Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System

Jean Yang
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory

Chris Hawblitzel
Microsoft Research

Abstract

Typed assembly language (TAL) and Hoare logic can verify the absence of many kinds of errors in low-level code. We use TAL and Hoare logic to achieve highly automated, static verification of the safety of a new operating system called Verve. Our techniques and tools mechanically verify the safety of every assembly language instruction in the operating system, run-time system, drivers, and applications (in fact, every part of the system software except the boot loader). Verve consists of a "Nucleus" that provides primitive access to hardware and memory, a kernel that builds services on top of the Nucleus, and applications that run on top of the kernel. The Nucleus, written in verified assembly language, implements allocation, garbage collection, multiple stacks, interrupt handling, and device access. The kernel, written in C# and compiled to TAL, builds higher-level services, such as preemptive threads, on top of the Nucleus. A TAL checker verifies the safety of the kernel and applications. A Hoare-style verifier with an automated theorem prover verifies both the safety and correctness of the Nucleus. Verve is, to the best of our knowledge, the first operating system mechanically verified to guarantee both type and memory safety. More generally, Verve's approach demonstrates a practical way to mix high-level typed code with low-level untyped code in a verifiably safe manner.

Categories and Subject Descriptors D.2.4 [*SOFTWARE ENGINEERING*]: Software/Program Verification

General Terms Verification

Keywords Operating system, run-time system, verification, type safety

1. Introduction

High-level computer applications build on services provided by lower-level software layers, such as operating systems and language run-time systems. These lower-level software layers should be reliable and secure. Without reliability, users endure frustration and potential data loss when the system software crashes. Without security, users are vulnerable to attacks from the network, which often exploit low-level bugs such as buffer overflows to take over a user's computer. Unfortunately, today's low-level software still suffers from a steady stream of bugs, often leaving computers vulnerable to attack until the bugs are patched.

Many projects have proposed using safe languages to increase the reliability and security of low-level systems. Safe languages ensure type safety and memory safety: accesses to data are guaranteed to be well-typed and guaranteed not to overflow memory boundaries or dereference dangling pointers. This safety rules out many common bugs, such as buffer overflow vulnerabilities. Unfortunately, even if a language is safe, implementations of the language's underlying run-time system might have bugs that undermine the safety. For example, such bugs have left web browsers open to attack.

This paper presents Verve, an operating system and run-time system that we have verified to ensure type and memory safety. Verve has a simple mantra: every assembly language instruction in the software stack must be mechanically verified for safety. This includes every instruction of every piece of software except the boot loader: applications, device drivers, thread scheduler, interrupt handler, allocator, garbage collector, etc.

The goal of formally verifying low-level OS and run-time system code is not new. Nevertheless, very little mechanically verified low-level OS and run-time system code exists, and that code still requires man-years of effort to verify [9, 14]. This paper argues that recent programming language and theorem-proving technologies reduce this effort substantially, making it practical to verify strong properties throughout a complex system. The key idea is to split a traditional OS kernel into two layers: a critical low-level "Nucleus," which exports essential runtime abstractions of the underlying hardware and memory, and a higher-level kernel, which provides more fully-fledged services. Because of these two distinct layers, we can leverage two distinct automated technologies to verify Verve: TAL (typed assembly language [18]) and automated SMT (satisfiability modulo theories) theorem provers. Specifically, we verify the Nucleus using automated theorem proving (based on Hoare Logic) and we ensure the safety of the kernel using TAL (generated from C#). Note that this two-layer approach is not specific to just Verve, but should apply more generally to systems that want to mix lower-level untyped code with higher-level typed code in a verifiably safe way.

A complete Verve system consists of a Nucleus, a kernel, and one or more applications. We wrote the kernel and applications in safe C#, which is automatically compiled to TAL. An existing TAL checker [6] verifies this TAL (again, automatically). We wrote the Nucleus directly in assembly language, hand-annotating it with assertions (preconditions, postconditions, and loop invariants). An existing Hoare-style program verifier called Boogie [2] verifies the assembly language against a specification of safety and correctness. This ensures the safety and correctness of the Nucleus's implemen-

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.

tation, including safe interaction with the TAL code and safe interaction with hardware (including memory, interrupts, timer, keyboard, and screen). Boogie relies on Z3 [7], an automated SMT theorem prover, to check that the assertions are satisfied. Writing the assertions requires human effort, but once they are written, Boogie and Z3 verify them completely automatically. As a result, the Verve Nucleus requires only 2-3 lines of proof annotation per executable statement, an order of magnitude less than similar projects based on interactive theorem provers [9, 14].

The Verve operating system demonstrates the following:

- It is, to the best of our knowledge, the first operating system mechanically verified to ensure type safety. Furthermore, every assembly language instruction that runs after booting is statically verified (we do not have to trust a high-level-language compiler, nor do we have to trust any unverified library code).
- It is a real system: it boots and runs on real, off-the-shelf x86 hardware, and supports realistic language features, including classes, virtual methods, arrays, and preemptive threads.
- It is efficient: it supports efficient TAL code generated by an optimizing C#-to-TAL compiler, Bartok [6], using Bartok's native layouts for objects, method tables, and arrays. It incorporates the code from earlier verified garbage collectors [13], which, as shown in [13], can run realistic macro-benchmarks at near the performance of Bartok's native garbage collectors.
- It demonstrates that *automated* techniques (TAL and automated theorem proving) are powerful enough to verify the safety of the complex, low-level code that makes up an operating system and run-time system. Furthermore, it demonstrates that a small amount of code verified with automated theorem proving can support an arbitrary large amount of TAL code.

In its current implementation, Verve is a small system and has many limitations. It lacks support for many C# features: exception handling, for example, is implemented by killing a thread entirely, rather than with try/catch. It lacks the standard .NET class library, since the library's implementation currently contains much unsafe code. It lacks dynamic loading of code. It runs only on a single processor. Although it protects applications from each other using type safety, it lacks a more comprehensive isolation mechanism between applications, such as Java Isolates, C# AppDomains, or Singularity SIPs [8]. The verification does not guarantee termination. Finally, Verve uses verified garbage collectors [13] that are stop-the-world rather than incremental or real-time, and Verve keeps interrupts disabled throughout the collection.

Except for multi-processor support, none of the limitations in Verve's present implementation are fundamental. We expect that with more time, the high degree of automation in Verve's verification will allow Verve to scale to a more realistic feature set, such as a large library of safe code and a verified incremental garbage collector.

1.1 Availability

All of the Verve source code is freely available for download or browsing at the following URL (browse the latest version under "Source Code" to see the "verify" directory, which contains Verve):

`http://www.codeplex.com/singularity`

2. Tools for constructing a safe OS

Two complementary verification technologies, TAL and automated theorem proving, drive Verve's design. On one hand, TAL is relatively easy to generate, since the compiler automatically turns C# code into TAL code, relying only on lightweight type annotations

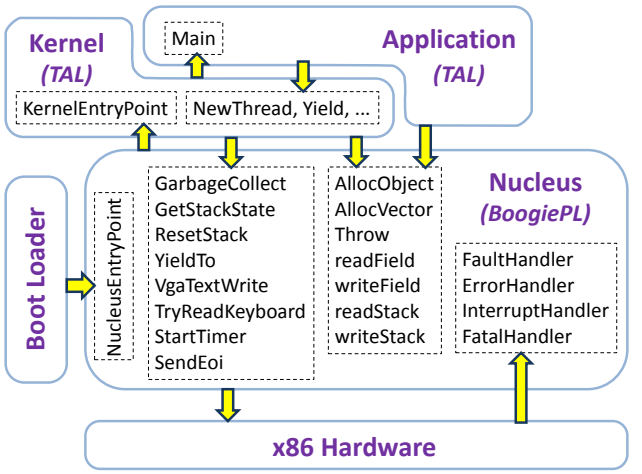

Figure 1. ***Verve structure, showing all 20 functions exported by the Nucleus***

already present in the C# code. This enables TAL to scale easily to large amounts of code. For Verve, we use the Bartok compiler [6] to generate TAL code.

On the other hand, automated theorem provers can verify deeper logical properties about the code than a typical TAL type system can express. Leveraging this power requires more effort, though, in the form of heavyweight programmer-supplied preconditions, postconditions, and loop invariants. To exploit the tradeoff between TAL and automated theorem proving, we decided to split the Verve operating system code into two parts, shown in Figure 1: a Nucleus, verified with automated theorem proving, and a kernel, verified with TAL. The difficulty of theorem proving motivated the balance between the two parts: only the functionality that TAL could not verify as safe went into the Nucleus; all other code went into the kernel.

