6-1-1996

Implementation and Analysis of Software Based Fault Isolation

Scott M. Silver
Dartmouth College

Implementation and Analysis of Software Based Fault Isolation

Scott M. Silver

Dartmouth Computer Science Technical Report PCS-TR96-287
Senior Honors Thesis

Dartmouth College
June 1996

Abstract

Extensible applications rely upon user-supplied, untrusted modules to extend their functionality. To remain reliable, applications must isolate themselves from user modules. One method places each user module in a separate address space (process), which uses hardware virtual memory support to isolate the user process. Costly inter-process communication, however, prohibits frequent communication between the application and the untrusted module. We implemented and analyzed a software method for isolating an application from user modules. The technique uses a single address space. We provide a logical address space and per-module access to system resources for each module. Our software technique is a two-step process. First, we augment a module's code so that it cannot access any address outside of an assigned range. Second, we prevent the module from using system calls to access resources outside of its fault domain.

This method for software isolation has two particular advantages over processes. First, for frequently communicating modules, we significantly reduce context switch time. Thus, we demonstrate near-optimal inter-module communication using software fault isolation. Second, our software-based techniques provide an efficient and expedient solution in situations where only one address space is available (e.g., kernel, or a single-address-space operating system).

1.0 Terms

fault domain - an address space and a set of inter-fault domain calls. A module assigned to a fault domain cannot directly access any resource outside of its fault domain.

trusted module - a collection of functions permitted to access any address or resource allocated to a fault domain.

host module - a trusted module that owns, and exists in, a fault domain. A host module provides the runtime assistance needed to load, initialize, and execute untrusted modules.

untrusted module - a module that contains a set of functions with defined entry points. The implementations are not trusted, so they are restricted to their own fault domain.

hardware fault domain - same as fault domain, except the address space is allocated to it by the operating system, which uses a memory management unit to enforce the address space.

logical context switch - the saving of the current context and a subsequent switch to another logical fault domain. This context switch does not require kernel services. Fewer data are saved, and no (full) address space change is required.

2.0 Introduction

2.1 Applications

An extensible application (trusted host) can load and execute untrusted, user modules at runtime. Faults caused by an untrusted module should not halt or otherwise corrupt the host. To isolate faults between a host and a user module, operating systems provide fault domains, such as processes. Processes use separate address spaces to create fault domains. The host and an untrusted module communicate using various forms of inter-process communication. Unfortunately, both processes and inter-process communication consume significant resources for frequent communication. In addition, some operating systems (such as MacOS) do not provide separate address spaces. Recent trends to modularize software in both industry and academia have prompted the need for efficient fault isolation.

Extensible operating systems import user modules into the kernel to improve performance and extend functionality [engler:exokernel, bershad:spin, osdi:panel]. The goal is to simultaneously increase performance and allow modularity while keeping the host (kernel) isolated from the module. For example, some kernels can import small user modules to efficiently dispatch packets as they arrive from networking hardware [mccaine:packet]. To maintain the reliability of the kernel, these packet filters are interpreted. Software fault isolation could allow these packet filters to be translated to native code and execute in a separate logical fault domain [wahbe:sbfi]. More ambitious systems, such as the SPIN operating system, allow user modules to aggressively extend the kernel [bershad:spin]. Security and reliability is maintained by the use of special compilers and type-safe languages at the cost of speed and platform and language neutrality.

In industry, the World Wide Web, despite its plethora of security problems, is taking the world by storm [hpp:blackw]. Untrusted modules such as Java Applets are currently fault isolated through interpretation of type-safe bytecodes [colusa:omniware, hpp:blackw, gosling:java]. To increase the speed of an applet it may be necessary to translate it to native code [adl-t:omnimobile]. Once the applet is executing in native code, it must execute in an isolated environment to protect the browser from the applet.

2.2 Introduction to Software-Based Fault Isolation

An application or operating system that can dynamically load *untrusted* modules at run time needs to create an environment in which untrusted modules can run efficiently. In addition to good performance, untrusted modules must not interfere or otherwise corrupt the host module. This means that an untrusted module must be in a separate *fault domain* from its host. A fault is caused when an illegal action is performed (e.g., access to a forbidden resource).

The classic method of isolating two domains uses hardware assisted virtual memory and UNIX processes (hereafter referred to as processes) to create separate address spaces and separate resource permissions on a per-process basis. For a host module to communicate with an untrusted module or vice versa, some form of inter-domain communication is used. Pipes or remote procedure calls (RPC) are the most common [birrel:rpc].

Using multiple processes for multiple untrusted modules often yields unacceptable performance for frequently communicating modules, due to the high cost of inter-process communication and context switches. In addition, inter-process control transfer which is inherent in most process communication, does not necessarily scale with a processor's integer performance [wahbe:sfi]. The cost of an RPC for frequently communicating domains is prohibitive. An RPC requires: copying arguments from caller to callee, a trap into the kernel, a context switch, copying the return value, and another context switch. Even the fastest RPC is several orders of magnitude slower than a function call [bershad:lrpc].

In this paper, we explore recent methods for creating and maintaining software-enforced fault domains [wahbe:sfi]. We remove the requirement for separate address spaces by creating logical fault domains within a single address space provided by a single process.

On a UNIX-like operating system, two sets of resources need to be protected: memory and everything else. A technique called *software fault isolation* is used to enforce the logical assignment of address space to an untrusted module. The idea is to restrict a module to a set of valid addresses assigned to the fault domain. Since all other resources (besides CPU utilization) are accessed through system calls, per-domain access to system calls protects other resources. For example, the system calls, `read`, `write`, `open`, and `ioctl` provide access to files and other devices. To restrict access to those resources, it is only necessary to reroute `read`, `write`, and `open` through access controlling functions.

On other platforms various CPU specific instructions must also be protected. For example, on a 68000 running MacOS, both the A-line and F-line exception handlers must be rerouted through an access controlling function. A-line and F-line instructions provide access to operating system services [moto:68k].

Multiple fault domains existing in one address space allows for near-optimal RPC performance. An RPC is a transfer of control from one fault domain to another. In the process model RPC performance is bound by resources consumed by context-switching, marshalling the arguments from one address space to the the other, and system calling. Under the single process model, an RPC requires only the time to copy arguments, and to set up the lighter software-enforced fault context. We reduce the cost of these activities, and thus the cost of an RPC, through software fault isolation techniques.

3.0 General Ideas

In this section we describe the concepts behind software fault isolation and sandboxing. In Section 5.0 we describe our implementation.

