The following paper was originally published in the
Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems
Portland, Oregon, June 1997

# A Tool for Constructing Safe Extensible C++ Systems

Christopher Small
Harvard University

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

# A Tool for Constructing Safe Extensible C++ Systems

Christopher Small
*Harvard University*

## Abstract

The boundary between application and system is becoming increasingly permeable. Extensible applications, such as web browsers, database systems, and operating systems, demonstrate the value of allowing end-users to extend and modify the behavior of what was formerly considered to be a static, inviolate system. Unfortunately, flexibility often comes with a cost: systems unprotected from misbehaved end-user extensions are fragile and prone to instability.

Object-oriented programming models are a good fit for the development of this kind of system. An extensions can be designed as a refinement of an existing class, and loaded into a running system. In our model, when code is downloaded into the system, it is used to replace a virtual function on an existing C++ object. Because our tool is source-language neutral, it can be used to build safe extensible systems written in other languages as well.

There are three methods commonly used to make end-user extensions safe: restrict the extension language (e.g., Java), interpret the extension language (e.g., Tcl), or combine run-time checks with a trusted environment. The third technique is the one discussed here; it offers the twin benefits of the flexibility to implement extensions in an unsafe language, such as C++, and the performance of compiled code.

MiSFIT, the Minimal i386 Software Fault Isolation Tool, can be used as the central component of a tool set for building safe extensible systems in C++. MiSFIT transforms C++ code, compiled by g++, into safe binary code. Combined with a runtime support library, the overhead of MiSFIT is an order of magnitude lower than the overhead of interpreted Java, and permits safe extensible systems to be written in C++.

## 1 Introduction

*Software fault isolation* is a technique for transforming code written in an otherwise unsafe language (e.g., C or C++) into safe compiled code. At transformation time, each read, write, and jump instruction is analyzed and, if necessary, transformed to ensure that it will not reach outside the memory region assigned to the code.

Two other techniques for ensuring the safety of code are *safe languages* and *interpreted systems*. Safe languages, such as Java and Modula-3, are designed to make it difficult or impossible to write code that performs illegal or unsafe operations. By definition, safe languages are restricted; C++, which allows unchecked array accesses, pointer arithmetic, and arbitrary casting, is implicitly unsafe.

Scripting languages, such as Tcl and Perl, enforce safety by validating each data access as it takes place. Although great strides are being made to improve the performance of interpreted languages through the use of dynamic code generation [Hölze94], the performance overhead is at least a factor of two to ten over native compiled code.

In earlier work [Small96], we measured byte-code interpreted Java taking ten to seventy times longer than compiled C code performing the same task[1]. The overhead of software fault isolation is an order of magnitude less than that of interpretation, and SFI techniques have the advantage of operating on assembler-level code, so they can be used with any source language.

Although a small number of software fault isolation tools exist, and the underlying techniques are not complex, no tools have been made freely available on commodity platforms such as the x86. MiSFIT, the Minimal i386 Software Fault Isolation Tool, developed for use with the VINO extensible operating system, is such a tool. VINO is a new operating system, written in C++, designed around the idea that system policies can be modified, and kernel components reused, by downloading extensions written by untrusted end-users and protected by MiSFIT.

MiSFIT includes runtime support necessary to create a *sandbox* in which the downloaded code will run. Additional code (not provided as part of MiSFIT) is needed to load the extension into the base system, verify

This work was supported by a grant from Sun Microsystems Laboratories and by the John Parker Scholarship Fund.

1. The tests run in that paper were re-run with MiSFIT for this paper. The results are found in Section 6.1.

that the code was processed by MiSFIT, and offer a library of routines that can be called by the extension.

MiSFIT accepts x86 assembler code, produced by the Gnu C++ compiler, as input, and produces fault-isolated x86 assembler code as output. MiSFIT can be used as a component of a safe code system, allowing otherwise untrusted code to be linked to and run in the context of an extensible application or system. For example, MiSFIT can fault isolate dynamically linked extensions to world-wide web browsers (e.g., Netscape Navigator), kernel extensions (which are supported by a variety of current systems, such as Solaris, NetBSD, MS-DOS and Windows/NT), and client code linked to a database server (e.g., the Illustra database server [Bloor96]).

Software fault isolation techniques can be implemented in a compiler pass [Silver96], a filter between the compiler and assembler (as in the case of MiSFIT), or a binary editing tool [Wahbe93]. MiSFIT works as an assembler-level filter for several reasons. First, not writing a binary editing tool simplified the task tremendously, as there was no need to parse, disassemble, patch, and reassemble x86 binary code. Another motivation was that it conforms to the Unix tool-oriented approach for building systems. By not adding it to g++, MiSFIT has a degree of compiler independence. Although MiSFIT makes a (small) number of assumptions about the format of its input, it could easily be modified to work with output from other compilers, such as lcc or Microsoft C++.

