Efficient Software-Based Fault Isolation

Robert Wahbe   Steven Lucco   Thomas E. Anderson   Susan L. Graham

Computer Science Division
University of California
Berkeley, CA 94720

Abstract

One way to provide fault isolation among cooperating software modules is to place each in its own address space. However, for tightly-coupled modules, this solution incurs prohibitive context switch overhead. In this paper, we present a software approach to implementing fault isolation within a single address space. Our approach has two parts. First, we load the code and data for a distrusted module into its own *fault domain*, a logically separate portion of the application's address space. Second, we modify the object code of a distrusted module to prevent it from writing or jumping to an address outside its fault domain. Both these software operations are portable and programming language independent.

Our approach poses a tradeoff relative to hardware fault isolation: substantially faster communication between fault domains, at a cost of slightly increased execution time for distrusted modules. We demonstrate that for frequently communicating modules, implementing fault isolation in software rather than hardware can substantially improve end-to-end application performance.

1 Introduction

Application programs often achieve extensibility by incorporating independently developed software modules. However, faults in extension code can render a software system unreliable, or even dangerous, since such faults could corrupt permanent data. To increase the reliability of these applications, an operating system can provide services that prevent faults in distrusted modules from corrupting application data. Such *fault isolation* services also facilitate software development by helping to identify sources of system failure.

For example, the POSTGRES database manager includes an extensible type system [Sto87]. Using this facility, POSTGRES queries can refer to general-purpose code that defines constructors, destructors, and predicates for user-defined data types such as geometric objects. Without fault isolation, any query that uses extension code could interfere with an unrelated query or corrupt the database.

Similarly, recent operating system research has focused on making it easier for third party vendors to enhance parts of the operating system. An example is micro-kernel design; parts of the operating system are implemented as user-level servers that can be easily modified or replaced. More generally, several systems have added extension code into the operating system, for example, the BSD network packet filter [MRA87, MJ93], application-specific virtual memory management [HC92], and Active Messages [vCGS92]. Among industry systems, Microsoft's Object Linking and Embedding system [Cla92] can link together independently developed software modules. Also, the Quark Xpress desktop publishing system [Dys92] is structured to support incorporation of

---

This work was supported in part by the National Science Foundation (CDA-8722788), Defense Advanced Research Projects Agency (DARPA) under grant MDA972-92-J-1028 and contracts DABT63-92-C-0026 and N00600-93-C-2481, the Digital Equipment Corporation (the Systems Research Center and the External Research Program), and the AT&T Foundation. Anderson was also supported by a National Science Foundation Young Investigator Award. The content of the paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

To appear in the Proceedings of the Symposium on Operating System Principles, 1993.

Email: {rwahbe, lucco, tea, graham}@cs.berkeley.edu

general-purpose third party code. As with POSTGRES, faults in extension modules can render any of these systems unreliable.

One way to provide fault isolation among cooperating software modules is to place each in its own address space. Using Remote Procedure Call (RPC) [BN84], modules in separate address spaces can call into each other through a normal procedure call interface. Hardware page tables prevent the code in one address space from corrupting the contents of another.

Unfortunately, there is a high performance cost to providing fault isolation through separate address spaces. Transferring control across protection boundaries is expensive, and does not necessarily scale with improvements in a processor's integer performance [ALBL91]. A cross-address-space RPC requires at least: a trap into the operating system kernel, copying each argument from the caller to the callee, saving and restoring registers, switching hardware address spaces (on many machines, flushing the translation lookaside buffer), and a trap back to user level. These operations must be repeated upon RPC return. The execution time overhead of an RPC, even with a highly optimized implementation, will often be two to three orders of magnitude greater than the execution time overhead of a normal procedure call [BALL90, ALBL91].

The goal of our work is to make fault isolation cheap enough that system developers can ignore its performance effect in choosing which modules to place in separate fault domains. In many cases where fault isolation would be useful, cross-domain procedure calls are frequent yet involve only a moderate amount of computation per call. In this situation it is impractical to isolate each logically separate module within its own address space, because of the cost of crossing hardware protection boundaries.

We propose a software approach to implementing fault isolation within a single address space. Our approach has two parts. First, we load the code and data for a distrusted module into its own *fault domain*, a logically separate portion of the application's address space. A fault domain, in addition to comprising a contiguous region of memory within an address space, has a unique identifier which is used to control its access to process resources such as file descriptors. Second, we modify the object code of a distrusted module to prevent it from writing or jumping to an address outside its fault domain. Program modules isolated in separate software-enforced fault domains can not modify each other's data or execute each other's code except through an explicit cross-fault-domain RPC interface.