3.1 Protecting Memory

To protect memory we must determine a general technique for restricting a module to an arbitrary region in memory. We prefix an *untrusted instruction* with an additional sequence of instructions to check the target effective address. An unsafe instruction is a load, branch, or store whose target address cannot be determined before runtime. For example, program-counter (PC) relative addresses can be verified at compile time. In contrast, arbitrary jumps, like returns from functions cannot be verified at compile time. Thus, the inserted code checks to see that the target address is within the segment. If not, a fault is generated. Consider Figure 1 to be part of an untrusted module. The goal of the technique is to prevent the instruction from reading memory outside of its designated segment. Depending on the desired level of isolation and performance, we can chose not to check the target address of loads. In our implementation, however, we chose to check the target address of loads.

FIGURE 1. Pseudo code for an unsafe instruction. Since we cannot determine the address contained by r0 at runtime, this instruction is unsafe.

```
(1)      load r2, (r0) ; load contents of memory address r0 into r2
```

In Figure 2 the general prefix sequence is shown. The “trap” pseudo-instruction is a system dependent action. In most UNIX implementations, the “trap” would most likely send a signal to the untrusted module that performed the illegal action.

FIGURE 2. Pseudo code for isolating an address in software. We prefix the instruction in line (3) because it is unsafe. Line (3) will not execute unless it passes the legality check in line (1).

```
(1)      legal = check r0      ; is r2 in this fault domain
(2)      trap if (not legal)   ; no? notify parent
(3)      load r2, (r0)         ; yes? allow load to proceed as usual
```

The speed of an isolated load or store depends upon the complexity and length of the check code. To reduce the complexity of the check, we restrict the domain of the target address to a contiguous segment of memory. Thus, in a simple implementation, we need two extra variables to either hold fence posts of the segment or base and bounds addresses. Figure 3 depicts the use of fence posts, both of which are held in registers. Line (1) in Figure 2 expands to lines (1) - (4) in Figure 3. This technique is called segment matching [wahbe:sfi]. Notice that this technique requires 4 instructions.

FIGURE 3. Pseudo code for segment matching. We compare the address contained in the target register (r0), to beginning and end of the segment (contained in rBeginning and rEnd). We trap if the target address is not between those two “fence posts”.

```
(1)      compare r0, rBeginning ; r2 < beginning?
(2)      blt illegal            ; yes? illegal access
(3)      compare r0, rEnd       ; r2 > end
(4)      bgt  illegal           ; yes? illegal access
(5)      load r2, (r0)          ; perform load
(6)     illegal:
(7)      trap                   ; notify parent
```

If we no longer require that faults actually be identified, but only guarantee that the target address be valid (but not necessarily the computed one) we can remove line (2) in Figure 2. This technique is called *sandboxing* [wahbe:sfi]. Figure 4 shows the pseudo code for sandboxing. Within a segment the top `k` bits of all addresses, called the segment identifier, are constant. If we simply set the top `k` bits of each target address to the segment identifier we guarantee that the address must be in that segment. If the target address were an illegal address, it would be prevented from accessing memory outside of the fault domain. Notice that this technique takes only two instructions. The first instruction clears the segment identifier of the target address, and the second instruction sets the segment identifier of the target address. For this technique to be efficient, the masks should be stored in dedicated registers since two load penalties for every target instruction may be unacceptable.

FIGURE 4. Pseudo code for sandboxing. We drop the requirement to identify the location of the fault. Line (1) clears the segment identifier of the address contained in r0. Line (2) sets the segment identifier. Line (3) can now only access addresses within the segment.

```
(1)    r0 = r0 AND rClearSegmentIdentifier ; clear seg identifier
(2)    r0 = r0 OR rSetSegmentIdentifier    ; set seg identifier
(3)    load r2, (r0)                       ; perform load
```

We still have not addressed several outstanding security problems. An untrusted module could arrange to jump directly to the load and avoid the prefix code. A module could arrange this by using code that detects that it is sandboxed, and then arranging a branch to skip over the check code. This problem is solved by reserving both code and data registers, that is, disallowing user code to use these registers. The “data register” is used as the target register for every load or store. The “code register” is used as the target register for every indirect branch or jump. Thus, each load or store uses the special data register, which can contain only a valid data segment address. This eliminates a module from containing self-modifying code (on systems that map those pages writable). However, a module could generate code in its heap, static storage, or stack that performs illegal instructions. Since the special register reserved for indirect branches and jumps can contain only a valid address in the code segment, not the data segment, a module cannot jump to generated code.

Implicitly, we stated that there are separate “code” and “data” segments. It follows that there are also separate data and code “segment setting masks”. All of these masks are stored in reserved registers for fast access. The total reserved register count is 5: one dedicated code register, one dedicated data register, one clear segment mask, one set code segment mask, and one set data segment mask.

The result of reserving 5 registers on a RISC architecture with at least 32 registers is marginal [wahbe:sfi]. On older CISC architectures, however, such as the Intel x86 with only 8 general-purpose registers, the penalty might be too great.

A module can be sandboxed at runtime or at compile time. If the module is sandboxed at runtime, then the sandboxer can be highly architecture dependent. Sandboxing at runtime might also shorten the verification step (see Section 4.5). If a module is sandboxed at compile time, the runtime must determine if a module is correctly sandboxed. One method uses a verifier to actually check the module and declare it safe or unsafe. Another method involves creating a trusted compiler that cryptographically signs the sandboxed module with a private key. Since the public key of the compiler is known, authentication will only succeed if the module has not been modified. This method is used in the SPIN operating system [bershad:spin].

By using public key encryption and a compiler to sandbox a module, it is easier to create a more platform-independent sandboxer. A sandboxer created by modifying a compiler also makes it possible to use later optimization stages of the compiler. This is the technique we chose.

3.2 Runtime Support

The runtime support primarily provides services for loading untrusted modules and making cross domain (RPC) calls. In addition to executing untrusted modules, the runtime code limits access to system calls and other resources.

An untrusted module is assigned a segment of the host's address space and is loaded into that segment. If an operating supports access to individual page permissions, the code pages are marked executable and readable, but not writable. The data pages are marked readable and writable, but not executable. If the module is already sandboxed, it is verified by either checking the signature of the compiler or by using a verifier (see Section 4.5).

To catch system calls, the runtime code must use the sandboxer or some other mechanism to cause system calls to pass through some procedure that can screen access to potentially dangerous functions.

3.3 Crossing Fault Domains

Due to the cost of process-level context switching traditional RPCs are slow. Our goal is to create RPCs that perform near-optimally. An optimal RPC would take exactly the amount of time that a regular function call takes.

There are three types of fault domain crossings: host to untrusted, untrusted to host, and untrusted to untrusted.

