LockDown: An Operating System for Achieving Service Continuity by Quarantining Principals

Gedare Bloom
Computer Science
Howard University
Washington, DC
gedare.bloom@howard.edu

Gabriel Parmer
Computer Science
George Washington University
Washington, DC
gparmer@gwu.edu

Rahul Simha
Computer Science
George Washington University
Washington, DC
simha@gwu.edu

ABSTRACT

This paper introduces quarantine, a new security primitive for an operating system to use in order to protect information and isolate malicious behavior. Quarantine's core feature is the ability to fork a protection domain on-the-fly to isolate a specific principal's execution of untrusted code without risk of a compromise spreading. Forking enables the OS to ensure service continuity by permitting even high-risk operations to proceed, albeit subject to greater scrutiny and constraints. Quarantine even partitions executing threads that share resources into isolated protection domains. We discuss the design and implementation of quarantine within the LockDown OS, a security-focused evolution of the Composite component-based microkernel OS. Initial performance results for quarantine show that about 98% of the overhead comes from the cost of copying memory to the new protection domain.

CCS Concepts

•Security and privacy → Operating systems security;

Keywords

access control, confinement, microkernel, protection

1. INTRODUCTION

Computer system security is a tale as old as time-sharing. The plot features the protagonist, a humble operating system, and motley antagonists attempting to gainsay OS authority. The OS has at its disposal the primary advantage of privilege, which underlies the vast array of access control solutions that have been discovered and handed down across the generations. Alas, the adage "power corrupts" fittingly describes the current situation in which OSs have become large, unwieldy, bloated, complex beasts that are no longer fit to wear the crown of privilege. Tragically, vulnerabilities in commodity OS services are commonplace, and computer systems are no longer presumed secure until proven vulnerable.

In this paper, we explore a systemic problem with traditional approaches to computer security—namely, that the principle of least privilege has not been applied sufficiently well within OS services.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EUROSEC'16, April 18-21 2016, London, United Kingdom

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4295-7/16/04. . . $15.00

DOI: http://dx.doi.org/10.1145/2905760.2905764

Compromised system software compromises all software in the system. For example, consider an application that unwittingly downloads a malicious executable that mounts a privilege escalation attack, such as CVE-2013-2094. This particular attack exploits an obscure bug in the Linux kernel to get root access via a seemingly harmless syscall. The fundamental problem is that authorization in modern systems is generally limited to checking proper access to high-level objects such as files and devices, whereas exploitable bugs can occur anywhere in OS code and circumvent authorization. This problem is well-known and stimulates research in the primary direction of information flow control.

The mandatory access control frameworks used by military systems in the 60s and 70s underlie information flow control [3]. In a nutshell, a lattice of partially-ordered labels are given to principals (subjects) and domains (objects), where the ordering determines whether or not a principal may access a domain. Labels are a combination of a security level—*e.g.* a clearance or taint—and a compartment. For a given access request to a domain, a reference monitor checks all of the labels on that domain to see if the principal has labels that grant permission for the given request. Labels with different compartments are not comparable, and the reference monitor denies access requests with two labels that are not comparable by default. A sufficiently privileged operator carefully assigns labels to all of the principals and domains in a system, and the reference monitor uses the ordering rules to prevent any information from leaking except through permitted and known channels.

Four main problems for the security of an information flow control system include (1) the initialization stages including label assignment and policy definitions must be trusted, (2) dynamic modification of label security levels must be trusted, (3) the reference monitor that evaluates access requests must be trusted, (4) and (unknown) covert channels may exist. These problems affect all known implementations. The first problem is passed off to the end user, while the rest are mitigated in part through maintaining a small, simple trusted computing base (TCB); formal proofs of TCB correctness or the security invariants; and runtime heuristics meant to reduce the attack surface.

A traditional, monolithic OS is no longer sufficient for guaranteeing security even with information flow control. The primary reason for such insufficiency is that the TCB of a monolithic OS is too big: vulnerable code executes with supervisor permission and can compromise the mechanisms that enforce information flow control. For example, SELinux [10] is a hook-based framework that supports mandatory access control using file and process labels. With SELinux, the TCB contains the Linux kernel, which is large, complex, and, from a security standpoint, untrustworthy.

