Improving Usability via Access Decomposition and Policy Refinement

Neal H. Walfield and Marcus Brinkmann
{neal,marcus}@gnu.org

Abstract

Commodity operating systems fail to meet the security, resource management and integration expectations of users. We propose a unified solution based on a capability framework as it supports fine grained objects, straightforward access propagation and virtualizable interfaces and explore how to improve resource use via access decomposition and policy refinement with minimum interposition. We argue that only a small static number of scheduling policies are needed in practice and advocate hierarchical policy specification and central realization.

1 Introduction

Commodity operating systems provide inadequate protection mechanisms preventing users from articulating some useful security policies; they expose resource abstractions which, in hiding resource multiplexing, reduce efficiency, limit application adaptability and impede the realization of real-time properties; and they lack extensibility, reducing consistency of mechanism and integration. Although the research community has explored these problems individually, the resulting models often ignore one of these concerns limiting applicability.

We have chosen an object capability system as our foundation. By conflating designation and authorization, thereby eliminating principal identifiers and shared name spaces, such a system enables fine grained authorization and simplifies access propagation. This is necessary for the dynamic realization of the principle of least privilege (POLP). Its virtualizability enables better integration by permitting untrusted extensions without necessitating parallel worlds. To enable better use of resources, we explore how to decompose authority and refine policy without expensive and inaccurate interposition.

2 Motivation

2.1 Security and Protection

When Alice launches a program on a commodity operating system, the program instance typically runs with her full authority. A web browser, although its core functionality only depends on a network connection, a window on which to render content and a fixed number of resources known in advance, has access to all of her resources and can even control other program instances running on her behalf. Alice has no protection from unauthorized disclosure, tampering or disruption of service.

Although Alice may have learned through experience to trust her web browser, it operates on externally supplied data and loads plug-ins. It also contains bugs [22]. By finding a weakness in the code which processes external input, an attacker is able to confuse Alice's web browser and gain all of her authority. Such attacks are so easy, these compromised machines, so-called bots, can be purchased on the black market for $0.04 each [30].

2.2 Resource Management

Commodity operating systems transparently manage scarce resources, ostensibly relieving applications of this complexity. The exposed resource abstractions, however, have loose access characteristics, i.e., a large gap between average and worst case access times, frustrating application efforts to maximize performance and realize real-time properties.

2.2.1 Efficient resource usage

The ever increasing performance gap between backing store and main memory makes paging increasingly expensive. Many applications possess information which can significantly improve scheduling but is inaccessible to a scheduler which only monitors behavior.

Performance Garbage collectors and databases are two classes of applications evaluated against the clock. Although these applications often have access patterns which differ significantly from those which the operating system can detect and some are able to predict their own access patterns, exploiting this local information requires either having closely guarded privilege to use, e.g., `mlock`, or relying on implementation details [29, 1, 16]. Although extensions exist to provide mechanisms for application input, they are not expressive, e.g., `madvise`, or cooperative, making the memory manager vulnerable to malicious applications [16].

Caching Many applications are able to save a significant number of CPU cycles as well as power by caching calculated data and intermediate results for opportunistic reuse. As the data is often large, e.g., a decompressed JPEG, there is a significant opportunity cost associated with such caching: the data occupies memory possibly causing more valuable data to be paged; and the operating system may page the cached data which may be more expensive than simply recomputing it on demand.

Because commodity operating systems provide no way for applications to prevent this paging and because applications do not know how much memory is idle, they must act conservatively. GQView, a popular image viewer for GNOME, maintains, by default, a 10 MB cache of rendered images [11]. gThumb, another image viewer for GNOME, keeps a static cache of four images and preloads the image following and that previous to the requested image [4]. Neither application pro-actively frees its cache.

Nokia, in the development of their Internet Tablet platform, Maemo, acknowledged this problem and introduced a feedback mechanism, an event source, allowing cooperative applications to obtain global contention and to adapt their resource use accordingly [21].

Virtualization Virtual machine monitors (VMMs) are currently used for resource consolidation with compartments sometimes requiring guaranteed levels of quality of service [33]. They are also being used as an isolation mechanism for the enforcement of security policies [23]. As this technology is improved, it is plausible that it will be applied at a finer granularity and made available to users. This will require mechanisms for limited breaching of the isolation barrier to enable composition and collaboration. (Such VMMs would increasingly resemble reference monitors [2].) To provide quality of service and enable efficiency, such a scheme would require that resources be strictly accounted, easily decomposed, delegatable and dynamically reallocatable.

