The Cake is a Lie: Privilege Rings as a Policy Resource

Sergey Bratus[1], Peter C. Johnson[1], Michael E. Locasto[2], Ashwin Ramaswamy[1], and Sean W. Smith[1]

[1]Dartmouth College

[2]George Mason University

ABSTRACT

Components of commodity OS kernels typically execute at the same privilege level. Consequently, the compromise of even a single component undermines the trustworthiness of the entire kernel and its ability to enforce separation between user-level processes. Reliably containing the extent of a compromised kernel component is a problem to which few practical solutions exist.

While many approaches have been proposed to reduce the need to trust large portions of the kernel, most of these approaches represent exotic reorganizations of the hardware or OS kernel that are either not applicable to commodity systems or are relatively complex and difficult to debug in their own right (e.g., microkernels).

We propose simple, natural modifications to commodity—`x86`—hardware that enable *vertical isolation* down through the kernel *without* the use of virtualization or major OS rewrites; specifically, extending and reinterpreting the `x86` segmentation mechanism, extending the existing Current Privilege Level and Descriptor Privilege Level fields. We believe our proposal is a compelling alternative to traditional virtualization because the hardware virtualizes *permissions*, not I/O.

Categories and Subject Descriptors

C.5.0 [**Computer Systems Organization**]: Computer System ImplementationGeneral; B.7.1 [**Hardware**]: Types and Design Styles-Gate Arrays

1. INTRODUCTION

Commodity operating systems typically provide isolation between user-level processes by placing each into a separate virtual memory context, between which the kernel enforces IPC and file system access restrictions. Mechanisms such as Linux Vservers, Solaris Zones, and BSD jails take this a step further to separate *groups* of processes from one another. In these systems, the kernel data structures that describe a process (e.g., Linux's `task_struct` or Solaris' `proc_t`) and the various objects it creates are extended with a context label; the kernel consults rules governing these labels before performing system calls.

This approach to isolation appears to work well for *user-level* process separation , but it does not provide any degree of *kernel-level* separation in case of a kernel compromise. Once malcode gains access to kernel privileges (such as the ability to read and write raw kernel memory), the kernel can no longer guarantee correct operation. Although kernels can be hardened with various monitoring and protection mechanisms, the problem of "protecting ring zero from ring zero" is notoriously hard.

A recently re-popularized approach to achieving kernel-level separation utilizes a *hypervisor*: a reference monitor executing *beneath* the OS kernel to intercept and govern I/O-related operations [8, 9]. Widely considered more trustworthy than general-purpose OS kernels due to their smaller size and limited functionality, hypervisors arguably just push the problem of separating data between compartments lower, albeit into a smaller TCB [14]. Unfortunately, hypervisors have grown into sizable beasts themselves, where security is less easy to verify. In addition, hypervisors introduce another layer of protection and policy that is less intuitive to programmers.

Taking the idea of data protection and isolation a step further, neither hypervisors nor the traditional UNIX protection model allow a process to protect user data from code executing *within the same process context*. Under the traditional model, from the point of view of integrity protection, all user data is created equal: no user-level data structure can be protected more than another. Unfortunately, there is no generic mechanism for relegating the security of user data (e.g., for cryptographic keys, linker tables, etc.) to the kernel; those requiring such security are welcome to write their own kernel modules and define their own APIs.

We see these two areas—intra- and inter-process isolation—as related, despite the fact that they do not appear to be treated as such by existing systems. In light of this observation, we propose a simple hardware solution to maintain separation between processes, kernel components, and even compartments within a single process, *even if one of those compartments is compromised*. Our design extends and reinterprets the Current Privilege Level (CPL) and Descriptor Privilege Level (DPL) bits in the x86 segmentation mechanism, which have long been part of 32-bit `x86` architecture (since at least i386), and should therefore be familiar to processor designers and straightforward for them to implement.