Host to an untrusted module: the host must set up the reserved sandboxing registers, save its own context, restore sandbox context (the special sandboxing registers, and the stack pointer), swap stack pointers, possibly copy arguments, and call the desired function. A host may also wish to zero out all non-used registers to avoid leaking information from the host to the untrusted module. Upon return the context and stack pointer are restored, and the return value is copied if necessary. In the case of an application binary interface (ABI) that allows arguments to be passed in registers (and has no recursion), no argument copying is necessary.

Untrusted module to host: Note that all indirect branches or jumps must be to a valid code address within the untrusted module's segment. So, to jump out of its fault domain, an untrusted module must jump through a carefully constructed jump table. If the CPU on which the system is running has a *direct* jump or branch instruction in which the target address is encoded in the instruction itself we can create a "hard coded" jump table. The jump table would consist of a series of direct jumps. At load time, the host modifies each jump instruction in the table so that each is hard coded to jump to a particular address. In addition the page that contains the jump table is marked unwritable in the host code to the untrusted module. If the CPU does not have a direct jump instruction, however, we can employ a special jump table and a special sequence of instructions. The basic idea is to place a jump table in an unwritable page somewhere in the untrusted module's valid data segment (see Section 4.3). We describe this technique in more detail in Section 5.2.2.

3.4 Optimization

We can make several optimizations to sandboxed code. Special registers like the stack pointer (SP) and the frame pointer can be sandboxed once for multiple unsafe instructions before a control transfer. The sandboxer can verify that loads from only small (valid) offsets from the SP are made, and thus safely sandbox the SP once at the beginning of a function.

Often CPUs provide indexed addressing modes. The indexes are usually limited to small ranges, like 16 bits. Compilers often use indexed addressing modes for accessing arrays in loops and for accessing elements from a known reference register (e.g. the SP). If we place guard pages (pages marked not readable, writable, or executable) the size of the index range at the beginning and end of each of our data and code segments, we only have to sandbox the base address (as opposed to the index) once, outside of the loop.

3.5 Verification

We must verify that an untrusted module is sandboxed before we execute it. There are two main methods for verifying a module. If we sandbox untrusted modules at compile time we must either build a separate application which analyzes the module, or use a trusted compiler that cryptographically signs an untrusted module after it has finished. At load time, we simply verify the signature of the compiler and execute the module.

Wahbe et. al. present a method for verification at runtime [wahbe:sfi].

4.0 Implementation

The current implementation of our software-based fault isolation system is designed for the DEC Alpha under Digital Unix 3.2 (formerly OSF). The primary goal of this design is to explore the possibility of a portable sandboxer and runtime. We implemented two versions. The first sandboxes the intermediate language representation (used by the GCC, the Gnu C Compiler) of a module [stallman:gcc]. This version works for much common code; for a complete solution, we found it necessary to modify the Alpha-specific code generator of GCC.

4.1 Sandboxing Using RTL

The first sandboxer is a modified form of GCC. We added sandboxing as the first “optimization” pass on the intermediate translated form (RTL, register transfer language). Since RTL is designed to be architecture neutral, our hope is that this sandboxer can be easily ported to other architectures.

GCC is the compiler created and distributed by the Free Software Foundation. It is designed to execute on many platforms and compile many languages to binaries on multiple target platforms and architectures. It is primarily used on UNIX and UNIX-derivative platforms, but also runs under Windows and MacOS. By choosing GCC as the sandboxer, we are limiting our sandboxer to platforms on which GCC runs and to any language front end which is written for GCC. There are many such platforms, however, so it is more portable than a binary-code or assembly-code patcher.

GCC translates a source file to RTL on a function-by-function basis. Next, multiple optimization passes run on the RTL such as code motion and loop unrolling. We view sandboxing as a special “optimizer”, which actually outputs slower (but safer) RTL. Since the sandboxing techniques we described are language and (in general) architecture neutral, we can sandbox at the intermediate language level of compilation.

There are two main pieces to the sandboxing code. First we must arrange for the reservation of the special sandboxing registers. Second, we must create an optimizer that augments the RTL translation of the current function.

4.1.1 Reserving Registers

We need to reserve the five dedicated registers used in sandboxing. Since GCC is retargetable to many platforms, it contains no register quantity dependencies. We needed only to set a mask indicating the desired registers.

4.1.2 Augmenting Register Transfer Language

The sandboxing optimizer pass executes directly after a function is translated to RTL. RTL is the intermediate language upon which all optimizations (except peephole optimizers) occur.

The following is excerpted from the GCC Manual [stallman:gcc]:

> Most of the work of the compiler is done on an intermediate representation called register transfer language. In this language, the instructions to be output are described, pretty much one by one, in an algebraic form that describes what the instruction does.
>
> RTL is inspired by Lisp lists. It has both an internal form, made up of structures that point at other structures, and a textual form that is used in the machine description and in printed debugging dumps. The textual form uses nested parentheses to indicate the pointers in the internal form.

RTL assumes an infinite number of pseudo (or virtual) registers. Register assignment is performed as an optimization pass. Analysis is performed to assign a hard register or a “stack slot” (i.e. allocated from the local variable space on the stack) to each pseudo-register.

Figure 5 shows an *RTL expression* (RTX). The operator is followed by a colon and then a *machine mode*. A machine mode indicates the size of the data and the representation used for it. In Figure 5 the indicated machine mode is “`DI`”, which means double integer. A double integer is a two’s complement 8-byte integer. The expression shown does not actually “affect” any registers or memory. In order to actually change state, a “side-effect” operator is necessary. In Figure 6 the `set` side-effect operator assigns the value of the second operand to the first operand.

FIGURE 5. An RTL Expression (RTX) whose value is the bitwise and of the contents of pseudo-register 74 and the constant -1. Because of the machine mode (DI), the constant -1 will be treated as an 8-byte two’s complement number (0xFFFFFFFFFFFFFFFF).

```
(and:DI (reg:DI 74) (const_int -1))
```

FIGURE 6. The use of the set operator in RTL. Pseudo-register 75 is assigned the value of the bitwise AND of the value of pseudo-register 74 and -1.

```
(set (reg:DI 75) (and:DI (reg:DI 74) (const_int -1)))
```

To sandbox a function, we must first determine which expressions need to be sandboxed. Since we are sandboxing loads, stores, and indirect branches and jumps, we must look for expressions that access memory. Figure 7 shows such an expression. The `mem` operator returns an address based on its operand. The expression in Figure 7 sets the value at the address contained in pseudo-register 74 to -1.

The necessary prefix code requires an `and` and an `or`. In Figure 8, a sequence of expressions shows a sandboxed store.

