrupt handlers are all implemented at this level. Interrupt handling is enabled/disabled explicitly using `sti/cli`. At the lowest level (Level C), the thread scheduler and the context switching routine manipulate the threads' execution contexts, which are stored in thread queues (on the heap). Interrupts are invisible at this level because they are always disabled. Libraries implemented at a lower level are exposed as abstract primitives for the level above it; their operational semantics in the high-level abstract machine serves as formal specifications for the low-level implementation. The stratified system model gives us a systematic and principled approach for controlling complexity. We can focus on a subset of language features at each level and certify different software components using specialized program logics.

Our proposed work is to integrate the above framework [28] with VMM and also to support realistic scheduler, realistic stack management, multilevel interrupts, and various other event or signal handlers. For our multicore extension, multiple scheduler may run at the same time so access to the thread queues must be synchronized as well; turning off interrupts can no longer prevent thread interleaving so synchronization libraries must be implemented differently.

Information Flow Control and IPC (BT3,OT1-2) Information flow control is a new kernel mechanism, first introduced in HiStar [100], to implement protection in traditional kernels and to enforce application-level security policies. Subsequent work [101] shows that it is possible to simplify the HiStar kernel greatly by introducing a separate security monitor. Both of these work, however, still require a nontrivial sized trusted computing base, and neither spells out the exact formal property the kernel can guarantee and what requirements any kernel extension must satisfy.

We plan to formalize and incorporate HiStar-style information flow control into our certified kernels. We will take the language-based approach (as done in Jif [72]) but instead of relying on a type system, we will design new specialized program logics to certify low-level label manipulation code. We will first formalize the traditional notion of the principals as well as the HiStar-style categories as a basic language primitive. We will then extend previous work on Concurrent Separation Logic (CSL) [80] and Local Rely Guarantees (LRG) [23] to reason about multi-threaded shared memory programs but with explicit domain separation and security labels. Interprocess communication can be either implemented as cross-domain procedure calls or as messaging passing via contract-based channels [22]. Security labels and declassification constraints can be formalized as resource invariants or rely-guarantee conditions. Finally, we will integrate the certified label mechanisms with our certified virtual memory and thread management modules.

Optimistic Concurrency and File System (BT6,OT1-2) Optimistic concurrency algorithms [52, 53] (e.g., non-blocking synchronization, software transactional memory) allow concurrent access

to shared data and ensure data consistency by performing dynamic conflict detection. These algorithms can be more efficient than coarse-grained lock-based synchronization if there is sufficient data independence. However, the design of the algorithms has to consider many more thread-interleaving scenarios than coarse-grained synchronization. The algorithms are usually complex and error-prone. Their correctness is usually far from obvious and is hard to verify too.

In our recent work [44], we have developed a new program logic that uses invariants on historical execution traces to reason about optimistic concurrency algorithms. The logic extends previous work on LRG by introducing past tense temporal operators in the assertion language. The pre- and post-conditions in our judgments only specify the program state at the current program point and the execution history before this point. It allows us to specify historical events directly without using history variables, which makes the verification process more modular and intuitive.

We have successfully applied our new logic to certify a few classic optimistic algorithms [16, 69]. Our proposed work is to extend the history logic (or design new DSLs) to certify a large class of optimistic concurrent data structures commonly used in kernel programming. We will also apply new DSLs to certify the file system implementation and also the history-based accountability and recovery mechanism described in Section 2.1. For the multicore extension, we plan to extend the history logic to include assertions on future traces; this will allow us to reason about the liveness properties of certain critical kernel routines; we will also look into concurrent reference counting [63] and real-time garbage collection algorithms [83].

2.3 Formal Methods for Automated Analysis and Proofs

The end goal of certified kernels is to establish the important requirement specifications for all the kernel modules, abstract machines, and libraries. It uses a rich meta logic and a large set of DSLs (implemented on top of the OCAP in the meta logic) to capture deep kernel invariants and to spell out claims required for building a resilient and secure host. Automated analysis and proof construction are extremely important and desirable to make the whole approach scalable.

Unfortunately, existing proof assistants or theorem proving tools do not offer good support for certified programming. Automated theorem provers and SMT solvers such as Z3 [13] often depend on a rather restricted logic—this is in conflict with the rich meta logic (which makes heavy uses of quantifiers) required for supporting the OCAP framework; they also do not provide any interface or generate witnesses for linking with other tools. Proof assistants such as Coq [54]—which we used heavily in the past—provide rather poor support to automation.