MiSFIT takes the strategy of being platform specific and language neutral; the Java Virtual Machine is both platform neutral and language neutral. We found that any need we had for platform independence was outweighed by our need for high performance and the ability to write extensions in C++.

## 2 SFI Is Not Enough

MiSFIT is not a complete solution to the problem of protection from misbehaved extensions.

First, protection from errant writes and calls is not sufficient; the application or kernel must provide a safe interface to the extension, or a safe environment in which it can run. Protection against illegal stores is useless if the extension can call **bcopy()** with arbitrary arguments. Safe equivalents of many other commonly used routines, such as **read(), write(),** and **printf()** will also be needed.

Second, and more importantly, software fault isolation (or any other memory protection mechanism) is not a substitute for a resource management strategy. An extension should not be allowed to allocate memory, obtain a lock for a critical data structure, or even be given the freedom to run on the CPU, unless some mechanism is provided for the resource to be revoked if the extension fails to release it in a reasonable amount of time. In related work [Seltzer96], we explored wrapping each extension invocation in a transaction; if the extension aborted, or failed to complete promptly, our system could abort the transaction and nullify any changes made by the extension.

The third way in which MiSFIT is not a complete solution is that it, by itself, does not ensure that a given piece of binary code has been processed by MiSFIT. There are at least two methods for solving this problem. First, extension writers can distribute source code for their extensions, and the person installing the extension could compile and MiSFIT the code before installing it. This technique may be reasonable for installing operating system extensions, as is done now with loadable kernel modules in NetBSD and Linux.

The second method is more end-user-friendly, but is logistically more complex. Code processed by MiSFIT would be given a cryptographic digital signature, either by the tool itself or by a signing authority. This signature would then be checked at load time. In order to support this scheme it would be necessary to find a trustworthy authority willing to MiSFIT and sign code, or somehow safely hide the apparatus for generating the signature within MiSFIT itself.

Although there are pieces missing from MiSFIT to make it a complete environment for building extensible systems, they are both technically tractable and application specific. For our project (the VINO extensible operating system [Seltzer94]), we have developed a protected runtime environment, resource management infrastructure, and code signature scheme[2] for use with MiSFIT. Other applications of MiSFIT would necessarily have a different safe runtime environment and resource management infrastructure.

The remainder of this paper focuses on related work, the architecture of MiSFIT, and its runtime support. Section 3 contains a discussion of related work in extension technology. Section 4 discusses the design and implementation of MiSFIT, and Section 5 covers the related runtime support. Section 6 includes the overhead of MiSFIT on benchmark programs. Section 7 discusses what has been left out of MiSFIT, and the paper conclude in Section 8.

## 3 Related Work

The term Software Fault Isolation was introduced by Wahbe et al. [Wahbe93]. They proposed a type of software fault isolation, *sandboxing*, which has low overhead on a processor with a large number of registers.

---

2. Our code signature implementation uses the RSAREF library [RSA], which is export controlled.

Their tool was originally targeted for the MIPS and Alpha processors. The initial results for this work show overheads of roughly five percent to ten percent.

A follow-on to that work is the Omniware Portable Code system. The Omniware compiler generates portable code for an abstract virtual machine (OmniVM) which is translated to native fault-isolated code at runtime [Adl96]. Along with the source language independence provided by software fault isolation techniques, the Omniware system also offers target-independent portable code.

Silver has developed a version of gcc which generates software fault isolated code for the DEC Alpha processor [Silver96]. Most of the modifications to gcc were made in the machine-independent portion of the compiler, although some changes were needed in the machine dependent portion of the code. The author reports that the implementation is dependent upon a large number of registers being available for use by the tool; a port to x86, which has a severely limited register set, appears to be difficult, if not impossible.

Several other researchers in the area of extensible operating systems have developed one-off software fault isolation tools, including Banerji [Banerji96], Engler [Engler95], and Mazieres [Mazieres96]. Unfortunately these tools suffer from working on less widely used platforms, working only with domain-specific languages, or not being publicly available.

Some extensible systems designers have followed a different route, proposing that extensions be written in a safe language (e.g., the SPIN operating system [Bershad95], which uses Modula-3 [Nelson91], and Netscape Navigator, which uses Java [Gosling96]). Safe languages can perform as well or better than software-fault-isolated unsafe languages, but have the two disadvantages that there is no possibility of reusing existing C or C++ code, and that programmers need to develop extensions in the safe language, and not the more familiar and common unsafe languages. The performance overhead of Modula-3 relative to compiled C or C++ appears to be negligible, but Java (which is most often interpreted by a virtual machine) is 20 to 50 times slower than equivalent compiled C code [Small96].

