



MiSFIT: A Freely Available Tool for Building Safe Extensible Systems

Citation

Small, Christopher. 1996. MiSFIT: A Freely Available Tool for Building Safe Extensible Systems. Harvard Computer Science Group Technical Report TR-07-96.

Link

<http://nrs.harvard.edu/urn-3:HUL.InstRepos:24827567>

Terms of use

This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material (LAA), as set forth at

<https://harvardwiki.atlassian.net/wiki/external/NGY5NDE4ZjgzNTc5NDQzMGIzZWZhMGFIOWI2M2EwYTg>

Accessibility

<https://accessibility.huit.harvard.edu/digital-accessibility-policy>

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. [Submit a story](#)

MiSFIT: A Freely Available Tool for Building Safe Extensible Systems

Christopher Small

Harvard University Center for Research in Computing Technology

Technical Report 07-96

Abstract

The boundary between application and system is becoming increasingly permeable. Extensible database systems, operating systems, and applications, such as web browsers, are demonstrating the value of allowing end-users to extend and modify the behavior of what formerly was 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.

There are three common methods for making end-user extensions safe: restrict the extension language, interpret the extension language, or add run-time checks to binary code that ensure the safety of an otherwise unsafe program. The third technique, *software fault isolation*, offers the twin benefits of the performance of compiled code and the flexibility to choose an unsafe language, such as C or C++.

MiSFIT, a tool for software fault isolation of x86 code, transforms unsafe C or C++ into safe binary code. The performance overhead of using MiSFIT to protect against stray writes and arbitrary function calls is low, on the order of ten percent.

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. Each read, write, and jump instruction is modified or isolated to ensure that it will not reach outside the assigned memory region.

Two other techniques for ensuring the safety of code are safe languages and interpreted systems. Safe languages, such as Modula-3 and Java are designed to make it difficult or impossible to write code that performs illegal or unsafe operations. Scripting languages, such as Tcl and Perl, enforce safety by validating each data access as it takes place.

By definition, safe languages are restricted; C and C++, which allow unchecked array accesses, pointer arithmetic, and arbitrary casting, are unsafe. And the

performance of interpreted languages suffers from the overhead of interpretation. Although great strides are being made in improving 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.

Software fault isolation techniques have the advantage of operating on assembler-level code, so they can be used with any source language. And software fault isolation adds a much smaller overhead than interpretation, from five to one hundred percent [Wahbe93].

Although a small number of software fault isolation tools exist, and the techniques are not complex, no tools have been made freely available on commodity platforms such as the x86.

MiSFIT, the Minimal intel Software Fault Isolation Tool, operates on the assembler output of the Gnu C and C++ compilers (gcc and g++). It accepts x86 assembler code as input, and produces fault-isolated x86 assembler code as output. It 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. MiSFIT can be used to isolate dynamically linked kernel extensions (which are supported by a variety of current systems, such as Solaris, NetBSD, MS-DOS and Windows/NT), client code linked to a database server (e.g., the POSTGRES database [Ston87]) and extensions to world-wide web browsers (e.g., Netscape Navigator).

Software fault isolation techniques can be implemented in a compiler pass [Sil96], a filter between the compiler and assembler, or a binary editing tool [Wahbe93]. We implemented MiSFIT as an assembler-level filter for several reasons. First, working at the symbolic assembler level simplified the task tremendously; there was no need to parse, disassemble, patch, and reassemble x86 binary code. Second, it allows MiSFIT a degree of compiler independence; although MiSFIT makes a small number of assumptions about the format of its input, the tool could easily be ported to accept input from other compilers, e.g., lcc or a PC C compiler. We discuss the architecture of MiSFIT in more detail in Section 3.

This research was supported by Sun Microsystems Laboratories.

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

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.

As another example, an extension should not be allowed to obtain a lock for a critical data structure, unless some mechanism is provided for the lock to be revoked if the extension fails to release it in a reasonable amount of time. In related work [Selt96] 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. We found that the combined overhead of MiSFIT and the transaction system was acceptably low for our needs.

