MiSFIT: A Tool for Constructing Safe Extensible C++ Systems

Christopher Small and Margo Seltzer
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 extension can be designed as a refinement to 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 a component of a tool set for building safe extensible systems in C++. MiSFIT transforms C++ code, compiled by the Gnu C++ compiler, 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,[1] the performance overhead is at least a factor of two to ten more than native compiled code.

In earlier work, we measured byte-code interpreted Java taking ten to seventy times longer than compiled C code performing the same task.[2] 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.

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

MiSFIT accepts as input x86 assembler code from the Gnu C++ compiler, and produces as output fault-isolated x86 assembler code. 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).

Software fault isolation techniques can be implemented in a compiler pass,[3] a filter between the compiler and assembler, or a binary editing tool.[4] MiSFIT was implemented as an assembler-level filter for several reasons.

First, the task was tremendously simplified. An x86 binary editing tool needs to parse, disassemble, patch, and reassemble x86 binary code. A new pass for g++ would require spending time learning and understanding

the current version of g++, and tracking changes to g++ as it evolves.

Second, our implementation model conforms to the Unix tool-oriented approach for building systems. By not building MiSFIT into g++, it gains 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++. 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 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.[5]

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) we have developed a protected runtime environment, resource management infrastructure, and code signature scheme for use with MiSFIT. Other applications of MiSFIT would necessarily have a different safe runtime environment and resource management infrastructure.

3 Related Work

The term Software Fault Isolation was introduced by Wahbe et al..[4] They proposed a type of software fault isolation, *sandboxing*, which has low overhead on a processor with a large number of registers. Their tool was originally targeted for the MIPS and Alpha processors. The initial results of their work show overheads of roughly five 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.[6] 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, the Gnu C compiler, that generates software fault isolated code for the DEC Alpha processor.[3] 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 implementation takes advantage of the large number of registers available on the Alpha processor. A port to x86, which has a severely limited register set, would be difficult.

Several other researchers in the area of extensible operating systems have developed one-off software fault isolation tools, including Banerji[7] and Engler.[8] 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,[9] which uses Modula-3, and Netscape Navigator, which uses Java). 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 per-

formance overhead of Modula-3 relative to compiled C or C++ appears to be negligible, but interpreted Java is 20 to 50 times slower than equivalent compiled C code.[2]

Netscape Navigator supports extensions written in Java. Because of this, a complete implementation of the Java interpreter and its runtime environment must be available on each platform. It is arguably less work to construct a simple software fault isolation tool for a given platform than it is to develop or port a Java interpreter and runtime environment.

In previous work we have seen interpreted Java running ten to seventy times slower than compiled C. Several "just-in-time" native code compilers for Java are available, which convert Java bytecode into native code as it is loaded or first run. "Just-in-time" compiled code has been shown to take two to ten times as long as conventionally compiled code,[1] which would give Java roughly the same performance as code protected by MiSFIT.

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 techniques can be used in concert with digital signatures, to guarantee both the identity of the provider and the safety of the code.

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.)

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 and, if necessary, transformed. Once the module has been processed, simple peephole optimization is performed (to remove any redundancies introduced by the transformations).

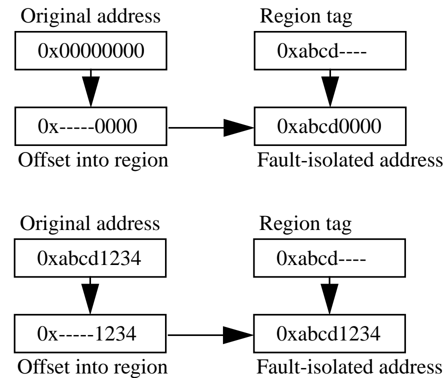

Figure 1: Example Transformations. In this example, the region tag is the top sixteen bits of the address and has the value 0xabcd. In the first example, the original address is invalid, so the fault-isolated address is different. In the second example, the original address is within the region, so the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

4.1 Indirect Loads and Stores

Loads and stores that use an indirect address (one computed at run-time) are potentially unsafe. MiSFIT inserts code to sandbox arguments of these instructions to force the indirect address, to 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 overlap, but it is not necessary that they do so.) In order to preclude the code from modifying itself (and thus potentially circumventing MiSFIT's transformations), the writable region must not overlap with the region assigned to the extension's code.