The Netscape Navigator world-wide web browser is an interesting example of an extensible system. The current release (3.0) supports two types of extensions: those written in Java (a safe language) and JavaScript (an interpreted scripting language). In order for Netscape Navigator to support extensions written in Java on all platforms, a complete implementation of the Java interpreter and runtime environment must be developed on each platform. It is arguably less work to construct a simple software fault isolation tool for a hardware architecture than to develop or port an interpreter and runtime environment.

Although in previous work we have measured interpreted Java as running ten to seventy times slower than compiled C, several companies plan to release "just-in-time" native code compilers for Java[3]. These compilers would convert Java bytecode into native code as it is loaded (or first run). The overhead of running "just-in-time" compiled code has been measured at two to ten times that of regular compiled code [Hölze94], which would give Java roughly the same performance as software fault isolated code.

Microsoft offers the ActiveX extension mechanism, which provides no technical guarantee of safety, but instead supplies only a method for verifying the identity of the provider of the code through the use of digital signatures. Software fault isolation can work in concert with digital signatures, to guarantee both the identity of the provider and the safety of the code.

The design of the VINO extensible operating system, which is the primary testbed for MiSFIT, is described in more detail in other work [Seltzer94, Seltzer96].

## 4 MiSFIT Design and Implementation

Software fault isolation can be used to protect against illegal jumps, stores, and loads. Protecting against illegal stores and jumps is necessary for correctness, but protection from illegal reads is usually a security issue, not a correctness issue. (If an extension can read outside its memory bounds, it may be able to find data it should not be allowed to see, but if an extension can write or jump to an arbitrary location in memory, the stability and correctness of the host program can be compromised.[4])

MiSFIT can be used to fault isolate indirect loads, stores, and calls. It acts as a filter, sitting between the compiler and the assembler. MiSFIT scans the output of the compiler and builds an in-memory representation for the module. It then processes each instruction of the module in turn. If any implicitly unsafe instruction (e.g., **halt**) appears, the module is rejected. The arguments for each store, call, and (optionally) load instruction are examined. (Constants and general-purpose registers are implicitly safe.) Once the module has been processed, simple peephole optimization is performed (to remove

3. Symantec has shipped a just-in-time compiler, and Sun has announced plans to do so.

4. This is not necessarily the case inside the operating system kernel; on some hardware, such as the x86, device registers are mapped into memory and reset themselves after being read.

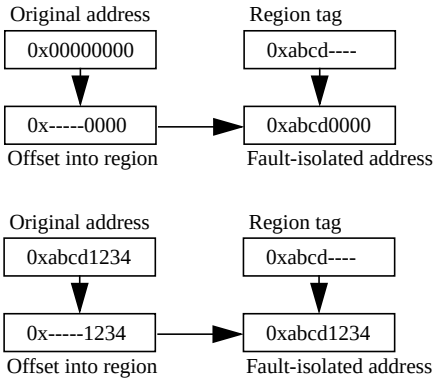


**Figure 1: Example Transformations.** In this example, the region tag is the top sixteen bits of the address and has the value 0xabcd. In the first example, the original address is invalid, so the fault-isolated address is different. In the second example, the original address is within the region, so the fault-isolated address is the same as the original address.

any redundancies introduced by the SFI transformation) and a new copy of the module is written out.

### 4.1 Indirect Loads and Stores

Loads and stores that use an indirect address that is computed at run-time are potentially unsafe. MiSFIT inserts code to *sandbox* [Wahbe93] arguments of these instructions to force the indirect address fall within a legal range.

Each user extension is assigned a contiguous region of memory into which it can write, and a region from which it can read. (These regions would normally at least overlap, if not be the same, but it is not necessary.)

MiSFIT requires that the size of each memory region be a power of two; because of this, the high bits of each address in the memory region (the *region tag*) will be the same. To sandbox a memory reference, MiSFIT simply sets the high bits of the reference to the region tag of its associated memory region. Any load or store that would have accessed memory outside its region is thus forced to fall somewhere inside the extension's memory region. Note that if the fault isolated target address was already in the extension's memory region, it does not change. The fault isolated address differs from the original target address only if the original target address was outside the extension's memory region (and therefore illegal). Examples of this transformation are given in Figure 1.

There is one more detail: in order to preclude the code from (unsafely) modifying itself, the writable

```
    movl eax,0(edx)          ; do the store
is transformed into:
    andl $0xffff,edx         ; clear old region tag
    orl destmask,edx         ; set our region tag
    movl eax,0(edx)          ; do the store

    movl eax,12(ebx,ecx) ; do the store
is transformed into:
    pushl edx                ; obtain scratch register
    leal 12(ebx,ecx),edx     ; load target address
    andl $0xffff,edx         ; clear old region tag
    orl destmask,edx         ; set our region tag
    movl eax,0(edx)          ; do the store
    popl edx                 ; restore scratch register
```

**Figure 2: Sandboxing transformations for a store instruction.** In the first case the target is a simple indirection through a register; in the second case it is a complex indirection, so a scratch register is first made available and the target is loaded into the scratch register before sandboxing. In this example, the size of the assigned memory region is 64KB (the argument to the andl is 0xffff). Note that all of the added instructions take one cycle on the Pentium (assuming that the stack targets of the push and pop are in the first level cache). Note: the general format of x86 assembler instructions is *instr src, dest*.

region should be chosen so that it does not overlap the code space assigned to the extension.