The second way in which MiSFIT is not a complete solution to the problem is that it is a stand-alone tool; we include no mechanism for ensuring that a given piece of binary code has been processed by MiSFIT. We see MiSFIT as part of an end-to-end solution of the problem of extension safety. We see two ways in which MiSFIT can be used. One method is to distribute source code for extensions, and have the person installing the extension compile and MiSFIT the code before installing it. (This method is reasonable for installing operating system extensions, as is done now with loadable kernel modules in NetBSD and Linux.) The second method we see is to have the compilation system apply a cryptographic digital signature to the generated binary code, which could be checked at the time the extension is installed.

The remainder of the paper discusses the architecture of MiSFIT and the overhead associated with using it in several environments. In Section 2, we discuss related work in safe extensible technology. In Section 3, we discuss the design and implementation of MiSFIT; the following section includes the overhead of MiSFIT on benchmark programs. In Section 5, we discuss the use of MiSFIT for constructing a safe extension to the NetBSD kernel. We conclude in Section 6.

2 Related Work

The term Software Fault Isolation was first put forward 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 (e.g., a modern RISC processor). Their tool was targeted for the MIPS and Alpha processors.

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. Omniware was developed by Colusa Software, which has been acquired by Microsoft. The future availability of Omniware as a product is not clear.

Silver has developed a version of gcc which generates software fault isolated code for the DEC Alpha processor [Sil96]. 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. He 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 an extremely limited register set, is thought 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 [Ban96], Engler [Engl95], and Mazieres [personal communication]. 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 uses Modula-3 [Bers95]). Safe languages can perform as well or better than software fault isolated unsafe languages, but have the disadvantages that there is no possibility of reusing existing code, and that programmers need to develop extensions in the safe language, and not more familiar and common unsafe languages.

The Netscape Navigator world-wide web browser is an interesting case in point. The current release 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. 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. In addition, by using software fault isolation techniques, existing C and C++ code could be used extension developers.

3 MiSFIT Design and Implementation

Software fault isolation can be used to protect against illegal stores, loads, and jumps/calls. Protecting against illegal stores and jumps is necessary for correctness. Note that 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; if an extension can write or jump to an arbitrary location in memory, the stability and correctness of the program can be compromised.¹

MiSFIT can be used to fault isolate indirect loads, stores, and calls. It acts as a filter, sitting between the compiler and the assembler. It 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. Validity checking of label arguments is postponed until link time; the external symbols of the module need to be checked against the symbols exported by the linked-to system.

3.1 Indirect Loads and Stores

Loads and stores that indirect through a register, memory location, or run-time computed value are potentially unsafe. MiSFIT inserts code to *sandbox* [Wahbe93] arguments of this type to force such operands to fall within legal ranges.

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 a requirement.) We require 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 are the same. To sandbox a memory reference, we force the high bits of the reference to match those of its associated memory region.

MiSFIT modifies the extension code in the following way. It inserts code to first load the target address into a register. The high bits of the associated memory region are then masked into the register. The register is then used in place of the operand in the original instruction.

1. This is not necessarily the case inside the operating system kernel; on some hardware, device registers reset themselves after being read.

Depending on the form of the original instruction, this technique adds two or five simple instructions (which each take one cycle on the Pentium). If the original operand is an indirection through a single register (with no constant offset) two instructions are needed, an AND to clear the high bits of the register and an OR to set the high bits to the appropriate value. If the original operand is not in a register, we need to add five instructions; we obtain a scratch register (by pushing its current value on the stack), load the effective target address into the scratch register, mask the high bits as above, and restore the previous value to the scratch register. Examples of these transformations are shown in Figure .

```
movl %eax,0(%edx)
becomes:
andl $0xffff,%edx
orl destmask,%edx
movl %eax,0(%edx)

movl %eax,12(%ebx,%ecx)
becomes:
pushl %edx
leal 12(%ebx,%ecx),%edx
andl $0xffff,%edx
orl destmask,%edx
movl %eax,0(%edx)
popl %edx
```

Figure 1: 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, we assume that the size of the assigned memory region is 4KB (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*.

It would be possible to save the push and pop instructions if we were able to analyze the code and determine statically if there was a dead register² that could be used as a scratch register. We found that the performance impact of the transformation as described above is sufficiently low that we have not yet been tempted to perform this optimization.

On the x86, an alternative to sandboxing exists. The bound instruction checks that a value falls within a specified range. It appears to have been designed to be used for array bounds checking; it performs a signed,

2. A dead register is one that will not be read again before being written.

rather than unsigned, comparison. Note that we could arrange that 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.