The Nucleus's source code is not expressed in TAL, but rather in Boogie's programming language, called BoogiePL (or just Boogie), so that the Boogie verifier can check it. Since the Nucleus code consists of assembly language instructions, these assembly language instructions must appear in a form that the Boogie verifier can understand. As described in detail below, we decided to encode assembly language instructions as sequences of calls to BoogiePL procedures (e.g. an "Add" procedure, a "Load" procedure, etc.), so that the Boogie verifier can check that each instruction's precondition is satisfied. After Boogie verification, a separate tool called "BoogieAsm", developed for an earlier project [13], extracts standard assembly language instructions from the BoogiePL code. A standard assembler then turns these instructions into an object file.

Rather than hand-code all of the Nucleus in assembly language, we wrote some of the less performance-critical parts in a high-level extension of BoogiePL that we call "Beat". Our (non-optimizing) Beat compiler transforms verifiable Beat expressions and statements into verifiable BoogiePL assembly language instructions.

In Figure 2 we show the trusted and untrusted components of our system. Besides the boot loader, the only trusted components are the tools used to verify, assemble, and link the verified Nucleus and kernel. Note that none of our compilers are part of our trusted computing base: we do not need to trust the compilers to ensure the correctness of the Nucleus and safety of the Verve system as a whole. As shown in Figure 2, the TAL checker and Boogie/Z3 verifier check that the output of the compilers conforms to the TAL type system and the Nucleus specification, so we just need to trust these checkers.

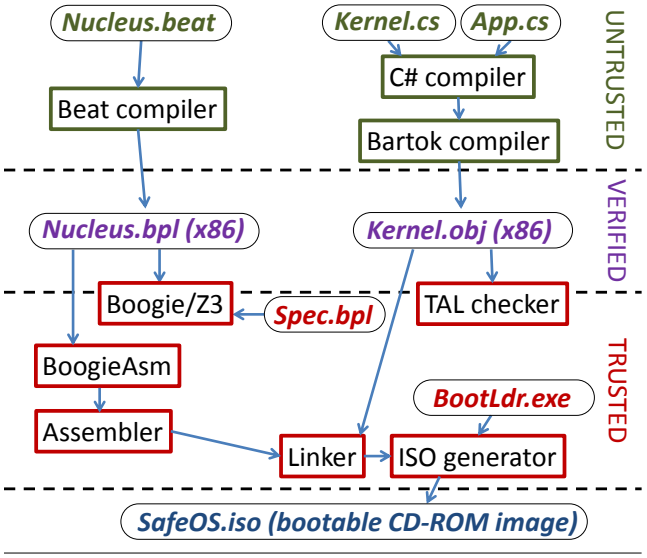

Figure 2. ***Building the Verve system: trusted, untrusted components***

Beyond the TAL checker and Boogie/Z3 verifiers, Figure 2 shows additional components in Verve's trusted computing base: the assembler, the linker, the ISO CD-ROM image generator, and the boot loader. In addition, the trusted computing base includes the specification of correctness for the Nucleus's BoogiePL code. This includes specifications of the behavior of functions exported by the Nucleus, shown in Figure 1. (For example, the specification of "YieldTo" ensures that the Nucleus sets the stack pointer to the top of the correct stack during a yield.) It also includes specifications for assembly language instructions and for interaction with hardware devices and memory; we took some of these specifications from existing work [13], and wrote some of them from scratch. All Boogie specifications are written as first-order logic formulas in the BoogiePL language.

By expressing and checking properties at a low level (assembly language), we can ensure non-trivial properties with high confidence. The bulk of this paper focuses on these properties, with an emphasis on the specification and verification of the Nucleus's correctness properties. The next section discusses the Nucleus's design, and subsequent sections discuss specification and verification.

3. The Nucleus interface

The core of our verification is the Nucleus, which provides a verified interface to the low-level functionality of the operating system. We verify the Nucleus using Hoare logic in Boogie, based on a trusted specification for x86 assembly language instructions. In Verve, all access to low-level functionality must occur through the Nucleus — the kernel's TAL code and application's TAL code can only access low-level functionality indirectly, through the Nucleus. For example, TAL code cannot directly access device registers. Furthermore, even though TAL code can directly read and write words of memory, it can only read and write words designated as safe-for-TAL by the Nucleus's garbage collector.

The Nucleus consists of a minimal set of functions necessary to support the TAL code that runs above it. We wanted a minimal set because even with an automated theorem prover, Hoare-style verification is still hard work; less code in the Nucleus means less code to verify. At the same time, the set has to guarantee safety in the presence of arbitrary TAL code; it can assume that the TAL code is well typed, but can make no further assumptions about the behavior of the TAL code. For example, when an interrupt occurs, the Nucleus attempts to transfer control to a designated TAL interrupt handler. The Nucleus cannot assume that this handler is in the correct state to handle the interrupt, and must therefore check the handler's state at run-time (see section 4.5).

One design decision greatly simplified the Nucleus: following a design used in recent micro-kernels [11, 14], no Nucleus function ever blocks. In other words, every Nucleus function performs a finite (and usually small) amount of work and then returns. The Nucleus may, however, return to a different thread than the thread that invoked the function. This allows the kernel built on top of the Nucleus to implement blocking thread operations, such as waiting on a semaphore.

These design decisions led us to a small Nucleus API consisting of just 20 functions, all shown in Figure 1. These 20 functions, implemented with a total of about 1500 x86 instructions, include 3 memory management functions (AllocObject, AllocVector, GarbageCollect), one very limited exception handling function (Throw), 3 stack management functions (GetStackState, ResetStack, YieldTo), and 4 device access functions (VgaTextWrite, TryReadKeyboard, StartTimer, SendEoi). Most of the functions are intended for use only by the kernel, not by applications. However, applications may call AllocObject and AllocVector directly.

The Nucleus exports four pseudo-functions, readField, writeField, readStack, and writeStack, to the kernel and applications. These functions, described further in section 4, contain no executable code, but their verification guarantees that the kernel and applications will find the values that they expect when they try to access fields or objects or slots on the current stack. The Nucleus exports another 4 functions to the hardware for handling faults and interrupts: FatalHandler halts the system, while FaultHandler, ErrorHandler, and InterruptHandler yield execution to kernel TAL code running on a designated interrupt-handling stack. Finally, the Nucleus exports an entry point, NucleusEntryPoint to the boot loader.

As more devices are added to the system, more Nucleus functions may be required. However, a minimal Nucleus only needs to include the portion of the device interfaces critical the rest of the system's safety; if desired, Verve could use an I/O MMU to protect the system from devices, minimizing the device code that needs to reside in the Nucleus.

Following the approach taken by the recent verified L4 microkernel, seL4 [14], Verve keeps interrupts disabled throughout the execution of any single Nucleus function. (On the other hand, interrupts may be enabled during the TAL kernel's execution, with no loss of safety.) Since Nucleus functions do not block, Verve still guarantees that eventually, interrupts will always be re-enabled, and usually will be re-enabled very quickly. However, Verve's current implementation sacrifices real-time interrupt handling because of one particularly long function: "GarbageCollect", which performs an entire stop-the-world garbage collection. This is currently a substantial limitation for the responsiveness of the system, but it is not fundamental to Verve's approach. Given a verified incremental collector, Verve could reduce the execution time of "GarbageCollect" to, say, just the time required to scan the stacks (or even a single stack), rather than the time required to garbage collect the whole heap. (Alternatively, Verve could poll for interrupts periodically, as in seL4. However, *delivering* these interrupts to the kernel TAL code would still require that the garbage collector reach a state that is safe for the kernel.)

The next two sections describes how Verve specifies, implements, and verifies the Nucleus functions listed above.

4. The Nucleus specification

To verify that the Nucleus behaves correctly, we have to specify what correct behavior is. Formally, this specification con-

sists of preconditions and postconditions for each of the 20 functions exported by the Nucleus (Figure 1). The preconditions reflect the guarantees made by other components of the system when calling the Nucleus. For example, the precondition to `NucleusEntryPoint` describes the state of memory when the Nucleus begins execution; the (trusted) boot loader is responsible for establishing this precondition. The preconditions for functions exported to the kernel and applications describe the state of registers and the current stack when making a call to the Nucleus; the (trusted) TAL checker is responsible for guaranteeing that these preconditions hold when the (untrusted) kernel and applications transfer control to the Nucleus. (Note that any property not guaranteed by the TAL checker cannot be assumed by the Nucleus's preconditions for functions exported to the kernel and applications; as a result, the Nucleus must occasionally perform run-time checks to validate the values passed from the kernel and applications.)