FIGURE 7. A store and a load

```
(set (mem:DI (reg:DI 74))
 (const_int -1))

(set (reg:DI 74)
 (mem:DI (reg:DI 74)))
```

FIGURE 8. A sandboxed store. The first expression sets the segment identifier, the second expression sets the data segment identifier. The third instruction is the store (which now cannot access any address outside the data segment).

```
(set (reg:DI $kDedDataReg)
 (and:DI (reg:DI 74) (reg:DI $kDedClearSegReg))

(set (reg:DI $kDedDataReg)
 (ior:DI (reg:DI 74) (reg:DI $kDedDataSetSegReg))

(set (mem:DI (reg:DI 74)) (const_int -1))
```

4.1.3 Problems with Sandboxing and RTL

We encountered two problems with sandboxing RTL. First, target addresses of RTL `mem` expressions can be system (GCC) dependent. Second, various system imposed runtime requirements are not visible in RTL.

The target address of an RTX is often a simple `reg`, or an expression which represents “register + offset” addressing (See Figure 9). However, to optimize for addressing modes, the RTL generated for some target addresses on an Alpha are non-standard. Figure 10 depicts a valid memory reference which has no meaning outside of the Alpha machine description. Our normal tactic would move the target address into one of the dedicated sandboxing registers, and then proceed to sandbox the address. Unfortunately, since this expression can only be translated as a target address of a `mem` expression, the code generator fails.

FIGURE 9. A typical mem RTX which uses offset addressing

```
(mem:DI (plus:DI (reg:DI 74)
  (const_int 4))
```

FIGURE 10. A mem RTX which uses a non-standard target address.

```
(mem:DI (and:DI (plus:DI (reg:DI 74)
  (const_int 4)) (const_int 8))
```

Runtime imposed constraints, such as function prologues, function epilogues, and global constant loading are not expressed in RTL. For each function, the prologue and epilogue code is automatically generated by an architecture dependent file. No function epilogue RTL code is ever seen by our sandboxing optimizer. Thus, we can never sandbox the frame or stack pointers in the prologue (or epilogue). Also, according the the Digital Unix Application Binary Interface, for each global (as opposed to static) function call, the program needs to load certain values from a global offset table (a runtime specific global table). This sequence includes several loads from memory that are never represented in RTL code.

To address these problems, we augment machine description files.

4.1.4 Sandboxing Using Machine Description Files

GCC is retargetable to many platforms by using machine description files for each system. A machine description file is a series of *rules*. A rule corresponds to one or many low level virtual instructions defined by GCC. For example, one instruction is “`movsi`”. The rule for “`movsi`” describes how to move a single integer (for many sources and destinations). A rule can either be a series of system dependent assembly instructions or a series of RTL functions. Figure 11 shows a “`nop`” rule for the Alpha.

FIGURE 11. A nop instruction from the Alpha machine description file. This defines the internal GCC instruction “nop” as the Alpha instruction “bis $31,$31,$31”.

```
(define_insn "nop"
  [(const_int 0)]
  ""
  "bis $31,$31,$31"
  [(set_attr "type" "iaddlog")])
```

We only need to modify the load, store, and jump GCC pseudo instructions to include sandboxing. Figure 12 shows the original and modified rules for a load. A machine-description also includes the various functions for creating function prologues and epilogues. It is a matter of simply adding the sandboxing code, as we did for stores.

FIGURE 12. The first rule describes how to move from memory to a register. The %0 and %1 operators get replaced with the actual address and target registers, respectively. In the second rule, the sequence has been modified to include sandboxing code. (The code shown here has been simplified). Thus, for each move from memory to register both sandboxing and loading code will be generated.

```
(define_insn "movdi"
  [(set (match_operand:DI 0 "general_operand" "r")
        (match_operand:DI 1 "input_operand" "m"))]
  "register_operand (operands[0], DImode)
   || reg_or_0_operand (operands[1], DImode)"
  "ldq %1, (%0)”)

(define_insn "movdi"
  [(set (match_operand:DI 0 "general_operand" "r")
        (match_operand:DI 1 "input_operand" "m"))]
  "register_operand (operands[0], DImode)
   || reg_or_0_operand (operands[1], DImode)"
  "and %0,       $kClearSegReg, $kDedDataReg \;
   bis $kDedReg, $kSetSegReg,   $kDedDataReg \;
   ldq %1,       ($kDedReg)”)
```

4.2 Runtime Support

As we discussed in Section 4.2 the runtime provides all of the “operating system” services for the software fault domain. It is the most system dependent piece of the fault-isolation system. There are two main pieces to the runtime. First, we need a method to trap system calls and modify an untrusted module’s memory. Our solution uses the “`/proc`” file system [dig:proc, sgi:proc]. The `/proc` file system provides exclusive access to an untrusted module’s memory, and also allows the trapping of its system calls. We leverage our system on top of the system runtime shared object services (i.e., shared libraries). Second, we need to verify and load a module, potentially resolve symbols (i.e., runtime symbol binding), and set up the sandboxing context for an untrusted module. We use the system shared library loader (`loader`).

4.2.1 Loading an Untrusted Module

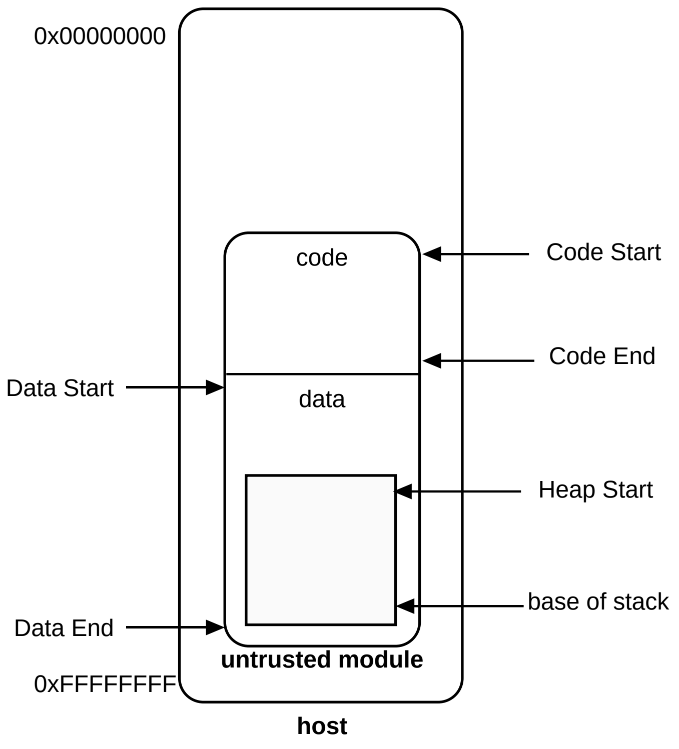
Before actually loading an untrusted module, we authenticate its creator. By checking the signature we can determine if our trusted linker and compiler was used. If we cannot authenticate an untrusted module it is not permitted to execute.

