Nested Java Processes: OS Structure for Mobile Code

Patrick Tullmann *Jay Lepreau*

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

`tullmann,lepreau@cs.utah.edu`
`http://www.cs.utah.edu/projects/flux/`

Abstract

The majority of work on protection in single-language mobile code environments focuses on information security issues and depends on the language environment for solutions to the problems of resource management and process isolation. We believe that what is needed in these environments are not ad-hoc or incremental changes but a coherent approach to security, failure isolation, and resource management. Protection, separation, and control of the resources used by mutually untrusting components, applets, applications, or agents are exactly the same problems faced by multi-user operating systems. We believe that real solutions will come only if an OS model is uniformly applied to these environments. We present *Alta* , our prototype Java-based system patterned on Fluke, a highly structured, hardware-based OS, and report on its features appropriate to mobile code.

1 Operating System Model Required

In the last European SIGOPS Workshop, our paper [15] argued that the local operating system is an essential foundation for global applications. We described the many demands that a reasonably well functioning distributed system places on the local OS, and particularly emphasized end-system security in the widespread presence of mobile code. The focus of that paper was on making the case for the importance of the local OS, and outlining an appropriate OS for that environment: the Fluke [9] operating system, an OS based on a recursive virtual machine model, analogous to the Cambridge CAP Computer [28], but implemented by a microkernel instead of special hardware.

In this paper we assume that the importance of the local OS to distributed applications is evident. From that base, we endeavor to make four points concerning platforms for mixed trust components and mobile code:

(i) A coherent, structured approach is required, driven by a full-blown OS model; not simply language-level patches.

(ii) Existing security-oriented approaches fall short in resource management.

(iii) Applying an OS model is feasible, based on our initial experiences with Alta.

(iv) Alta provides features useful for mobile code, including hierarchical resource management and flexible object sharing.

1.1 Application Scenario

In 1997 MCI developed and distributed its Denial of Service Tracker (DoSTracker) [17], after getting their router vendor to add the required interfaces and code to the routers. DoSTracker works as follows: many denial of service attacks involve generating packets that spoof the IP address of the victim's host. For example, fabricating broadcast packets will generate a storm of replies to the "sender." When a customer reports an attack on a particular host, their ISP runs DoSTracker on a machine connected to the victim's router, giving it the victim's IP address. DoSTracker hops from router to router, following spoofed broadcast packets "upstream" to the actual source. Problems arise when the path leads into another Internet carrier's hosts—a different administrative and technical domain—whose routers may well not support the required interfaces.

Similar hard to predict problems arise constantly in network management, and solutions are difficult to deploy quickly, and almost impossible to standardize. A first step to providing network administrators with a solution to these problems might let them run mobile programs

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army under contract number DABT63–94–C–0058, and Air Force Research Laboratory, Rome Research Site, USAF, under agreement number F30602–96–2–0269. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

on the routers. This, of course, is one example of an active network [26]. One need not commit to the aggressive vision of active networks—code in any packet—to appreciate the value of supporting mobile code in routers. Network management is an application domain that could greatly profit from mobile code and dynamic composition of mobile components.

However, along with the solutions proffered by mobile code there must be strong security guarantees and flexible, hierarchical resource management. Consider the following realistic Internet-wide scenario of hierarchical trust and proportional share resource management. MCI reserves 80% of the resources in each of its routers for "real work" (i.e., forwarding packets). The other 20% is available on demand for management functions (such as DoSTracker), mobile code, or agents. 50% of *that* (i.e., 10% of the total) is reserved for MCI's own management routines, with the rest available to its customers. However, all customers are not equal, so MCI allocates 50% of that 10% to the 20-odd long-haul Internet carriers, such as Digex[1] or AT&T , and the other 50% to other customers (e.g., ISPs). The 5% allocated to the long-haul Internet carriers could again be split up equally among the carriers—effectively each internet carrier owns a modest 0.25% of every other carrier's available bandwidth. Digex manages its portion (on any carrier), allocating half to trusted (to Digex) requests from its own network management, and the other half to Digex customers. See Figure 1. Clearly, a hierarchical, extensible resource management model would provide the ability to recursively refine system allocation. Additionally, a stringent security infrastructure to authenticate and manage the mobile agents in such a system is required.

Figure 1: Hierarchical breakdown of the resource subdivisions of a hypothetical MCI router.

