A Tool for Constructing Safe Extensible C++ Systems

Christopher Small
Presented by Ian Murdock
November 1, 1999

1 Introduction

This paper describes the design and implementation of MiSFIT, a tool used to construct safe extensions in C++ for extensible applications and operating systems. Originally developed for the VINO extensible operating system, MiSFIT uses software fault isolation techniques to provide memory protection and is implemented as a filter between the compiler and assembler, taking the assembly output from the compiler and producing fault-isolated assembly code.

2 Related Work

MiSFIT uses software fault isolation, in concert with a safe runtime environment, to make application extensions safe. Other techniques include using a safe language (e.g., Modula-3), interpreting the extension language (e.g., Tcl), or using some combination of the two (e.g., Java). Safe languages perform as well as or better than software-fault-isolated unsafe languages, but have the drawback that users may only write extensions in the system-supported extension language. Interpreted languages have the same drawback and, additionally, have very high overhead. For example, current Java implementations are between 20 and 50 times slower than compiled C code. Dynamic code generation helps improve the performance of interpreted systems, bringing overheads down to a factor of two to ten over compiled code.

3 Operation

MiSFIT first assigns the extension a contiguous region of memory that is read-only and a region that is read-write (these regions may overlap). Each memory region has a size that is a power of two, so that all addresses within the memory region have the same high-order bits, called the region tag. MiSFIT then scans each assembly instruction in turn. If an implicitly unsafe instruction (e.g., HALT) is encountered, the extension is rejected. The arguments for each jump and store (and, optionally, load\(^1\)) instruction are examined, to ensure that memory accesses and control transfers fall within the bounds of the extension’s memory region, and software fault isolation transformations are applied as necessary. After the entire extension has been processed, simple peephole optimization is performed to remove redundant instructions introduced by the fault isolation transformations, and the modified assembly code is passed to the assembler for translation into machine code. To ensure that an extension cannot modify itself to “undo” the transformations, the extension’s code is placed in the read-only memory region.

The following subsections address a few of the issues that complicate the fault isolation process.

\(^1\)Limiting jumps and stores to the bounds of the memory region is necessary for correctness, whereas limiting loads is normally a security issue, not a correctness issue.
3.1 Indirect loads and stores

Indirect loads and stores are potentially unsafe. Without care, an extension could arrange to read or write an arbitrary memory location simply by setting the indirection register accordingly. To prevent this, MiSFIT sandboxes indirect loads and stores, setting the high-order bits of the address such that the memory reference is forced to be within the extension’s memory region (i.e., setting them to the region tag). For example, if the region tag is 0xabcd, the address 0x00000000 is sandboxed to 0xabcd0000, while the address 0xabcd1234 is sandboxed to 0xabcd1234. Note that both addresses, after sandboxing, fall properly within the extension’s memory region.

MiSFIT requires either two or five instructions to sandbox an indirect memory reference. If the target address is in a register, MiSFIT (1) clears the high-orders bits of the address; and (2) sets them to the region tag. If the target address is not in a register, MiSFIT (1) pushes the contents of a register onto the stack to obtain a scratch register; (2) loads the target address into the scratch register; (3) and (4) sets the region tag as above, then performs the store using the scratch register; and (5) pops the saved register from the stack to restore its original value.

3.2 Virtual functions calls

C++ virtual functions calls are implemented using indirect jumps. Again, without some care, an extension could arrange to jump to arbitrary points in the code or data, such as past the sandboxing instructions inserted before indirect stores, compromising the safety of the application being extended. To prevent this, MiSFIT inserts code to verify that the address of each indirect call is legal. The application’s runtime system provides MiSFIT with a list of functions that extensions may call, and MiSFIT generates an open hash table that allows indirect addresses to be checked for validity.

The overhead of this approach depends on the density of the hash table; a densely-populated table will result in more probes on average than one that is populated sparsely. The author reports that this approach adds approximately ten to fifteen cycles to each indirect call, though this figure seems to be based on the assumption that all memory accesses needed to search the hash table hit in the L1 cache, which may be true for subsequent probes of the table after the first (due to the adjacency property of open hash tables), but it is probably not true for the first itself.

3.3 Global variables

MiSFIT sandboxes all memory references, including references to global variables. Thus, global variables must be located in an extension’s memory region if it needs to access them. This approach presents a problem if multiple extensions need to access the same global variable. This problem applies to other shared state as well, such as virtual function tables.

One workaround is to implement accessor functions that allow extensions to manipulate the shared state, rather than allowing extensions direct access to it. Another approach is to map shared state into the extension’s memory region. This approach is used by VINO to provide extensions with shared access to virtual function tables.

3.4 Stack

The stack is used not only to store local variables, but also saved state like the return address of the function and callee-saved registers. An extension must not be allowed to change these values, or it may compromise the safety of the surrounding system. For example, an extension could overwrite the return address on the stack with an arbitrary address to cause control to return to a disallowed location. A further complication is that the stack is normally not in the same area of memory than the code and static data; in order for sandboxing to work, the stack needs to be located into the extension’s memory region, as with global variables.

To solve these problems, MiSFIT provides each extension with its own stack located within its assigned memory region. When the extension is called, the application stores the return address and callee-saved registers on its own stack, and then switches to the extension’s stack. On exit, the stack is switched back, callee-saved registers are restored, and flow of control is returned to the appropriate location.
A similar problem arises with the heap; in order for an extension to safely allocate and deallocate memory, it must do so within its memory region. Thus, MiSFIT provides each extension with its own heap, located inside the extension’s memory region, and replaces the `new` and `delete` operations with its own versions that manipulate it.

4 Other issues

It is important to note that MiSFIT is not a complete mechanism for providing safe extensibility on its own. There are several other issues that need to be addressed to completely guarantee safety.

First of all, the application must provide a safe interface to the surrounding system or a safe environment in which the extension may run, or the safety of the application may be compromised. For example, it is useless to provide memory protection inside the extension if it may call `bcopy` with arbitrary arguments.

Second, the application being extended must carefully control the resources available to an extension. Even if memory protection and a safe environment are provided, the extension may still adversely affect the surrounding system by acquiring a lock and never releasing it, or by going into an infinite loop.

Finally, because MiSFIT performs its instrumentation at compile time, there must be a mechanism in place for verifying that code being loaded has been instrumented properly. The paper suggests adding a cryptographic digital signature to code generated by MiSFIT, and verifying that the signature is valid when code is loaded. It is not clear from the paper, however, if this has been implemented.

All of these issues are addressed in other work.

5 Performance

The authors ran several benchmarks on code processed by MiSFIT to measure its overhead. In summary, the overhead for write-call protection is small, on the order of a few percent, while the overhead for read-write-call protection is substantially higher, resulting in code that is between 1.4 and 3.2 times slower than code that is unprotected. This is acceptable, given the high overhead of alternatives such as Java, which is 20 to 50 times slower than compiled C code.

6 Conclusion

MiSFIT is a tool for constructing safe extensions in C++ for extensible applications and operating systems. Working in combination with a safe runtime system, MiSFIT allows users to write extensions that impose a small amount of overhead to provide write and call protection, and an acceptable amount of additional overhead to protect reads as well when compared to alternatives such as Java.