When writing the Nucleus's specification, we are not interested so much in the Nucleus's private, internal behaviors, but rather how the Nucleus interacts with other components of the system. For example, the Verve specification of garbage collection does not specify which algorithm the garbage collector should implement (e.g. copying or mark-sweep), since the algorithm is an internal behavior that is irrelevant to the other components of the system. Instead, following the approach of McCreight et al [17], the specification simply says that the garbage collector must maintain the well-formedness of the stacks and heap, so that subsequent reads and writes to the stack and heap other components of the system behave as expected. (Internally, the garbage collectors used by Verve [13] do have stronger invariants about specific details of the algorithm, but these invariants are kept hidden from the other Verve system components.)

The Nucleus interacts with five components: memory, hardware devices, the boot loader, interrupt handling, and TAL code (kernel and application code). Memory and hardware devices export functionality to the Nucleus, such as the ability to read memory locations and write hardware registers. Verve must verify that the Nucleus satisfies the preconditions to each operation on memory and hardware. In turn, the Nucleus exports functionality to the boot loader (the Nucleus entry point), the interrupt handling (the Nucleus's interrupt handlers), and the TAL code (AllocObject, YieldTo, etc.).

4.1 Specification logistics

Verve expresses the specification for interacting with these components as first-order logical formulas in BoogiePL [2]. These formulas follow C/Java/C# syntax and consist of:

- arithmetic operators: `+`, `-`, `*`, `>`, `==`, `!=`, ...
- boolean operators: `!`, `&&`, `||`, `==>`, ...
- variables: `foo`, `Bar`, `old(foo)`, ...
- boolean constants: `true`, `false`
- integer constants: `5`, ...
- bit vector contants: `5bv16`, `5bv32`, ...
- function application: `Factorial(5)`, `Max(3,7)`, `IsOdd(9)`, ...
- array indexing: `foo[3]`, ...
- array update: `foo[3 := Bar]`, ...
- quantified formulas: `(∀i:int::foo[i]==Factorial(i))`, ...

For a variable `foo`, the expression `old(foo)` refers to the value of `foo` at the beginning of the procedure. BoogieAsm also enforces the convention that capitalized variable names correspond with read-only variables. For instance, the variable corresponding to the instruction pointer `Eip` is read-only, while variable corresponding to the register `eax` is directly readable and writable.

BoogiePL bit vectors correspond to integers in C/Java/C#, which are limited to numbers that fit inside a fixed number of bits. BoogiePL integers, on the other hand, are unbounded mathematical integers. BoogiePL arrays are unbounded mathematical maps from some type (usually integers) to some other type. Unlike arrays in C/Java/C#, BoogiePL arrays are immutable values (there are no references to arrays and arrays are not updated in place). An array update expression `a[x := y]` creates a new array, which is equal to the old array `a` at all locations except `x`, where it contains a new value, `y`. For example, `(a[x := y])[x] == y` and `(a[x := y])[x + 1] == a[x + 1])`.

BoogiePL procedures have preconditions and postconditions, written as BoogiePL logical formulas:

```
var a:int, b:int;
procedure P(x:int, y:int)
  requires a < b && x < y;
  modifies a, b;
  ensures a < b && a == x + old(a);
{
  a := a + x;
  b := b + y;
}
```

In this example, the procedure `P` can only be called in a state where global variable `a` is less than global variable `b`, and the parameter `x` is less than the parameter `y`. Upon exit, the procedure's postconditions ensure that `a` is still less than `b`, and that `a` is equal to `x` plus the old version of `a` (before `P` executed). Note that the procedure must explicitly reveal all the global variables that it modifies ("`modifies a, b;`" in this example), so that callers to the procedure will be aware of the modifications.

To verify that a program's code obeys its specification, the Boogie tool relies on the Z3 [7] automated theorem prover. Z3 automatically checks logical formulas involving linear arithmetic (addition, subtraction, comparison), arrays, bit vectors, and functions. Z3 also checks quantified formulas (formulas with `forall` and `exists`); however, Z3 relies on programmer-supplied hints in the form of *triggers* to help with quantifiers, since checking quantified formulas with arithmetic and arrays is undecidable in general. Earlier work [13] describes how Verve's garbage collectors use triggers.

For each function exported to the boot loader, interrupt handling, and TAL code, the Nucleus implements a BoogiePL procedure whose specification is given in terms of "requires", "ensures", and "modifies" clauses. The Nucleus implements these procedures in terms of more primitive hardware procedures: each BoogiePL procedure exported from the memory and hardware devices to the Nucleus corresponds to exactly one assembly language instruction, such as an instruction to read a single memory location or write to a single hardware register.

In order to convey concretely what the Nucleus specification and verification entail, subsections 4.2-4.6 describe in detail the BoogiePL specifications of the Nucleus's interaction with the memory (4.2), hardware devices (4.3), boot loader (4.4), interrupt handling (4.5), and TAL code (4.6). Note that the interfaces in subsections 4.2-4.5 are for use only by the Nucleus, not by kernel and application code. Kernel and application code can only make use of these interfaces indirectly, via calls to the Nucleus.

4.2 Memory

Verve's initial memory layout is set by the boot loader. Verve uses the boot loader from the Singularity project [8], which sets up an initial virtual-memory address space (i.e. it sets up a page table), loads the executable image into memory, and jumps to the executable's entry point, passing detailed information about the mem-

ory layout to the entry point. The boot-loader-supplied address space simply maps virtual memory directly to physical memory, except for a small range of low addresses that are left unmapped (to catch null pointer dereferences). A traditional operating system would create new virtual memory address spaces to protect applications from each other. However, unlike traditional operating systems, Verve guarantees type safety, and can therefore rely on type safety for protection. Because of this, Verve simply keeps the initial boot-loader-supplied address space.

Verve's mapped address space consists of three parts. First is the memory occupied by the executable image, including code, static fields, method tables, and memory layout information for the garbage collector. Verve may read this memory, but may write only to the static fields, not the code, method tables, or layout information. Second, the Verve specification reserves the memory just above the executable image for the interrupt table. Verve may write to the table, but it can only write values that obey the specification for interrupt handlers. Third, the remaining memory above the interrupt handler is general-purpose memory, free for arbitrary use; the Nucleus may read and write it at any time as it wishes (as long as it ensures the well-formedness of the heap and current stack before transferring control to TAL code, as discussed in section 4.6).

The specification describes the state of general-purpose memory using a global variable `Mem`, which is an array that maps integer byte addresses to integer values. For any 4-byte-aligned address `i` in general-purpose memory, `Mem[i]` contains the 32-bit memory contents stored at address `i`, represented as an integer in the range $0 \ldots 2^{32} - 1$. The memory exports two operations to the Nucleus, `Load` and `Store`:

```
procedure Load(ptr:int) returns (val:int);
  requires memAddr(ptr);
  requires Aligned(ptr);
  modifies Eip;
  ensures word(val);
  ensures val == Mem[ptr];

procedure Store(ptr:int, val:int);
  requires memAddr(ptr);
  requires Aligned(ptr);
  requires word(val);
  modifies Eip, Mem;
  ensures Mem == old(Mem)[ptr := val];
```

Each of these two operations requires a 4-byte-aligned pointer ("`Aligned(...)`") to memory inside the general-purpose memory region ("`memAddr(...)`"). The loaded or stored value must be in the range $0 \ldots 2^{32} - 1$ ("`word(...)`"). Any `Store` operation updates the contents of `Mem`, so that subsequent `Load` operations are guaranteed to see the updated value. Loads and stores have an additional side effect, noted in the modifies clause: they modify the current instruction pointer (program counter), "`Eip`".

The executable image memory exports its own load/store interface to the Nucleus, but with a store operation that applies only to static fields.