The problem of complex, trusted software is not new. To stem the growing complexity of OSs that provided information flow con-

trol, Rushby [20] introduced the idea of a separation kernel over 3 decades ago. Nearly synonymous with the notion of a hypervisor, the separation kernel abstracts the physical machine so that multiple, independent OS and application software stacks can be resident on the same platform. The separation kernel provides only the minimum needed to enforce information flow control between the partitioned software stacks. Usually, separation kernels keep the explicit flows small or nonexistent, and make sharing data among partitions difficult with large overhead. In addition, the separation kernel has an opaque view over the resource usage of the partitions, which frustrates efficient resource management. As such, a separation kernel is best matched to mutually independent applications with statically configured resource limits.

Concepts of microkernels are also related to the separation kernel, especially the idea of putting more of the system services in user-space. Microkernels afford a lower-cost communication pathway, and have less stringent rules about information flow. Notwithstanding, capability-based microkernels—notably EROS [22] and seL4 [14]—enforce information flow control through kernel capability management and restriction of capability propagation.

Decentralized information flow control [15] is an approach that descends directly from the lattice-based theory of information flow control. Distributing the policy management enables each principal in the system to control access to the resources it owns, which in turn reduces the size and complexity of the trusted computing base containing the remaining information flow control mechanisms. Using this approach, Asbestos [5] and HiStar [29] are two related OSs that enforce information flow control with an interface for applications to generate and own new compartments and manage labels for those compartments. A compartment's owner can modify the security level of labels for that compartment, which enables software outside of the trusted computing base to exert control over portions of the information flow control policy. HiStar shows how to use ownership to create explicit control flow for kernel-level software, and how to manage resources such that superuser permission is not required for system administration.

Our hypothesis is that current secure OSs enforcing information flow control are critically flawed due to one key factor: the granularity at which these systems apply and enforce labels. At issue here is that the minimum size of the principal and domain are artifacts of the system design and its underlying protection mechanism. Thus, the reference monitor cannot enforce arbitrarily small principals and domains on commodity hardware when the typical principal is a process, and the two common minimum domains are the memory page and file descriptor. The minimum granularity is a problem because of modern, massively threaded programs that run in a single process executing on behalf of multiple users and accessing resources that do not correspond with memory pages or file descriptors, for example a web server like Apache handles requests for all users in the same process with the same privileges and accesses resources such as CPU time and network devices on behalf of users. Furthermore, such applications may require *service continuity*: a detected attack in one request should not impact the rest. Custom hardware can support information flow control at a much finer granularity [25, 30, 26], yet achieving security with service continuity using off-the-shelf hardware within a single process exhibiting highly dynamic behavior motivates rethinking the assumptions behind access control.

The contributions of this paper are two-fold. First, we introduce quarantine as a novel primitive for dynamically isolating a principal within a new, replicated protection domain. Second, we describe the design decisions and implementation challenges for using quarantine within the COMPOSITE component-based OS. We present preliminary results measuring the performance of quarantine in its first, unoptimized incarnation.

2. LOCKDOWN

In this paper, we introduce the LOCKDOWN operating system, a novel approach to OS design that builds from the COMPOSITE OS as a base layer. LOCKDOWN introduces new mechanisms to COMPOSITE with a focus on the end goal of service continuity. To achieve this goal, LOCKDOWN explicitly tracks dependencies and interactions between principals and dynamically isolates suspicious principals from the rest of the system. This isolation uses a new OS primitive we call *quarantine*. When a high risk of compromise is identified, the principals in question are dynamically disentangled and peeled apart from the rest of the system.

Quarantine is, in some ways, analogous to modifying security labels in an information flow control scheme. The difference, however, is that LOCKDOWN also moves or copies the domains that a quarantined principal previously was accessing. Hence, quarantine effectively creates a new, isolated set of domains for the affected principals, a concept that we liken to a dynamic separation kernel or sandboxing. A key step in LOCKDOWN's quarantine is to *fork* a protection domain, which shares many of the same actions as its POSIX namesake. Importantly, quarantine is an asynchronous event that may occur at any time, which is the source of much of the design complexity. In the following we review the relevant background material for understanding the base COMPOSITE system before discussing the salient challenges involved in designing the quarantine primitive.

2.1 Background: COMPOSITE

COMPOSITE is a microkernel design based on fine-grained decomposition of the OS into modular, isolated components. The COMPOSITE OS is a small kernel with system-level services for managing resource access policies implemented in user-level *components*, *e.g.* scheduling, synchronization, file systems, memory management, and events. These system components can expose hierarchical control of resources thus enabling efficient subsystem composition [17] and permitting flexible tradeoffs between isolation and sharing.