Engineering and economy of scale Devices, especially consumer electronics, are being increasingly sold based not on their resource abundance but on their functionality. This is in tension with the desire to reduce engineering costs by using commodity operating systems with moderate increases in resource requirements relative to more specialized systems. One might expect such a tradeoff to be quickly mitigated by the ever increasing abundance of processing power and memory and their respective decreases in cost. Contrariwise, Linksys recently revised their popular router to use vxworks instead of GNU/Linux and were able to halve the 16MB RAM and 4MB of flash thereby increasing profitability despite the engineering costs [20]. This argument, that it is significantly cheaper to improve the software than to increase the available resources, has also been made by the developers of the One Laptop Per Child project [13].

2.2.2 Real-time properties

Real-time and adaptive applications need a mechanism to obtain statistical guarantees regarding resource schedules [9]. Although POSIX provides mechanisms such as `mlock` to allocate physical memory, this privilege is closely held to prevent misuse and abuse. Yet, few commodity applications actually require such firm guarantees. To work around this deficiency, application developers often take advantage of implementation details. Reliance on this is problematic as the behavior is not part of the API contract and can change. The authors of Cedega, a Windows emulator for games, encountered this when Linux's CPU scheduler was modified [31].

2.3 Integration

Integration depends on uniform access mechanisms. One of the most visible interfaces is the virtual file system (VFS). Normally, users are not able to provide new implementations or start new file system instances which integrate into the VFS as this is normally only available to programs running in the kernel. Although it is possible to upload code into the kernel, it is undesirable as it is not able to be constrained.

This situation has led to the development of parallel interfaces. The GNOME project's GnomeVFS and KDE's KIO-Slave both expose a new VFS to to their respective applications so as to more seamlessly integrate interesting file systems such as those accessible over ftp and ssh. Yet these technologies are, at best, only half integrated: an application that does not make use of, e.g., the KDE VFS exposes a very different file system layout. The Linux developers have also acknowledged this and recently introduced an API to allow users to safely provide their own file systems running in user space.

3 A System Structure

We have selected a capability based framework [7, 34, 15, 27, 12] as it appears to provide the necessary foundational mechanisms. The power of capabilities lies in their bundling of authorization and designation. This permits fine grained objects and enables access to be propagated in a single step unlike, e.g., on an ACL based system where designation and authorization are separated, frustrating delegation [19].

Below, we briefly address how capabilities help solve the aforementioned problems and outline the additional mechanisms required to build a system.

3.1 Protection and Security

On commodity operating systems, programs run with all of the authority of the user who started them. This is excessive. A more secure mode of operation would be one where users are able to delegate just the authority a program instance requires to carry out their intent, the principle of *least privilege* (POLP) [24]. Thus, when a program goes awry, damage would be restricted to those resources to which it has access.

Although it is technically possible to achieve such controlled sharing, e.g., on Unix using an additional UID and `chroot`, it is so unwieldy to configure as to be used only by experts in special scenarios. A successful mechanism must be consistent with the principle of *fail-safe defaults*: it must be the default and require effort to violate [24]. In addition, for interactive programs where a reasonable minimum authority cannot be calculated a priori, it must be straightforward for the user to delegate additional access rights after it has started, *dynamic* POLP.

Capability systems can provide this with the help of a so-called powerbox [28, 25]. Instead of creating an `open` or `save` dialog, the application invokes the user's trusted powerbox, which, having all of the user's author-

ity, interacts with the user and then delegates access to the selected resources to the program. This change is largely invisible to users and applications.

As capabilities are held by processes, a mechanism needs to be provided for their recovery, for programs to be able to restore their configuration on system restart. A desktop manager, for instance, would like to remember what programs were running; applications would like to record resources in use. This configuration management problem is referred to as *trusted recovery* [8] and is ignored by commodity operating systems as all of a user's programs run in the same trust domain.

Currently, applications store file names, however, this requires that program instances run with the full authority of the user in violation of POLP. Having program instances remember delegations and replay them on restart is fragile and complicated by the fact that delegations are made to program instances and several instances of a program may run in different trust domains. Instead, this problem can be circumvented by making the system persistent: the access graph need, then, never be recreated. Although seemingly overkill, this is already the aim of desktop managers and is directly realized by many laptops and an increasing number of desktops in the form of suspend or hibernate. To this end, EROS uses a single level store to realize orthogonal persistence [26]. Another approach is exportable state [32, 14].