VeriML and Type-Safe Proof Scripts (BT7,OT2) We have recently developed a prototype implementation of VeriML [92]—a novel language design that couples a type-safe effectful computational language with first-class support for manipulating logical terms such as propositions and proofs. Our main idea is to integrate a rich logical framework—similar to the one in Coq—inside a computational language inspired by ML. This way, the OCAP-style certified linking can be supported and the trusted computing base of the verification process is kept at a minimum. The design for VeriML is such that the added features are orthogonal to the rest of the computational language, and also do not require significant additions to the logic language, so soundness is guaranteed.

VeriML can programmatically reflect on logical terms so that we can write a large number of

procedures (e.g., tactics, decision procedures, and automated provers) tailored to solving different proof obligations. VeriML also provides an unrestricted programming model for developing these procedures, that permits the use of features such as non-termination and mutable references. The reason for this is that even simple decision procedures might make essential use of imperative data structures and might have complex termination arguments. One such example are decision procedures for the theory of equality with uninterpreted functions. By enabling an unrestricted programming model, porting such procedures does not require significant re-engineering.

We plan to extend VeriML into a more realistic tool: this includes building more user-friendly surface syntax, providing support for finite precision integers, and also extending the base language with more powerful context matching construct. We will then implement a large number of automated decision procedures (like Z3) in VeriML. We will also implement our new OCAP framework, environment modeling, and all the DSLs in VeriML. Because VeriML uses a logic language that is a subset of Coq's calculus inductive constructions, and both can provide explicit proof witnesses, linking these two tools should be fairly easy.

Another effort we want to pursue is to use type-safe proof scripts to serve as proof witnesses. When building certified kernels, we ideally want some sort of certificate that can be easily checked by a third party. Under Foundational PCC [3, 50], this certificate takes the form of a proof object of some suitable logic. In Coq, proof objects are made feasible in terms of memory consumption, by including a notion of computation inside the logic (i.e., automatic $\beta\iota$ reductions) and trading derivation steps for computation. This complicates proof checking, increasing the overall TCB. We propose to use a simpler, well-established logic that does not include such a notion of computation as the meta logic, which further reduces the TCB. With VeriML, we can write procedures producing proofs in this logic in a type-safe manner. VeriML programs that produce proofs can be seen as type-safe proof scripts, whose types guarantee that proof objects establishing the desired properties exist, if these programs terminate. The user is free to trust some of these proof-producing procedures, so as to have them not generate full proof objects; still, type safety guarantees that such proof objects exist in principle. We are therefore led to "tunable-size" proof objects, where users can trade memory consumption for increased assurance.

Automated Program Verifiers and Analysis Tools (BT8,OT1-2) We anticipate that our kernel code will be written in variants or subsets of assembly and C, each of which is extended with a set of library primitives (implemented by code at an abstraction layer below), and each is equipped with a DSL to lift programming to a higher abstraction layer. A large portion of these code, however, can be certified using the traditional automated program verification method as done by Boogie [7]. Given a C-like program annotated with proper loop invariants and assertions for all procedures, the automated verifier can calculate a verification condition (for the program) and feed it to a Z3-like SMT solver to determine whether the program indeed satisfies the required specification. The existing Boogie-Z3 framework is not extensible because it uses a rather limited logic. With VeriML, we can implement the Boogie-like program verifiers, the Z3-like solvers, and the proof scripts for higher-order logic, in a single typed framework. In fact, the program verifier itself is just yet another type-safe proof tactic (e.g., for an Hoare triple). Depending on the needs of the kernel code, we also plan to implement automated static analysis tool for synthesize loop invariants and produce proof witnesses. The advantage of VeriML is that we can decompose a

large program into different components, each may be certified using different tactics, yet in the end they can all be linked together—in VeriML—to form the proof-witness for the entire program.

To turn certified C programs into certified assembly components suitable for linking in the OCAP framework, we also plan to use VeriML to build a certifying compiler [75] for a small subset of C. Leroy [62] has already built a certified compiler named CompCert in Coq, but it does not handle concurrent programs and it is unclear how the generated code from CompCert can be linked with other certified assembly code. The VeriML type system allows us to express precisely what program properties needs to be preserved during compilation. For the Option, we plan to extend the basic tools developed under the base to handle more realistic languages.

3 Prior Work

Zhong Shao During the past 10 years, Dr. Shao and his FLINT group at Yale have worked extensively on various aspects of proof-carrying code [49, 50, 71, 89, 98], certified OS kernels and runtime systems [10, 28–30, 66, 68, 78], certified low-level programming [25–27, 31, 77, 98, 99], optimistic concurrency [44], relaxed memory models [32], certifying compilers [59, 60, 85, 87, 88], and proof assistants and automated theorem proving [92]. Here are a list of prior projects that are most related to the current proposal.