Components have well-defined interfaces based on function definitions. Calling a function in a component's interface causes a *component invocation* using synchronous thread migration [16]. We say that a client component invokes a server component. The client must have a user-level capability for the specific function it calls in the server. Kernel code mediates component invocations using kernel-level capabilities that relate to a static, directed acyclic graph of components.

Protection in COMPOSITE uses page tables to enforce isolation between components. By default, each thread is a principal and the (protection) domain is a single component, although grouping components is possible using mutable protection domains [18]. The design goals for COMPOSITE focus on configurability, predictability, reliability, and scalability. LOCKDOWN adds continuity and security.

2.2 LOCKDOWN Terminology

- Q: the quarantine manager, a system-level service component that implements the quarantine interface.
- P: a principal.
- T: a thread.
- O: a component that gets forked (O for original).
- C: a client component that invokes O, written $C \to O$, read as

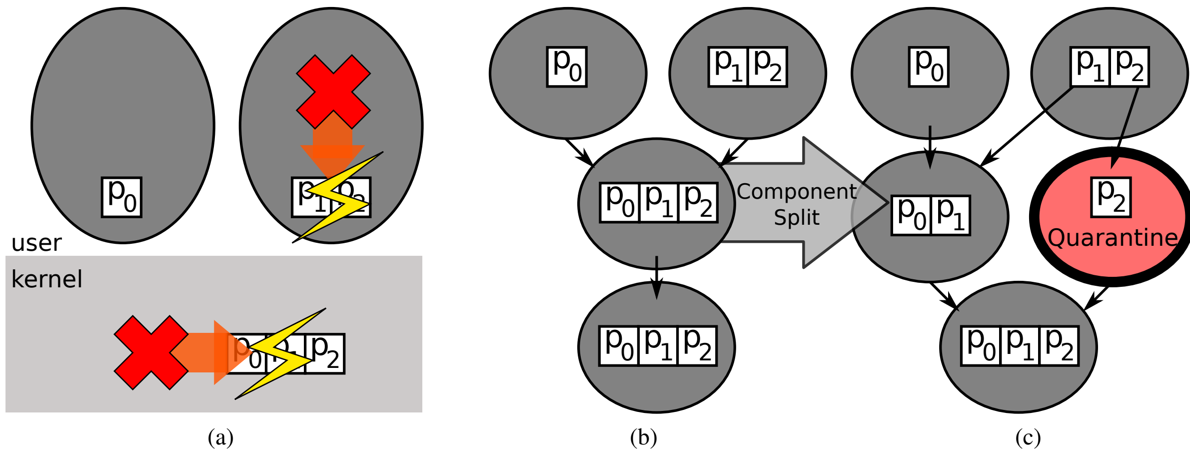

Figure 1: Example of *component forking* in LockDown. Circles are protection domains, and each is annotated with the principals (p_0, p_1 and p_2) that are vulnerable within that domain. (a) Conventional systems where compromises (red Xs) in processes responsible for more than one principal (*e.g.* in a webserver) impact all of them; compromises in the kernel impact all principals. (b) LockDown system (simplified) with four components, and dependencies are arrows. The middle component (*e.g.* networking stack) can be *forked* into two (c) when a principal (p_2 here) is suspicious or high-risk. This dynamic quarantining separates the suspicious principal from the others and mitigates the impact of a potential compromise in that component.

C invokes O. Equivalently, C depends on O.

- S: a server component that O invokes, i.e. $O \to S$. Note that O depends on S.
- F: a forked copy of O.

The main complexity when quarantining a principal P is for Q to disentangle execution state for that principal within a set of components it may access. For simplicity, we refer to a single such component O at a time, but O may be a set of such components. Disentanglement involves three distinct phases: clearing threads, forking components, and routing principals. The following explains the design of each phase.

2.3 Clearing Threads

The first phase is a preparatory step to ensure that forking O will not cause spurious faults or errors. A primary complication when forking O is handling a thread T holding a lock in O, because the critical section is protecting some data that is not necessarily in a globally consistent state. If a fork happened—see below for the technical details—with a held lock, then either the shared data will be inconsistent or there will be a deadlock in whichever component (O or F) does not resume T.