MiSFIT requires that each memory region be a power of two bytes in size. Because of this, the high bits of all addresses in the memory region (the *region tag*) will be the same. To sandbox a memory reference, MiSFIT inserts code that sets the high bits of the reference so that it matches 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, the target address 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 shown in Figure 1.

MiSFIT modifies the loads and stores in the following way. First, it inserts code to load the target address

```
    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 here is *instr src, dest*.

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. 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 were able to determine that there was a register available for use 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. Runtime code (provided with MiSFIT) creates this table at program startup, using the application's symbol table to find the start address of each function that extensions may call.

Although there may be an arbitrarily large number of valid target addresses, search time is limited by storing the valid addresses in a sparse, open addressed, hash table, which has near-linear search time.

An open addressed hash table is implemented as an array, and the hash value of the key gives the index of the array to check. When an item is added to the table, the insertion function hashes the key to some value n. If location n of the table is already in use, the insertion function check locations $n+1$, $n+2$, and so on, until it finds a free slot. When searching the table, the search function hashes the key, yielding n, and then checks 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 finds the key (signifying success) or finds 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 is at an adjacent memory location (at index $n+1$), it is likely to be in the cache. So, even if the search function 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 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. We do not quantify this cost in the tests discussed in this paper.

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 the read-only region of each extension.

4.4 Block Instructions

Unlike RISC processors, the x86 includes memory-to-memory move and comparison instructions, **movs** and **cmps**. These instructions can be used to construct *block* move and compare sequences, using the x86 **rep** instruction as a prefix. 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 used to perform structure copies and in-line expansions of **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 **movs** instructions. Block instructions are typically used for copying or moving more than four data 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 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 can write to arbitrary locations on the stack, the return address of the function can be 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 stores the return address of the function in per-thread state stored outside the extension's writable region. MiSFIT then replaces each **ret** instruction with a jump to a support routine that loads the saved return address and jumps to it. If the extension overwrites the contents of its stack, the system still returns to the correct location.

Similarly, to ensure that register values are preserved across the invocation of the extension, MiSFIT saves the contents of all callee-saved registers on entry to an extension, and restores them 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 the extension is loaded. 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 so, 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; the same type of external reference marker is generated by the assembler for all reads and writes. If there is no read protection, but there is write protection, there is no way for the dynamic linker to determine 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 Optimizations

In general, we have found that the performance of MiSFIT-protected code is close enough to that of unprotected code that we have not been tempted to implement complex code optimizations. However, two simple code optimizations are performed.

The first is a peephole optimization that removes or performs strength reduction on pop/push instruction pairs, which are generated by the sandboxing code in order to obtain a scratch register. If the register that is popped and immediately pushed is known to be dead at the point of the pop (because the immediately following code is sandboxing code that will overwrite it) both instructions can be removed. If the register is not known to be dead, the pop/push pair is transformed into a single instruction that loads the value from the top of the stack into the register.

The second optimization performs live/dead analysis of the condition codes (stored in the x86 flags register) and uses this information to determine whether it is necessary to save and restore the condition codes when a register is sandboxed. The latter optimization is both effective and important: we have found no instances where the state of the condition codes needs to be saved when sandboxing, and saving and restoring the flags register on the x86 is an expensive operation.

4.8 Stubs

When an extension is invoked, a small stub function, similar to an RPC stub, is called. This stub is responsible for configuring the extension's sandbox, saving callee-saved registers and initializing the global variables that hold the region tags for the read and write regions assigned to the extension. It copies any arguments passed to the extension onto the extension's stack, switches the stack pointer to the extension's stack, and jumps to the extension. When the extension completes, the stub copies any returned values to the caller's stack and then returns to the caller.