In their PLDI'08 and JAR'09 papers [28, 30], Shao and his students successfully developed a sequence of specialized program logics (DSLs) for certifying low-level system programs involving both hardware interrupts and preemptive threads. They showed that enabling and disabling interrupts can be formalized precisely using simple ownership-transfer semantics, and the same technique also extends to the concurrent setting. By carefully reasoning about the interaction among interrupt handlers, context switching, and synchronization libraries, they are able to—for the first time—successfully certify a preemptive thread implementation and a large number of common synchronization primitives. The Coq implementation of this certified thread implementation is made publicly available. In their VSTTE'08 paper [29], Shao and his students showed how to apply their OCAP framework and domain-specific logics to build end-to-end certified systems.

In their PLDI'07 and TASE'07 papers [66, 68], Shao and his students have successfully developed new program logics and used them to mechanically verify assembly implementations of mark-sweep, copying and incremental copying garbage collectors (with read or write barriers) in Coq. They have also verified sample mutator programs and linked them with any of the GCs to produce a fully-verified garbage-collected program. The Coq implementation of these certified garbage collectors was made publicly available and later extended to support more realistic certified collectors by researchers at Portland State University [67] and Microsoft Research [51, 96].

In their PLDI'07 paper [10], Shao and his student have successfully developed a set of DSLs for supporting modular verification of general von-Neumann machine code with runtime code manipulation. They have used these logics to certify a realistic OS boot loader that can directly boot on stock x86 hardware and a wide range of applications requiring dynamic loading and linking, runtime code generation and optimization, dynamic code encryption and obfuscation.

In their PLDI'06 paper [31], Shao and his students have successfully developed a set of DSLs for supporting modular verification of low-level programs involving stack-based control abstrac-

tion, ranging from simple function call/return, stack cutting and unwinding, coroutines, to thread context switch. In their POPL'06 paper [77], Shao and his student have successfully developed new program logics for certifying embedded code pointers; in the subsequent TPHOL'07 paper [78], they presented a realistic x86 implementation of the machine context management library.

In their ICFP'10 paper [92], Shao and his student designed and developed a prototype implementation of VeriML —a novel language design that couples a type-safe effectful computational language with first-class support for manipulating logical terms such as propositions and proofs.

In their CONCUR'10 paper [44], Shao and his students a novel program logic that uses invariants on history traces to reason about optimistic concurrency algorithms. They have successfully used it to certify Michael's non-blocking stack algorithm [69] as well as an TL2-style time-stamp-based implementation of Software Transactional Memory [16].

Bryan Ford Dr. Ford has a long track record in building experimental kernels [6, 38, 40, 41], and other relevant research such as lightweight kernel IPC mechanisms [42], novel CPU scheduling architectures [43], and code sandboxing via dynamic instruction translation [36, 39].

In his Fluke kernel [40, 41], Ford and his colleagues designed and built a research kernel from the ground up around the concept of *recursive virtual machines*, as embodied in a novel *nested process model*. This hybrid microkernel/VM model foreshadowed today's popular hypervisors, and enabled virtual machines to be nested more deeply, without the severe slowdown or reductions in functionality commonly observed when recursively nesting conventional virtual machines. Efficient recursive virtualization enables more flexible system composition while retaining the full power and security of the virtual machine model: with full recursive virtualization, for example, an untrusted Java applet downloaded into a Web browser might have the capability of creating and supervising a nested virtual machine, emulating a complete x86 PC and hosting a native guest OS logically running "within" the Java applet. Since this system composition flexibility is showing ever-increasing importance in today's world of ubiquitous virtualization and cobe mobility, our proposed kernel design will incorporate and further develop these system structuring concepts.

In his Flux OS Kit [38], Ford and his colleagues explored methods of building modular kernels out of "plug-and-play" components, in many cases adapted from existing operating systems and encapsulated into libraries with clean, well-defined interfaces. Our proposed kernel design in a sense takes the OS Kit one step further, building a modular kernel structure whose component interfaces are *formally* defined and suitable for modular certification.