Q deals with this complication by helping T through the critical section before forking O. Q exports the lock interface with an implementation to facilitate tracking threads and locks in order to help threads finish execution through critical sections within O. This help requires that Q knows every T that holds a lock in O, and that Q can manipulate T to execute prior to forking O. This manipulation involves boosting T's priority (giving inheritance to any lock T contends) until T releases the lock in O. The priority boost occurs during disentanglement. Q finds the owners of locks held in O and places the running thread (executing the disentanglement) in the list of waiters for the owner to exit the critical section. This list blocks the thread with priority inheritance, thus T inherits the running thread's priority, which is currently hard-wired to the highest priority. When T releases the lock its priority drops, Q is notified via its lock implementation, and disentanglement proceeds.

2.4 Forking Components

Component forking relocates one or more principals from one protection domain into a different protection domain with copies of resources. We logically sub-divide forking into the following five steps.

1. Copying protection domain O into F. Q replicates all component memory in O into F including code and data.
2. Setting capabilities for F. After copying O, Q may remove (or add) invocation capabilities to F. Currently, this feature is not used, but the mechanism exists.
3. Preparing for post-fork fixups. After forking completes, component invocations need to route correctly to either O or F, see below for the details. To help LockDown detect that a fork has occurred and that the kernel capabilities store outdated routing information, we add two *fork counters* to kernel capabilities, one each for the send and receive sides. Q increments the counters on the send-side for all of O's capabilities, which are copied to F. Q also finds every invocation capability to O, *i.e.* every possible client of O, and increments the receive-side counter for each of them. Q also keeps a mapping between F and O to facilitate tracking a forked component back to its original source component.
4. Fixing server metadata for F. After copying O into F, an invocation $F \to S$ may rely on state established by some previous invocation $O \to S$. This state normally is associated with O's component identifier, which is passed as an argument in the invocation to S. We call this state *server metadata*. The problem to solve then is when and what state to copy, relocate, or recreate in S on behalf of F.

   Q needs to handle some system services like memory management directly during the fork operation. Other system services and application-level components handle the fixup using an *upcall* that executes synchronously with component invocation to S. Since requests from O may update the metadata after a fork, either $F \to S$ or $O \to S$ may trigger the lazy update of this metadata.

   A tricky situation exists in case S depends on another server, say T, and there exist invocations $S \to T$ that pass the component identifier of O. (An example is the *valloc* component, which allocates virtual address space in a destination compo-

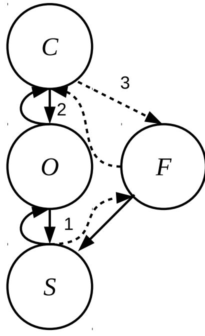

Figure 2: Routing decisions after a fork. Component invocations shown as straight lines. Return paths (using return capabilities in the kernel) shown as curved lines pointing back toward the invoking component. The dashed lines show routing decisions subsequent to forking. 1: the return capability for an invocation made to S prior to the fork may be re-routed to F. 2: F may get the return capability for an invocation made to O prior to the fork. 3: new invocations from C to O may be re-routed to F.

nent that may differ from the invoking component.) In this case, T may be storing metadata related to O that needs to also be fixed transitively due to the fork. We currently address this problem by manually invoking T's upcall entry point if needed. We also considered that this problem goes away if S is forked as well, which following through all of the dependencies of S will effectively create an isolated subsystem similar to a partition of a separation kernel.

5. Migrating threads from O to F, if needed. One possibility is that P already executes within O. In this case, Q could migrate the thread associated with P so that it resumes execution within F instead. Although the mechanisms for thread migration exist, LockDown does not currently migrate threads when forking components.

After these five steps have completed the forked component is ready for quarantined principals. However, the act of forking does not in itself direct the quarantined principals to the forked component. Instead, the capabilities for quarantined principals update lazily when attempting to access the original component, which we call routing the principals.

2.5 Routing Principals

Routing in Composite happens directly from an invocation capability. For example, component C has an invocation capability that, in the kernel, names O as the destination. LockDown requires solving the problem that simply mapping an invocation capability to a destination component cannot support features such as dynamic (per-request/per-thread) routing. The only possibility would be to choose whether to map the invocation capability to O or F, and doing so on every invocation would be cumbersome. Thus, LockDown routes components dynamically by making use of the existing user-level fault handling framework in Composite to update user-level capabilities impacted by a fork. See Figure 2 for a diagram and description of the routing decisions that occur after a fork happens.