Finally, the interrupt table exports a store operation that allows the Nucleus to write to the interrupt descriptor table (IDT). On x86 processors, the interrupt descriptor table contains a sequence of 8-byte entries that describe what code the processor should jump to when receiving various kinds of faults and interrupts. This is a very sensitive data structure, since an invalid entry could cause a jump to an arbitrary address in memory, which would be unsafe. We had originally hoped to use general-purpose memory for the interrupt table, and to guarantee the well-formedness of the interrupt table whenever the Nucleus transfers control to TAL code. Under this hope, the Nucleus would be allowed to arbitrarily modify the interrupt table temporarily, which would be safe while interrupts are disabled. However, some x86 platforms support "non-maskable interrupts", which can occur even with interrupts disabled. If such an interrupt ever occurred, we'd feel safer transferring control to a designated fatal error handler than allowing arbitrary behavior. Therefore, the interrupt table exports an interface that only allows stores of approved entries:

```
procedure IdtStore(entry:int, offset:int,
                   handler:int, ptr:int, val:int);
  requires 0 <= entry && entry < 256;
  requires (offset == 0 && val == IdtWord0(handler))
        || (offset == 4 && val == IdtWord4(handler));
  requires IsHandlerForEntry(entry, handler);
  requires ptr == idtLo + 8 * entry + offset;
  modifies Eip, IdtMem, IdtMemOk;
  ensures IdtMem == old(IdtMem)[ptr := val];
  ensures IdtMemOk == old(IdtMemOk)[ptr := true];
```

Like `Store`, `IdtStore` corresponds to a single x86 store instruction. Using `IdtStore`, the Nucleus may write to either 4-byte word of any 8-byte entry in the table, as long as the word describes a valid interrupt handler for the entry. After a valid word is written to address `ptr`, the specification updates an array of booleans "`IdtMemOk`" to reflect that the word at address `ptr` is now valid. Valid words obey a strange Intel specification that splits the handler's address across the two words in 16-bit pieces; we show the BoogiePL for this specification just to demonstrate that Verve can deal with such low-level architectural details safely (`and`, `or`, and `shl` are 32-bit bitwise AND/OR/SHIFT-LEFT):

```
function IdtWord0(handler:int) returns(int) {
  or(shl(CSS, 16), and(handler, 0x0000ffff))
}
function IdtWord4(handler:int) returns(int) {
  or(and(handler, 0xffff0000), 0x8e00)
}
```

4.3 Hardware devices

Verve currently supports four hardware devices: a programmable interrupt controller (PIC), a programmable interval timer (PIT), a VGA text screen, and a keyboard. Verve specifies the interaction with this hardware using unbounded streams of events. The Nucleus delivers events to the PIC, PIT, and screen, and it receives events from the keyboard. For the screen, the events are commands to draw a character at a particular position on the screen. For the keyboard, events are keystrokes received from the keyboard. For the PIC and PIT, particular sequences of events initialize interrupt handling and start timers.

We present the keyboard specification as an example. Verve represents the stream of events from the keyboard as an immutable array `KbdEvents` mapping event sequence numbers (represented as integers, starting from 0) to events (also represented as integers). As the Nucleus queries the keyboard, it discovers more and more events from the stream. Two indices into the array, `KbdAvailable` and `KbdDone`, indicate the state of the Nucleus's interaction with the keyboard. Events 0...`KbdDone-1` have already been read by the Nucleus, while events `KbdDone`...`KbdAvailable-1` are available to read but have not yet been read.

Two operations, `KbdStatusIn8` and `KbdDataIn8`, query the keyboard. Each of these procedures represents a single 8-bit x86 assembly language I/O instruction, and BoogieAsm translates each call to these procedures into a single x86 "in" instruction. By invoking `KbdStatusIn8`, the Nucleus discovers the current state of

KbdAvailable and KbdDone. If this operation places a 0 in the eax register's lowest bit, then no events are available; if the operation places a 1 in eax's lowest bit, then at least one event is available. If the Nucleus can prove that at least one event is available, it may call KbdDataIn8 to receive the first available event.

```
var KbdEvents:[int]int;
var KbdAvailable:int, KbdDone:int;
procedure KbdStatusIn8();
  modifies Eip, eax, KbdAvailable;
  ensures and(eax,1)==0 ==> KbdAvailable==KbdDone;
  ensures and(eax,1)!=0 ==> KbdAvailable> KbdDone;
procedure KbdDataIn8();
  requires KbdAvailable > KbdDone;
  modifies Eip, eax, KbdDone;
  ensures KbdDone == old(KbdDone) + 1;
  ensures and(eax,255) == KbdEvents[old(KbdDone)];
```

For example, the Nucleus implementation of the TryReadKeyboard function first calls KeyboardStatusIn8, and then performs a bitwise AND operation to discover the status:

```
implementation TryReadKeyboard() {
  call KeyboardStatusIn8();
  call eax := And(eax, 1);
...
```

4.4 Boot loader and Nucleus initialization

Interaction with the boot loader, interrupt handling, and TAL code is specified by preconditions and postconditions on Nucleus-implemented procedures. The first such procedure to execute is the Nucleus entry point, NucleusEntryPoint. When booting completes, the boot loader transfers control to NucleusEntryPoint. (After this control transfer, no boot loader code ever runs again.) The Nucleus entry point implementation must obey the following specification (for brevity, we omit some of the requires, modifies, and ensures clauses):

```
procedure NucleusEntryPoint(...);
  requires...idtLo == ro32(ro32(ecx+40)+72+0)
          && memHi == ro32(ro32(ecx+40)+72+8)+idtLo
          ...
  requires RET == ReturnToAddr(KernelEntryPoint);
  requires S == 0;
  ...
  ensures esp == StackHi(S) - 4 && ebp == 0;
  ensures StackCheckInv(S, StackCheck);
  ensures IdtOk && PicOk(...) && TimerOk(...);
  ensures NucleusInv(S,
              StackState[S:=StackRunning],...);
```

The boot loader passes information about available memory in the ecx register. This information supplies the Nucleus with the bounds of the memory above the executable image, which ranges from the low end of the interrupt table (idtLo) to the high end of general-purpose memory. (The function ro32 maps each address in read-only memory to the 32-bit value stored at that address.)

The "RET" value specifies how the procedure must return. It equals one of two values: ReturnToAddr(i), which specifies that the procedure must perform a normal return (the x86 ret instruction) to address i, or ReturnToInterrupted(i, cs, eflags), which specifies that the procedure must perform an interrupt return (the x86 iretd instruction) to return address i, restoring code segment cs and status flags eflags. The specification shown above requires that the Nucleus entry point return to the TAL kernel entry point. (Notice that Nucleus functions do not necessarily return to their direct caller; the Nucleus entry point returns to TAL, not to the boot loader.) Furthermore, the Nucleus entry point must set correct initial stack pointer (esp) and frame pointer (ebp) values. It must also set a global variable StackCheck with the address of the low end of the stack; TAL functions compare the stack pointer to StackCheck to check for stack overflow.

Several postconditions ensure that devices and interrupts are set up correctly: IdtOk guarantees that the Nucleus has completely filled in the interrupt table and set up the x86's pointer to the interrupt table, PicOk guarantees that the Nucleus has initialized the programmable interrupt controller, and TimerOk guarantees that the Nucleus has initialized the programmable interval timer.

One of the Nucleus's key roles is to manage multiple stacks, so that the TAL kernel can implement multiple threads. (To distinguish the Nucleus's functionality from the kernel's functionality, we say that the Nucleus implements "stacks" and the kernel implements "threads".) The specification uses two variables, S and StackState, to specify the current state of the Verve stacks. Stacks are numbered starting from 0, and S contains the current running stack. The Nucleus entry point specification S=0 indicates that the nucleus should run the TAL kernel entry point in stack 0. At any time, each stack s is in one of four states, specified by StackState[s]: empty, running, yielded, or interrupted. Initially, stack 0 is set to running (StackRunning), and all other stacks are empty. When TAL code is interrupted, its current stack's state changes to interrupted. When TAL code voluntarily yields, its current stack's state changes to yielded.

NucleusEntryPoint's final postcondition sets up the Nucleus's private invariant, NucleusInv. This invariant is a Nucleus-specified function that takes as arguments the Nucleus's private global variables, along with some specification variables like Mem, S, and StackState. The Nucleus may define NucleusInv any way it wants. If it defines too weak an invariant (e.g. NucleusInv(...) = true), though, then the Nucleus will be not have enough information to implement other Nucleus functions, such as allocation. On the other hand, if the invariant is too strong (e.g. NucleusInv(...) = false), the Nucleus entry point will not be able to ensure it in the first place. For successful verification, the Nucleus must define an invariant strong enough to ensure the well-formedness of the stacks and heaps, as well as the well-formedness of the Nucleus's own internal data structures.

4.5 Interrupt handling

After the TAL kernel begins execution in stack 0, it voluntarily yields control to another stack and enables interrupts. When a stack voluntarily yields, its state changes from StackRunning to StackYielded(_ebp, _esp, _eip), where _ebp, _esp, and _eip indicate the values that the stack pointer, frame pointer, and instruction pointer must be restored with in order to resume the stack's execution.