We have identified several programming-language-independent transformation strategies that can render object code unable to escape its own code and data segments. In this paper, we concentrate on a simple transformation technique, called *sandboxing*, that only slightly increases the execution time of the modified object code. We also investigate techniques that provide more debugging information but which incur greater execution time overhead.

Our approach poses a tradeoff relative to hardware-based fault isolation. Because we eliminate the need to cross hardware boundaries, we can offer substantially lower-cost RPC between fault domains. A safe RPC in our prototype implementation takes roughly 1.1μs on a DECstation 5000/240 and roughly 0.8μs on a DEC Alpha 400, more than an order of magnitude faster than any existing RPC system. This reduction in RPC time comes at a cost of slightly increased distrusted module execution time. On a test suite including the the C SPEC92 benchmarks, sandboxing incurs an average of 4% execution time overhead on both the DECstation and the Alpha.

Software-enforced fault isolation may seem to be counter-intuitive: we are slowing down the common case (normal execution) to speed up the uncommon case (cross-domain communication). But for frequently communicating fault domains, our approach can offer substantially better end-to-end performance. To demonstrate this, we applied software-enforced fault isolation to the POSTGRES database system running the Sequoia 2000 benchmark. The benchmark makes use of the POSTGRES extensible data type system to define geometric operators. For this benchmark, the software approach reduced fault isolation overhead by more than a factor of three on a DECstation 5000/240.

A software approach also provides a tradeoff between performance and level of distrust. If some modules in a program are trusted while others are distrusted (as may be the case with extension code), only the distrusted modules incur any execution time overhead. Code in trusted domains can run at full speed. Similarly, it is possible to use our techniques to implement full security, preventing distrusted code from even *reading* data outside of its domain, at a cost of higher execution time overhead. We quantify this effect in Section 5.

The remainder of the paper is organized as follows. Section 2 provides some examples of systems that require frequent communication between fault domains. Section 3 outlines how we modify object code to prevent it from generating illegal addresses. Section 4 describes how we implement low latency cross-fault-domain RPC. Section 5 presents performance results for our prototype, and finally Section 6 discusses some related work.

2 Background

In this section, we characterize in more detail the type of application that can benefit from software-enforced fault isolation. We defer further description of the POSTGRES extensible type system until Section 5, which gives performance measurements for this application.

The operating systems community has focused considerable attention on supporting kernel extensibility. For example, the UNIX vnode interface is designed to make it easy to add a new file system into UNIX [Kle86]. Unfortunately, it is too expensive to forward every file system operation to user level, so typically new file system implementations are added directly into the kernel. (The Andrew file system is largely implemented at user level, but it maintains a kernel cache for performance [HKM+88].) Epoch's tertiary storage file system [Web93] is one example of operating system kernel code developed by a third party vendor.

Another example is user-programmable high performance I/O systems. If data is arriving on an I/O channel at a high enough rate, performance will be degraded substantially if control has to be transferred to user level to manipulate the incoming data [FP93]. Similarly, Active Messages provide high performance message handling in distributed-memory multiprocessors [vCGS92]. Typically, the message handlers are application-specific, but unless the network controller can be accessed from user level [Thi92], the message handlers must be compiled into the kernel for reasonable performance.

A user-level example is the Quark Xpress desktop publishing system. One can purchase third party software that will extend this system to perform functions unforeseen by its original designers [Dys92]. At the same time, this extensibility has caused Quark a number of problems. Because of the lack of efficient fault domains on the personal computers where Quark Xpress runs, extension modules can corrupt Quark's internal data structures. Hence, bugs in third party code can make the Quark system appear unreliable, because end-users do not distinguish among sources of system failure.

All these examples share two characteristics. First, using hardware fault isolation would result in a significant portion of the overall execution time being spent in operating system context switch code. Second, only a small amount of code is distrusted; most of the execution time is spent in trusted code. In this situation, software fault isolation is likely to be more efficient than hardware fault isolation because it sharply reduces the time spent crossing fault domain boundaries, while only slightly increasing the time spent executing the distrusted part of the application. Section 5 quantifies this trade-off between domain-crossing overhead and application execution time overhead, and demonstrates that even if domain-crossing overhead represents a modest proportion of the total application execution time, software-enforced fault isolation is cost effective.

3 Software-Enforced Fault Isolation

In this section, we outline several *software encapsulation* techniques for transforming a distrusted module so that it can not escape its fault domain. We first describe a technique that allows users to pinpoint the location of faults within a software module. Next, we introduce a technique, called *sandboxing*, that can isolate a distrusted module while only slightly increasing its execution time. Section 5 provides a performance analysis of this techinique. Finally, we present a software encapsulation technique that allows cooperating fault domains to share memory. The remainder of this discussion assumes we are operating on a RISC load/store architecture, although our techniques could be extended to handle CISCs. Section 4 describes how we implement safe and efficient cross-fault-domain RPC.