During component invocation, the kernel checks if the fork counters are non-zero, and if so a fork has occurred on either end of the invocation. This check triggers fault handling into Q, which then can fix routing in C or upcall into S to fix server-side metadata. The fault handler in Q helps determine whether to route requests from C to O or F, and uses a syscall to adjust the fork counter.

To detect forks during the return paths, LockDown stores an *epoch number* associated with the fork counter within the invocation stack during the invocation kernel code. The kernel compares the epoch with the current counter values during the return path. A mismatch triggers a routing fault in the client.

Fault handlers in Composite, *e.g.* the page fault handler, manipulate the return address on the client-side of the invocation into the faulty component by adjusting it to a fault return path in the client. The Composite fault return path simply sets the return value to a magic number. (In prior work that extends Composite for fault tolerance [24], the fault return path sets a special variable to 1, which is then checked by interface code.)

LockDown changes fault handling in two key ways. First, the fault handler sets a return value by manipulating the return register in the invoking component, C. The return value conveys information back to C about the fault. When the client returns at the modified return address, the fault return path in C checks the value of the return register and determines whether updated routing information exists. Subsequent invocations do not cause a routing fault unless another fork occurs of either the client or server component.

2.6 Summary

Taken together, the three actions of clearing threads, forking components, and routing principals has the intended effect of quarantining the re-routed principals. Non-quarantined principals will still experience a routing fault, but are otherwise unaffected by the forked component or quarantined principal.

3. EVALUATION

LockDown is in a nascent stage and the early efforts focus on achieving a functional level of principal quarantine. In this section, we present the preliminary results of quarantine performance. The data were gathered on an Intel i7-2760QM running at 2.4 GHz with only one core enabled and compiler optimizations (gcc -O3). We use two microbenchmarks that measure the constituent costs of quarantine. The first microbenchmark measures the time taken to fork a component, and the second measures the time taken to fork and invoke a component. In both cases the forked component is invoked once before the first fork, and it does not have any resident threads when the quarantine begins. The second benchmark induces principal routing overhead from the fault handling induced by the invocation to the forked component. Each microbenchmark repeats the measurement 50 times, hence each conducts 50 quarantine operations.

Benchmark	Mean Time (μs)	Std. Dev. (μs)
1: Fork Only	1886.99	269.4
2: Fork and Route	1886.78	269.3

Table 1: Time taken by the fork portion of quarantine, and by fork followed by route. Note that the cost to dynamically route a principal following a fork is minuscule when compared to the fork cost itself.

The data show that forking one component takes approximately 2 milliseconds of execution time. We investigated this cost further and found that about 98% of this time (1858 μs) is spent copying the data from the original component to its fork. Though unsurprising, this finding demonstrates that the system infrastructure for quarantine is efficient: on the order of tens of microseconds besides

the cost of copying memory.

4. RELATED WORK

A fundamental way to increase system security is to break up software into isolated chunks with heightened access control constraining their interaction to provide strong information flow control [21, 5, 29]. This isolation is effective as it is a direct implementation of the principle of least privilege. LOCKDOWN uses this approach by inheriting it from the COMPOSITE component-based system. COMPOSITE goes further than existing systems by ensuring that even system schedulers, memory mappers, and synchronization are component-scoped. LOCKDOWN goes beyond these existing systems by providing isolation and protection along the dimensions of *principals* as well as that of components (functional units) to dynamically quarantine principals from each other and prevent system-wide, and principal-wide, compromises.

Techniques related to sandboxing also overlap with the notion of quarantine, in that supervisor software creates an isolated protection domain to execute untrusted code. In general, the prior art for sandboxing uses a static approach: a protection domain is precreated for the untrusted code before it executes. LOCKDOWN fundamentally differs in that the isolated domain is created on-the-fly, which enables seamless, self-reconfiguring protection of a running system.

Library operating systems (libOS) are similar in spirit to hypervisor-based separation kernels [19, 11, 1]. Graphene [27] identifies the need for fine-grained libOS stacks to support multithreaded applications, and uses a sandboxing approach to enforce a simple access control policy that prevents communication across protection domains. Our work differs in a few key ways, the primary being that communication is possible between protection domains. Key to our approach is the fine-grained modularization of system services that enables an efficient quarantine operation. It would be of interest to explore quarantine in the context of a libOS.