MiSFIT currently generates sandboxing code, although an earlier version used the bound technique. We found that the sandboxing method has superior performance. The measurements presented in this paper were taken using the sandboxing technique.

3.2 Indirect Calls

When an indirect function call takes place (i.e. through a function pointer) we must verify that the target address is a function that the extension is permitted to call. We accomplish this by generating a table of valid function targets at compile time, and searching the table for the presence of a target address on an indirect call. Although there may be an arbitrarily large number of valid target addresses, we limit the search time by storing the valid addresses in a sparse hash table. By reducing the density of the table we can reduce the number of probes needed on each call nearly to unity. With a table that has a 50% density we see an average of less 1.5 probes. The overhead of each probe is on the order of six to ten cycles, adding on average approximately ten to fifteen cycles to each indirect function call.

We found that this type of call is infrequently made in typical C code. One common place it is used is when calling `qsort()` and similar functions; this type of function takes a pointer to a comparison function as an argument and calls the comparison function while sorting. Indirect calls are much more 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. The per-invocation cost remains low, however, because the number of probes into the sparse hash table is more or less independent of size, but instead dependent upon the density of the table.

3.3 Global Data

Because we sandbox global memory references, any data accessible to the extension must be placed in the memory region assigned to the extension. If there is glo-

bal data that the extension should be able to access, the data should be placed in the memory region assigned to the extension.

This limitation is problematic if multiple extensions are to be granted access to a single global datum. Our solution is for the system 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; 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 our three tests we do not attempt to quantify this added cost; in the third, the cost is built in to the overall model, and not factored and measured separately.

3.4 Stack Protection

Protecting the contents of the stack is more 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; our 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, we 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, we 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, we replace each `call` instruction within the extension with a call to a support routine that saves the return address outside the

extension's memory region and jumps to the called function. We then replace 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, we store the contents of all callee-saved registers on entry to the extension, and reload these values when it returns.

3.5 Block Instructions

The x86 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; 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 **strcpy()** and **bcopy()**.

Our solution to transform 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 our technique is preferable.

3.6 Dynamic Linking

MiSFIT modifies the operands of load, store, and call instructions which 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, 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 which entrypoints a given extension should be given access to.

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 can generate 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 MiSFIT Overhead

We compare 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. Both write-call and read-write-call performance numbers are included; as pointed out above, read-protection is typically a requirement for security, not for correctness.

Where it is possible, we compare the performance of MiSFIT protected code with that of Omniware-protected code. This is not precisely an apples-to-apples comparison; Omniware is a portable code system that performs run-time compilation and software fault isolation; MiSFIT is an x86-specific software fault isolation tool. However, the overhead of Omniware (which is impressively low) gives us a performance target.

4.1 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. We include 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 easily fits into main memory. We report times relative to the unprotected version of the code. The results are found in Table 1.

Test	MiSFIT Write-Call Protected (MiSFIT/ unprotected)	MiSFIT Read-Write-Call Protected (MiSFIT/ unprotected)
<i>hotlist</i>	1.00	3.2
<i>lld</i>	1.07	1.4
<i>md5</i>	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.

We see that 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; the read-write-call versions of *lld* and *md5* do as well, but the read-write-call version of *hotlist* does not.

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 we see an overhead of over 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.

The write-call overhead seen here is similar to that found when running an early version of Omniware with write-call protection on a Sparc [Small96]. The relative run-time for the Omniware protected version of *hotlist*

was 1.4, for *lld* was 1.16, and for *md5* was 1.5. Note that these results are not directly comparable to the MiSFIT overhead reported here; the earlier results came from running an older version of Omniware on a different processor. (In the following section we compare MiSFIT with a more current version of Omniware.)

4.2 SPECInt92

We compiled several SPECInt92 benchmarks with 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, along with Omniware Pentium performance numbers for the same tests. The Omniware numbers were obtained from recently published work [Adl96].

Although it is unlikely that anyone would want to load a SPEC benchmark into a database server or web browser, these results give a feeling for the overhead imposed by MiSFIT on “typical” code.

Test	MiSFIT Write-Call	MiSFIT Read-Write-Call	Omniware Write-Call
<i>compress</i>	1.09	1.26	1.02
<i>espresso</i>	1.15	1.76	<i>not avail</i>
<i>eqntott</i>	1.02	1.68	1.06
<i>li</i>	1.17	1.61	1.11

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

To conservatively estimate the overhead imposed by MiSFIT, our results include only time spent at user level.