We divide an application's virtual address space into segments, aligned so that all virtual addresses within a segment share a unique pattern of upper bits, called the *segment identifier*. A fault domain consists of two segments, one for a distrusted module's code, the other for its static data, heap and stack. The specific segment addresses are determined at load time.

Software encapsulation transforms a distrusted module's object code so that it can jump only to targets in its code segment, and write only to addresses within its data segment. Hence, all legal jump targets in the distrusted module have the same upper bit pattern (segment identifier); similarly, all legal data addresses generated by the distrusted module share the same segment identifier. Separate code and data segments are necessary to prevent a module from modifying its code segment[1]. It is possible for an address with the correct segment identifier to be illegal, for instance if it refers to an unmapped page. This is caught by the normal operating system page fault mechanism.

3.1 Segment Matching

An *unsafe instruction* is any instruction that jumps to or stores to an address that can not be statically ver-

---

[1] Our system supports dynamic linking through a special interface.

ified to be within the correct segment. Most control transfer instructions, such as program-counter-relative branches, can be statically verified. Stores to static variables often use an immediate addressing mode and can be statically verified. However, jumps through registers, most commonly used to implement procedure returns, and stores that use a register to hold their target address, can not be statically verified.

A straightforward approach to preventing the use of illegal addresses is to insert checking code before every unsafe instruction. The checking code determines whether the unsafe instruction's target address has the correct segment identifier. If the check fails, the inserted code will trap to a system error routine outside the distrusted module's fault domain. We call this software encapsulation technique *segment matching*.

On typical RISC architectures, segment matching requires four instructions. Figure 1 lists a pseudo-code fragment for segment matching. The first instruction in this fragment moves the store target address into a *dedicated register*. Dedicated registers are used only by inserted code and are never modified by code in the distrusted module. They are necessary because code elsewhere in the distrusted module may arrange to jump directly to the unsafe store instruction, bypassing the inserted check. Hence, we transform all unsafe store and jump instructions to use a dedicated register.

All the software encapsulation techniques presented in this paper require dedicated registers[2]. Segment matching requires four dedicated registers: one to hold addresses in the code segment, one to hold addresses in the data segment, one to hold the segment shift amount, and one to hold the segment identifier.

Using dedicated registers may have an impact on the execution time of the distrusted module. However, since most modern RISC architectures, including the MIPS and Alpha, have at least 32 registers, we can retarget the compiler to use a smaller register set with minimal performance impact. For example, Section 5 shows that, on the DECstation 5000/240, reducing by five registers the register set available to a C compiler (gcc) did not have a significant effect on the average execution time of the SPEC92 benchmarks.

3.2 Address Sandboxing

The segment matching technique has the advantage that it can pinpoint the offending instruction. This capability is useful during software development. We can reduce runtime overhead still further, at the cost of providing no information about the source of faults.

```
dedicated-reg ⇐ target address
    Move target address into dedicated register.
scratch-reg ⇐ (dedicated-reg>>shift-reg)
    Right-shift address to get segment identifier.
    scratch-reg is not a dedicated register.
    shift-reg is a dedicated register.
compare scratch-reg and segment-reg
    segment-reg is a dedicated register.
trap if not equal
    Trap if store address is outside of segment.
store instruction uses dedicated-reg
```

Figure 1: Assembly pseudo code for segment matching.

```
dedicated-reg ⇐ target-reg&and-mask-reg
    Use dedicated register and-mask-reg
    to clear segment identifier bits.
dedicated-reg ⇐ dedicated-reg|segment-reg
    Use dedicated register segment-reg
    to set segment identifier bits.
store instruction uses dedicated-reg
```

Figure 2: Assembly pseudo code to sandbox address in `target-reg`.

Before each unsafe instruction we simply insert code that *sets* the upper bits of the target address to the correct segment identifier. We call this *sandboxing* the address. Sandboxing does not catch illegal addresses; it merely prevents them from affecting any fault domain other than the one generating the address.

Address sandboxing requires insertion of two arithmetic instructions before each unsafe store or jump instruction. The first inserted instruction clears the segment identifier bits and stores the result in a dedicated register. The second instruction sets the segment identifier to the correct value. Figure 2 lists the pseudo-code to perform this operation. As with segment matching, we modify the unsafe store or jump instruction to use the dedicated register. Since we are using a dedicated register, the distrusted module code can not produce an illegal address even by jumping to the second instruction in the sandboxing sequence; since the upper bits of the dedicated register will already contain the correct segment identifier, this second instruction will have no effect. Section 3.6 presents a simple algorithm that can verify that an object code module has been correctly sandboxed.

