Introduction

Conventional methods of software development attempt to find errors through software testing, but tests are not a certain guarantee of correctness. In contrast, in formal verification, the source code of a program is written alongside a mathematical proof of correctness. This proof shows that the source code conforms to a stated specification. With a “deep specification” that completely characterizes the desired behavior of a program, a proof of correctness can show that a piece of software is, in essence, “bug free.”

This project extended the functionality of virtual memory mapping in mCertiKOS, a formally verified operating system kernel. IOMMU (Input-output memory management unit) chips permit hardware translation of virtual and physical addresses for devices using direct memory mapping for input and output with support for virtualization. In particular, Intel’s VT-d chip is one such IOMMU featured in much recent hardware. In this project, selective features of the VT-d interface have been ported to mCertiKOS. Future work to finish porting the remainder of the VT-d interface, and subsequently to verify the correctness of this code, will permit mCertiKOS to program and use the VT-d chip to perform input and output on external devices.
Background

Formal verification

In formal verification, programs are accompanied by a proof of correctness. The software is written in a particular programming language (in this case, a subset of C). The semantics of that programming language are then formalized in a logical “metalanguage” (in this case, Coq); in addition, a specification for the program is written in the metalanguage. The metalanguage should be able to express the target language, as well as to manipulate assertions about the behavior of programs in that language. It is then possible to prove that the assertions given by the specification are, in fact, satisfied by the program.

Formal verification is far more labor-intensive than conventional software testing, but unlike testing, it provides a checkable guarantee of correctness. It is common in formal verification for proofs to occupy by more than an order of magnitude more lines of code than the original source code. As a result, formal verification is not practicable for entire large systems; rather it should be used to provide an absolute guarantee of correctness in situations where security is absolutely critical.

Coq

mCertiKOS is verified in the Coq programming language. Coq is a functional language that permits the user-defined syntax extensions and data structures. The features can easily be used to represent another programming language, and programs written in that other language. In addition, Coq provides a number of built-in constructs that allow the user to write and automatically verify proofs. Coq’s proof facilities provide a number of utilities for automating certain aspects of the proof-writing process, allowing verification of proofs that would be too tedious or long to write out by hand.

Clight/CompCert/Clightgen

The CompCert project has already formalized the semantics of the C programming language in Coq and produced a formally verified compiler for C. However, certain aspects of C pose difficulties in proof writing. In particular, side effects during expression evaluation substantially complicate reasoning about code. Clight is a restricted subset of
C programming language that excludes these aspects. The Clightgen utility uses CompCert to translate unverified C source code into Clight, and automatically produces Coq code that can be operated upon for verification.

**mCertiKOS/VeriKOS**

mCertiKOS is an operating system microkernel, meaning that it provides the minimal amount of functionality necessary to support an operating system. Most of the functionality of traditional monolithic kernels is run in the user-space, rather than within the kernel itself. mCertiKOS is only a few thousand lines of code, making it feasible to verify.

The functionality traditionally demanded of an operating system is instead provided through mCertiKOS’s support of virtualization; another full operating system, such as Linux, can be run as a guest OS, with mCertiKOS acting as a virtual machine hypervisor. In this fashion, unverified code might be run in virtual machines, to which mCertiKOS might provide a guarantee of certain security-critical properties, like isolation, between unverified systems.

Zhong Shao’s group has already produced an unverified microkernel, called VeriKOS, which contains the functionality desired in mCertiKOS, including IOMMU. This code must be ported from VeriKOS to mCertiKOS in a way that makes it possible to write modular specifications and proofs for it. Therefore, the task of this project includes refactoring that implementation and redesigning its interface; the new interface can then be applied to ported code in order to permit IOMMU to be verified and to run within mCertiKOS. (Much of this interface design was completed in Fall 2014 through an unrelated group’s term project for CS 428: Language-based Security.)

**IOMMU**

Devices often interact with computer software through memory mapping, in which programs can read from or write to the device as though the device were part of main memory. The device may have direct memory access (DMA) to a section of main memory, permitting it to read from or write to that section of memory without requiring it to go through the CPU.

While these protocols provide efficient methods of input and output for devices, they do not inherently provide support for virtual memory systems, nor for security restrictions on
the device’s access to restricted sections of memory. Such translation to virtual memory and checking of access restrictions can be done in software, but computation is expensive.

Instead, IOMMU provides this translation and security checking in hardware. Intel’s VT-d system permits the operating system to program the physical VT-d chip to permit it to configure particular memory sections called “domains.” These domains can be assigned to devices, and security is enforced by prohibiting devices from reading and writing outside of domains which have been assigned to them. It can then remap DMA onto these domains. In addition to remapping DMA, VT-d also permits interrupt requests to be remapped.[3]

The IOMMU chip is programmed by the operating system at startup with the information it needs to set up these domains, as well as with pointers to tables in main memory containing information on the devices attached to the computer. This setup code is supported by a variety of input and output protocols and data structures allowing the operating system to communicate with the chip through the ACPI and PCI interfaces and MSI data structures. These components, as well as the implementation of IOMMU are discussed further in the Layering section.