MiSFIT modifies the loads and stores in the following way. First, it inserts code to load the target address into a register (if it is not already in a register). The high bits of the register are then cleared, and the region tag of the associated memory region is then OR'd into the register. The register is then used in place of the operand in the original instruction.

Depending on whether the target address was already in a register, this technique adds either two or five instructions[5]. If the original operand is an indirection through a single register (with no constant offset) only two instructions are needed, an AND to clear the high bits of the register and an OR to set the region tag. If the target address is not already in a register, MiSFIT inserts five instructions: MiSFIT obtains a scratch register (by pushing its current value on the stack), loads the effective target address into the scratch register, masks in the region tag as above, and restores the scratch register.

Examples of these transformations are shown in Figure 2. Note that in the second case it would be possible to save the scratch register push and pop if MiSFIT

5. Each instruction is executed in one cycle on the Pentium, assuming all memory references hit in the L1 cache.

were able to determine that there was a dead register[6] available that could be used as a scratch register. The MiSFIT performance impact is low enough that we have not yet been tempted to perform this optimization.

### 4.2 Virtual Function Calls

When a virtual function call takes place MiSFIT must verify that the target address is one that the extension is permitted to call. If the extension were allowed to indirectly call to any address, it not only might obtain access to an unsafe function, it also might jump into the middle of an instruction or into data space, which would open all sorts of security and safety holes.

MiSFIT restricts the extension by searching a table of valid function targets on each indirect call from an extension. The builder of the base system provides a file with the names of the functions that an extension may call; an auxiliary tool (provided with MiSFIT) determines the start address of each of these functions at link time, and places the addresses into a table that is linked into the base system.

Although there may be an arbitrarily large number of valid target addresses, the tool greatly limits search time by storing the valid addresses in a sparsely populated open addressed hash table [Cormen90]. An open addressed hash table is implemented as an array; the hash value of the key gives the index of the array to check. When the tool adds items to the hash table, if the key hashes to *n* and location *n* of the table is already in use, it check locations *n+1*, *n+2*, and so on, until it finds a free slot for the value. When searching for a key in the table, the search function hashes the key, yielding *n*, and then check location *n* of the table. If location *n* has a value (but not the key) it checks location *n+1*, *n+2*, and so on, until it either find the key (signifying success) or find an empty slot (signifying failure).

One subtle advantage of using an open addressed hash table is that if the search function does not find the key at location *n*, because the next location checked (at index *n+1)* is at an adjacent memory location, it is likely to be in the cache. So, even if it fails on the first probe of the table, the cost of subsequent probes is reduced.

By decreasing the density of the table, it is possible to reduce the number of probes needed nearly to unity (the theoretical minimum). With a table that has a 50% density (half the slots are empty) an average of fewer than 1.5 probes per indirect call are required. The overhead of each probe is roughly six to ten cycles (assuming everything hits in the L1 cache), adding, on average, approximately ten to fifteen cycles to each indirect function call.

Indirect calls are common in C++ code, as virtual functions are implemented as indirect calls. When protecting C++ code with MiSFIT the table of valid function targets can become quite large, but the per-invocation cost remains low, because the number of probes into the table is independent of the size of the table, depending only on its density, which is under MiSFIT's control.

### 4.3 Global Data, Virtual Function Tables

Because MiSFIT sandboxes global memory references, any data accessible to the extension must be placed in the memory region assigned to the extension. If there is global data that the extension should be able to access, the data should be placed in the memory region assigned to the extension. This applies not only to global program data, but other shared state, such as virtual function tables.

The restriction on global program data is a problem if multiple extensions are to be granted access to the same datum. A work-around is for the application to provide functions to access the data; each extension will be given permission to call these accessor functions, and use them instead of directly reading and writing the data.

This technique has an impact on performance that is difficult to quantify, as the cost is a function of the amount of data that is protected in this way, the frequency of access, and the type of interface the functions provide. In two of the three tests discussed in this paper this cost is not to quantified; in the third, the cost is built in to the overall model, but not factored and measured separately.

Virtual function tables are a different matter. If MiSFIT is configured to use read protection, virtual function tables need to be in a region of memory that is readable by the extension. The solution we have chosen for VINO is to store all virtual function tables in a contiguous region of memory (by making a one-line change to g++), and mapping that region into each extensions read-only region.