1.2 The Process

When addressing the control of mobile code and foreign components, the information security issues raised by such code receive the lion's share of attention from researchers, developers, and the press. These issues are indeed important, the efforts are valuable, and progress is being made. However, little attention has been paid to a distinct but important aspect of controlling foreign code: resource management. Although solely providing information security may be sufficient to incorporate foreign components into an interactive browser running on a desktop OS, more ambitious visions of distributed components clearly require advances to control the resources available to foreign code.

We argue that these mobile code systems need a coherent, structured approach to both information security and resource management. We need not look far: the "process" abstraction has historically been the unit of protection and resource management in operating systems. Typically, the operating system has used a hardware memory management unit (MMU) to keep processes separate. Many language-based systems, such as the kernel extensions in SPIN [3] and Java applets in Web browsers, use a type-safe implementation language to provide memory protection in place of a hardware MMU, so that the resulting protection abstraction will be more flexible and lighter weight than a traditional process. But a process is more than just a layer of separation, and most of current language-based systems are lacking a comparable abstraction. Indeed, a Java applet's execution environment is little more than "threads" with a few ad-hoc constraints—a situation that provides almost no control over resource use, and has led to numerous security problems. Although steps are being taken to improve the security situation in Java [12, 27], that is not enough. Architects of multi-user, safe-language environments need to confront the need for a first-class abstraction for protection and resource management—the "process."

1.3 Existing Safe-language Systems

Using a type-safe language to provide similar services as an MMU is not novel, but previous approaches have either not been designed for malicious, multi-user code or address only one of security and resource control.

Software-based Approaches

Java started with the "sandbox," which was limited to providing all-or-nothing access control, depending on whether the source of the code was local or remote. JavaSoft relaxed the sandbox model in JDK 1.2, which introduces access control lists (ACLs) and signed code [12]. However, policies in the JDK are expressed via ACLs (making it error prone at large scale, just like ACLs in Unix) and there is no notion of user authentication—principals are tied to code signatures. Additionally, protection domains are only created implicitly, through code loading.

[1] Digex is an example of a modest-sized Internet carrier that leases capacity from the big long-lines carriers.

Numerous other approaches to Java security exist [27], including more-or-less traditional capabilities, "stack introspection" which intuits the current principal based on the call stack, and name space management, which manipulates the class name space to control suspect code's access to "dangerous" functionality. Although the vendors and developers who have produced systems based on these approaches make claims otherwise, each of these three approaches is only a *mechanism* for controlling code. With one exception noted below, as currently applied they have not yielded comprehensive designs.

Recent work designed extensions to the type system of a (non-Java) language to support controls on information flow [18]. Refreshingly, the authors do not claim that this approach provides a complete solution to security woes, but contemplate using this approach to augment the security of applications on both traditional and non-traditional operating systems.

Sun's original JavaOS [25] was a standalone OS written almost entirely in Java. Although it billed itself as a first-class OS for Java applications, from the scant literature available, it appeared to provide a single JVM with no particular separation between applications running on it. Recently, Sun has migrated their JavaOS to three products built on the ChorusOS (neé Chorus [21]), while little public documentation is available on these new systems, we believe they are targeted to the same environments as the original JavaOS and will not be tailored to management of multiple, mutually untrusting applications, and instead focused on the controlled environments of consumer appliances and centrally-managed network computing.

The Oberon language/OS [30] shares many of Java's features, although it is a non-preemptive, single-threaded system. Protection between tasks is enforced by the language, but the exposure of global state to all top-level procedure calls ("commands") and the uninterruptibility of those commands means isolation is not enforceable, and malicious code can take free reign of the system. The Juice project [11] uses Oberon and Slim Binaries to provide a Java-like environment for downloaded code. Juice addresses many of the performance and security problems associated with large Java binaries and the necessity of run-time security checks in the JVM. Juice does not, as far as we know, provide resource controls.

Cornell's J-Kernel [13] and the OpenGroup's Conversant [2] are other efforts, besides ours, that fully recognize the need for first-class notion of protection domain in safe-language-based systems. The J-Kernel's capability-based system concentrates on clean domain termination and provides this by enforcing complete separation of processes (i.e., no sharing)—even between domains that have some degree of trust between them. Conversant goes a step further and modifies the Virtual Machine to put each application in wholly separate address ranges and provides per-process garbage collection threads. But, denying the ability to easily share objects between processes defeats the greatest advantage of safe-language based systems: safe, flexible sharing.