Upon receiving an interrupt or fault, the Nucleus's interrupt and fault handlers (InterruptHandler, FaultHandler, and ErrorHandler) transfer control back to the TAL code in stack 0, which then services the interrupt or fault. To specify this behavior, the handler procedure's preconditions and postconditions force the interrupt and fault handlers to restore stack 0's stack and frame pointers and return to stack 0's instruction pointer:

```
procedure InterruptHandler(...);
  requires NucleusInv(S, ...);
  requires (Tag(StackState[0])==STACK_YIELDED ==>
              RET == ReturnToAddr(eip)
           && StackState[0]==StackYielded(
                              _ebp, _esp, _eip));
```

```
  ...
  ensures (Tag(StackState[0])==STACK_YIELDED ==>
             NucleusInv(0, ...)
           && ebp == _ebp
           && esp == _esp);
```

The interrupted stack changes from state `StackRunning` to state:

```
StackInterrupted(_eax,_ebx,_ecx,_edx,_esi,_edi,
                 _ebp,_esp,_eip,_cs,_efl)
```

The values `_eax`...`_efl` indicate the x86 registers that must be restored to resume the interrupted stack's execution. (Verve does not yet handle floating point code, so no floating point state needs to be restored.)

Verve verifies the correctness of the Nucleus, but only verifies the safety of the kernel. As a result, a buggy TAL kernel might leave stack 0 in some state besides yielded. (For example, a buggy kernel might enable interrupts while running in stack 0, which could cause stack 0 to be in the running state when an interrupt occurs.) To ensure safety even in the presence of a buggy kernel, the Nucleus must check stack 0's state at run-time; if it not in the yielded state, the Nucleus halts the system. `InterruptHandler`'s precondition enforces this run-time check: it allows the stack to be in any state, but `RET` is only known in the case where the state is yielded, so that `InterruptHandler`'s implementation has to check that the state is yielded to use `RET` and return to the kernel.

4.6 TAL kernel and applications

The Nucleus exports various functions directly to TAL code. First, it exports three stack manipulation functions to the TAL kernel: `GetStackState`, `ResetStack`, and `YieldTo`. `GetStackState` simply returns the state of any stack. `ResetStack` changes the state of a stack to empty; the kernel may use this to terminate an unused thread. Finally, `YieldTo` transfers control to another stack. The kernel uses `YieldTo` to implement thread scheduling; for example, the kernel's timer interrupt handler calls `YieldTo` to preempt one thread and switch to another thread. The exact behavior of `YieldTo` depends on the state of the target stack that is being yielded to. If the target stack is in the yielded or interrupted state, `YieldTo` restores the target stack's state and resumes its execution. If the target stack is in the empty state, `YieldTo` runs the TAL kernel's entry point in the target stack; the kernel uses this to start new threads. If the target state is in the running state, then the stack is switching to itself; in this case, `YieldTo` simply returns. For brevity, we omit most of the specification for `YieldTo` here; it looks much like `InterruptHandler`'s specification from section 4.5, but with cases for all four stack states, rather than just for the yielded state:

```
procedure YieldTo(...);
  requires NucleusInv(S, ...);
  requires ScanStackInv(S, ..., esp, ebp);
  requires
    ( StackState[s]==StackRunning && s==S
      && RET==ReturnToAddr(Mem[esp]))
  || (StackState[s]==StackYielded(_ebp,_esp,_eip)
      && RET==ReturnToAddr(_eip))
  || (StackState[s]==StackInterrupted(_eax,...,_efl)
      && RET==ReturnToInterrupted(_eip,_cs,_efl))
  || (StackState[s]==StackEmpty
      && RET==ReturnToAddr(KernelEntryPoint)
      && ...);
...
```

`YieldTo` differs from `InterruptHandler` in one crucial respect. When TAL code voluntarily calls a Nucleus function, the TAL code leaves its stack contents in a state that the verified garbage collector can scan: the garbage collector follows the chain of saved frame pointers until it reaches a saved frame pointer equal to 0. For each frame, the collector uses the frame's return address as an index into tables of GC layout information. By contrast, an interrupted stack is not necessarily scannable. The requirement `ScanStackInv(...)` expresses the layout of the stack that allows scanning by the garbage collector, and the TAL checker enforces that the TAL code satisfies this requirement.

The Nucleus exports several run-time system functions to both TAL kernel and TAL application code: `AllocObject`, `AllocVector`, `GarbageCollect`, and `Throw`. Currently, Verve implements only trivial exception handling: when TAL tries to throw an exception, `Throw` simply terminates the current stack by setting its state to empty and transferring control to stack 0. Verve takes its allocation and garbage collection implementations (both mark-sweep and copying collection) directly from another project on verified garbage collection [13]; more information about them can be found there. Verve makes only minor changes to the allocation and GC implementation. First, the allocators return null when the heap is full, rather than invoking garbage collection directly. This allows the TAL kernel to schedule the remaining threads to prepare for garbage collection, as described in section 6. Second, the original verified collectors scanned only a single stack; Verve adds a loop that scans all the stacks.

Following earlier work by [17], the verified garbage collectors [13] export functions `readField` and `writeField` that read and write heap object fields on behalf of the TAL code. More precisely, these functions grant the TAL code permission to read and write fields of objects, by ensuring that the fields reside at valid memory locations and, for reads, that the fields contain the correct value. The "correct value" is defined by an abstract heap that the specification maintains [13, 17]. The key invariant in the garbage collector verification is that the concrete values stored in memory accurately reflect the abstract heap whenever the TAL code is running.

Verve extends the abstract heap with multiple abstract stacks, each consisting of zero or more abstract stack frames. Prior work [13, 17] used auxiliary variables to describe the contents of the abstract heap. In the same way, Verve uses auxiliary variables `FrameCount` and `FrameAbs` to describe the number of frames in each stack and the abstract contents of each word of each frame of each stack. The Nucleus exports functions `readStack` and `writeStack` that guarantee that the concrete stack contents in `Mem` match the abstract contents. As in prior work [13, 17], this matching is loose enough to allow the garbage collector to update concrete pointers as objects move in memory (since the copying collector moves objects). In the specification for `readStack`, the `InteriorValue` predicate matches a concrete value `val` at memory address `ptr` to the abstract value at offset `j` in the abstract frame `frame` of the current stack `S`:

```
procedure readStack(ptr:int, frame:int, j:int)
            returns(val:int);
  requires StackState[S] == StackRunning;
  requires NucleusInv(S,...);
  requires 0 <= frame < FrameCounts[S];
  requires ...
  ensures ...
  ensures val == Mem[ptr];
  ensures InteriorValue(val,...,
                        FrameAbs[S][frame][j],...);
```

(The name "InteriorValue" reflects the fact that a stack value might be an "interior" pointer to the inside of an object, which the

garbage collector must properly track [13].) Note that `readStack` does not modify the instruction pointer, `Eip`, because it contains no instructions (the same is true for `readField`, `writeField`, and `writeStack`). Because of this, the TAL code need not generate any code to call `readStack` at run-time. Instead, it simply reads the data at `Mem[ptr]` directly.

Finally, the Nucleus exports device management functions to the TAL kernel: `VgaTextWrite`, `TryReadKeyboard`, `StartTimer`, and `SendEoi` (send end-of-interrupt). The specification for `TryReadKeyboard`, for example, requires that the Nucleus return a keystroke (in the range 0-255) if one is available, and otherwise return the value 256:

```
procedure TryReadKeyboard();
  ...
  ensures KbdAvailable==old(KbdDone) ==> eax==256;
  ensures KbdAvailable> old(KbdDone) ==>
              eax==KbdEvents[old(KbdDone)];
```

5. The Nucleus implementation and verification

Verve follows earlier work [13] by using BoogiePL to express verified assembly language instructions, but improves on the earlier work by generating much of the verified assembly language automatically from higher-level source code. We illustrate the verified assembly language code with a small, but complete, example — the verified source code implementing `TryReadKeyboard`:

```
implementation TryReadKeyboard() {
  call KeyboardStatusIn8();
  call eax := And(eax, 1);
  call Go(); if (eax != 0) {goto skip;}
    call eax:=Mov(256);
    call Ret(old(RET)); return;
  skip:
  call KeyboardDataIn8();
  call eax := And(eax, 255);
  call Ret(old(RET)); return;
}
```

To verify this, we simply run the Boogie tool on the source code, which queries the Z3 theorem prover to check that the procedure satisfies its postconditions, and that all calls inside the procedure satisfy the necessary preconditions. Given the BoogiePL source code, this process is entirely automatic, requiring no scripts or human interactive assistance to guide the theorem prover.