To leverage off of system components we use system loader libraries to load shared libraries. This does not give us as much flexibility or fine grained control over where the module is loaded, but it does mean we need not handle relocation ourselves. We still need to know where the code and data sections are mapped.

To determine the location of the code and data segments we use a "secure" linker (i.e., one that signs), to place runtime object code before and after the user module. The piece of object code that prefixes the user module contains the unsandboxed RPC stubs (see below), and two marker symbol pairs. One symbol pair is a start and end data symbol, the other is a start and end code symbol. By finding the addresses of these symbols at runtime, after the loader maps the object, we can find the start of the code and data sections. See Figure 13.

The suffix object code also includes a static heap (see Figure 13). We make this heap large, for it will also contain the untrusted module's stack. A third marker contains the size of and location of the heap. Under Digital UNIX, an unusually large data area will not cause any of the corresponding pages to be mapped until they are touched. Once the heap is setup, the last step is to setup the stack. Currently the stack starts at the end of the static heap.

FIGURE 13. The text in quotes represents symbols whose addresses resolve to the beginning and end of each segment and also identify the malloc heap (and stack).



4.2.2 Crossing Fault Domains

Host to Untrusted Module

First, the called function must be resolved to the untrusted module. We use a simple interface to the Alpha shared library loader to determine the location of a function (by name). Even though the current implementation actually tries to find the symbol each time, this clearly can be performed one at run time. If we have a stub for each function called, then the function address can be filled in by the loader itself, instead of our doing it by hand. Since stubs are currently generated by hand, however, we did not make this optimization.

To call a function in the untrusted module, in addition to normal function call register saving, the sandbox register context must be restored, stack pointer swapped, arguments not passed in registers copied to the untrusted module's stack, and a jump to the actual function. On a DEC Alpha this adds 10 extra instructions (ignoring the copying of parameters, which is one store per parameter) to the normal procedure call cost. Upon return, in addition to the normal function return and register restoration, we need only swap the stack pointer and copy arguments not passed into registers back into their stack slots on the host stack. We do not need to save the untrusted modules register context, because those registers are inaccessible to it.

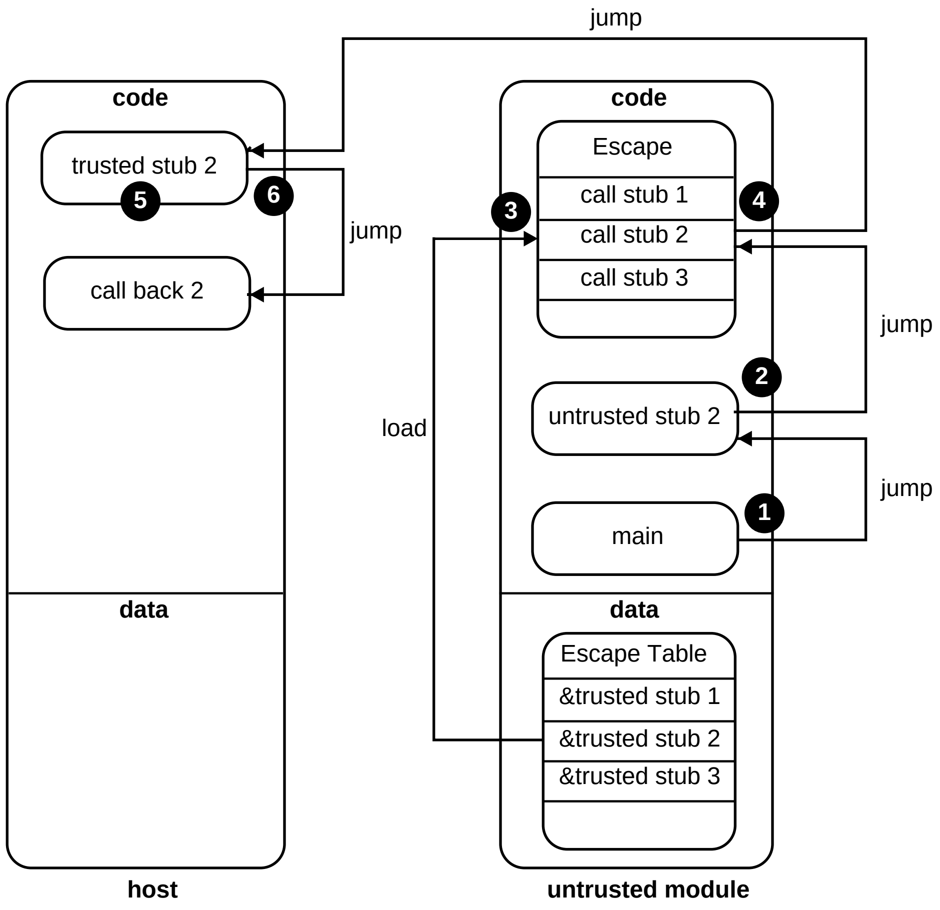
Untrusted Module to Host

The control transfer from untrusted module to host is trickier because the target of the jump or branch instruction will lie outside of its code segment. If *direct* branches and jumps are available the technique described in Section 4.3 can be used. Unfortunately many platforms that support direct addressing limit the number of bits available for the direct jump (because it is assembled into the instruction), thus limiting the distance in memory of the RPC sending stub from the target RPC receiving stub [may:powerpc]. To gain the full range of addressable memory on a given CPU or if the given CPU does not support direct jumps or branches, another technique is necessary.

The technique has two pieces. First, we require a jump table. The table must be located in a page mapped readable, but not writable or executable, in the untrusted module's data segment. (We can drop the not executable requirement by insisting that any data in the table not be an unsafe instruction.) The second part is a special code sequence which prevents a rogue module from using the table load to jump to an illegal address. We call this sequence of code `Escape`, and the jump table the `EscapeTable`.

There are four functions that make up an untrusted-to-host RPC: the untrusted stub, the code that loads from the jump table and jumps to the host stub, the host stub, and the host procedure. Figure 15 shows an overview of the process.

FIGURE 14. Overview of an untrusted to host RPC. This demonstrates the RPC to call a function called “call back 2”. The steps are labelled in order of execution. Step 1, a function called main in the untrusted module calls the untrusted stub 2. Step 2, the stub calls the correct Escape function. Step 3, the correct escape sequence loads the address of trusted stub 2, and jumps to it (Step 4). The control transfer is complete at step 5. At Step 5, arguments are copied on to the host stack, and the stack pointers are swapped. Step 6, the function call back 2 is executed.