3.2 Resource Management

More effective use of resources can be achieved by providing resources with tighter access characteristics, exposing the resource schedule and inexpensive decomposition and delegation. We temper the solution space with the requirement that mechanisms and policies must also be safe. To achieve this, we prefer specificity over generality through the elimination of unmotivated functionality. This is in contrast to extensible kernels whose emphasis is on generality, which has been criticized as introducing unjustified complexity frustrating safety [10].

Exokernels, a class of extensible kernels, aim to securely export physical resources at as fine a granularity as possible and hide as few policy decisions as feasible including resource revocation [18]. To achieve this, an exokernel uses so-called visible revocation. We observe two shortcomings with this approach.

When an application is chosen to yield memory, it receives an upcall and is given a set amount of time to return some amount of memory. To avoid creating a functional dependency on the correct behavior of applications, the kernel must impose a deadline. When this is too short, an otherwise correct application will generate a spurious fault. When this is too long, a malicious application may be able to induce a denial of service. Although spurious faults can be avoided by taking the position that all code in the page-out path must be hard real-time capable, writing correct real-time capable code is notoriously difficult and it induces conservative behavior reducing the possibility of best effort optimizations.

Second, because managing resources requires resources that the kernel may choose to reclaim at any time an application must make provisions to allow a third party to manage these latter resources on its behalf.

These shortcomings motivate the reintroduction of transparent paging and could be viewed as a failure of exokernel principles to generalize. We sacrifice generality and instead aim to allow applications to drive resource management in a consistent, straightforward fashion.

Distribution Policy There are three parties interested in specifying scheduling policy on others: supervisors, developers and users.

A system administrator would like the available resources to be distributed according to some rather static fairness property among users and the various system services (or VMMs). This should not be understood to mean that the allocations are static but that the allocation policy changes relatively infrequently.

Developers who build systems with a number of activities generally statically assign priorities to them. A multimedia player or game engine, for instance, would assign the audio decoder a higher priority than the video decoder as people are more sensitive to audio jitter.

Users, to ensure that they always remain in control, have as their top priorities the event input thread and the window manager. These trusted applications would be run under a highest priority first (HPF) regime. The balance would be aggregated under a lower priority and scheduled according to, e.g., a proportional share policy.

The distribution of priorities among these applications depends on a user's priorities, i.e., it is a function of real world tasks and goals. Having the user change application priorities manually is cumbersome. Fortunately, priorities can often be inferred from a user's actions. The application with the focus likely has a high importance to the user and should therefore have a high priority. Minimized applications are likely less important. For some applications, e.g., an audio player, the user may desire a permanently high priority independent of their respective window states. To accommodate this, the user must have the possibility to override the priority which can be remembered by the window manager. The distributor could also provide hints about the appropriate policy.

We postulate that a small number of fixed policies is sufficient for most useful scheduling scenarios.

Multiplexing Policy As resources are scarce, applications have an interest in multiplexing what is available: determining how CPU is used and which data is held in memory. This can be achieved with scheduler activations [3] and providing control over the eviction policy.

Additionally, applications which need to articulate scheduling parameters such as duration and jitter, in particular, real-time and adaptive applications, also need to be supported via, e.g., imprecise computation [17].

Framework In none of the presented distribution scenarios does the parent process need to participate in admission or allocation: it is sufficient for it to describe a policy. Likewise, applications do not generally care what the controlling policy is: they request schedules which are either admitted or not. As such, we allow policy to be articulated hierarchically but centralize admission control and scheduling thereby circumventing the process hierarchy for the realization of this mechanism.

By separating the specification of policy from scheduling, the latter can be determined quickly and more accurately. A highly nested process need not request a schedule from its parent which must translate the request to its parent's vocabulary, etc.; it directly requests a schedule from the system scheduler. Likewise, when the schedule must be changed either due to policy (based, e.g., on contention) or due to a policy change, schedules can be quickly recalculated and processes directly informed.

As the scheduling hierarchy can be complex, the calculation of schedules can become complicated. We contend, however, that policy change is relatively infrequent compared with admission requests and resource usage reducing potential overhead.

Adherence to schedules as well as reduction of cross talk require accurate resource accounting of both the resources a principal directly uses as well as those it indirectly uses, i.e., those allocated by servers on its behalf.

To achieve this, we introduce a mechanism, resource pools, similar to EROS's space banks [26] and resource containers [5]. We account memory and backing store individually, unlike EROS.