**Layering**

In order to permit formal verification of the source code, it must be cleanly structured in a way that allows the proof to be written in modular “layers.” The proof for the lowest layer may then be written first, and the proofs for higher layers built upon lower layers. Complicated, unlayered source code may be too difficult to verify, so the original VeriKOS source code has been refactored into explicit layers.

These layers are grouped and described individually below. Detailed information is also provided on the specific parts of these layers that are relevant to or accessed by VT-d.

**ACPI**

ACPI (advanced configuration and power interface) provides access to the system descriptor tables in hardware. These tables provide pointers to other system descriptor tables, which in turn contain information about the computer’s hardware for input and output. Of particular interest to IOMMU, ACPI contains pointers to tables that describe the computer’s units for hardware DMAR (direct memory access remapping). The particular,
physical hardware units of interest are DRHDs and RMRRs.

**DRHD:** DMA remapping hardware unit. These structures permit direct memory access requests from devices to be translated, and enable memory domains (see prior IOMMU subsection) to be assigned to each device.

**RMRR:** Reserved memory regions. Older devices and standards, such as USB, may not support DRHD and instead require reserved memory regions; each reserved memory region is a section of 4kB, or a multiple of 4kB, reserved exclusively for DMA for a particular device[4].

IOMMU uses the ACPI interface to scan for DRHDs and RMRRs, and stores pointers to the DRHDs and RMRRs that it finds in its own tables.

IOMMU also scans for two other types of units: ATSR (root port address translation service capability reporting) and RHSA (remapping hardware static affinity). However, these are not implemented in VeriKOS or in the ported mCertKOS versions of VT-d.

ACPI implements these table using a root pointer and table:

**RSDP:** Root system descriptor pointer. A data structure containing a pointer to the RDST and information about the table. In porting ACPI to mCertiKOS, the RSDP has been abstracted away; programs should access the RSDT through an interface provided with setters and getters, not by using the pointer in the RSDP.

**RSDT:** Root system description table. This table consists of a header containing its length and information about the hardware, and contiguous data blocks, each containing a pointer to another system descriptor table. The tables of interest to IOMMU are DRHDs and RMRRs, described above.

**XSDT:** XSDT is similar to the RDST, but provides 64-bit addressing for 64-bit systems.

The ACPI source code has been divided into two layers in mCertiKOS.

**acpi:** The *acpi* layer provides minimal abstraction from the hardware, although it abstracts away the RSDP. Instead, it searches for the tables with “probe” functions and remembers their locations. Other programs can use the probe function cause *acpi* to recall the location of the table, and by passing as an argument the number returned by the probe, and an offset to the *acpi* layer in order to access an entry in the RSDT or XSDT.

**acpi_info:** The *acpi_info* layer further abstracts the ACPI. No distinction is made between the RDST and XSDT. Rather, calling functions specify which table they want
by the offset. Further, a single function is provided to callers that want to access these tables; this function returns a magic number which can then be passed as an argument, along with an offset, to setter and getter functions in order to access the tables.

In general, calling programs (like IOMMU) should use the interface provided by the `acpi_info` layer. As part of this project, the VT-d code has been modified to use this interface.

**LAPIC/APIC**

LAPIC/APIC ([Local] advanced programmable interrupt control) communicates interrupts to the CPU. VT-d uses MSI data structures to store information about these interrupts; these data structures are described below.

LAPIC refers to the component of APIC that is in the processor itself, since APIC may be located elsewhere in the system. IOMMU is concerned only with LAPIC.

Information about the APIC units are contained in the MADT (multiple APIC description table).

LAPIC has been divided into two layers.

- **lapic**: This layer provides初始化 procedures and constants for LAPIC.

- **acpi_lapic**: This layer provides access to the MADT through setters and getters. It also includes setters and getters for LAPIC structures contained in the MADT. This provides the interface to LAPIC through which other programs should access it.

The way that VT-d uses LAPIC still needs to be modified to account for the differences in the trap handler between VeriKOS and mCertiKOS, and to account for the simplified interface of `lapic`. In particular, there is only one CPU in mCertiKOS, and the LAPIC_ID is fixed as a constant.

**MSI**

MSI (message signaled interrupt) refers to the style of interrupt supported by PCI, described below. However, the actual source code is quite simple, consisting of two data structures and the setters and getters for these data structures. IOMMU uses MSI to communicate interrupts from devices to the processor.
**msg_data:** contains information about the interrupt itself, including whether it is edge- or level-triggered in hardware, and whether it is asserting or de-asserting the interrupt.