The untrusted stub jumps (step 1) to a particular place in `Escape`, which loads (step 2) a value as a known offset into the `EscapeTable`, and then jumps to that address (step 3). If we simply performed a load and a jump through any (non dedicated) register, a rogue module could place an illegal address in the target register of the jump instruction and skip the load, and therefore jump to an illegal address. If the target register of the jump is our dedicated code register, we guarantee that the instruction will never jump to an illegal address if the rogue module skips the load and starts at line (3) in Figure 15. This sequence is illustrated in Figure 14. However, it is still not safe. If the untrusted module starts at (2), we will load the value of an assembled instruction into `rCode`, and jump there.

FIGURE 15. Simple version of Escape. This is a DEC Alpha assembly sequence for loading an entry from the table and jumping to it. kOffs0 is a constant whose value is the index of the stub entry in the table. We use the rCode register (the special register reserved for indirect jumps) to avoid jumping to an address, other than the desired address, outside of this fault domain. This version is still not safe. If the module jumps directly to (2), we will jump to whatever address happens to the assembled value of the instruction at rCode.

```
(1)     lda rCode, kOffs0(EscapeTable); load address of entry
(2)     ldq rCode, (rCode)            ; load address from jump table
(3)     jmp rZero, (rCode)            ; jump to the stub
```

To address the problems of the simple version, we use the sequence depicted in Figure 16. We can prove that the indirect jump in line (7) will only exit the fault domain to an address in the untrusted module's code segment or an address listed in the `EscapeTable`. First, we know that upon entry to any line in `Escape` we know that `rCode` contains some valid code address and `rData` contains some valid data address. The correct path is to start at line (1) and in line (7) we will jump to the stub address as in Figure 15.

If we start at (2), `rData` could contain any valid data address. Thus `rCode` will be loaded with a potentially illegal value. In lines (3) and (4), however, we try to load the same table entry as in (1) and (2). Since `rData` will now contain the address of the stub, only two things can happen in the compare sequence in lines (5) and (6). First, in (6) we take the branch, then `rData` is not equal to `rCode`, thus the untrusted module must have skipped line (1), resulting in a trap. If we do not take the branch, thus `rData` equals `rCode`. Since `rData` must contain the stub address, the jump in (7) will give us the desired result.

If we start at (3), in order to take the branch in (6), `rCode` must already contain the value which will be loaded from the table in (4). Thus, `rCode` will contain desired stub address.

If we start in (4), `rCode` must equal the value at address `rData` if we take the jump in (7); since, `rCode` is in the untrusted module's code segment, this does not succeed in jumping to an illegal destination.

If we start in (5), `rCode` must equal `rData` in order to take the jump in (7). Again, `rCode` must be a location within the untrusted module's code segment.

If we start in (6), `rData` will equal 0 in order to take the jump in (7). Again, since we know `rCode` will be a valid code address, line (7) will jump somewhere valid in the module's code space.

If we start in (7), `rCode` must contain some valid code address, thus we will just jump to a valid code address in the module's segment.

Since we used `rData` as our target registers for the loads, we guarantee that no loads will take place out of the valid data section.

Thus, the escape code is robust, i.e., the untrusted module may only jump to a designated exit point (listed in the `EscapeTable`) or to some location in its own code segment.

The current implementation stores the table in the code segment, and not in the data segment. This way we know that the segment is not writable. In order to guarantee that the untrusted module cannot jump to an entry in the table and perform an illegal operation, we require that no entry contains an invalid instruction.

FIGURE 16. A robust version of Escape. See the text for full explanation. This code is not the one actually used, since we need to deal with some Alpha runtime issues (like gp and pv). Those details were eliminated for clarity. kOffs0 is a constant whose value is 0. rZero is the zero register on the Alpha. rCode is the dedicated code register. rData is the dedicated data register.

```
(1)      lda  rData, kOffs0(EscapeTable)  ; load address of entry
(2)      ldq  rCode, (rData)              ; rCode = stub address
(3)      lda  rData, kOffs0(EscapeTable)  ; load address of entry
(4)      ldq  rData, (rData)              ; rData = stub address
(5)      subq rData, rCode, rData         ; rData = rData - rCode
(6)      bne  rData, illegal              ; if (!rData) illegal
(7)      jmp  rZero, (rCode)              ; else, jump away
(8)    illegal:
(9)      trap                             ; do trap stuff here
```

4.2.3 /proc and Modifying Untrusted Modules

The "`/proc`" file system (or `procfs`) allows processes to be manipulated as files. It enables processes with the correct permissions to control the behavior of victim process. An entry in the "/proc" directory is created for each valid process. The file is treated like any standard file, and manipulated using open, close, write, ioctl, etc. The functions that actually read and write data to the file give access to the processes's memory. The first address of the processes's memory corresponds to the first byte in the file.

Procfs provides various other services to the controlling process through the use of ioctl. For example, we can stop a process, cause a process to run again, or stop a process from calling a set of system calls. Fortunately the procfs is available under Digital Unix, IRIX, BSD, and Linux. Since procfs provides a somewhat general interface to the process, we can duplicate its structure (the parts we need) on platforms that do not yet support procfs (e.g., AIX).

We use procfs to fill in the EscapeTable (see Section 4.3) with the various addresses of the callbacks. This is quite simple using procfs. We simply lseek to the base address of the table, and write the addresses of the trusted stubs into the table.

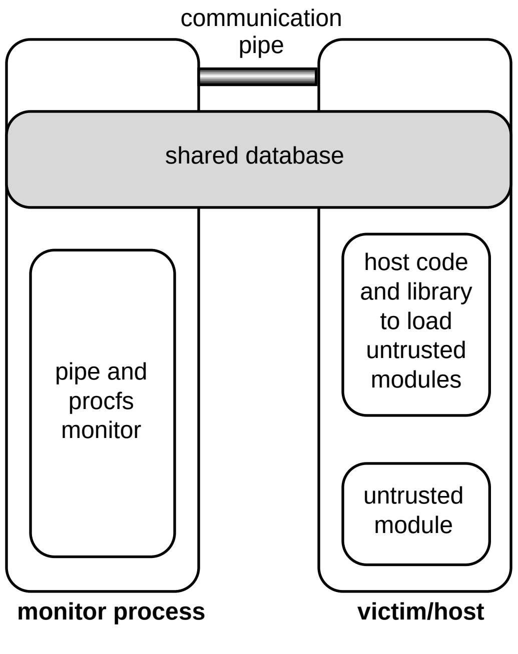
To trap system calls, we use the procfs to ask the operating system to stop on a set of system calls for the process or thread executing the untrusted module. The host supplies a mask (a large mask) specifying which systems calls for a particular module should be trapped. Ideally, the host would supply a table of replacement routines for the replaced system calls. Our current implementation, however, does not replace any system calls, but faults if the module makes any unsafe system calls. In future versions a replacement set of system calls will be associated with each untrusted module context.