A resource pool specifies a scheduling policy for resources allocated against it. A new pool can be derived from an existing pool and delegated. The policy applied to the derived pool can refine the policy imposed by the parent. As such, resource pools form a hierarchy and children are strictly dominated by their parents.

Resource pools are used for controlling inferior processes. A process derives a resource pool from its own and specifies any scheduling parameters. It then runs the child out of this inferior pool and passes it a weakened form, which does not allow control of the scheduling policy. The child allocates all resources out of this pool. At any time, the parent can destroy the derived pool and, in doing so, destroy the child and everything that it allocated including temporary files and other processes.

Pools are also passed to servers when the server must allocate resources on behalf of a client, e.g., memory for session state (although, we try to avoid sessions when possible). This improves the ability of the server to honor any quality of service guarantees and provides a way for the client to reclaim resources if the server misbehaves.

Revocation When a process's schedule has changed, action may need to be taken to reclaim resources. For instance, when a process's memory allocation is reduced, pages may need to be saved. Similarly, when resources need to be multiplexed, a scheduling decision must be made. Allowing applications control of this policy is essential to exploiting local information, improving performance and meeting real-time requirements.

We have noted that rendered data can be recomputed without loss of information and, thus, can be discarded without negative consequences. Caching this data in idle memory is desirable as it can significantly improve best-effort applications. As commodity operating systems are unable to distinguish this memory from normal anonymous memory, they must page it.

We propose two mechanisms which permit the memory manager to be able to distinguish such data and to be able to discard it with no negative consequences. We introduce a function which allows applications to mark data as being discardable. This allows the memory manager to simply discard it when it is chosen for eviction. When a thread next accesses the virtual memory region, it receives a fault indicating what has happened allowing the application to recompute the data. These mechanisms never require that the memory manager wait on a response from the application: when the application must act, it is in response to a fault.

To allow increased control of how the rest of memory is paged, i.e., the eviction policy, the application assigns priorities to allocated memory. When the manager evicts a page, it selects the lowest priority page. If there are multiple pages, then it selects the one approximately least recently used.

When a page is evicted, the application can request to receive an event the next time it is scheduled. This requires resources for the manager to hold which page was evicted necessitating care. When the virtual memory is again referenced, the operating system can send a fault to the application or transparently page it back in.

3.3 Integration

Capability systems enable fine grained virtualization: whether a kernel or user object, its methods are accessed using the same mechanism, capability invocation. Further, as each service is typically encapsulated by a different object, a single service can be proxied, extended or monitored without imposing overhead on other services.

4 Conclusion

We have identified a number of problems with commodity operating systems: they fail to provide adequate protection; their resource management strategies lead to in-

efficient resource use and cannot be effectively used in meeting real-time properties; and they lack integration.

We propose a class of operating systems which may be able to solve the most egregious of these. Based on a capability framework, such a system permits the realization of dynamic POLP and virtualizable interfaces. To improve resource scheduling, we provide applications with more control over the scheduling policy. We argue that only a small number of scheduling policies are required in practice. Thus sacrificing generality for safety, we permit applications to articulate scheduling policy to a centralized scheduler. This permits access decomposition and policy refinement without process interposition.

We acknowledge that radical new designs will not be accepted if users cannot run even one or two of their legacy applications. The Hurd, another multi-server system, successfully provided a high degree of API compatibility via a so-called fat C library which implemented the legacy interfaces in terms of Hurd mechanisms [6].

5 Acknowledgements

We thank: J. Shapiro for many discussions about security and system design; and T. Schwinge for feedback.

References

[1] ALONSO, R., AND APPEL, A. W. An advisor for flexible working sets. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems* (1990).

[2] ANDERSON, J. P. Computer security technology planning study. Tech. rep., Electronic Systems Division, Oct. 1972.

[3] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM symposium on Operating systems principles* (1991).

[4] BACCHILEGA, P. gThumb v2.8.0. `http://gthumb.sf.net`, Nov. 2006.

[5] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In *3rd USENIX Symposium on Operating Systems Design and Implementation* (Feb. 1999).

[6] BUSHNELL, M. Towards a new strategy of OS design. *GNU's Bulletin 1*, 16 (Jan. 1994).

[7] DENNIS, J. B., AND VAN HORN, E. C. Programming semantics for multiprogrammed computations. *Communications of the ACM 9*, 3 (Mar. 1966), 143–155.

[8] DEPARTMENT OF DEFENSE. *Trusted Computer System Evaluation Criteria DOD 5200.28-STD*. Dec. 1985.