### 4.4 Block Instructions

The x86 instruction set includes memory-to-memory move and comparison instructions, **movs** and **cmps**, which take four or five clock cycles on the Pentium. The same goal can be accomplished by four one cycle instructions (assuming a scratch register is available). However, the memory to memory instructions have the advantage that they can be used to construct *block* move and compare sequences. The x86 **rep** instruction can be used as a prefix to the memory-to-memory instructions;

6. A dead register is one that will not be read again before it is written.

the **rep** prefix instructs the processor to repeat the memory-to-memory instruction for *count* times, where *count* is the value in the %ecx register. The block move instruction sequence has a lower per-move overhead than a sequence or loop of individual memory-to-memory move instructions, and can be generated by compilers to perform structure copies and in-line expansions of common C library functions such as **strcmp()** and **bcopy**().

MiSFIT transforms the base addresses and repeat count of arguments to the block instruction, sandboxing the compound instruction as a whole. Although this adds a high fixed overhead to the block instruction (roughly 26 cycles), there is no per-element cost. The alternative, transforming the block instruction into a loop and sandboxing the instructions in the loop, has a high per-element overhead; the break-even point for the two techniques is at three or four iterations. Block instructions are typically used for copying or moving more than four elements, so the fixed overhead imposed by MiSFIT's technique is preferable.

### 4.5 Saved Registers and Return Addresses

Protecting the contents of the stack is also problematic. The stack is used not only for local variables (which must be accessible to the user extension) but also saved registers and the function return address (which should not be accessible to the user extension). If the user extension could write to arbitrary locations on the stack, the return address of the function could be overwritten and set to an arbitrary value, circumventing the call protection offered by MiSFIT.

A second problem is that the process stack is normally not in the same region of memory as the heap and global data; MiSFIT's technique depends on all valid memory references falling within a single region of memory. In a multi-threaded environment (either a multi-threaded operating system kernel or multi-threaded end-user application) each thread of control is assigned its own stack. In environments where the extension can be run as a separate thread of control, MiSFIT can co-locate the stack assigned to the thread (i.e. assigned to the extension) with the memory region assigned to the extension. Then all valid memory references made by the extension will fall within a single region.

In environments where there is a single thread of control, MiSFIT can provide the same type of protection by providing each extension with its own stack, located in its memory region. When the extension is invoked, the application switches to the stack associated with the extension. When the extension returns to the application, the process switches back to the original stack.

To solve the problem of an extension overwriting a return address on the stack, MiSFIT replaces each **call** instruction within the extension with a call to a support routine that saves the return address in a separate stack outside the extension's memory region and then jumps to the called function. MiSFIT then replaces each **ret** instruction with a jump to a second support routine that loads the saved return address and jumps to it. In this way, even if the extension misbehaves and overwrites the return address, the system returns to the correct location. To ensure that register values are preserved across the invocation of the extension, MiSFIT stores the contents of all callee-saved registers on entry to the extension, and reloads these values when it returns.

### 4.6 Dynamic Linking

MiSFIT modifies the operands of load, store, and call instructions that are computed at runtime. It does not modify operands that are labels, assuming that references to addresses within the module (i.e. local jumps, and loads and stores of module-level variables) are implicitly safe (generated by the compiler), and references to addresses outside the module will be checked by the dynamic linker when they are resolved. This implies that the dynamic linker is responsible for keeping track of which symbols may be linked to by an extension. Under some circumstances it may be the case that not all extensions will be given access to the same set of entrypoints. If this is the case, the dynamic linker is responsible for determining to which entrypoints a given extension should be given access.

Relinquishing responsibility for protecting external symbols has a limitation. The assembler does not mark external symbols as being for read or write use; a single external reference is generated for all reads and writes. If there is no read protection, but there is write protection, there is no way for the linker to discern which references are source (read) references and which are destination (write) references – in other words, which should be allowed, and which should be disallowed.

To solve this problem, MiSFIT generates a table of addresses of instructions that write operands that are labels. The dynamic linker can use the information in this table, in addition with the external reference table, to differentiate between read references and write references at link time.

### 4.7 An Alternative to Sandboxing on the x86

On the x86, an alternative to sandboxing exists. The **bound** instruction checks that a value falls within a specified range; if it does not, a trap occurs. If this trap can be caught, the ill-behaved extension can be stopped before it does any damage. It appears that the **bound** instruction was designed to be used for array bounds

checking. since it performs a signed (rather than unsigned) comparison. This does not preclude using it for SFI. MiSFIT might arrange for all parts of the region of memory assigned to an extension have the same sign (i.e. not cross the border between location 0x7fffffff and location 0x80000000), so the signed nature of the comparison would not be a problem. The bound instruction takes more cycles than the instructions needed to set the high bits of a register (eight vs. two); however, instead of neutering an illegal load or store the bound instruction would trap an illegal memory access.

This paper includes results of running tests comparing the performance of MiSFIT using the **bound** instruction and the sandboxing technique described in Section 4.1. The results show that the sandboxing method has superior performance, which is not surprising, considering the cost of the **bound** instruction in comparison with the cost of sandboxing.