More recently, Ford's Vx32 sandbox [39] and VXA archival storage system [36] developed new techniques for lightweight sandboxing of untrusted, native x86 code within a user-level process, enabling untrusted application plug-ins to run safely but with full performance and unrestricted choice of programming language, unlike typesafe language environments like Java and C#. Ford's Vx32 work in part inspired both Google's Native Client [97] and Microsoft's Xax [17] environments for sandboxed native Web applications. At least one of the protected code executors for the proposed OS kernel (see Section 2.1) will build on these native code sandboxing techniques, enabling untrusted code written in legacy languages to interact with the kernel and other processes efficiently under fine-grained protection schemes, by avoiding the high cost of hardware-supported protection changes (kernel entry/exit) when entering or leaving sandboxed code.

Finally, Ford's ongoing work in system-enforced deterministic execution [4–6] will serve as a building block for the history-based recovery and accountability mechanisms in the proposed kernel, enabling the efficient logging, spot-checking, and replay of past computations and data to verify system integrity and recover from hardware or application-level failures or compromises.

In addition to his work in system building, Dr. Ford's work has crossed into programming language theory and practice, in his work on Parsing Expression Grammars [35] and Packrat Parsing [34]. This experience and understanding of formal techniques will be crucial in enabling a close collaboration with Dr. Shao for the proposed kernel design and certification work.

Most of Ford's research results has been publicly released in open source software, including the prototypes resulting from the projects listed above. Many of his research results have seen substantial direct use and indirect impact in the community, such as his OS Kit [38], Vx32 sandbox [36, 39], parsing expression grammars [34, 35], and Structured Stream Transport [37].

4 Comparison with Current Technology

OS Kernel Verification The seL4 group [58] recently demonstrated that it is possible to verify a nontrivial sized microkernel. The seL4 kernel consists of 8700 lines of C and 600 lines of assembler. They have proved a refinement property for the 7500 lines of their C code (meaning that they satisfy some high-level specifications); from this property they can derive the basic memory- and control-flow safety properties. Their work is impressive in that all the proofs were done inside a high-assurance mechanized proof assistant (Isabelle/HOL [81]).

Our proposed work differs from seL4 in many significant ways. First, the focus of our project is not to verify a specific kernel, but rather to take the certified kernel approach to facilitate the co-development of new kernels suitable for the resilient, adaptive, and secure hosts. Second, our proposed work will eliminate many major limitations in the current seL4 effort. For example, their virtual memory manager (in C), the 1200 lines of the initialization code (in C), and the 600 lines assembly code are not verified; they have significantly modified the kernel so that it does not support any concurrency (in the kernel) and provides no explicit memory allocation; all resource management is pushed outside the microkernel; general interrupts are not supported; the capability system in seL4 is less general than the information flow control mechanism in HiStar; these restrictions are precisely those addressed by our research tasks described in Section 2. Third, our DSL-centric approach and new VeriML-based proof environment will make it easier to maintain, develop, and scale our certified kernel effort. Even with all of the above limitations, the seL4 effort still took more than 20-person years. We believe that we can do much better.

One evidence that automated proving tools can dramatically cut down the proof effort is the recent Verve project [96] at Microsoft Research. The Verve kernel consists of mostly C# code taken from the Singularity kernel [55] plus a small nucleus written in 1400 lines assembly. Verve managed to prove the type safety of the entire kernel by combining the partial correctness proof of the nucleus and the type-safety guarantee from a certifying C# compiler (for the rest of the kernel). By using powerful automated proving tools (e.g., Boogie and Z3), Verve managed to certify the nucleus in 9 person-months. Of course, the Verve effort does have quite a few caveats: the linking between the nucleus and the rest of the kernel is not verified; the kernel does not support interrupts;

the rest of the kernel is only shown to be type safe which is far from enough for certifying the integrity of most of the kernel mechanisms (for security and resilience).

Our proposed work would combine the benefits of automated theorem proving with the modularity and expressiveness of the VeriML/OCAP-based certified programming framework. Both the Verve and seL4 kernels only work for uniprocessors which our multicore extension will address. We have the right set of new technologies (see Section 2) which fit together naturally. We have the experience and the right software base to do this (see Section 3).

Extensible Operating Systems Many approaches have been explored to building extensible operating systems. The microkernel design philosophy [64, 65] is to minimize the functionality in the kernel itself so that most system functionality, implemented in user-level servers, can be modified and extended without modifying the kernel. Exokernels [21] take this philosophy even further by exposing bare, "abstraction-free" protected resource access mechanisms in the kernel's basic API. Our approach builds on lessons from microkernels and exokernels but we will also address modularity and formal decomposition of functionality *within* our certified kernel.