[9] DOMJAN, H., AND GROSS, T. R. Managing resource reservations and admission control for adaptive applications. In *30th International Conference on Parallel Processing* (2001).

[10] DRUSCHEL, P., PAI, V. S., AND ZWAENEPOEL, W. Extensible kernels are leading os research astray. *Proceedings of the 6th Workshop on Hot Topics in Operating Systems* (May 1997).

[11] ELLIS, J. GQView v2.0.4. `http://gqview.sf.net`, Dec. 2006.

[12] FORD, B., HIBLER, M., LEPREAU, J., TULLMANN, P., BACK, G., AND CLAWSON, S. Microkernels meet recursive virtual machines. *2nd USENIX Symposium on Operating systems design and implementation* (Oct. 1996).

[13] GETTY, J. $100 laptop / OLPC (One Laptop Per Child). `http://gettysfamily.org/wordpress/?p=11`, Nov. 2005.

[14] HAEBERLEN, A., AND ELPHINSTONE, K. User-level management of kernel memory. In *Proceedings of the Eighth Asia-Pacific Computer Systems Architecture Conference* (Sept. 2003).

[15] HARDY, N. The KeyKOS architecture. In *Operating Systems Review* (Oct. 1985), vol. 19, pp. 8–25.

[16] HERTZ, M., FENG, Y., AND BERGER, E. D. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (June 2005).

[17] HULL, D., FENG, W., AND LIU, J. W. S. Operating system support for imprecise computation. In *AAAI Fall Symposium on Flexible Computation* (Nov. 1996).

[18] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., BRICEO, H. M., HUNT, R., MAZIRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application performance and flexibility on exokernel systems. *16th Symposium on Operating Systems Principles* (1997).

[19] MILLER, M. S., TULLOH, B., AND SHAPIRO, J. S. The structure of authority: Why security is not a seperable concern. In *MOZ 2004 Workshop* (2005), vol. 3389 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag.

[20] NEEDLEMAN, R. Technology marches backward. `http://reviews.cnet.com/4520-3000_7-6542073.html`, June 2006.

[21] NOKIA. Maemo, the application development platform for the Nokia 770 internet tablet. `http://maemo.org`.

[22] OSTRAND, T., WEYUKER, E., AND BELL, R. Where the bugs are. In *ACM SIGSOFT International Symposium on Software Testing and Analysis* (2004), pp. 86–96.

[23] SAILER, R., JAEGER, T., VALDEZ, E., CCERES, R., PEREZ, R., BERGER, S., GRIFFIN, J., AND VAN DOORN, L. Building a MAC-based security architecture for the Xen opensource hypervisor. *21st Annual Computer Security Applications Conference* (Dec. 2005).

[24] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. In *Proceedings of the IEEE* (1975), vol. 63, pp. 1278–1308.

[25] SEABORN, M. Plash: tools for practical least privilege. `http://plash.beasts.org`.

[26] SHAPIRO, J. S., AND ADAMS, J. Design evolution of the EROS single-level store. In *2002 USENIX Annual Technical Conference* (2002), pp. 59–72.

[27] SHAPIRO, J. S., SMITH, J. M., AND FARBER, D. J. EROS: a fast capability system. In *Symposium on Operating Systems Principles* (1999), pp. 170–185.

[28] STIEGLER, M., KARP, A. H., YEE, K.-P., AND MILLER, M. Polaris: Virus safe computing for Windows XP. *Communications of the ACM 49*, 9 (2006), 83–88.

[29] STONEBRAKER, M. Operating system support for database management. *Communications of the ACM 24*, 7 (July 1981), 412–418.

[30] THOMAS, R., AND MARTIN, J. The underground economy: priceless. *;login: 31*, 6 (Dec. 2006).

[31] TRANSGAMING. September development status and voting report. `http://www.transgaming.com/showthread.php?news=126`, 2004.

[32] TULLMANN, P., LEPREAU, J., FORD, B., AND HIBLER, M. User-level checkpointing through exportable kernel state. *IEEE International Workshop on Object-Orientation in Operating Systems* (Oct. 1996).

[33] WALDSPURGER, C. A. Memory resource management in VMware ESX server. *5th Symposium on Operating Systems Design and Implementation* (Dec. 2002).

[34] WULF, W., COHEN, E., CORWIN, W., JONES, A., LEVIN, R., PIERSON, C., AND POLLACK, F. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM 17*, 6 (June 1974), 337–345.