These extensions are a novel hardware feature we believe would significantly improve isolation support in commodity systems at a comparatively small OS software (re-)engineering cost. We stress that, in itself, this proposal does not suggest a new operating system design paradigm in the spirit of microkernels, capabilities, or hypervirtualization. Rather, it is an attempt to find a "hardware game-changer" that poses only reasonable OS software engineering challenges — that is to say, challenges which we know can be solved in practice without sacrificing too much performance or undertaking extensive changes in the kernel code base. Even though we cannot comprehensively review them within the scope of this paper, we acknowledge many recent advances in securing operating systems as a motivation, and we review some of the most related work (e.g., Nooks) in Section 6.

2. X86 SEGMENTATION

Operating system security in UNIX is, historically, closely tied to the kernel's ability to keep processes separate, and the *in*ability of processes to breach the user/kernel barrier outside controlled and well-defined interfaces. The former is achieved by placing process context structures under the kernel control and the latter is enforced by limiting ingress points to code running with kernel privileges (i.e., call gates). In an ideal world, these together ensure that no process can act on another process' data except via mediation by trusted code.

The x86 architecture provides so-called privilege "rings" to achieve this protection model, the enforcement of which is at the level of memory "segments". A description of the x86 segmentation mechanism and associated protections follows.

In x86 protected mode, all memory references are subject to a two-step conversion: from "logical" address within a segment to "linear" address within the process' virtual address space, then from linear to physical address via the page table system. Each segment is described by a *segment descriptor*, which includes a two-bit *descriptor privilege level* (DPL). On all memory references, the DPL is compared to the two-bit *current privilege level* (CPL); if CPL $>$ DPL, the reference causes a general protection fault, disallowing access. CPL and DPL values are directly analogous to the four x86 rings.

Segment descriptors are stored in two types of tables: the global descriptor table (GDT) or one of many local descriptor tables (LDTs), both of which have DPL=0, thereby preventing user processes (CPL=3) from manipulating descriptor privilege levels. (Each local descriptor table has the distinction of being described as a segment unto itself.) Instructions generally run at a CPL matching the DPL of the code segment; the CPL can only be changed when program control is transferred to a different segment (e.g., software interrupt, or CALL or JMP using a far pointer), which results in a kernel trap and is therefore subject to operating system mediation. To create the user-level process context, the kernel populates a set of segment registers (CS, DS, ES, FS, and SS) with segment selectors that point to segment descriptors with DPL=3, thus ensuring that user process code runs with CPL=3 which disallows access to kernel data (DPL=0).

We visualize the separation achieved by x86 segmentation as a two-layer cake sandwiching the system of call gates (Figure 1). The top layer is the user level, with processes separated by (kernel-enforced) vertical cuts stopping at the layer boundary. Malware infecting a "slice" of this cake cannot access data or interact with code across cuts. If, however, the malware manages to cross from the top later to the bottom (the kernel), it is free to burrow as it pleases. We note that mechanisms like SELinux, Linux Vservers, and Solaris Zones do not provide a game-changer in this respect, either.

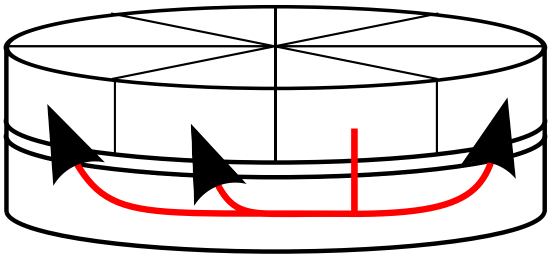

Figure 1: Cake visualization of existing x86 segmentation mechanism: malware crossing the barrier into the kernel is able to affect other processes.

X86 segmentation provides a *horizontal* separation between user processes and the kernel—traversed via closely-guarded call gates—and the kernel is depended upon to enforce separation between user processes and to provide data integrity guarantees. Unfortunately, this horizontal separation is insufficient if a user process gains supervisor privileges (i.e., CPL=0), either through a direct kernel barrier breach or via other mechanisms such as by sending a malformed network frame that exploits a vulnerability in the kernel network stack, allowing it to modify both kernel and user data willy-nilly. While we would like to believe that the user-kernel barrier is solid, experience teaches us to not depend on it.