Another well-explored approach to extensibility is to allow applications to download extensions into the kernel: e.g., via domain-specific languages [70], typesafe languages [8], static software fault isolation [93], and architecture-specific tricks [12]. All of these approaches address only a rigid, low-level notion of *safety*, however: e.g., ensuring an extension cannot harm the kernel or other processes via wild pointer writes. Our approach to kernel extensibility, in contrast, focuses on supporting certified extensions that are not only *safe* but also *semantically correct* according to a formal specification of their required functionality: e.g., a memory management extension must never allocate the same block to two purposes at once; a scheduler extension must not lose track of threads or run more than one thread at a time; a protected executor extension must not only ensure that untrusted code can be run safely, but that it can be preempted if it loops forever.

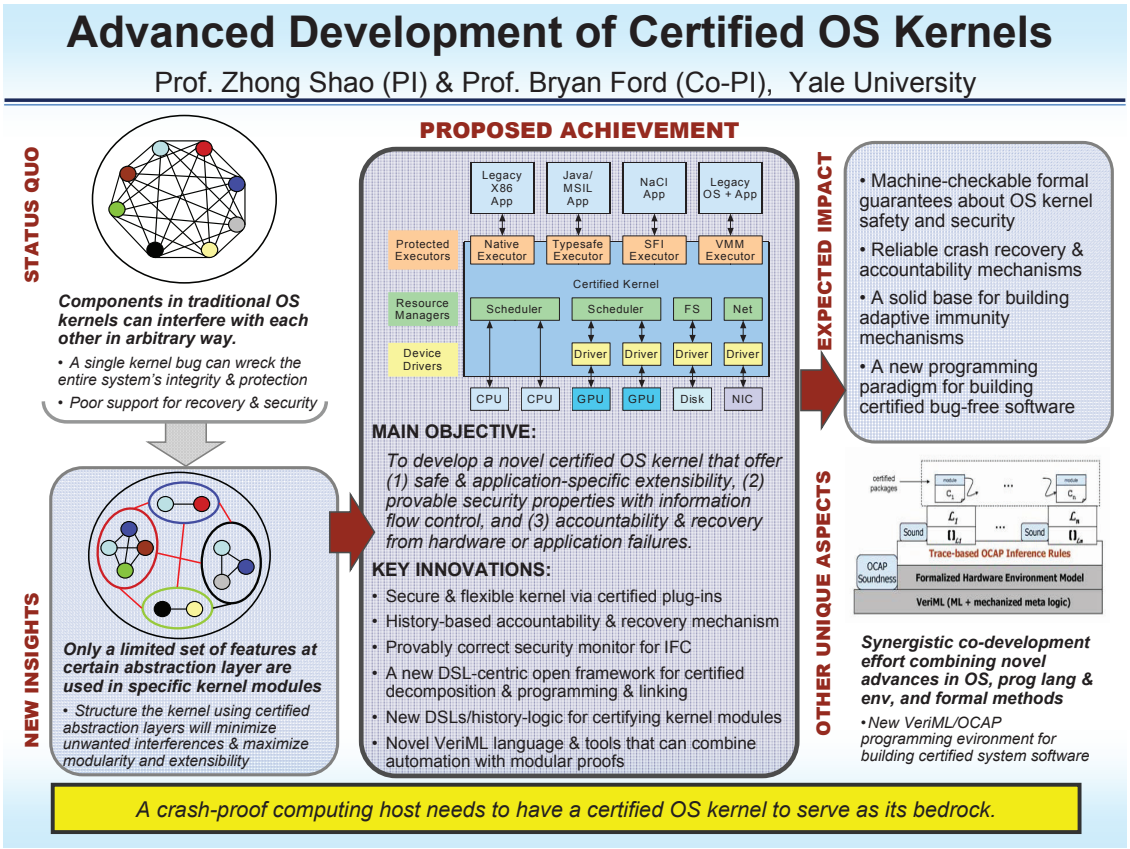
Programming Languages and Formal Methods Mainstream languages today offer rather poor support to certified programming. JML [61] and Spec# [7] extended an existing object-oriented language with assertions, but their assertion language is rather limited; they also took the one-logic-for-all view which is not suitable for capturing diverse kernel abstraction layers and plug-ins. Although high-level languages (e.g., Java and C#) have their benefits, they are not suitable for low-level kernel programming. They require a very complex native interface library which makes end-to-end kernel verification much more difficult. They also provide a rather rigid concurrency library which is not what we want in the context of an extensible kernel. Our DSL-centric OCAP framework will address these problems: it supports certified programming at a higher abstraction level, yet without sacrificing the flexibility, the code performance, and the firm control of hardware resources. The DSL approach is also important for supporting domain-specific decision procedures which are not possible in the one-logic-for-all or one-language-for-all approach.