Each statement in the verified BoogiePL code corresponds to 0, 1, or 2 assembly language instructions. ("If" statements require 2 instructions, a compare and branch. Dynamically checked arithmetic statements also require 2 instructions, an arithmetic operation followed by a jump-on-overflow.) BoogieAsm, a tool developed for earlier work [13], transforms this verified BoogiePL source code into valid assembly code:

```
_?TryReadKeyboard proc
    in al, 064h
    and eax, 1
    cmp eax, 0
    jne TryReadKeyboard$skip
    mov eax, 256
    ret
  TryReadKeyboard$skip:
    in al, 060h
    and eax, 255
    ret
```

BoogieAsm checks that the source code contains no circular definitions of constants or functions (which would cause the verification to be unsound), and no recursive definitions of procedures (which the translator currently cannot generate code for [13]). BoogieAsm also checks that the verified source code conforms to a restricted subset of the BoogiePL syntax. For example, before performing an "if" or "return" statement, the code must perform `call Go()` or `call Ret(old(RET))` operations to update the global variables that reflect the machine state:

```
procedure Go();
  modifies Eip;

procedure Ret(oldRET:ReturnTo);
  requires oldRET == ReturnToAddr(Mem[esp]);
  requires Aligned(esp);
  modifies Eip, esp;
  ensures esp == old(esp) + 4;
  ensures Aligned(esp);
```

5.1 Beat

To relieve some of the burden of writing detailed, annotated assembly and to clarify the code, we developed a small extension of BoogiePL called Beat, which we compile to BoogiePL. Beat provides some modest conveniences, such as defining named aliases for x86 assembly language registers, and very simple high-level-to-assembly-level compilation of statements and expressions. This aids readability, since named variables and structured control constructs ("if/else", "while") are easier to read than unstructured assembly language branches, without straying too far from the low-level assembly language model. For non-performance-critical code, we used Beat's high-level statements rather than writing assembly language instructions directly. As a very simple example of using Beat, here is an alternate implementation of `TryReadKeyboard`, rewritten to use Beat's structured "if/else" construct:

```
implementation TryReadKeyboard() {
  call KeyboardStatusIn8();
  call eax := And(eax, 1);
  if (eax == 0) {
    eax := 256;
  } else {
    call KeyboardDataIn8();
    call eax := And(eax, 255);
  }
  call Ret(old(RET)); return;
}
```

As shown in Figure 2, the Beat compiler generates (untrusted) BoogiePL assembly language code for the Nucleus from (untrusted) Beat code. The close correspondence between the Beat code and the generated BoogiePL assembly language code ensures that verifiable Beat compiles to verifiable BoogiePL assembly language.

5.2 The Nucleus Invariant

As mentioned in section 4.4, the Nucleus is free to choose its own internal invariant, `NucleusInv`, that describes the state of its internal data structures. Verve's definition of `NucleusInv` consists of two parts. The first part holds the garbage collector's internal invariants, as described in more detail in [13]. The second part holds the invariants about stacks. For example, the second part contains an invariant for each stack `s` in the yielded state:

```
Tag(StackState[s])==STACK_YIELDED ==>
...
  && Aligned(StackEsp(s,...))
  && StackState[s]==StackYielded(StackEbp(s,...),...)
  && ScanStackInv(s,...,StackEbp(s,...))
```

This invariant says that each yielded stack has an aligned stack pointer (`Aligned(...)`), contains the appropriate `ebp` register, `esp` register, and return address (`StackState[s]==...`), and is scannable by the garbage collector (`ScanStackInv(...)`). The Nucleus keeps an internal data structure that holds each stack's saved context, including the saved `esp` register, the saved `ebp` register, etc. This data structure occupies `TSize` bytes of memory per stack and starts at address `tLo`. The Nucleus defines the functions `StackEsp`, `StackEbp`, etc. to describe the layout of the saved context for each stack `s`:

```
function StackEsp(s,...){tMems[s][tLo+s*TSize+4]}
function StackEbp(s,...){tMems[s][tLo+s*TSize+8]}
function StackEax(s,...){tMems[s][tLo+s*TSize+12]}
...
```

In the definitions above, `tMems` is an auxiliary variable describing the subset of memory devoted to storing saved stack contexts; for this subset, `tMems[s][...]` is equal to `Mem[...]`. The Nucleus uses `Load` and `Store` operations to read and write the saved contexts. The `NucleusInv` invariant makes sure that each saved context contains correctly saved registers, so that the Nucleus's implementation of `InterruptHandler` and `YieldTo` can verifiably restore each saved context.

6. Kernel

On top of the Nucleus, Verve provides a simple kernel, written in C# and compiled to TAL. This kernel follows closely in the footsteps of other operating systems developed in safe languages [1, 4, 8, 20], so this section focuses on the interaction of the kernel with the Nucleus.

The kernel implements round-robin preemptive threading on top of Nucleus stacks, allowing threads to block on semaphores. The kernel scheduler maintains two queues: a ready queue of threads that are ready to run, and a collection queue of threads waiting for the next garbage collection. A running thread may voluntarily ask the kernel to yield. In this case, the thread goes to the back of the ready queue. The scheduler then selects another thread from the front of the ready queue and calls the Nucleus `YieldTo` function to transfer control to the newly running thread.

The kernel TAL code may execute the x86 disable-interrupt and enable-interrupt instructions whenever it wishes. While performing scheduling operations, the kernel keeps interrupts disabled. It enables interrupts before transferring control to application TAL code. Thus, a thread running application code may get interrupted by a timer interrupt. When this happens, the Nucleus transfers control back to the kernel TAL code running on stack 0. This code uses the Nucleus `GetStackState` function to discover that the previously running thread was interrupted. It then moves the interrupted thread to the back of the scheduler queue, and calls `YieldTo` to yield control to a thread from the front of the ready queue.

When an application asks the kernel to spawn a new thread, the kernel allocates an empty Nucleus stack, if available, and places the new thread in the ready queue. When the empty thread runs (via a scheduler call to `YieldTo`), it enters the kernel's entry point, `KernelEntryPoint`, which calls the application code. When the application code is finished running, it may return back to `KernelEntryPoint`, which marks the thread as "exited" and yields control back to the kernel's stack 0 code. The stack 0 code then calls `ResetThread` to mark the exited thread's stack as empty (and thus available for future threads to use). (Note that `KernelEntryPoint` is not allowed to return, since it sits in the bottom-most frame of its stack and has no return address to return to. The TAL checker verifies that `KernelEntryPoint` never tries to return, simply by assigning `KernelEntryPoint` a TAL type that lacks a return address to return to.)

Applications may allocate and share semaphore objects to coordinate execution among threads. The Verve keyboard driver, for example, consists of a thread that polls for keyboard events and signals a semaphore upon receiving an event. Each semaphore supports two operations, `Wait` and `Signal`. If a running thread requests to wait on a semaphore, the kernel scheduler moves the thread into the semaphore's private wait queue, so that the thread blocks waiting for someone to signal the thread. A subsequent request to signal the semaphore may release the thread from the wait queue into the ready queue.

The kernel's scheduler also coordinates garbage collection. We modified the Bartok compiler so that at each C# allocation site, the compiler generates TAL code to check the allocator's return value. If the value is null, the TAL code calls the kernel to block awaiting garbage collection, then jumps back to retry the allocation after the collection. The kernel maintains a collection queue of threads waiting for garbage collection. Following existing Bartok design [6] (and Singularity design [8]), before performing a collection Verve waits for each thread in the system to block on a semaphore or at an allocation site. This ensures that the collector is able to scan the stack of every thread. It does raise the possibility that one thread in an infinite loop could starve the other threads of garbage collections. If this is a concern, Bartok could be modified to poll for collection requests at backwards branches, at a small run-time cost and with larger GC tables. Alternatively, if Bartok were able to generate GC information for all program counter values in the program, we could add a precondition to `InterruptHandler` saying that an interrupted stack is scannable, so that the kernel need not wait for threads to block before calling the garbage collector.

7. Measurements

This section describes Verve's performance. We believe our measurements are, along with seL4, the first measurements showing efficient execution of realistic, verified kernels on real-world hardware. (Feng et al [9] do not present performance results. On an ARM processor, the seL4 implementation [14] reports just one micro-benchmark: a one-way IPC time of 224 cycles (448 cycles round-trip). Note that, as of the time of publication [14], this was the time for a "fast path" IPC that had not yet been fully verified. All the results we present for Verve are for fully verified code—that is, code that has been fully checked against a specification.)

We also describe the size of the Nucleus implementation (including annotations) and the verification time. The small size of the annotated Nucleus implementation and the relatively small amount of time for automated verification suggests that this is feasible as a general approach for developing verified low-level systems.