3. VERTICAL ISOLATION

We begin by proposing that the CPL and DPL check-and-trap condition should verify *equality*, instead of the implied ordering of levels that exists currently. This reinterpretation would create 4 *sibling* isolated security contexts rather than the 4 security *levels* that the current x86 measurement logic (i.e., total order on the 2-bit values) provides. In fact, by imposing *partial orders* we could create and enforce more complex relationships between contexts (such as those of lattice-based Bell-Lapadula and Biba models). Of course, providing only 4 security contexts is limiting, therefore in addition to reinterpretation, we also propose extending the width of these fields. This is by no means a revolutionary idea, it is true, but we believe it is simple enough—to both hardware and software engineers—and provides sufficiently tangible benefits to justify itself. For backward compatibility, two of these extended bits will continue serving kernel-user context separation and call-gates.

We call the contexts created by our proposed extension of the CPL/DPL mechanism *vertical segments*. To return to the cake analogy, we can visualize the additional slices as now running through the kernel, as well (Figure 2). Conceptually, this means CPL and DPL can be thought of instead as *code privilege label* and *data privilege label*, allowing the creation of many labeled code/data contexts. That is, the context identifier of the code (CPL) determines which data

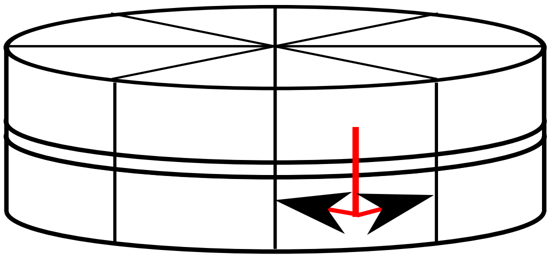

Figure 2: Cake visualization of our proposed modification to x86 segmentation: malware crossing the barrier into the kernel is *unable* to affect other contexts due to differing labels.

it is allowed to access, and the context identifier of the data (DPL) determines which code is allowed to access it.

Regarding specifics of implementation, the extended CPL bits would be best accommodated in a separate processor register, rather than as a subfield of the status register; the extended DPL may remain a part of the segment tables, and will require hidden latch registers for efficiency.

These alterations require changes to both the processor hardware and operating system software, but as we will argue shortly, they are minimal compared to the isolation gains.

4. A COMPELLING ALTERNATIVE

Contract-based Compartment Isolation.

In evaluating our proposal, we find it instructive to examine systems with similar goals. In particular, consider Linux Vservers and Solaris Zones, which achieve context separation (i.e., process hierarchies roughly corresponding to separate physical machines with separate root contexts) by applying distinguishing labels to kernel data structures of processes belonging to different contexts. System calls and other IPC code executing on behalf of contexts with differing labels are therefore prevented from reporting on or affecting the context in question. Such isolation is enforced by the kernel-level implementation of, e.g., Vservers and Zones.

The underlying philosophy of existing designs can be summed up as "**A context cannot access what is explicitly labeled as off-limits to it or to which it lacks pointers**". In other words, labelling and linking is just for well-behaved kernel code to observe the internal **contract** that data does not flow between contexts. This is all well and good, assuming the arbiter of contracts is trusted, but hypervisors and similar systems have grown beyond the point of practical security audit. We note that the many flavors of *capabilities*-based systems (e.g., [15]), unless such capabilities are enforced by hardware, rely on the same isolation philosophy.

The attacker community, however, has long established[1] that scanning raw memory to recover pointers to critical data structures can be done very efficiently. Such scanning methods can be quite sophisticated, avoiding causing memory faults on intentionally unmapped address ranges and other defensive hacks.

[1]Relevant techniques have been published as early as Phrack 58 (2001).

Therefore, to break *contract-based* isolation, the attacker need only introduce code into the kernel that ignores the contract. This fragility demonstrates that such isolation mechanisms are inadequate for extending context separation into the kernel.