The stub generator is driven by annotated C++ class declarations, which specify parameters as input, output, or in/out. The stub generator uses standard techniques for creating the stubs. Unlike RPC stubs, arguments do not need to be marshalled, but simply copied from the caller's stack to the extension's stack (on entry) and the results copied back (on exit), hence the MiSFIT stubs can have less overhead than RPC stubs.

5 MiSFIT Overhead

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

The following tests were run on a 200MHz Pentium Pro running BSD/OS 3.0. All results are the mean of 20 runs, and in all cases the standard deviation was less than 1.1%, which yields a 95% confidence interval of less than 2.5%. The values reported are relative to unprotected code. In order to best estimate the overhead of protection, time spent in the operating system (and hence unprotected code) was factored out.

5.1 Operating System Extension Benchmarks

In previous work we examined the suitability of various extension technologies for constructing operating system extensions.[2] 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.[10]
- *md5:* compute the MD5 checksum of 1MB of data.

Each test and its data fit into main memory. The results are shown in Table 1.

The write-call overhead for these tests ranges from 11% to 27%, but the overhead for read-write-call protection can be much higher, for example, 144% for *hotlist*.

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 far below the break-even point, and the read-write-call protected of *lld* and *md5* are both below it. The read-write-call protected *hotlist* is at or just above the break-even point for this example extension.

The performance of the write-call protected *hotlist* is close to that of the unprotected version. This is because there are only a small number of calls and very few write instructions executed during the test. The core of the test repeatedly scans a linked list of page descriptors, hence 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 over 140%.

The *lld* test has a write-call overhead of 27%, and read protection adds another 24%. 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 23% write-call overhead, and an additional 14% overhead for read protection.

Test	MiSFIT Write-Call Protected (MiSFIT/ unprotected)	MiSFIT Read-Write-Call Protected (MiSFIT/ unprotected)
hotlist	1.11	2.44
lld	1.27	1.51
md5	1.23	1.37

Table 1: Overhead of MiSFIT protection on operating system extension benchmarks.

5.2 SPECInt Benchmarks

This experiment shows the performance overhead of MiSFIT protection on several SPEC benchmarks from the 1992 and 1995 integer suites. We ran tests using write-call and read-write-call protection. The results are shown in Table 2, and are relative to unprotected code.

It is highly unlikely that anyone would want to load a SPEC benchmark into a web browser or database server. However, 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.

The write-call MiSFIT overhead for the SPECInt code is comparable to that of MiSFIT on the operating system extension benchmarks, ranging from a factor of 28% to 59%. As is seen above, the overhead of read-write-call protection is higher than the overhead for write-call protection, from 73% to 93%.

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.

Test	MiSFIT Write-Call Protected (MiSFIT/ unprotected)	MiSFIT Read-Write-Call Protected (MiSFIT/ unprotected)
compress95	1.59	1.73
eqntott92	1.28	1.76
espresso92	1.63	1.80
go95	1.45	1.93

Table 2: Overhead of MiSFIT protection on SPECInt benchmarks. Results only include time spent at user level.

5.3 Performance Summary

With read-write-call protection, MiSFIT-protected code takes from 37% to 144% times as long as unprotected code. Although this overhead may seem large, it should be compared to the overhead of an interpreted language, which can be 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.

6 Putting Together the Pieces

As discussed in Section 2, SFI in and of itself 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 are usually dynamically linked, and to further that end, a minimal ELF dynamic linker is included with the MiSFIT distribution. The dynamic linker can load object files that have been processed by MiSFIT into a running program. The linker is derived from the VINO graft loader.

7 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.

8 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 MiSFIT source code or the benchmarks used in this paper.

References

1. U. Hölze and D. Ungar, "Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback", *PLDI '94,* Orlando, FL, June 1994.
2. C. Small and M. Seltzer, "A Comparison of OS Extension Technologies," *Proc. 1996 USENIX Technical Conference,* New Orleans, LA, January 1996, pp 41-54.
3. S. Silver, "Implementation and Analysis of Software-Based Fault Isolation," Dartmouth College Technical Report PCS-TR96-287, 1996.
4. R. Wahbe et al., "Efficient Software-Based Fault Isolation," *Proc. 14th SOSP,* Asheville, NC, December 1993, pp. 203–216.
5. M. Seltzer et al., "Dealing With Disaster: Surviving Misbehaved Kernel Extensions," *Proc. 2nd OSDI,* Seattle, WA, October 1996, pp. 203–216.
6. A. Adl-Tabatabai et al., "Efficient and Language-Independent Mobile Programs," *PLDI '96*, Philadelphia, PA, May 1996, pp. 127-136
7. A. Banerji et al., "Quantitative Analysis of Protection Options," University of Dame Technical Report TR-96-20, 1996.
8. D. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. 15th SOSP*, Copper Mountain, CO, December 1995, pp. 251–266.
9. B. Bershad et al., "Extensibility, Safety, and Performance in the SPIN Operating System," *Proc. 15th SOSP*, Copper Mountain, CO, December 1995, pp. 267–284.
10. W. de Jonge, M. F. Kaashoek, and W. Hsieh, "The Logical Disk: A New Approach to Improving File Systems," *Proc. 14th SOSP*, Asheville, NC, December 1993, pp. 15–28.