We see that 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, we can expect to see a higher overhead. The overhead seen is, of course, a function of the ratio of protected instructions to unprotected instructions.

5 Performance of a Kernel Extension

To given an example of the performance of MiSFIT in-situ, we constructed an extension to the NetBSD kernel that adds a system call to short-circuit data copies from

one file descriptor to another. This example has been explored before, as a possible feature to add to existing systems [Fall93] or as an example extension for extensible operating systems [Bers95].

NetBSD supports dynamic addition of system calls while the kernel is running. We added a new system call that takes as arguments two file descriptors and a count; the new call copies count bytes from the first file descriptor to the second, without requiring multiple kernel-user protection boundary crossings. We protected this code with MiSFIT and measured the performance.

We saw that not only was the MiSFIT overhead small, the savings relative to a user level process was also quite low. We attribute this to the fact that our extension was using a fairly high-level interface within the kernel; it was essentially calling the read and write system calls, albeit from kernel level.

Test	Time
<i>user-level syscall interface</i>	45.4s (1%)
<i>unprotected kernel-level syscall interface</i>	45.2s (0.4%)
<i>MiSFIT kernel-level syscall interface</i>	45.2s (0.1%)

Table 3: Copying 8MB of data from one file to another on the local disk, using 8KB blocks. The user-level test repeatedly invoked the read and write system calls. The kernel-level syscall interface tests called the functions that implement the read and write system calls. Standard deviations are listed in parenthesis; the mean of five runs is reported.

The test platform was a 486 DX2-66 with 8MB of 60ns memory and a 3600RPM IDE disk. The results are summarized in Table 3.

Looking at the first three rows of the table, we see that although the benefit of moving the copy loop into the kernel saved very little time (roughly half a percent, on the order of the standard deviation), the overhead added by MiSFIT in this case was not measurable.

6 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.

7 Availability

MiSFIT is covered by a BSD-style license, and is available for public use. Contact the author (chris@eecs.harvard.edu).

Bibliography

- [Adl96] Adl-Tabatabai, A., Langdale, G., Lucco, S., Wahbe, R., “Efficient and Language-Independent Mobile Programs,” *PLDI* ‘96, 127-136 (1996).
- [Ban96] Banerji, A., Panteleenko, V., Wyant, G., Cohn, D., “Quantitative Analysis of Protection Options,” University of Notre Dame Technical Report TR-96-20 (1996).
- [Bers95] 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,” *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.
- [DeJon93] de Jonge, W., Kaashoek, M. F., Hsieh, W., “The Logical Disk: A New Approach to Improving File Systems,” *Proceedings of the 14th SOSP*, 15–28, Asheville, NC (December 1993).
- [Engl95] Engler, D., Kaashoek, M. F., O’Toole, J., “Exokernel: An Operating System Architecture for Application-Level Resource Management,” *Proceedings of Fifteenth SOSP*, 251–266 (1995).
- [Fall93] Fall, K., Pasquale, J., “Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability,” *1993 Winter USENIX Conference*, 327–334, San Diego, CA (January 1993).
- [Hölz94] Hölz, U., Ungar, D., “Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback,” *Proceedings of the 1994 SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, FL (June 1994).
- [Nels91] *Systems Programming with Modula-3*, Nelson, G., ed., Prentice Hall, Englewood Cliffs, NJ (1991).
- [RFC1321] Rivest, R., “The MD5 Message-Digest Algorithm,” *Network Working Group RFC 1321* (April 1992).

- [Selt96] Seltzer, M., Endo, Y., Small, C., Smith, K., “Dealing With Disaster: Surviving Misbehaved Kernel Extensions,” *Proceedings of the Second OSDI* (October 1996).
- [Sil96] 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,” *Proceedings of the 1996 USENIX Technical Conference*, New Orleans, LA, 41-54 (1996).
- [Ston87] Stonebraker, M., “Extensibility in POSTGRES,” *IEEE Data Engineering*, September 1987.
- [Wahbe93] Wahbe, R., Lucco, S., Anderson, T., Graham, S., “Efficient Software-Based Fault Isolation,” *Proceedings of the 14th SOSP*, 203–216 Asheville, NC (December 1993).