The LTac language [14, 15] available in the Coq proof assistant is an obvious point of reference for our VeriML work. LTac is an untyped domain-specific language that can be used to define new tactics by combining existing ones, employing pattern matching on propositions and proof contexts. Our VeriML is strongly typed, statically guarantees correct behavior with regards to binding, and gives access to a richer set of programming constructs, including effectful ones; this, we believe, enables the development of more robust and complex tactics and decision procedures.

The comparison with the LCF approach [47] is interesting both from a technical as well as from a historical standpoint, seeing how ML was originally developed toward the same goals as VeriML aims to accomplish. The LCF approach to building a theorem prover for a meta logic would amount to building a library inside ML that contained implementations for each axiom, yielding a term of the abstract *thm* datatype. VeriML is different in that the equivalent of the *thm* datatype is dependent on the proposition that the theorem shows. Essentially, where such manipulation is done in an untyped manner following the usual LCF approach, it is done in a strongly typed way in VeriML. We believe that this leads to a more principled and modular programming paradigm.

In recent years many languages with rich dependent type systems [11, 33, 73, 79] have been proposed. Our approach differs from these languages in that we are not primarily interested in certifying properties of code written in VeriML itself. We rather view VeriML as a foundational "meta" language for an "extensible" proof assistant, where proofs about code written in other (however richly typed or untyped) languages can be developed in a scalable manner.

5 Summary Chart



References

[1] O. Acıiçmez. Yet another microarchitectural attack: Exploiting I-cache. In *Conference on Computer and Communications Security*, Nov. 2007.

[2] O. Acıiçmez, Çetin Kaya Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. In *CT-RSA*, Feb. 2007.

[3] A. W. Appel. Foundational proof-carrying code. In *Proc. 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[4] A. Aviram and B. Ford. Determinating timing channels in statistically multiplexed clouds, Mar. 2010. `http://arxiv.org/abs/1003.5303`.

[5] A. Aviram and B. Ford. Deterministic consistency: A programming model for shared memory parallelism, Feb. 2010. `http://arxiv.org/abs/0912.0926`.

[6] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism, May 2010. `http://arxiv.org/abs/1005.3450`.

[7] M. Barnett, B.-Y. E. Chang, R. DeLine, B. J. 0002, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Symosium on Formal Methods for Components and Objects (FMCO'05)*, pages 364–387, 2005.

[8] B. N. Bershad et al. Extensibility, safety and performance in the SPIN operating system. In *15th ACM Symposium on Operating System Principles*, 1995.

[9] D. Brumley and D. Boneh. Remote timing attacks are practical. In *12th USENIX Security Symposium*, Aug. 2003.

[10] H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *Proc. 2007 ACM Conference on Programming Language Design and Implementation*, pages 66–77, 2007.

[11] C. Chen and H. Xi. Combining programming with theorem proving. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, page 77. ACM, 2005.

[12] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *17th ACM Symposium on Operating System Principles*, pages 140–153, Dec. 1999.

[13] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, pages 337–340, 2008.

[14] D. Delahaye. A tactic language for the system Coq. *Lecture notes in computer science*, pages 85–95, 2000.

[15] D. Delahaye. A proof dedicated meta-language. *Electronic Notes in Theoretical Computer Science*, 70(2):96–109, 2002.

[16] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. 20th Int'l Symp. on Distributed Computing (DISC'06)*, pages 194–208, 2006.

[17] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the Web. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2008.

[18] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *5th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[19] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay for multiprocessor virtual machines. In *Virtual Execution Environments (VEE)*, Mar. 2008.

[20] P. Efstathopoulos et al. Labels and event processes in the Asbestos operating system. In *20th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.

[21] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *15th ACM Symposium on Operating Systems Principles (SOSP)*, Dec. 1995.

[22] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity os. In *Proceedings of the 2006 EuroSys Conference*, pages 177–190, 2006.

[23] X. Feng. Local rely-guarantee reasoning. In *Proc. 36th ACM Symposium on Principles of Programming Languages*, page (to appear), 2009.

[24] X. Feng. *An Open Framework for Certified System Software*. PhD thesis, Department of Computer Science, Yale University, December 2007.

[25] X. Feng, R. Ferreira, and Z. 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, Apr. 2007.

[26] X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *Proc. 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 67–78, Jan. 2007.

[27] X. Feng and Z. Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *Proc. 2005 ACM SIGPLAN International Conference on Functional Programming*, pages 254–267. ACM Press, September 2005.