# 5 Runtime Support

MiSFIT includes runtime support for linking extension code as new virtual functions to existing objects, setting up the state of an extension, and managing free store for the extension.

## 5.1 Virtual Function Table Manipulation

In the MiSFIT model, an extension is used to modify the behavior of a single object, by replacing a virtual function of that object. MiSFIT accomplishes this by making a copy of the virtual function table for that object and writing a new value into the slot corresponding to the replaced function.

The process by which an extension is called is somewhat baroque. MiSFIT can not just replace the address of the old function with the address of the newly loaded function in the virtual function table. As outlined above, when an extension is called its sandbox needs to be configured.

## 5.2 Calling The Extension

When an extension is installed, a small assembler stub function (similar to a closure) is created. This stub is responsible for configuring the sandbox and calling the extension. The stub is specific to the particular extension, because it includes the addresses of the extension's sandbox regions, as well as the address of the extension function itself.

The stub sets up the sandbox for the extension. It first saves callee-saved registers (as MiSFIT does not trust the extension to do so). The stub sets up the global variables that hold the region tags for the read and write (source and destination) regions assigned to the extension, and copies any arguments passed to the extension onto the extension's stack. It switches to the extension's stack, and jumps to the extension.

When the extension completes, it jumps to the returns stub (remember that the extension's **ret** instruction was replaced by this jump, as described in Section 4.5), which switches to the regular stack, loads the saved registers, and returns to the base system.

The runtime support code also includes the function that implements safe indirect calls (as described in Section 4.2). MiSFIT replaces indirect calls with code that loads the target function address and calls the hash table lookup function. If the function address is not found in the hash table by the lookup function, the function calls an **abort** function, which is responsible for cleaning up after the extension.

## 5.3 Extension Free Store Management

As the code running in an extension cannot reach outside its bounds, if it were to allocate storage using **new** it would not be able to read from nor write to that storage. MiSFIT provides a small heap in the data area assigned to the extension, and simple implementations of the built-in **new** and **delete** functions. When MiSFIT is processing an extension, it replaces any calls to the built-in **new** and **delete** functions with calls to the MiSFIT versions.

# 6 MiSFIT Overhead

This section compares the performance of unprotected code (written in C or C++) with the MiSFIT-protected versions. Times are reported as a percentage of the unprotected versions. Performance numbers for both write-call (where store and call instructions are protected) and read-write-call (where load, store, and call instructions are protected) tests are included. As pointed out above, read protection is typically a requirement for security, not for correctness.

## 6.1 Operating System Extensions

In previous work [Small96], we examined the suitability of various extension technologies for constructing operating system extensions. Three tests were developed and used, with each test representing a class of possible OS extensions. Following is a short description of each test; for more detail, the reader is directed to the earlier paper.

- *hotlist:* choose which page to evict from a linked list of page descriptors.
- *lld:* simulate the operation of a logical disk layer [DeJon93].
- *md5:* compute the MD5 checksum [RFC1321] of 1MB of data.

The tests were run on a 120MHz Pentium with 64MB of EDO memory, running BSD/OS 2.1. Each test

and its data fit into main memory. Times are reported relative to the unprotected version of the code. The results are found in Table 1.

The write-call overhead for these tests is low, at most 10%. The overhead for read-write-call protection can be much higher, over 200%.

In our earlier work we computed a break-even point for each operating system extension. If the cost of using the extension is below the break-even point, the extension will improve overall system performance; if it above this point, it will degrade system performance. The three write-call protected tests fall below the break-even point, as do the read-write-call versions of *lld* and *md5*, but the read-write-call version of *hotlist* does not.

| Test | MiSFIT Write-Call Protected (MiSFIT/ unprotected) | MiSFIT Read-Write-Call Protected (MiSFIT/ unprotected) |
|---|---|---|
| *hotlist* | 1.00 | 3.2 |
| *lld* | 1.07 | 1.4 |
| *md5* | 1.09 | 1.7 |

**Table 1:** Relative overhead of MiSFIT-protected code to unprotected code on operating system extension benchmarks. The cost of isolating writes and indirect writes is low, under 10%, but the cost of protecting reads as well can be prohibitively high.

The performance of the write-call protected *hotlist* is equivalent to the unprotected version. This is because there are very few protected write instructions executed during the test. Because the kernel of the test repeatedly scans a linked list of page descriptors, the number of read instructions executed is very high. This bias is reflected in the performance of the read-write-call protected version of this test, where the overhead is more than 200%.

The *lld* test has a noticeable but small write-call overhead of 7%; read protection adds another 33%. This test is not as read-intensive as *hotlist*, so the added overhead of read protection is much lower. The *md5* test has similar performance characteristics, with a sub-10% write-call overhead, and an additional 60% overhead for read protection.