Hardware-supported Approaches

Two groups have independently developed a Java applet execution model that requires the applets to run on dedicated, specially protected and isolated hosts: the "Java Playground" [16] and Digitivity's "Cage" [7]. However, this model is severely flawed: it imposes inherent limits on the functionality achievable from mobile code, essentially limiting execution to the restrictive Java applet model (a model useful for little more than "dancing pigs"), disallowing the richer semantics that could be made available if the local system provided the requisite security and resource controls. Additionally, these systems can only guarantees to "your" system that it will protected from "their" code, and cannot provide any guarantees (security-wise or resource-wise) to the code running on them. The Kimera [22] system has similarities, in that the verifier and compiler must be run on specially isolated hosts, but the resulting native code is allowed to run in the browser on the desktop.

A number of microkernel style OS's have been built which provide either relatively high security or resource management for mobile code, but not both. For example, the Lava system [14], provides tight security between different JVMs, at acceptable cost, but apparently no particular resource management. Nemesis [20] was one of the earlier systems to concentrate on QoS issues, achieving good performance and good control. Eclipse [5] similarly concentrates on QoS, but for multiple resources. Joust [19], a JVM integrated into the Scout operating system, provides CPU scheduling but no memory resource control, and as of yet, no explicit security mechanisms, though the associated Escort effort [24] is working on it.

Finally, in contrast with influential systems such as Eden [1] and Emerald [4], the current tendency in language-oriented operating systems is to relegate OS design to an afterthought, relying on the features of "advanced languages" for security and structure. Eden and Emerald indeed relied on special languages, but devoted primary effort to OS structure.

2 An OS Structure for Mobile Components

2.1 The Archetype

We chose our own system, Fluke, as a model, not simply because we believe in its design, but because we understand its details. Any coherently structured OS which provides both security and resource controls might serve well as a model. For supporting mobile components, we believe Fluke offers advantages over existing systems because it lends support to highly structured systems, provides extensible resource management, and optionally provides policy-flexible mandatory security mechanisms.

The key aspect of Fluke is that it supports hierarchically structured processes: the "nested process model." Fluke is designed to make it straightforward for arbitrary user-level processes to completely encapsulate child environments and provide useful services to those environments. This design optionally exposes all of the resources required by a process to the creator of that process. For example, in Fluke a user-level server which completely controls the memory resources of child environments is able to provide virtual memory services. The model's CPU inheritance scheduling framework [10] allows normal, unprivileged threads to schedule child threads—permitting widely different scheduling policies to coexist in a single system. CPU use, memory use, and file system and network access can be mediated by any ancestor. In the interests of performance, the kernel *enables* process hierarchy, but does not *require* it. For example, a process can create multiple children and closely manage their file system accesses, while leaving memory and CPU management to its ancestor. Figure 2 demonstrates such variant hierarchies for different resources in the same process tree. The interfaces for memory management and CPU scheduling are provided directly by kernel operations, while "higher level" operations, like file system and network access, are managed entirely through a traditional capability system and IPC to user-level servers.

Flask is a security-enhanced version of Fluke [23], developed with collaborators at the U.S. Department of Defense. Flask provides a mandatory security framework for the Fluke kernel and associated security-aware servers. Flask also provides a mechanism-independent authentication subsystem, used by the kernel to establish and verify a principal's identity, and a separable cryptographic subsystem which isolates applications from the details of cryptographic algorithms. A key feature of Flask is the separation of security enforcement from security policy decisions. In Flask, the kernel and servers within the TCB (trusted computing base) contain permission checks which inquire of a "security policy server" whether a given (subject, operation, object) triple is permitted. The policy server's decision is based on the security contexts of the involved objects, and is enforced by the kernel or the object's server. This complete separation of policy and mechanism makes the Flask model relevant to diverse communities with widely varying security policies.

2.2 Alta: Nested Processes in Java

Using the Fluke model, we are developing Alta, a Java-based system which uses software protection mechanisms to maintain the safety and separation of processes. One of our goals is to determine the extent to which a Java-based system can provide the same environment as a traditional microkernel+servers system. While the design of Fluke assumed a traditional hardware protection based microkernel, its abstractions turn out to be independent of that environment, with a few minor exceptions.[2] A second and important goal is to bring the nested process model to Java. A system with such properties is well matched to the needs of mobile code and distributed applications. Considering that mobile code systems will tend to use platform-independent safe languages for other reasons, moving OS functionality into the safe language environment will avoid the overhead and indirection associated with starting and maintaining multiple environments. For example, the Lava system provides IPC performance that is remarkable for a hardware protection based system, but to effectively support mobile Java code it must instantiate a Java Virtual Machine in each process and processes must communicate with the JDK's RMI.