Contract to Enforcement.

Our proposal embodies a switch from voluntary or advisory contracts within the kernel to memory trap-based enforcement. Instead of storing context labels inside data structures, we advocate creating data structures in separate "vertical segments" of virtual memory, and enforcing access restrictions as part of the segment address translation phase.

Projects such as Linux Vserver do, however, provide broad shoulders upon which our proposal stands, by having already identified locations within kernel code where contract-affected data structures are created and must be shared. We note that the necessary changes to memory allocation code appear lightweight when compared to those needed for full virtualization. Specifics will be presented in section 5.

Application Security Policy.

We have thus far discussed vertical isolation as a means of separating context within the kernel, but it also corresponds to a broader concept intuitive to application developers: that of a particular segment of code "owning" a specific set of data, confident that the data's integrity is protected by only allowing access to trusted, owning code.

This approach to improving code trustworthiness is demonstrated firstly by compiler scoping rules such as `static` file-scope variables in C, which are loosely enforced at the compiler level. Additionally, the object-oriented programming paradigm introduces the concept of encapsulation, which could be thought of as object-level isolation. However, this isolation verification ends as soon as the compiler terminates; in short, application developers cannot translate knowledge about memory objects into policy regarding their protection.

Since few data access-related facts are as intuitively clear and relevant to a program's expected behavior as the code-data ownership relations, we believe developers will appreciate a programming primitive to express them. This primitive would help separate code and data into slices and support trapping execution when developers' expectations are violated by cross-slice accesses.

One way to implement this capability is to extend the compiler such that code is subdivided into `.text` subsegments and data into `.data` subsegments, labeled with our extended CPL/DPL bits, with data only accessible to code with a matching label. Internally, labels would be assigned to individual vertical segments and the resulting binary loaded and memory-mapped accordingly. The inspiring flexibility of the ELF format makes describing such mappings straightforward.

Limiting cross-thread data sharing abuse.

Threads allow different code segments to share data without the overhead of IPC. Although threads provide a powerful advancement in programming language semantics, they encourage oversharing, i.e., reads and writes far beyond the programmer's intentions, where sharing is neither needed nor wanted. This is due to the fact that address space sharing between threads has no granularity: threads share the entire address space. Our vertical segment mechanism introduces support for increased protection granularity *within a threaded process*, far more effectively than, e.g., compiler-based object-oriented encapsulation.

Eliminating policy "hooks".

Taking another angle on application-level protections, our proposal frees the developer from having to align their application's security assertions with the operating system's system calls, as in FLASK-based systems such as SELinux. Developers no longer require intimate OS-level knowledge, nor are they limited to equating "trusted data" with "kernel-held data".

.

Thus our proposed modifications can be applied not just to kernel-level contexts, but also within processes, which to this point have been generally considered undividable contexts unto themselves, with a modest learning curve to developers. In fact, our design proceeds from the motivating assumption that a policy mechanism requiring the policy authors to understand its many measurement points is bound to be a usability failure.

5. A PRACTICAL CONSIDERATION

We argue that hardware modifications to enable our proposed reinterpretation and extension of the CPL and DPL bits is minimal; how much software re-engineering effort is then required? Although, at first blush, the answer appears to demand a non-trivial software re-design, we can harness existing kernel-based process virtualization systems to achieve our goals.

These systems, including VServer[17] and ZAP[13], have already performed the steps necessary to provide separation within the kernel: they maintain and manage the kernel notion of a process "context" or "namespace" (e.g., VServer adds the `struct vx_info` construct); extend some existing data structures and system calls; and introduce into the kernel a default context that accommodates the init process and one or more management system calls. (In this paper, we are not concerned with the necessary kernel modifications required to enforce resource limitations.)

Container Arch.	OpenVZ	Linux VServer	ZAP
Lines of code	∼92,000	∼8,700	∼27,000
New Files	80	50	86
Files touched	920	300	134
Size (Kb)	3,445	820	653

Table 1: Comparison of code modifications for some process virtualization systems: OpenVZ for Linux 2.6.26; VServer vs2.2.0.7 for Linux 2.6.22.19; ZAP version for Linux 2.4.