4.2.4 /proc, the Untrusted Module Database and Software Structure

The procfs requires either a threaded or a two process software design. To stop (and resume) a thread or process, another process (called the monitor) or thread must be able to wake it up. We chose a two process design for expediency (there was example code available). At start-up a host that wishes to load untrusted modules calls a library function that forks off a process that subsequently opens the procfs file for the host process. This file is opened with the permission "O_EXCL" which means that only one process can have a file descriptor to the file for that process (so that the untrusted module cannot modify itself). We call the process opened the victim. Thus, the host (and its loaded untrusted module) is the victim. We create two communication channels between the victim and the `monitor`. The first is a pipe (which is faked to be bidirectional, by using two pipes), and the second is a shared memory area. The shared memory area is the central database where all the current information about an untrusted module is stored. Concurrent write access to the data base is controlled using messages communicated over the pipe. We chose not to use semaphores for expediency. The current software allows for only one possible writer to the database. This will need to be changed in the event of a threaded model.

Figure 17 shows the architecture of the complete system. The `monitor` process waits for activity on the file descriptor attached to the communication pipe or the descriptor attached to the `monitor` process (using the `poll` system call, which actually blocks). The victim can send 4 different messages to the `monitor`: `new untrusted module`, `unlock database`, `lock database`, and `shutdown`.

FIGURE 17. The Architecture of the Software Based Fault Isolation System



The "`new untrusted module`" message is sent to the `monitor` when a new module has been loaded into memory and placed in the database. Included in this message is the address of the `EscapeTable`. Next, the `monitor` writes the callback addresses into the table.

"`Lock database`" and "`unlock database`" messages are sent to guarantee exclusive access to the database. These can be used to modify the database and avoid race conditions.

The "`shutdown`" message is sent when the host terminates. Any cleanup should be performed. Currently no cleanup is necessary.

5.0 Performance

To evaluate the performance of software-enforced fault isolation we used our prototype system on a 133Mhz DEC Alpha 3000 300X running Digital Unix 3.2. We evaluate the speed of our RPC mechanism, and determine the cost of fault isolation on various benchmarks.

5.1 Sandboxing Overhead

We measured the execution times and self-reported values (such as FLOPS) for various benchmarks. We treated each benchmark as an untrusted module, and sandboxed all of its code. Table 1 shows the changes in execution time and benchmark value. We use non-sandboxed code as the baseline performance. The overhead of register reservation (recall the five dedicated registers) and the overhead of the additional sandboxing instructions is reported. The value for sandboxing overhead includes the the reserved register overhead. Positive values indicate that the benchmark ran slower. In general, floating point benchmarks slowed down less than integer benchmarks. We used Al Aburto's set of widely used benchmarks [aburto:bench].

Table 1 contains various anomalies. For example, `flops1` reports that the sandboxed version was actually 7.58% faster than non-sandboxed code. In many cases the code with a reduced register executed faster than the normal code. In each of these cases the result was nominal and not statistically significant. Notice that register reservation overhead is about 2-3%, however, while sandboxing overhead is about 20 - 30% (though as low as 1%).

The benchmarks we tested used little or no I/O. The more I/O an application uses, in general, the lower the cost of sandboxing.

TABLE 1. Sandboxing and Register Reservation Overhead. Column 1 indicates the name and type of the benchmark. An “int” designates a primarily integer-based benchmark. An “fp” designates a primarily floating point-based benchmark. Columns 2 and 3 compares the performance of each benchmark compiled without the use of the dedicated sandboxing registers to the performance of the benchmark normally compiled. Positive values indicate slower performance. A “benchmark value” is the self-reported value. For example, the flops benchmark reports “FLOPS” or floating point operations per second. Columns 3 and 4 indicate the performance of benchmarks compiled with the sandboxing registers reserved and the sandboxing code inserted.

benchmark		reserved register overhead		sandboxing overhead	
name/type		benchmark value (%)	elapsed time(%)	benchmark value (%)	elapsed time(%)
`c4`	int	4.68	4.88	35.1	68.6
`dhry1`	int	0.361	0.33	29.6	42.0
`dhry2`	int	32.6	20.1	30.8	23.3
`fft`	fp		0.359		0.894
`flops1`	fp	0.956	-0.327	-7.58	1.85
`flops2`	fp	0.277		1.70	
`flops3`	fp	0.147		1.90	
`flops4`	fp	0.005		2.50	
`hanoi`	int	-0.514	-0.797	39.2	18.1
`heapsort`	int	5.44	7.03	16.6	55.3
`clinpack1`	fp	-1.69	-0.714	24.2	27.9
`clinpack2`	fp	2.34	2.45	29.1	32.5
`clinpack3`	fp	0.316	0.291	26.0	31.0
`clinpack4`	fp	-0.551	0.794	27.6	33.7
`mm1`	fp	0.00	0.00	35.7	34.0
`mm2`	fp	0.00	-2.68	20.0	22.5
`mm3`	fp	9.91	10.9	27.3	29.4
`mm4`	fp	0.00	-6.91	42.9	26.8
`nsieve`	int		3.53		27.9
`queens`	int		1.07		38.2
`tfftdp`	fp	-2.06	-2.03	0.155	0.0103
average		2.90	2.13	21.2	28.6

5.2 Cross Fault Domain Performance

We measured the performance of host-to-untrusted-module RPCs, and untrusted-module-to-host RPCs. Each RPC took no arguments and returned no value. For comparison, we compared the execution time for void function calls determined by Wahbe, et al [wahbe:sfi]. Function performance performance provides a lower bound on the performance of an RPC. In addition, we timed the round trip cost of sending a byte between two processes using the `pipe` mechanism. Table 2 shows the results of our experiments. Notice that our RPCs are 2 orders of magnitude faster than the byte sending test. This is due to our extremely inexpensive context switches, conservative register saving, and no traps into the kernel.

We attribute the difference in costs between the two RPCs in Columns 1 and 2 to our implementation. In a future implementation this cost would be equal.

TABLE 2. RPC Performance. Column 1 indicates the cost of a C procedure call with no parameters and no return value. Column 2 shows the cost of host-to-untrusted-module RPC. Column 3 shows the cost of an untrusted-to-host-module RPC. Column 3 shows the cost of sending a byte round trip between two processes using the pipe mechanism.

C void procedure call	Host to Untrusted Module	Untrusted to Host Module	Pipe Byte Sending
0.06 µs	5.32 µs	3.22 µs	141.1 µs

6.0 Limitations, and future directions

6.1 Current System Limitations

The current system does not perform any verification or authentication of untrusted modules. To sign modules, we need to augment our trusted compiler to use a public key encryption system such as `pgp` [garfinkel:pgp].