In our Java-based implementation of the Fluke model we can leverage the flexibility afforded by a type-safe environment to improve and expand the nested process model. For example, in Fluke objects passed into the kernel have user-level and system-level portions. When a process passes in its user-level portion, the kernel must look up the system portion. Java's 'private' modifier on object fields provides the same service much more elegantly. As another example, type safety means that a process can provide an object handle to another process with the full confidence that the receiving process will only be able to access visible portions of the object. This contrasts with hardware based environments, where sharing is an all-or-nothing affair since it (generally) relies on raw memory page sharing.

We have extended the Fluke model to include direct object sharing via IPC, that is, IPC messages can contain raw byte data and capabilities (as in Fluke), but IPC messages can also contain Java object references. In Fluke, processes can directly share state via shared memory pages.

[2] The abstraction for sharing memory between processes explicitly used virtual addresses, which are only meaningful in a hardware protection environment.

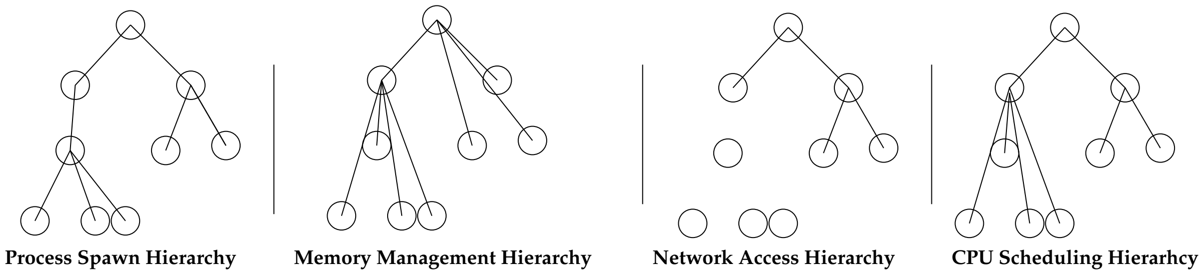

Figure 2: An example process hierarchy and the associated memory, CPU and Network access hierarchies. In all cases, the root of the system is the "kernel." The lines represent the relationship between a process and the ancestor process to which resource requests are directed, demonstrating that the hierarchies can be different for different resources.

In Java, sharing memory is accomplished by sharing object references. Sharing object references between domains can lead to problems with domain termination and memory usage accounting, as described in [13], but when inter-process trust allows, such flexibility is useful. Additionally, we feel these problems are the domain of applications and can be adequately handled at the application level. If a process needs to restrict the sharing of object references then it must interpose on the communication channels where the reference can be sent. This sort of interposition and monitoring is already required in capability based systems to prevent unwanted capability propagation.

In contrast to the argument that a capability-based system in Java would require restructuring all of the Java system classes [27], we contend that the Java system classes do an excellent job of containing and reusing the methods that need to be modified to support a capability system. For example, to control I/O operations only the basic classes `java.io.{File, FileInputStream, FileOutputStream, RandomAccessFile}` need to be changed. All of the other interfaces and classes in the `java.io` package rely on these classes to do file-based input and output.

Finally, Java defines some services and abstractions that have no analogue in Fluke. The two major differences are the garbage collector and dynamic class loading.

2.3 Class Loading

We have extended Alta to handle class loading as an external service in the same fashion as page faults in Fluke are handled. Outside of a core set of critical system classes which are pre-loaded into every process, the majority of classes are loaded on demand by the class fault handler for the process. Additionally, we have divorced the name of a class from its implementation, which allows processes to substitute classes. For example, a process could substitute `edu.utah.cs.npm.io.NoAccessInputStream` when a request for `java.io.InputStream` comes in, or it could substitute a version of the class that uses IPC to perform file access operations. The class fault mechanism relies on the link-time integrity checks performed by the virtual machine to guarantee that the class object provided in response to a class fault is appropriate.