The size of the implementation, in terms of kernel modifications, indicates that the effort required to re-engineer a system to enable process context separation is relatively high. Table 1 shows a comparison of the patch sizes for Linux kernel versions. The different technologies have different advantages in terms of performance and resource control the description of which goes beyond the scope of this paper.

For our purpose, we can peruse the existence of both container labels and data structures to enforce policies using the points in the kernel where objects are created via `kmalloc`; the birth of these objects are under the control of a certain "container", and so their label is straightforward to derive.

6. RELATED WORK

Hardware tagging of memory has been around for decades, but usually with the goal of supporting language constructs (e.g., Lisp [12]) rather than security. The ill-fated Intel iAPX-432 [4] implemented the concept of "roles" and "objects" at the hardware level and encouraged separation of OS duties similar to our concept of vertical isolation, but it never caught on. Burroughs 5000 machines [1] also employed a tagged architecture, but only used 3 bits for the tag and explicitly defined the meaning of all values, disallowing arbitrary semantics.

More recently, in the area of information flow control [5], projects such as Asbestos [21] apply labels to processes and use these labels to control read and write privileges to other processes, though these controls do not extend below the user-kernel barrier. HiStar [24] builds on Asbestos and applies the idea of tainting to trace and control information flow; like Asbestos, this protection does not extend to kernel-space. Loki [25] brings hardware support — in the form of a modified SPARC architecture—to HiStar, implementing labeling with a 32-bit tag on every 32-bit memory word.

TIARA [16] proposes an entirely new processor, operating system, and middleware architecture — including tagged memory — that allows what the authors term a "zero-sized kernel" in which individual kernel components (scheduler, device drivers, *etc.*) are implemented and extended privileges separately. This design achieves an approximation of vertical isolation at the expense of near-term, real-world practicality.

Word-level memory isolation has been accomplished through major `x86` hardware modifications [22, 23] that extend processor page tables, translation logic, CPU pipelines, and register files. Such changes make these systems impractical for immediate deployment.

Certain fault-tolerant systems [15, 21] and extensible operating systems [2, 6] enforce module-level isolation, but they require a major rewrite of the OS (or are full OSes in their own right). In contrast, our approach is pragmatic enough, both in terms of hardware and software, to warrant attention from OS practitioners today.

User-level isolation and privilege separation is provided by either OS-virtualizers [17, 11] that modify the kernel to accommodate container labeling, or user-level monitors [3, 10, 7] that entail minimal kernel modifications. While the former category of systems does not guarantee isolation at the kernel level, the latter category does not enforce *full* program separation.

Nooks..

The *Nooks* [19, 20] system, which uses virtual memory techniques to isolate drivers for the sake of raising kernel *reliability*, is closely related to our proposal. The Nooks designers recognize that virtualizing physical devices by emulating separate virtual instances of them is not conducive to improving OS reliability, despite creating an illusion of isolation. Instead, Nooks concentrates on separating drivers from the main kernel by applying virtualization at their interface points. Nooks' success in isolating actual drivers shows that such decomposition of kernel code and data can be realistically achieved even in driver programming, not just in the general kernel contexts (as Linux Vserver and other work suggests).

From the security perspective, we similarly eschew device emulation as a productive isolation approach, and also propose to build better isolation on the virtual memory mechanism. However, there is an essential difference, which goes back to the difference between the reliability vs. security approaches.

In the case of Nooks and other reliability systems, the "adversary" is bugs, and the associated threat can be modeled by injecting random flaws. In the security domain, however, the "adversary" is malicious code, specifically written to take advantage of known opportunities to elevate its privileges. To wit, one of Nooks' authors notes: "**Nooks protects against bugs but not against malicious code**" [18].

Thus, whereas for Nooks using x86 hardware features such as rings and segmentation is undesirable due to reduced portability, performance, and being parenthetical to achieving the primary goal of improving reliability in presence of bugs, our attention is instead directed towards these protection mechanisms that would constrain malicious code from purposefully subverting kernel isolation.