[28] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proc. 2008 ACM Conference on Programming Language Design and Implementation*, pages 170–182, 2008. `flint.cs.yale.edu/publications/aimjar.html`.

[29] X. Feng, Z. Shao, Y. Guo, and Y. Dong. Combining domain-specific and foundational logics to verify complete software systems. In *Proc. 2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08)*, volume 5295 of *LNCS*, pages 54–69. Springer-Verlag, October 2008.

[30] X. Feng, Z. Shao, Y. Guo, and Y. Dong. Certifying low-level programs with hardware interrupts and preemptive threads. *J. Autom. Reasoning*, 42(2-4):301–347, 2009.

[31] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *Proc. 2006 ACM Conference on Programming Language Design and Implementation*, pages 401–414, June 2006.

[32] R. Ferreira, X. Feng, and Z. Shao. Parameterized memory models and concurrent separation logic. In *Proceedings of the 19th European Symposium on Programming Languages and Systems*, pages 267–286, 2010.

[33] S. Fogarty, E. Pasalic, J. Siek, and W. Taha. Concoqtion: indexed types now! In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 112–121. ACM New York, NY, USA, 2007.

[34] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *International Conference on Functional Programming (ICFP)*, Oct 2002.

[35] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *31st ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 2004.

[36] B. Ford. VXA: A virtual architecture for durable compressed archives. In *4th USENIX Conference on File and Storage Technologies (FAST)*, Dec. 2005.

[37] B. Ford. Structured streams: a new transport abstraction. In *ACM SIGCOMM*, Aug. 2007.

[38] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *16th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 1997.

[39] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conference*, June 2008.

[40] B. Ford, M. Hibler, J. Lepreau, R. McGrath, , and P. Tullmann. Interface and execution models in the Fluke kernel. In *3rd Symposium on Operating Systems Design and Implementation (OSDI)*, Feb. 1999.

[41] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–151, 1996.

[42] B. Ford and J. Lepreau. Evolving mach 3.0 to a migrating thread model. In *Winter 1994 USENIX Conference*, pages 97–114, Jan. 1994.

[43] B. Ford and S. R. Susarla. CPU inheritance scheduling. In *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, Oct. 1996.

[44] M. Fu, Y. Li, X. Feng, Z. Shao, and Y. 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`.

[45] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *10th Network and Distributed System Security Symposium*, Feb. 2003.

[46] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An extensibility system for commodity operating systems. In *USENIX Annual Technical Conference*, June 1998.

[47] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF: a mechanized logic of computation. *Springer-Verlag Berlin*, 10:11–25, 1979.

[48] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *21st ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[49] N. A. Hamid and Z. Shao. Interfacing hoare logic and type systems for foundational proof-carrying code. In *Proc. 17th International Conference on Theorem Proving in Higher Order Logics*, volume 3223 of *LNCS*, pages 118–135. Springer-Verlag, Sept. 2004.

[50] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. Seventeenth Annual IEEE Symposium on Logic In Computer Science (LICS'02)*, July 2002.

[51] C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. In *Proc. 36th ACM Symposium on Principles of Programming Languages*, pages 441–453, 2009.

[52] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[53] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th Annual Int'l Symp. on Computer Architecture (ISCA)*, pages 289–300, 1993.

[54] G. Huet, C. Paulin-Mohring, et al. The Coq proof assistant reference manual. The Coq release v6.3.1, May 2000.

[55] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *Operating Systems Review*, 41(2):37–49, 2007.

[56] Intel Corporation. Intel® 64 and IA-32 architectures software developer's manual, Mar. 2009.

[57] R. A. Kemmerer. Shared resource matrix methodology: An approach to identifying storage and timing channels. *Transactions on Computer Systems*, 1(3):256–277, Aug. 1983.

[58] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. 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.

[59] C. League, Z. Shao, and V. Trifonov. Type-preserving compilation of Featherweight Java. *ACM Transactions on Programming Languages and Systems*, 24(2):112–152, March 2002.

[60] C. League, Z. Shao, and V. Trifonov. Precision in practice: A type-preserving Java compiler. In *International Conference on Compiler Construction*, Apr. 2003.

[61] G. T. Leavens. Tutorial on jml, the java modeling language. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, page 573, 2007.

[62] X. Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Proceedings of the 33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, Jan. 2006.

[63] Y. Levanoni and E. Petrank. An on-the-fly reference-counting garbage collector for java. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, 2006.

[64] J. Liedtke. A persistent system in real use: experiences of the first 13 years. In *International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, 1993.

[65] J. Liedtke. On micro-kernel construction. In *15th ACM Symposium on Operating System Principles*, 1995.