Address sandboxing requires five dedicated registers. One register is used to hold the segment mask, two registers are used to hold the code and data segment

---

[2] For architectures with limited register sets, such as the 80386 [Int86], it is possible to encapsulate a module using no reserved registers by restricting control flow within a fault domain.

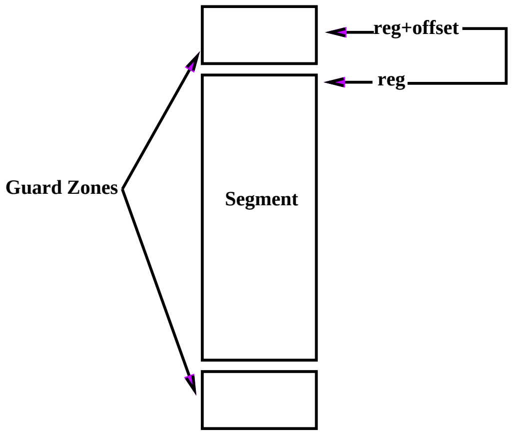

Figure 3: A segment with guard zones. The size of the guard zones covers the range of possible immediate offsets in register-plus-offset addressing modes.

identifiers, and two are used to hold the sandboxed code and data addresses.

3.3 Optimizations

The overhead of software encapsulation can be reduced by using conventional compiler optimizations. Our current prototype applies loop invariant code motion and instruction scheduling optimizations [ASU86, ACD74]. In addition to these conventional techniques, we employ a number of optimizations specialized to software encapsulation.

We can reduce the overhead of software encapsulation mechanisms by avoiding arithmetic that computes target addresses. For example, many RISC architectures include a register-plus-offset instruction mode, where the offset is an immediate constant in some limited range. On the MIPS architecture such offsets are limited to the range -64K to +64K. Consider the store instruction `store value,offset(reg)`, whose address `offset(reg)` uses the register-plus-offset addressing mode. Sandboxing this instruction requires three inserted instructions: one to sum `reg+offset` into the dedicated register, and two sandboxing instructions to set the segment identifier of the dedicated register.

Our prototype optimizes this case by sandboxing only the register `reg`, rather than the actual target address `reg+offset`, thereby saving an instruction. To support this optimization, the prototype establishes *guard zones* at the top and bottom of each segment. To create the guard zones, virtual memory pages adjacent to the segment are unmapped (see Figure 3).

We also reduce runtime overhead by treating the MIPS stack pointer as a dedicated register. We avoid sandboxing the uses of the stack pointer by sandboxing this register whenever it is set. Since uses of the stack pointer to form addresses are much more plentiful than changes to it, this optimization significantly improves performance.

Further, we can avoid sandboxing the stack pointer after it is modified by a small constant offset as long as the modified stack pointer is used as part of a load or store address before the next control transfer instruction. If the modified stack pointer has moved into a guard zone, the load or store instruction using it will cause a hardware address fault. On the DEC Alpha processor, we apply these optimizations to both the frame pointer and the stack pointer.

There are a number of further optimizations that could reduce sandboxing overhead. For example, the transformation tool could remove sandboxing sequences from loops, in cases where a store target address changes by only a small constant offset during each loop iteration. Our prototype does not yet implement these optimizations.

3.4 Process Resources

Because multiple fault domains share the same virtual address space, the fault domain implementation must prevent distrusted modules from corrupting resources that are allocated on a per-address-space basis. For example, if a fault domain is allowed to make system calls, it can close or delete files needed by other code executing in the address space, potentially causing the application as a whole to crash.

One solution is to modify the operating system to know about fault domains. On a system call or page fault, the kernel can use the program counter to determine the currently executing fault domain, and restrict resources accordingly.

To keep our prototype portable, we implemented an alternative approach. In addition to placing each distrusted module in a separate fault domain, we require distrusted modules to access system resources only through cross-fault-domain RPC. We reserve a fault domain to hold trusted *arbitration* code that determines whether a particular system call performed by some other fault domain is safe. If a distrusted module's object code performs a direct system call, we transform this call into the appropriate RPC call. In the case of an extensible application, the trusted portion of the application can make system calls directly and shares a fault domain with the arbitration code.

3.5 Data Sharing

Hardware fault isolation mechanisms can support data sharing among virtual address spaces by manipulating page table entries. Fault domains share an ad-

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase®
Smarter legal research.