Consequently, when looking for a game-changer, we focus on how these hardware features could be extended so that they could be used effectively for kernel security isolation without requiring major changes to existing code. The software engineering challenge of such OS changes is strongly similar, being in fact the matter of code and data decomposition; however, the actual protection mechanism that will take advantage of such decomposition must be very different, owing to the differences between reliability and security challenges.

7. CONCLUSION

We propose a simple modification to `x86`, extending and reinterpreting the CPL and DPL bits to provide memory separation as a first-class policy resource. While our proposal is hardly radical, we believe it would make a profound impact on process context isolation at the expense of modest hardware- and system-engineering effort. In this paper, we consider what effects the availability of this resource has in terms of providing true isolation within the kernel as well as support for the ability of application programmers to express natural data ownership properties. A number of interesting problems remain, including the handling of traps raised by a label mismatch (i.e., interpreting the context labels).

8. REFERENCES

[1] *The Descriptor - A Definition of the B 5000 Information Processing System*. Burroughs Corporation, 1961.

[2] B. N. Bershad, S. Savage, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *SOSP '95*, pages 267–284, 1995.

[3] D. Brumley and D. Song. Privtrans: automatically partitioning programs for privilege separation. In *USENIX Security '04*, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[4] G. W. Cox, W. M. Corwin, K. K. Lai, and F. J. Pollack. A unified model and implementation for interprocess communication in a multiprocessor environment. *SIGOPS Oper. Syst. Rev.*, 15(5):125–126, 1981.

[5] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[6] D. R. Engler, M. F. Kaashoek, and J. OâĂŹtoole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *SOSP '95*, pages 251–266, 1995.

[7] B. Ford and R. Cox. Vx32: lightweight user-level sandboxing on the x86. In *USENIX ATC '08*, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association.

[8] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are Virtual Machine Monitors Microkernels Done Right? In *HOTOS '05*, June 2005.

[9] P. A. Karger and D. R. Safford. Security and Performance Trade-Offs in I/O Operations for Virtual Machine Monitors. In *IBM Research Technical Report RC24500 (W0802-069)*, February 2008.

[10] D. Kilpatrick. Privman: A Library for Partitioning Applications. In *USENIX Technical Conference, FREENIX Track*, Berkeley, CA, USA, 2003. USENIX Association.

[11] S. Microsystems. Consolidating applications with Solaris containers. 2004.

[12] D. A. Moon. Architecture of the Symbolics 3600. *SIGARCH Comput. Archit. News*, 13(3):76–83, 1985.

[13] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *OSDI '02*, pages 361–376, Boston, MA, Dec. 2002.

[14] T. Roscoe, K. Elphinstone, and G. Heiser. Hype and Virtue. In *HOTOS '07*, May 2007.

[15] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a Fast Capability System. In *SOSP '99: Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 170–185, New York, NY, USA, 1999. ACM.

[16] H. Shrobe, T. Knight, and A. de Hon. TIARA: Trust Management, Intrusion-tolerance, Accountability, and Reconstitution Architecture. Technical Report MIT-CSAIL-TR-2007-028, MIT, May 2007.

[17] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, 2007.

[18] M. M. Swift. Improving the Reliability of Commodity Operating Systems, 2005.

[19] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Trans. Comput. Syst.*, 24(4):333–360, 2006.

[20] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23(1):77–110, 2005.

[21] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4):11, 2007.

[22] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *ASPLOS-X: 2002*, volume 37, New York, NY, USA, 2002. ACM Press.

[23] E. Witchel, J. Rhee, and K. Asanović. Mondrix: memory isolation for linux using mondriaan memory protection. *SIGOPS Oper. Syst. Rev.*, 39(5):31–44, 2005.

[24] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI '06*, Berkeley, CA, USA, 2006. USENIX Association.

[25] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *OSDI '08*, Berkeley, CA, USA, 2008. USENIX Association.