### 6.2 SPECInt92

This experiment shows the results of several SPECInt92 benchmarks processed by MiSFIT, using write-call and read-write-call protection. The performance of the MiSFIT-protected code relative to native code is reported in Table 2.

Although it is unlikely that anyone would want to load a SPEC benchmark into a web browser or database server, these results give a feeling for the overhead imposed by MiSFIT on "typical" code. (To better estimate the overhead imposed by MiSFIT, the tables only include time spent at user level.)

| Test | MiSFIT Write-Call Protected (MiSFIT/ unprotected) | MiSFIT Read-Write-Call Protected (MiSFIT/ unprotected) |
|---|---|---|
| *compress* | 1.09 | 1.26 |
| *espresso* | 1.15 | 1.76 |
| *eqntott* | 1.02 | 1.68 |
| *li* | 1.17 | 1.61 |

**Table 2:** Overhead of protection on SPECInt benchmarks for MiSFIT, relative to unprotected code. MiSFIT times are the mean of ten runs. Standard deviations were less than 1%, except for *compress*, where it was 2.6%.

The write-call MiSFIT overhead for the SPEC92Int code is comparable to that of MiSFIT on the operating system extension benchmarks, ranging from a factor of 1.02 to a factor of 1.17. As is seen above, the overhead of read-write-call protection is higher than the overhead for write-call protection, on the order of 1.26 to 1.76. This overhead is large, but still substantially less than that of an interpreted language.

For memory-intensive applications, such as data copies, a higher overhead should be expected. The overhead seen is, of course, a function of the ratio of protected instructions to unprotected instructions.

### 6.3 VINO Kernel Extensions

MiSFIT is used to protect the VINO operating system kernel from misbehaved end-user extensions. We measured the performance overhead of MiSFIT on four kernel extensions [Seltzer96], and include these results here. For these tests we used MiSFIT for read-write-call protection, and the overhead shown is in line with the overhead seen above.

The *Read-ahead* extension specifies which disk block to read next, by returning a value found in its memory region. This code performs little computation, so the overhead imposed by protecting its loads and stores dominate its performance.

The *Page Eviction* extension is similar to the *hotlist* extension described in Section 6.1, but instead of searching a linked list it searches an array. Because there is less pointer chasing, the overhead imposed by MiSFIT is lower.

Each time the *Scheduling* extension is called it searches a list of 64 process IDs. Because the code that traverses the list is trusted code (is part of the base sys-

tem, outside the extension itself, unlike in the case of *hotlist*), the overhead of using MiSFIT is much lower[7].

The fourth extension, which performs a simple encryption of a data stream, is data intensive. It copies 8KB of data from an input buffer to an output buffer, applying a trivial (XOR-style) encryption to the data.

This extension was designed to be a worst-case test for MiSFIT, with little computation done between each data load and store. The MiSFIT version of the code takes slightly more than twice as long as the unprotected code. It is theoretically possible for MiSFIT-protected code of this form to take as much as six times as long as protected code (remember MiSFIT can add five instructions for each load and store), but it is difficult, if not impossible, to construct a real-world example where every instruction is a load or store. One possible case is when data is being copied directly from one buffer to another (as is done in this example), but the overhead seen here is 100%, not 500%. In the case of a straight data copy (using the x86 **rep; movs** instruction pair), MiSFIT uses a different technique for fault isolation which has lower overhead (see Section 4.4).

| Test | MiSFIT Read-Write-Call Protected (MiSFIT/unprotected) |
|---|---|
| *Read-ahead* | 2.5 |
| *Page Eviction* | 1.2 |
| *Scheduling* | 1.1 |
| *Encryption* | 2.1 |

**Table 3: VINO Kernel Extensions:** MiSFIT was used to apply read-write-call protection, which causes overhead in line with the results seen above. Each test was run between 300 and 3000 times; the standard deviation of each result was less than 2.5%.

### 6.4 Performance Summary

With read-write-call protection MiSFIT protected code can take from 1.4 to 3.2 times as long as unprotected code. Although this overhead may seem large, it should be compared to the overhead of an interpreted safe language, such as current Java implementations (which are 20 to 50 times slower than compiled C code), or the disadvantage of writing extensions in an unfamiliar, but safe, compiled language, such as Modula-3.

7. Calling trusted code outside the extension is analogous to a Java application calling native methods, which are implemented in compiled C or C++.

## 7 What Is Missing

As shown in Section 2, SFI is not a complete solution. The MiSFIT package does not include a safe runtime support library, which would be specific to the base system. This support library would be responsible for ensuring that extensions do not violate their resource limitations.

Extensions do not have access to the global heap; a version of **malloc** (or **new**) is needed that allocates memory from a pool inside the extension's writable memory region.

MiSFIT does not include a dynamic linker. Depending on its application, a dynamic linker may already be part of the system (e.g., NetBSD). The dynamic linker, or some code-signing tool, would be responsible for verifying that the loaded code had been processed by MiSFIT.

