Critique of

"Integrating segmentation and paging protection for safe, efficient and transparent software extensions"

by Chieueh, Venkitachalam and Pradhan

November 21, 2000

1 Introduction

This paper introduces a protection mechanism christened *Palladium*, a prototype of which has been implemented on Linux. The aim is to take advantage of both segmentation and paging in the Intel x86 architecture to provide an efficient and secure method for executing potentially untrusted extension code.

I was somewhat disappointed with the quality of this paper, compared to the other paper we reviewed and compared to other papers that I have read. Most of this critique discusses these shortcomings, although in the process I also cover its achievements.

2 Two Ideas, One Paper

The paper intertwines discussion of two somewhat different extension mechanisms - one suitable for kernel extensions, the other for user extensions.

The kernel extension mechanism is based purely on segmentation - core kernel code and data are assigned the traditional Segment Privilege Level (SPL) of 0, while special segments are created for extensions, with SPL of 1. This higher, less privileged, SPL prevents the extensions from accessing and potentially corrupting kernel data. Any kernel services that need to be called from the extension can be accessed via call gates (registered cross-domain entrypoints supported by the hardware), in a similar fashion to system calls. Any data that needs to be shared is placed by the kernel into a special part of the extension's segment.

On the surface, the user extension scheme is similar, in that core application code and data are assigned an SPL of 2, while extensions are assigned a less privileged SPL of 3. Call gates can also be registered, to allow extensions to access application services. However, to facilitate easier sharing, the extension's segment now refers to the entire address space, and not just to the extension's own part of memory. Instead, Page Protection Level (PPL) is used to control access to memory - the extensions, running at SPL 3, can only access pages with PPL 1. Pages which are not shared with the extension are set to PPL 0, protecting them.

While the motivations for using this second scheme for user code are discussed - primarily easier sharing - the paper never explains why the improved scheme cannot be used for kernel code. As far as I can see the only reason is the limitations of the Intel architecture, which only differentiates SPL 3 when applying page-level protection. Running kernel extensions in SPL 3 (essentially, at user level) would potentially allow user programs to subvert them. The fact that this is a limitation of the architecture is something that should be pointed out, as it may be possible to implement the more advanced scheme for kernel code on other architectures such as HP PA-RISC.

In any case, at many points it is not clear which scheme is being discussed. The paper could have potentially been improved by separating the two mechanisms into two distinct sections, first discussing the segmentation idea which is at the core of both schemes, with relevant benchmarks, and only then introducing the sharing optimisations possible for user code via the use of Page Protection Levels.

3 Apples and Oranges

As is typical of many papers, some of the benchmarks can be misleading, as they are not clearly defined and compare data which it may not be sensible to directly compare.

For instance, Figure 7 compares two very different kernel packet filters - a compiled filter extension using the Palladium model, and an interpreted Berkeley Packet Filter (BPF) filter. The intent is clearly to demonstrate that protection via cross-domain call, and in particular the model employed by Palladium, is more efficient than protection via interpretation. Obviously as the number of terms (the work to be done) increases, the overhead of a single cross-domain call becomes less significant than the overhead of per-term interpretation.

While this is a feasible comparision in terms of demonstrating potentially 'real-world' performance, the two filters are no doubt very different, and the compiled code would have been optimised for the type of filter expressions being used (which are not specified in the paper). Also, I would argue that this graph is insufficient data for what is really the only direct benchmark of the kernel extension model. For instance, the unprotected call cost to the compiled code could have been measured.

Table 1, displaying the cycle costs of various stages of inter and intra domain calls, is poorly presented. The "Hardware" column should certainly be next to the "Inter" column, since it relates to the hardware cost of inter-domain not intra-domain calls. The paper states, without justification, that the difference between these measured and theoretical counts is "due to data/control pipeline hazards", having claimed before that all their tests are run with "warm caches". While this may indeed be the case, some of the differences seem fairly large, and L1 cache misses could also be involved - even though the L1 cache would no doubt be larger than the working set of the benchmark, conflict misses may still occur.

Table 2 shows the times for unprotected, Palladium and Linux RPC calls to a string reversing function, for various string sizes. While it demonstrates that Palladium incurs a procedure call overhead of approximately 0.59 μs, it is mainly measuring the performance of the function to reverse strings, rather than of Palladium. The comparison with Linux RPC, which is implemented via sockets and not optimised for intra-machine RPC, is not reasonable. It is obvious that Palladium, a hardware-supported protected procedure call mechanism, will outperform marshalling RPC over sockets by an order of magnitude.

4 The Fish-Eye Lens

From the outset, the paper is Intel and Linux centric. While this is partly justifiable, I would hope for a SOSP paper to consider such operating system design issues more generally.

There is a brief mention of HP PA-RISC as "providing the most comprehensive protection and security hardware among modern RISC machines", and it is implied that a similar scheme could be developed on that architecture, since it has privilege levels similar to Intel x86. However, this is not developed.

Similarly, the authors admit that Mungi has "fast protected procedure calls", which is what this paper is about, but nowhere is this developed or its design considered. Rather, it is dismissed in one sentence as only running MIPS R4600 processors.

Indeed, no benchmarks or direct comparisons are provided against any of the classic research operating systems which provide extension mechanisms and cross-domain procedure calls - for instance VINO or SPIN. The closest is a comparison with an L4 "request-reply IPC". This is slightly misleading as this is, of course, two IPC's in the L4 model. Some procedure calls may not need to return data and will thus be faster on L4.

5 Lies, Damn Lies and Exaggerations

The authors state that "explicit data copying is still inevitable in L4 for processes to share data". This is simply not true, and shows their lack of understanding of L4. L4 IPC allows page mappings to be transferred between processes, and indeed this is one of the most useful features.

"Extension programming is identical to kernel module programming, except that they can build on the set of core kernel services exposed to kernel extensions". From this it seems that we only gain the advantage of extra services, whereas the opposite is true. Most existing Linux kernel modules liberally use kernel functions. In the Palladium model, we are limited to the functions that are exported by the kernel, and incur an expensive cross-domain call to access them.

6 Unanswered Questions

It is stated that `seg_dlsym` "generates" the necessary stub code for cross-domain calls (`Prepare` and `Transfer`). It is not clear where these functions are generated, particularly as `Prepare` needs to be at SPL 2 and `Transfer` at SPL 3. Ideally, these would be located so as to avoid incurring extra TLB misses and wasting TLB entries. This paper noticably avoids discussing architecture issues such as TLB and cache optimisation.

It is also not clear why all applications are not run at SPL 2, rather than starting applications at SPL 3 and requiring an `init_PL` call for promotion. To prevent extensions accessing system calls, this then requires storing the application SPL in the process control block, possibly adding some small overhead to system calls to load it (although unlike in the highly optimised L4, the relative overhead for such a PCB load in Linux will be minor). The cursory reason given for not starting applications at SPL 2 is to "avoid disruptions to the large number of existing Linux applications". This does seem an odd statement, however, as applications cannot rely on the SPL which they are run at.

7 Conclusion

While the paper is interesting as far as presenting a feasible mechanism for intra-process access control on the Intel x86 architecture, it has a number of shortcomings. The structure is fairly ad hoc, with related work presented only cursorily and not really considered in the context of the proposed scheme. The two different mechanisms that are presented are intertwined excessively. The benchmarks are limited and not particularly demonstrative. Finally, the paper lies about L4 - a great sin in our research group.