System verification efforts have attempted to dramatically heighten confidence in systems by mathematically verifying a *small functional core* of the system [22, 9, 28, 13, 12]. Even systems that do not go so far as to provide mathematical verification (*e.g.* [8, 23, 7, 24]) go to great effort to minimize the complexity of the TCB. LOCKDOWN follows the same general approach by removing complexity and policy from the system kernel, and instead defining it in configurable components that can each manage disjoint sets of resources. Thus, in LOCKDOWN the TCB should approach the complexity of, or be smaller than, existing systems.

LOCKDOWN harnesses past work by extending capabilities, that are a known mechanism for access control in secure systems. Capabilities are tokens that designate access to an underlying resource [4], and are the backbone of many secure systems [2, 21, 6]. LOCKDOWN expands on previous component-based systems by integrating principal-specific information into the capabilities to provide another dimension of checking that enables expanded access control, forking, and dynamic principal routing after a quarantine event.

5. CONCLUSIONS AND FUTURE WORK

Although LOCKDOWN is now in a useable state, some of the design and implementation details remain either unspecified or incomplete. In addition to areas noted in the above, we are also working toward the necessary support to quarantine system-level components especially those that manage resources such as schedulers and memory managers. Our current focus however is on reducing the overhead of quarantine in our system by optimizing the cost to copy component memory, which currently consumes 98% of the time taken by forking. We are also working toward integrating the quarantine support with a web server to provide for fine-grained isolation of web requests as an application benchmark and case study. In the longer term, we are interested in comparing the quarantine primitive with other isolation techniques including static and dynamic approaches.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS 1149675, CNS 0934725, and ONR Award No. N00014-14-1-0386. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or ONR.

6. REFERENCES

[1] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing OS Extensions Safely and Efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 239–252, New York, NY, USA, 2013. ACM.

[2] A. C. Bomberger, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The keykos nanokernel architecture. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Berkeley, CA, USA, 1992. USENIX Association.

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

[4] J. B. Dennis and E. C. V. Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 26(1):29–35, 1983.

[5] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 17–30, New York, NY, USA, 2005. ACM Press.

[6] K. Elphinstone and G. Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 133–150, 2013.

[7] M. Engel and B. Döbel. The reliable computing base âĂŞ a paradigm for software-based reliability. In *Proceedings of Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES)*, 2012.

[8] M. Hohmuth, M. Peter, H. Hartig, and J. Shapiro. Reducing tcb size by using components untrusted small kernels – small kernels versus virtual machine monitors. In *Proceedings of the 11th ACM SIGOPS European Workshop*, 2004.

[9] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, Oct 2009. ACM.

[10] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[11] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM.

[12] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying security invariants in expressos. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[13] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. Rocksalt: better, faster, stronger sfi for the x86. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2012.

[14] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: From General Purpose to a Proof of Information Flow Enforcement. In *2013 IEEE Symposium on Security and Privacy (SP)*, pages 415–429, May 2013.

[15] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 129–142, New York, NY, USA, 1997. ACM Press.

[16] G. Parmer. The case for thread migration: Predictable IPC in a customizable and reliable OS. In *OSPERT*, 2010.

[17] G. Parmer and R. West. HiRes: A system for predictable hierarchical resource management. In *RTAS*, 2011.

[18] G. Parmer and R. West. Mutable protection domains: Adapting system fault isolation for reliability and efficiency. In *ACM Transactions on Software Engineering (TSE)*, July/August 2012.

[19] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library OS from the top down. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 291–304, 2011.

[20] J. M. Rushby. Design and Verification of Secure Systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. ACM.

[21] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999.

[22] J. S. Shapiro and S. Weber. Verifying the eros confinement mechanism. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 166, Washington, DC, USA, 2000. IEEE Computer Society.

[23] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing tcb complexity for security-sensitive applications: Three case studies. In *Proceedings of EuroSys 2006*, April 2006.

[24] J. Song, J. Wittrock, and G. Parmer. Predictable, efficient system-level fault tolerance in C^3. In *Proceedings of the 2013 34th IEEE Real-Time Systems Symposium (RTSS)*, pages 21–32, 2013.

[25] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *SIGPLAN Not.*, 39(11):85–96, Oct. 2004.

[26] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 189–200, New York, NY, USA, 2011. ACM.

[27] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[28] J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[29] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazieres. Making information flow explicit in histar. In *OSDI '06: Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 263–278, 2006.

[30] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 225–240, Berkeley, CA, USA, 2008. USENIX Association.