7.1 Performance

We wrote our micro-benchmarks in C#, compiled them to TAL, verified the TAL code, and linked them with the kernel and Nucleus. We then ran them on a 1.8 GHz AMD Athlon 64 3000+ with 1GB RAM, using the processor's cycle counters to measure time and averaging over multiple iterations, after warming caches. The benchmarks exercise the performance-critical interfaces exported by the Nucleus: stack management and memory management. All benchmarks were performed on two configurations — Verve built

with a copying collector, and Verve built with a mark-sweep collector:

	Copying (cycles)	MS (cycles)
2*YieldTo	98	98
2*Wait+2*Signal	216	216
Allocate 16-byte object	46	61
Allocate 1000-byte array	1289	1364
GC per 16-byte object (0MB live)	1	34
GC per 16-byte object (256MB live)	193	105

Our numbers compare favorably with round-trip inter-process communication times for even the fastest x86 micro-kernels, and compares very well with seL4's "fast path". The YieldTo benchmark shows that the Verve Nucleus requires 98 cycles to switch from one stack to another and back (49 cycles per invocation of YieldTo). The kernel builds thread scheduling and semaphores on top of the raw Nucleus YieldTo operation. Using semaphore wait and signal operations, it takes 216 cycles to switch from one thread to another and back (108 cycles per thread switch). This 216 cycles is actually considerably faster than the measured performance of the round-trip intra-process wait/signal time measured by Fahndrich et al [8], on the same hardware as our measurements were taken, for the Singularity operating system (2156 cycles), the Linux operating system (2390 cycles), and the Windows operating system (3554 cycles). In fairness, of course, Singularity, Linux, and Windows support far more features than Verve, and Verve might become slower as it grows to include more features. The wait/signal performance is comparable to the round-trip IPC performance of fast micro-kernels such as L4 (242 cycles on a 166 MHz Pentium [15]) and seL4 (448 cycles on an ARM processor [14]), although in fairness, IPC involves an address space switch as well as a thread switch.

We split the memory management measurements into allocation time (when no GC runs) and GC time. The allocation times show the time taken to allocate individual 16-byte objects and 1000-byte arrays in a configuration with very little fragmentation. (The mark-sweep times may become worse under heavy fragmentation.) The GC times show the time taken to perform a collection divided by the number of objects (all 16 bytes each) allocated in each collection cycle. One measurement shows the time with no live data (the collection clears out the heap entirely) and with 256MB of live data retained across each collection. (256MB is 50% of the maximum capacity of the copying collector and about 30% of the maximum capacity of the mark-sweep collector; as the live data approaches 100% of capacity, GC time per allocation approaches infinity.)

Because Verve does not support all C# features and libraries yet, we have not been able to port existing C# macro-benchmarks to Verve. The micro-benchmarks presented in the section, though, give us hope that Verve compares favorably with existing, unverified operating systems and run-time systems. Furthermore, the verified allocators and garbage collectors used by Verve have shown competitive performance on macro-benchmarks when compared to native garbage collectors [13]. The TAL code generated by Bartok has also shown competitive performance on macro-benchmarks [6].

7.2 Implementation and verification

We next present the size of various parts of the Nucleus specification and implementation. All measurements are lines of BoogiePL and/or Beat code, after removing blank lines and comment-only lines. The following table shows the size of various portions of the trusted specification:

Basic definitions	61
Memory and stacks	116
Interrupts and devices	111
x86 instructions	126
GC tables and layouts	317
Nucleus GC, allocation functions	239
Nucleus other functions	215
Total BoogiePL lines	**1185**

Overall, 1185 lines of BoogiePL is fairly large, but most of this is devoted to definitions about the hardware platform and memory layout. The GC table and layout information, originally defined by the Bartok compiler, occupies a substantial fraction of the specification. The specifications for all the functions exported by the Nucleus total 239 + 215 = 454 lines.

We measure the size of the Nucleus implementation for two configurations of Verve, one with the copying collector and one with the mark-sweep collector. (Note that the trusted specifications are the same for both collectors.) 1610 lines of BoogiePL are shared between the two configurations:

	Copying	Mark-Sweep
Shared BoogiePL lines	1610	1610
Private BoogiePL lines	2699	3243
Total BoogiePL lines	**4309**	**4854**
Specification BoogiePL lines	1185	1185
Total BoogiePL lines w/ spec	**5494**	**6039**
x86 instructions	1377	1489
BoogiePL/x86 ratio	**3.1**	**3.3**
BoogiePL+spec/x86 ratio	**4.0**	**4.1**

In total, each configuration contains about 4500 lines of BoogiePL. From these, BoogieAsm extracts about 1400 x86 instructions. (Note: the BoogieAsm tool can perform macro-inlining of assembly language blocks; we report the number of x86 instructions before inlining occurs.) This corresponds roughly to a 3-to-1 ratio (or 4-to-1 ratio, if the specification is included) of BoogiePL to x86 instructions (or, roughly, 2-to-1 or 3-to-1 ratio of non-executable annotation to executable code). This is about an order of magnitude fewer lines of annotation and script than related projects [9, 14].

Of course, some annotations are easier to write than others. In particular, annotations with quantifiers (`forall` and `exists`) require programmer-chosen triggers for the quantifiers. The 7552 lines of BoogiePL listed in the table above contain 389 quantifiers, each requiring a trigger. Fortunately, in practice, Verve's quantifiers tend use the same triggers over and over: across the 389 quantifiers, there were only 15 unique choices of triggers. Another issue is that Verve sometimes requires the BoogiePL code to contain explicit assertions of triggers, which serve as hints to the theorem prover to instantiate a quantified variable at some particular value (see [13] for details). The 7552 lines of BoogiePL listed above contain 528 such assertions — more than we'd like, but still a small fraction of the annotations.

Using Boogie/Z3 to verify all the Nucleus components, including both the mark-sweep and copying collectors, takes 272 seconds on a 2.4GHz Intel Core2 with 4GB of memory. The vast majority of this time is spent verifying the collectors; only 33 seconds were required to verify the system's other components.

This small verification time enormously aided the Nucleus design and implementation, because it gave us the freedom to experiment with different designs. Often we would finish implementing and verifying a feature using one design, only to become dissatisfied with the complexity of the interface or limitations in the design. In such cases, we were able to redesign and re-implement the feature in a matter of days rather than months, because we could make minor changes to large, Nucleus-wide invariants and then run the

automated theorem prover to quickly re-verify the entire Nucleus. In fact, some of our re-designs were dramatic: mid-way through the project, we switched from an implementation based on blocking Nucleus calls to an implementation based on non-blocking Nucleus calls. We also had to revise the garbage collector invariants to reflect the possibility of multiple stacks, which weren't present in the original verified GC implementations [13]. In the end, the Verve design, implementation, and verification described in this paper took just 9 person-months, spread between two people.

8. Related Work

The Verve project follows in a long line of operating system and run-time system verification efforts. More than 20 years ago, the Boyer-Moore mechanical theorem prover verified a small operating system and a small high-level language implementation [5]. These were not integrated into a single system, though, and each piece in isolation was quite limited. The operating system, Kit, was quite small, containing just a few hundred assembly language instructions. It supported a fixed number of preemptive threads, but did not implement dynamic memory allocation or thread allocation. It ran on an artificial machine rather than on standard hardware. The language, micro-Gypsy, was small enough that it did not require a significant run-time system, as it had no heap allocation or threads.

More recently, the seL4 project verified all of the C code for an entire microkernel [14]. The goals of seL4 and the Verve Nucleus are similar in some ways and different in others. Both work on uni-processors, providing preemption via interrupts. seL4 provides non-blocking system calls; this inspired us to move the Verve Nucleus to a non-blocking design as well. Both Verve and seL4 provide memory protection, but in different ways. Verve provides objects protected by type safety, while seL4 provides address spaces of pages protected by hardware page tables. Verve applications can communicate using shared objects and semaphores, while seL4 processes communicate through shared memory or message passing. seL4 verifies its C code, but its assembly language (600 lines of ARM assembler) is currently unverified. The seL4 microkernel contained 8700 lines of C code, substantially larger than earlier verified operating systems like Kit. On the other hand, the effort required was also large: they report 20 person-years of research devoted to developing their proofs, including 11 person-years specifically for the seL4 code base. The proof required 200,000 lines of Isabelle scripts — a 20-to-1 script-to-code ratio. We hope that while seL4 demonstrates that microkernels are within the reach of interactive theorem proving, Verve demonstrates that automated theorem proving can provide a less time-consuming alternative to interactive theorem proving for realistic systems software verification.

