# Transparent Fault Isolation in Plugins using Dynamic Compilation

Peter Feiner peter@cs.toronto.edu
Angela Demke Brown demke@cs.toronto.edu
Ashvin Goel ashvin@eecg.toronto.edu

University of Toronto

Many large programs are designed using a plugin architecture. The core of the program, which we call the application (even in the case of operating systems where it is normally referred to as the kernel), interacts with modules that it loads, which we call plugins, to implement the program's features. Faults within plugins can incur down time or manifest as errors that propagate into application state. By isolating faults in the plugins that cause them, the potential for errors propagating into application state is reduced. Several techniques exist for isolating faults in application plugins, but they all require some effort on the part of the plugin programmer to use. In this research, we implement a new fault isolation technique that is completely transparent to plugin programmers.

The effort required by existing fault isolation techniques ranges from simply recompiling plugins to rewriting the application's plugin interface. In general, fault isolation techniques try to prevent the improper modification of data and the arbitrary execution of code. Traditional software fault isolation (SFI), by way of a special compiler, restricts a plugin by checking that its writes address the plugin's memory and that its branches target the plugin's instructions. SFI only permits interaction between the application and the plugin through special pass-by-value interfaces, precluding commonplace shared memory communication.

Recently, isolation techniques have been developed that allow plugins to use existing application interfaces. Nooks [1] simulates shared memory by copying interface parameters between hardware isolated address spaces. However, Nooks is limited to plugins that do not access global application data structures. Byte-Granularity Isolation (BGI) [2] places restrictions on plugin memory operations for each byte in memory: a plugin may write to its local variables and heap, but it has limited access to application data. A plugin is only granted temporary access to application data structures while it interacts with the application, for example during the execution of a call by the application into a plugin. To keep track of these restrictions, BGI wraps the application's interfaces with permission-granting and permission-revoking code. To enforce the restrictions, BGI uses a special compiler that prefaces memory accesses with permission-checking instructions. To restrict control flow, BGI stores execution permissions for each byte in memory and prefaces dynamic branches with checking code.

BGI is attractive because it allows existing interfaces, but it is not transparent because it requires recompilation. Our technique follows BGI but removes the need for a special compiler. Rather than compiling plugin code to check permissions, we employ dynamic compilation to add permission-checking code to existing x86 plugin binaries. Note that program shepherding, which also uses dynamic compilation, only enforces control flow conventions – it does not limit instructions that modify memory.

Not requiring compilation introduces some tradeoffs. First, rules that BGI derives from plugin source code cannot be enforced by our technique. The additional rules limit how a plugin executes internally, for example, by limiting calls to function entry points. However, to isolate plugins, it is not necessary to prohibit behavior that does not affect the application – although prohibiting such behavior is useful while debugging plugins. Thus, in spite of having fewer rules, our technique is sound. Second, controlling code generation affords BGI many performance optimizations that cannot be easily applied to our technique. For instance, BGI reduces memory used to store permissions by laying code and data out in permission-sharing groups of 8 bytes. Our approach uses 8 times the memory of BGI for permission data, which reduces cache performance. In another optimization, BGI omits checks for local variable accesses. We are developing heuristics to identify such accesses and safely omit checks for them. Our heuristics involve a simple, yet novel, algorithm for tracking the boundaries of plugins' stacks.

Currently, we have implemented a prototype of our fault isolation technique and applied it to plugins for the Firefox Web browser. Microbenchmarks of our prototype demonstrate the potential for good performance. Our current work focuses on enhancing our implementation's performance by improving our heuristics for omitting local variable checks; improving cache performance by experimenting with data structure layouts; and reducing the number of dynamic branches in permission-checking instructions, to alleviate pressure on the branch prediction hardware.

[1] M. M. Swift, et al. Improving the Reliability of Commodity Operating Systems. In *SOSP* 2003.

[2] M. Castro, et al. Fast Byte-Granularity Software Fault Isolation. In *SOSP* 2009.