[66] C. Lin, A. McCreight, Z. Shao, Y. Chen, and Y. Guo. Foundational typed assembly language with certified garbage collection. In *Proc. 1st IEEE and IFIP International Symposium on Theoretical Aspects of Software Engineering*, pages 326–335, 2007.

[67] A. McCreight, T. Chevalier, and A. Tolmach. A certified framework for compiling and executing garbage-collected languages. In *Proc. 2010 ACM SIGPLAN International Conference on Functional Programming*, page (to appear). ACM Press, September 2010.

[68] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *Proc. 2007 ACM Conference on Programming Language Design and Implementation*, pages 468–479, 2007.

[69] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[70] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Symposium on Operating System Principles*, pages 39–51, Nov. 1987.

[71] S. Monnier, B. Saha, and Z. Shao. Principled scavenging. In *Proc. 2001 ACM Conference on Programming Language Design and Implementation*, pages 81–91, 2001.

[72] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symposium on Principles of Programming Languages*, pages 228–241, Jan. 1999.

[73] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In *Proc. 2006 ACM SIGPLAN International Conference on Functional Programming*, pages 62–73, Sept. 2006.

[74] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proc. 2nd USENIX Symposium on Operating System Design and Impl.*, pages 229–243, 1996.

[75] G. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proc. 1998 ACM Conference on Programming Language Design and Implementation*, pages 333–344, 1998.

[76] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–243, 1996.

[77] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. 33rd Symp. on Principles of Prog. Lang.*, pages 320–333, Jan. 2006.

[78] Z. Ni, D. Yu, and Z. Shao. Using xcap to certify realistic system code: Machine context management. In *Proc. 20th International Conference on Theorem Proving in Higher Order Logics*, volume 4732 of *LNCS*, pages 189–206. Springer-Verlag, Sept. 2007.

[79] U. Norell. Towards a practical programming language based on dependent type theory. Technical report, Goteborg University, 2007.

[80] P. W. O'Hearn. Resources, concurrency and local reasoning. In *Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 49–67, 2004.

[81] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[82] C. Percival. Cache missing for fun and profit. In *BSDCan*, May 2005.

[83] F. Pizlo, E. Petrank, and B. Steensgaard. A study of concurrent real-time garbage collectors. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 33–44, 2008.

[84] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *16th ACM Conference on Computer and Communications Security (CCS)*, pages 199–212, New York, NY, USA, 2009. ACM.

[85] Z. Shao. An overview of the FLINT/ML compiler. In *Proc. ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[86] Z. Shao. Certified software. *Communications of the ACM*, page (to appear), 2010. A preprint is available at `http://flint.cs.yale.edu/publications/certsoft.html`.

[87] Z. Shao, C. League, and S. Monnier. Implementing typed intermediate languages. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 313–323, 1998.

[88] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Proc. 29th ACM Symposium on Principles of Programming Languages*, pages 217–232, Jan. 2002.

[89] Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou. A type system for certified binaries. *ACM Transactions on Programming Languages and Systems*, 27(1):1–45, January 2005.

[90] C. Small and M. Seltzer. A comparison of OS extension technologies. In *USENIX Annual Technical Conference*, Jan. 1996.

[91] C. Small and M. Seltzer. MiSFIT: Constructing safe extensible systems. *IEEE Concurrency*, 6(3):34–41, 1998.

[92] A. Stampoulis and Z. Shao. VeriML:typed computation of logical terms inside a language with effects. In *Proc. 2010 ACM SIGPLAN International Conference on Functional Programming*, page (to appear), 2010. `flint.cs.yale.edu/publications/veriml.html`.

[93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, Dec. 1993.

[94] Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *22nd Annual Computer Security Applications Conference (ACSAC)*, Dec. 2006.

[95] J. C. Wray. An analysis of covert timing channels. In *IEEE Symposium on Security and Privacy*, May 1991.

[96] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proc. 2010 ACM Conference on Programming Language Design and Implementation*, pages 99–110, 2010.

[97] B. Yee et al. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, May 2009.

[98] D. Yu, N. A. Hamid, and Z. Shao. Building certified libaries for PCC: Dynamic storage allocation. In *Proc. 2003 European Symposium on Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 363–379. Springer-Verlag, Apr. 2003.

[99] D. Yu and Z. Shao. Verification of safety properties for concurrent assembly code. In *Proc. 2004 International Conference on Functional Programming (ICFP'04)*, September 2004.

[100] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2006.

[101] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *Proceedings 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 225–240, 2008.