The seL4 kernel is bigger than Verve's Nucleus, so seL4's correctness proof verifies more than the Verve Nucleus's correctness proof (except for seL4's unverified assembly language instructions). On the other hand, seL4 is still a microkernel — most traditional OS features are implemented outside the microkernel as unverified C/C++ code, and this unverified code isn't necessarily type safe (although it is protected by virtual memory at a page granularity). By contrast, Verve's type safety is a system-wide guarantee. (Of course, type safety restricts which programs can run – seL4's use of page-level protection provides a way to run unsafe programs as well as safe programs.)

The FLINT project sets an ambitious goal to build foundational certified machine code, where certification produces a proof about an executable program, and a very small proof checker can verify the proof [9]. Such a system would have a much smaller trusted computing base than Verve. The FLINT project uses powerful higher-order logic to verify properties of low-level code, but achieves less automation than Verve: their preemptive thread library (which, unlike Verve, proves the partial correctness of thread implementation, not just safety) requires 35,000 lines of script for about 300 lines of assembly language (plus many tens of thousands of lines of script to build up foundational lemmas about the program logic [10]). Separately, the FLINT project also created a certified system combining TAL with certified garbage collection [16]. However, the TAL/GC system was less realistic than Verve: it supported only objects containing exactly two words, it performed only conservative collection, and did not support multiple threads. A combination of automated proof and foundational certification would be valuable, to get the certainty of foundational certified code with the scalability of automated verification.

Like the Verve Nucleus, the H monad [12] exports a set of operating system primitives usable to develop an operating system in a high-level language. H was used to build the House operating system [12]. However, H itself is not formally verified, and relies on a large Haskell run-time system for correctness.

9. Conclusions and Future Work

Using a combination of typed assembly language and automated theorem proving, we have completely verified the safety of Verve at the assembly language level, and completely verified the partial correctness of Verve's Nucleus at the assembly language level. Both Verve and its verification are efficient. So what happens when we boot and run Verve? Since it's verified, did it run perfectly (or at least safely?) every time we ran it? We give a short answer ("almost"), and a slightly longer answer composed of two anecdotes:

Anecdote 1: The Debugger. Initially, we admitted two pieces of unverified code into Verve: the boot loader and a debugger stub, both written for earlier projects in C and C++. Our rationale was that neither piece would run any instructions in a booted, deployed system: the boot loader permanently exits after ceding control to the Nucleus, and the debugger stub can be stripped from deployed systems. (In the future, we could develop a verifiably safe debugger stub, but using existing code was very appealing.) The debugger stub allows a remote debugger to connect to the kernel, so that we could examine the kernel memory and debug any inadvertent traps that might occur. Unfortunately, the debugger stub turned out to cause more bugs than it fixed: it required its own memory, its own interrupt handling, and its own thread context definition, and we implemented these things wrong as often as right. Debugging the debugger stub itself was hardly fun, so after a while, we decided to banish the debugger stub from Verve entirely. Developing a kernel without a kernel debugger requires a bit of faith in the verification process. After all, any bugs that somehow slipped through the verification process would be very painful to find with no debugger. But the decision paid off: we felt far more secure about Verve's invariants without the presence of unverified code to undermine them. And after dropping the debugger, we encountered only one bug that broke type safety or violated the Nucleus's correctness guarantees, described in the next anecdote.

Anecdote 2: The Linker. After dropping the debugger, we started to get used to the verified Nucleus code working the first time for each new feature we implemented. When we set up the first interrupt table, fault handling worked correctly the first time. When we programmed and enabled the timer, preemptive interrupts worked correctly the first time. In fact, after dropping the debugger stub, everything worked the first time, except when we first ran the garbage collector. Intriguingly, the garbage collector did run correctly the first time when we configured Bartok to generate assembly language code, which we assembled to an object file with the standard assembler. But the GC broke when we configured Bartok to generate the object file directly (even though we ran our TAL checker on the object file): it failed to scan the stack frames' return addresses

correctly. To our surprise, this bug was due to a linking problem: although the assembler attached relocation information to the entries in the GC's return address table when generating object files, the object file generated directly by Bartok lacked this relocation information (because Bartok has its own, special linking process for GC tables, which doesn't rely on the relocations). Thus, the tables' return addresses were incorrect after linking and loading, causing the GC to fail. Once we fixed this problem, the GC worked perfectly with both assembler-generated and Bartok-generated object files.

9.1 Future work

Verve is still a small system, and there are many directions for it to grow. The first candidate is an incremental or real-time garbage collector. Such a collector would require read and/or write barriers; prior work [17] has addressed this issue for verified garbage collection, but it remains to be seen whether this work can be implemented efficiently in a realistic system.

Another large area of research is multicore and multiprocessor machines. For Verve's TAL code, multiprocessing poses few problems; Bartok TAL is thread-safe. For the Nucleus, any shared data between processors makes verification much harder. The Barrelfish project [3] advocates one approach that would ease multiprocessor verification: minimize sharing by treating the multiprocessor as a distributed system of cores or processors communicating via message passing. Another approach would be to increase the granularity of the Nucleus interface; processors could allocate large chunks from a centralized Nucleus memory manager, and the centralized memory manager could use locks to minimize concurrency. At least one TAL type system, for example, can verify allocation from thread-local chunks [19].

A third area of research is to incorporate virtual memory management into the Nucleus and kernel (following, for example, seL4 [14]). In its full generality, virtual memory would complicate the semantics of `Load` and `Store`, and would thus be a non-trivial extension to the system. In addition, swapping garbage collectable memory to disk would bring the disk and disk driver into the Nucleus's trusted computing base. However, some applications of virtual memory hardware, such as sandboxing unsafe code and using IO/MMUs to protect against untrusted device interfaces, could be implemented without requiring all of virtual memory's generality.

A fourth area of research is to verify stronger properties of the TAL kernel. High-level languages like JML and Spec# could provide light-weight verification, weaker than the Nucleus verification, but beyond standard Java/C# type safety. This might allow us to prove, for example, that threads correctly enter and exit the runnable queue, and that each runnable thread is scheduled promptly.

Finally, we would like to improve Verve's TAL checking to be more foundational (as in FLINT) and to support dynamic loading. For dynamic loading, a key question is whether to run the TAL checker (currently written in C#) in the kernel or port the checker to run in the Nucleus. The former would be easier to implement, but the latter would keep the Nucleus independent from the kernel and would allow more foundational verification of the checker itself.

Acknowledgments

We would like to thank Jeremy Condit, Galen Hunt, Ed Nightingale, Don Porter, Shaz Qadeer, Rustan Leino, Juan Chen, and David Tarditi for their suggestions and assistance.

References

[1] G. Back, W. C. Hsieh, and J. Lepreau. Processes in kaffeos: isolation, resource management, and sharing in Java. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 23–23, Berkeley, CA, USA, 2000. USENIX Association.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO)*, volume 4111 of *Lecture Notes in Computer Science*, 2006.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *SOSP '09*, pages 29–44, 2009.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283, New York, NY, USA, 1995. ACM.

[5] W. R. Bevier, W. A. H. Jr., J. S. Moore, and W. D. Young. An approach to systems verification. *J. Autom. Reasoning*, 5(4):411–428, 1989.

[6] J. Chen, C. Hawblitzel, F. Perry, M. Emmi, J. Condit, D. Coetzee, and P. Pratikakis. Type-preserving compilation for large-scale optimizing object-oriented compilers. *SIGPLAN Not.*, 43(6):183–192, 2008. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1379022.1375604.

[7] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[8] 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 *EuroSys*, pages 177–190, 2006.

[9] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI*, pages 170–182, 2008.

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

[11] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and execution models in the Fluke kernel. In *OSDI*, pages 101–115, 1999.

[12] T. Hallgren, M. P. Jones, R. Leslie, and A. P. Tolmach. A principled approach to operating system construction in Haskell. In *ICFP*, pages 116–128, 2005.

[13] C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. In *POPL*, pages 441–453, 2009.

[14] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, USA, Oct. 2009. ACM.

[15] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. Achieved ipc performance (still the foundation for extensibility). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, Cape Cod, MA, May 5–6 1997. URL `http://l4ka.org/publications/`.

[16] C. Lin, A. McCreight, Z. Shao, Y. Chen, and Y. Guo. Foundational typed assembly language with certified garbage collection. *Theoretical Aspects of Software Engineering*, 2007.

[17] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *PLDI*, pages 468–479, 2007.

[18] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *POPL '98: 25th ACM Symposium on Principles of Programming Languages*, pages 85–97, Jan. 1998.

[19] L. Petersen, R. Harper, K. Crary, and F. Pfenning. A type theory for memory allocation and data layout. In *POPL*, pages 172–184, 2003.

[20] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell. Pilot: an operating system for a personal computer. *Commun. ACM*, 23(2):81–92, 1980.