The method of using `procfs` to trap system calls requires an extra context switch for each system call. This is not necessary if we modify the untrusted module to call RPCs in place of system calls. The RPCs would call trusted code which would replicate many of the (permitted) system call services.

We have not performed any of the optimizations described in Section 4.4. In order to perform the “guard page” optimization we probably need to implement our own “loader” untrusted modules. As previously described, using the system loader does not give us the fine grained control we need to allow for the insertion of guard pages.

6.2 Future Directions

The current system only executes on an Alpha running Digital Unix. The system should be ported to other platforms to determine the system's portability. An Alpha runtime verifier should be created to determine the start-up costs of loading a module. In the case of a network module (such as a Java Applet), its purpose may be to execute a short piece of code and terminate. An large start-up time might outweigh the use of a runtime verifier.

Although we demonstrated a prototype system that can execute untrusted modules, we have not shown a full application of the technology. We suggest three types of applications: (1) an extensible kernel such as SPIN, but using sandboxed untrusted modules to insulate the kernel from the modules, (2) applications like Adobe Photoshop or Netscape Navigator that load modules to handle different types of data on single address space operating systems such as the MacOS, (3) extensible applications, such as databases, which require user modules to handle user defined types (on UNIX-type operating systems).

7.0 Related Work

Much of the original theory presented in this paper was created by Robert Wahbe, et. al. in their paper [wahbe:sfi].

Many researchers have tried to increase RPC performance [bershad:lrpc, birrel:rpc]. RPCs are bound by the hardware limit of two context switches, and two kernel traps. Implementations such as LRPC have approached the hardware limit, thus suggesting another method for performing RPC [bershad:lrpc].

Some operating systems use type safe languages, trusted compilers and trusted linkers to make untrusted modules secure [bershad:spin]. For example, SPIN uses a type safe language for modules linked into the kernel. Although this adequately protects the kernel, and provides good performance, it limits the extension of the operating system to languages not normally used for operating system development (such as Modula-3). In addition, many other researchers are working on extensible operating systems [osdi:panel, engler:exokernel, seltzer:case].

The Omniware system compiles source files to its own Omniware virtual machine (OmniwareVM) and runtime [colusa:omniware, adl-t:omnimobile]. When the compiled objects need to be executed, they are translated to native platform objects with inserted sandboxing checks to provide fault isolation. They report that sandboxed code runs on average 9% slower than non-sandboxed code (although they do not sandbox loads).

8.0 Conclusion

We described a system for software-based fault isolation. This method creates logical fault domains within a single address space. By using a single address space, RPCs between fault domains are extremely fast because they are not bound by context switch time. In addition software fault isolation provides a method for creating multiple address spaces under operating systems that support only a single address space.

To protect memory outside of an untrusted module's logical fault domain we used a technique called *sandboxing* was used. This technique augments the code of an untrusted module so that a module can only access memory within its own fault domain. To exit its fault domain the untrusted module makes a relatively inexpensive RPC. The runtime overhead for sandboxing an untrusted module is approximately 30% on a DEC Alpha. For frequently communicating modules or applications that perform large amounts of I/O the cost of sandboxing decreases (because for inter-process communication, context switch time take the majority of time), thus making it an attractive solution. In situations, such as a kernel, where only one address space is available, software-based fault isolation provides near native speeds for untrusted modules without decreasing kernel reliability.

9.0 Acknowledgments

Wahbe et. al, wrote the first paper describing much of the work presented here. The Coca Cola Company makes Diet Coke, under whose spell this research was conducted.

This thesis is, in part, a product of my strong support network of professors, friends, and family. Professor David Kotz's class on extendible operating systems introduced me to software-based fault isolation through Wahbe et. al's paper. As my advisor, this thesis is as much his product, as it is mine. Professor Daniela Rus kept me well fed, found me a lab of my very own, and always entertained me with her answers to my my pointed questioning (or badgering). Michael Taylor always listened to me answer my own questions. Angel Mendoza always allowed me to keep him from his work when I was detoxifying from caffeine. Kartik Raghavan and Claud**e** Gayle provided me with many diversions, interesting conversations, and sanity. Joy Drake loved me, even while I was doing this thesis. My parents, who unfailingly supported me, introduced me to computers.

10.0 References

[adl-t:omnimobile]	Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient Language-Independent Mobile Programs. *PLDI: ACM SIGPLAN Conference on Programming Languages, Design and Implementation*, May 1996.
[aburto:bench]	Al Aburto. Benchmark suite. `<ftp://ftp.nosc.mil/pub/aburto>`
[bershad:lrpc]	Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1), February 1990.
[bershad:spin]	Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gun Sirer. SPIN: An extensible microkernel for application-specific operating system services. *ACM Operating Systems Review*, 29(1):74-77, January 1995.
[bershad:spin2]	Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.
[birrell:rpc]	Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39-59, February 1984.
[colusa:omniware]	Colusa Software. *Colusa Software White Paper. Omniware: A Universal Substrate for Mobile Code*, 1995.
[digital:proc]	Digital Equipment Corporation. *Digital Unix 3.2 “/proc” Manual Page.*

[engler:exokernel]	Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.
[garfinkel:pgp]	Simson Garfinkel. *PGP: Pretty Good Privacy.* O'Reilly & Associates, March 1995.
[gosling:java]	James Gosling and Henry McGilton. *The Java language: A white paper.*. Sun Microsystems, 1994.
[hpp:blackw]	Home Page Press, Inc. http://www.hpp.com. *Black Widows - Sun Declares War.*, 1996.
[may:powerpc]	Cathy May, Ed Silha, Rick Simpson, and Hank Warren. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, Inc., 1994.
[mccaine:packet]	Steven McCanne and Van Jacobsen. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proceedings of the 1993 Winter USENIX Conference*, January 1993.
[moto:68k]	Motorola Inc. *Programmer's Reference Manual.*
[osdi:panel]	Panel: Radical Operating Systems Structures For Extensibility. In *Proceedings of the USENIX Association, First Symposium on Operating Systems Design and Implementation*, 1994.
[seltzer:case]	Margo Seltzer, Chris Small, and Keith Smith. The case for extensible operating systems. *Technical Report TR-16-95, Harvard University*, 1995.
[sgi:proc]	Silicon Graphics. *IRIX 5.3 "/proc" Manual Page.*
[sites:alpha]	Richard L. Sites. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[stall:gcc] Richard Stallman. *GCC Users and Programmers Manual*. Free Software Foundation, 1995

[wahbe:sfi] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203-216, 1993.