**msg_address:** contains information on how the interrupt should be sent, including whether it is a physical or logical interrupt and the destination of the interrupt.

**PCI**

PCI (peripheral component interrupt) physically transfers data between the device and the hardware. IOMMU must use the PCI interface to add devices, as well as to communicate by MSI.

PCI may have multiple buses, each of which has its own “namespace” in the sense that different devices may share the same identifier across different PCI buses, but device identifiers must be unique on a single bus. Each PCI bus supports up to 32 devices, where each device has up to 8 functions; IOMMU uses this information to map each bus to a “context table” (described in the *Initialization and Root Table Access* section of *Organization of IOMMU*).

Adit Sinha (aditya.sinha@yale.edu) has been working on porting PCI this semester; the IOMMU code should be checked to ensure it conforms to his revised interface.

**IOMMU**

IOMMU (in this case, Intel’s VT-d chip) specifically permits: devices to be assigned to virtual machines (i.e., guest operating systems running on the mCertiKOS hypervisor); DMA remapping; and interrupt remapping. (The purpose of IOMMU, as well as DMA remapping, is described in more detail in the *Background* section.)

IOMMU’s current interface includes a function for setup. After setup, the function `ats_gpa_to_hpa` (Address translation service for guest physical address to host physical address) provides address translation through DMA remapping.

A section of IOMMU that has not been fully implemented in VeriKOS, but which eventually could be, includes the function pointers in the struct `intel_vtd_ops`. These are not currently exposed in the header file, but once implemented, they would permit devices to be assigned and de-assigned and the root table to be modified.
IOMMU is currently being treated as a single layer for the purposes of porting setup code. However, if verification proves too difficult, it may be possible to separate out a layer for VT-d initialization, consisting of the functions called by `intel_vtd_setup` and another layer for other functionality, such as explicit address translation.

The implementation of IOMMU is described in the following section.

**Organization of IOMMU**

**Initialization and Root Table Access**

The IOMMU setup is done through the function `intel_vtd_setup`. Calling the setup function causes the number of DRHD and RMRR devices (see ACPI subsection in Layers) to be reset to 0. Two arrays, `drhds` and `rmrrs` will be used to store information about any DRHD or RMRR devices found.

VT-d then scans the ACPI tables for DRDHs and RMRRs through the function `acpi_detect_dmar`. A call is made to the ACPI interface to obtain a magic number for accessing the RSDT. By accessing the header, it computes the length of the RSDT; it is then possible to cycle through the table simply by incrementing the offset passed to the ACPI interface. Each entry is checked for the unit type. If the type is RMRR or DRHD, the entry is parsed by a dedicated method that copies the relevant fields of the data structure representing the unit into the `drhds` or `rmrrs` array. Action is not taken for other device types.

256 arrays of 4kB are then allocated, each representing a single context-entry table. Each context-entry table represents a different PCI bus. Each entry in a given context table for a particular bus represents a single function for a single device. The table will eventually map a function for a given device on a given bus to a particular domain and address translation structure for that domain; the table may be populated later by `vtd_add_device`.

A root table consisting of 256 entries is simultaneously allocated, and each entry is set to point to a single context-entry table.

Finally, the function `vtd_enable` is called. This function cycles through each DRHD information struct contained in `drhds`. Each DRHD struct contains a `regbase` field representing the address for the registers of that DRHD. It sets the root table field to match the root table that has been allocated; it also sets various registers to appropriate constants and
waits for the hardware unit to respond by clearing the register. It additionally sets fields in order to permit the unit to communicate interrupts through MSI; this function calls the operating system function `trap_handler_register` in `kern/trap.c`. This file has been changed from VeriKOS to mCertiKOS and will need to be updated.

**Address Translation/Checking**

The VT-d header file also exposes a function for directly calculating address translation, `ats_gpa_to_hpa`. This directly searches the tables for ATS and returns the translated address. Performing this computation explicitly is useful in VeriKOS for testing that the mapping produced by VT-d is identical to the one produced by VMX (a system that is unrelated to IOMMU) through VMX’s EPT (extended paging table). Once VMX is incorporated into this branch of mCertiKOS, it could be useful to be able to perform this test, but otherwise, it does not seem to make sense to port to mCertiKOS yet. Another method synchronizes ATS translation with EPT. Again, while VMX is absent from this branch of mCertiKOS, it cannot yet be ported.

**Operating on VT-d after Set-up**

`intel_vtd_ops`, which is not exposed by the `vtd.h` header file, contains functions that modify VT-d after set-up, allowing new devices to be assigned or de-assigned, and populating and de-populating the context-entry tables. However, in VeriKOS, device de-assignment is not implemented, and assignment to uses some temporary workarounds that do not support “any given MAXADDR” and which always rely on the first DRHD unit found.