One restriction that is not currently addressed, but should be, is the difficulty of passing arguments and returns by reference. When calling an extension, the calling stub pushes arguments onto the extension's stack, but these arguments are currently restricted to immediate values. If the base system wants to pass an argument by reference (via a pointer) there is currently no way to do so. Additionally, there is no way for an extension to pass back data other than as the return value of the function or by storing the results in its writable memory region for later retrieval by the base system.

The solution to this limitation is the application of standard techniques for marshalling and unmarshalling arguments for remote procedure calls. By specifying the number and types of parameters to the extensions with an interface definition language, extension-specific stub functions could be generated that would copy arguments into the extension's address space when it is called, and copy results back to the base system when it returns.

## 8 Conclusions

The overhead imposed by MiSFIT when it is used for write and call protection is small. It allows applications and kernels to be protected from end-user extensions written in otherwise unsafe languages. Unlike other tools, it is freely available. As part of an end-to-end solution to the problem of constructing an extensible system, MiSFIT can provide safety at low cost.

## 9 Availability

MiSFIT is covered by a BSD-style license, and is available for public use without fee. Contact the author (chris@eecs.harvard.edu) to obtain a copy of the code.

## 10 Acknowledgments and Apology

The members of the VINO Operating System project, Prof. Margo Seltzer, Keith Smith, Yasuhiro Endo, and David Holland, are implicitly included and thanked wherever first-person plural is used in this paper. Keith Smith's eagle-eyed proofreading of the final draft was greatly appreciated by the author.

To circumvent use of both the awkward-sounding first-person singular and the royal "we," work done by the author has been ascribed either to the paper itself or directly to MiSFIT. Only where completely unavoidable was passive voice used. The author apologizes, profusely, to the linguistically offended.

## Bibliography

[Adl96] Adl-Tabatabai, A., Langdale, G., Lucco, S., Wahbe, R., "Efficient and Language-Independent Mobile Programs," *PLDI '96*, Philadelphia, PA, 127-136, May 1996.

[Banerji96] Banerji, A., Panteleenko, V., Wyant, G., Cohn, D., "Quantitative Analysis of Protection Options," University of Dame Technical Report TR-96-20, 1996.

[Bloor96] Bloor, R., "The Capabilities of Illustra and its Integration with Informix DSA," http://www.informix.com/informix/corpinfo/zines/whitpprs/bloor/contents.htm

[Bershad95] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., Chambers, C., "Extensibility, Safety, and Performance in the SPIN Operating System," *Proc. 15th SOSP*, Copper Mountain, CO, 267–284, December 1995.

[Cormen90] Cormen, T., Leiserson, C., Rivest, R., *Introduction to Algorithms*, 232-241, The MIT Press, Cambridge MA, 1990.

[DeJonge93] de Jonge, W., Kaashoek, M. F., Hsieh, W., "The Logical Disk: A New Approach to Improving File Systems," *Proc. 14th SOSP*, Asheville, NC, 15–28, December 1993.

[Engler95] Engler, D., Kaashoek, M. F., O'Toole, J., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. 15th SOSP*, Copper Mountain, CO, 251–266, December 1995.

[Fall93] Fall, K., Pasquale, J., "Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability," *1993 Winter USENIX Conference*, San Diego, CA, 327–334, January 1993.

[Gosling96] Gosling, J., Joy, B., Steele, G., *The Java Language Specification*, Addison-Wesley, Reading, MA, 1996.

[Hölze94] Hölze, U., Ungar, D., "Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback", *PLDI '94*, Orlando, FL, June 1994.

[Mazieres96] Mazieres, D., personal communication.

[Nelson91] Nelson, G., ed., *Systems Programming with Modula-3*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm," *Network Working Group RFC 1321*, April 1992.

[RSA] ftp://ftp.rsa.com/rsaref

[Seltzer94] Seltzer, M., Endo, Y., Small, C., Smith, K., "An Introduction to the VINO Architecture," Harvard University Computer Science Technical Report TR-34-94, 1994.

[Seltzer96] Seltzer, M., Endo, Y., Small, C., Smith. K., "Dealing With Disaster: Surviving Misbehaved Kernel Extensions," *Proc. 2nd OSDI*, Seattle, WA, 213–228, October 1996.

[Silver96] Silver, S., "Implementation and Analysis of Software-Based Fault Isolation," Dartmouth College Technical Report PCS-TR96-287, 1996.

[Small96] Small, C., Seltzer, M., "A Comparison of OS Extension Technologies," *Proc. 1996 USENIX Technical Conference*, New Orleans, LA, 41-54, January 1996.

[Wahbe93] Wahbe, R., Lucco, S., Anderson, T., Graham, S., "Efficient Software-Based Fault Isolation," *Proc. 14th SOSP*, Asheville, NC, 203–216, December 1993.