Because these functions are not referenced by the header file or needed by other functions in the VT-d code, they have been removed from the mCertiKOS version in the interest of getting everything in mCertiKOS working first. However, they provide functionality that may prove necessary later, and they might be re-added as future work.

**Sections Ported and Code Modified**

As noted in the previous section, there are three major parts of the IOMMU code that might be relevant to port to mCertiKOS: setup, address translation/checking, and VT-d operations. It does not make sense to port address translation/checking until VMX is
supported, and it might be helpful to get VT-d operations working in VeriKOS before attempting to port them to mCertiKOS. Therefore, this project has focused exclusively on setup code. These other sections of the VT-d code have been removed from the mCertiKOS version so that they may be ported separately at a later time.

In addition to the removal of code dealing with address translation/checking and VT-d operation, the following changes have been made to the setup code:

**get_acpi_entll:** Added method in `acpi_info` layer to permit 64 bits to be read from the ACPI tables at once, an action frequently required in the VT-d implementation. This simply consists of two 32-bit reads shifted appropriately.

**intel_vtd_setup:** The main setup function has been modified in how the root table and context tables are allocated. In the VeriKOS version, each table is given a page through `mem_page_alloc`, which is not present in mCertiKOS. Instead, these tables are now implemented as 4kB C arrays.

**acpi_detect_dmar:** In VeriKOS, it was necessary for IOMMU to probe main memory to find the RSDP, then the RSDT or XSDT. In mCertiKOS, the RSDP has been abstracted away, and the `acpi` does the probing behind the scenes and remembers the table location. In the mCertiKOS version, the DMAR table (that is, RSDT or XSDT) is found using the `find_acpi_table` function, which causes acpi to return a magic number. This magic number can then be passed to setters and getters for accessing the table, rather than manipulating the table memory explicitly.

**acpi_parse_dmar:** Because `acpi_detect_dmar` no longer has explicit access to the RSDT, it is no longer possible to pass a pointer to that table in order to parse it, as is done in VeriKOS. Instead, the magic number from `acpi_detect_dmar` is passed as a parameter. Instead of scanning the table by pointer arithmetic, this function now performs the equivalent arithmetic on the offset passed to the getter method for the DMAR table, and checks that the offset is less than the length of the table as determined from the table header.

**drhd_one_parse:** The parse methods for individual DRHD units similarly cannot take pointers to the units themselves as arguments, since these pointers are no longer accessible. Instead, we pass a table offset, and arithmetic on the offset is used to determine the offsets of the fields of the DRHD entry that will be copied into the `drhds` array. The ACPI getter methods can then be used to fetch the information required and copy it into the array.
**rmrr**. Similar to the case with *drhd*, we pass an offset instead of a pointer and extract the fields of RMRR entries into *rmrrs* by using a getter method with appropriately calculated offsets.

**Future Work**

The next step is to finish porting setup code, of which only the *vtd_enable* function and its helper functions remain. This will require looking into the changes in the *trap* interface between VeriKOS and mCertiKOS and accordingly modifying the way MSI interrupts are set up.

As previously noted, it will be necessary to check that the code conforms to the new PCI interface written by Adit Sinha.

After porting setup code has been finished, it should be possible to check that all code compiles and runs on mCertiKOS by emulation in QEMU.

Formal verification of the setup code can then proceed, pending verification of the lower layers (ACPI, APIC, MSI, PCI).

On a longer term basis, the functionality of VT-d might be extended by finishing the implementation of, then porting, the functions contained in *intel_vtd_ops*.

**Conclusion**

The formal verification of mCertiKOS provides a checkable guarantee of correctness; this project began to extend the functionality of mCertiKOS by porting code supporting Intel’s VT-d chip for IOMMU. VT-d provides hardware-level support for address translation, interrupt translation, and enforcement of memory access restrictions by domains.

Implementing IOMMU with VT-d will permit mCertiKOS to interact with external devices through a well-defined interface that provides efficient input and output without the security liability of direct memory access.

This project resulted in much of the VT-d setup code being ported from VeriKOS to mCertiKOS. A set of immediate next steps has been defined. In moving forward in implementing further sections of IOMMU, and in subsequently formalizing and verifying
IOMMU in Coq, it is my hope that this report might elucidate the structure of the code implementing and supporting VT-d in mCertiKOS.

Acknowledgements

I owe many thanks to Zhong Shao for his advising and assistance; to Newman Wu and Adit Sinha for helping me make sense of what appeared, at first, impenetrable; and to Ariel Katz and Jordana Cepelewicz for their unwavering support.

References