2.4 Garbage Collection and Process Separation

Garbage collection poses a number of problems for any system attempting to precisely manage resources. All of these problems stem from fine-grained object sharing between domains. Our current design for the nested process model in Java provides processes with mechanisms to restrict inter-domain object sharing. (Note that inter-domain IPC-endpoint references do not cause these problems because of the way they are implemented by the system.) In this way, processes can allow mutually-trusting child processes to communicate freely, with the caveat that domain termination and GC costs are also tightly coupled. We outline these problems as open issues for which more elegant system-level solutions are needed.

First, domain termination is difficult in a garbage collected system with shared objects. In a traditional system the kernel can simply unmap all of the memory a process is using, but in a single-address space GC system, the memory originally associated with a process may be reachable by other processes. Second, in order to provide comprehensive resource management and accounting, the system must be able to charge domains for the garbage collection that they incur. The standard accounting policy is to treat garbage collection as a system service that is performed for the benefit of the system as a whole. Unfortunately, such a simple-minded policy makes the system vulnerable to poorly-behaved or malicious processes, and weakens the system's ability to provide strong QoS guarantees. For example, under an incremental GC scheme, allocations in a process A will reduce the number of allocations that a subsequent process B can perform without triggering a minor GC. Even if GC is "free" and not charged to B, the shared heap weakens the guarantees of timely execution that the system can provide to process B or any other process. Third, in a system that allows fine-grained sharing, simply charging the process that allocates an object for the memory and not the processes that use the object is too inflexible. Charging processes that use objects for the memory required would be more representative of the actual memory costs involved, but could lead to asynchronous memory exceptions. For example, consider a large object to which many domains hold a reference and where each domain is charged a portion of the total memory cost; if many of the domains release their reference to the object the memory cost for the remaining domains will asynchronously increase.

2.5 Status

We have built Alta as a modification to Kaffe [29], a freely available, Java bytecode-compatible virtual machine that supports both interpretive and translator (JIT) modes. We have enhanced `java.lang.Thread` and other core classes to support our semantics, by modifying the Java core libraries provided by the Kore project [6], a clean-room implementation of the basic java classes compatible with JDK version 1.0.2.

Alta currently supports multiple applications on a single virtual machine and correctly handles unknown class "faults:" an IPC message is generated to the parent process which can resolve the name to any class to which it has access. IPC, ports, capabilities ("References" in Fluke terminology), spaces and threads are all working objects. Alta enforces per-process memory limits, and supports direct reclamation of system objects in terminated domains. Extensions to the virtual machine and the Java objects to support CPU inheritance scheduling are under active development.

Alta runs atop normal OSs in user-mode, and will also run in kernel-mode when linked with our OSKit [8]. The latter system should provide a true Java-based OS appropriate for supporting distributed Java components. Finally, we will learn to what extent a particular OS structure can be built atop drastically different protection mechanisms. The initial results are promising.

Acknowledgements

We are especially grateful to Godmar Back and Wilson Hsieh for enlightening discussion and collaboration.

References

[1] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, Jan. 1985.

[2] P. Bernadat, L. Feeney, D. Lambright, and F. Travostino. Java Sandboxes meet Service Guarantees: Secure Partitioning of CPU and Memory. Technical Report TOGRI-TR9805, The Open Group Research Institute, 11 Cambridge Center, Cambridge, MA 02142, June 1998.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety, and Performance in the SPIN Operating System. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, Dec. 1995.

[4] A. P. Black, N. Huchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Trans on Software Engineering*, SE-13(1):65–76, 1987.

[5] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *Proc. of the 1998 USENIX Annual Technical Conf.*, New Orleans, LA, 1998.

[6] G. Clements and G. Morrison. Kore—an implementation of the Java(tm) core class libraries. ftp://sensei.co.uk/-misc/kore.tar.gz.

[7] Digitivity Corp. Digitivity CAGE, 1997. http://-www.digitivity.com/overview.html.

[8] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for OS and Language Research. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, pages 38–51, St. Malo, France, Oct. 1997.

[9] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, pages 137–151, Seattle, WA, Oct. 1996. USENIX Assoc.

[10] B. Ford and S. Susarla. CPU Inheritance Scheduling. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, pages 91–105, Seattle, WA, Oct. 1996. USENIX Assoc.

[11] M. Franz and T. Kistler. Juice web page. http://-www.ics.uci.edu/ juice/intro.html.

[12] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *USENIX Symp. on Internet Technologies and Systems Proc.*, pages 103–112, Monterey, CA, Dec. 1997. USENIX.

[13] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing Multiple Protection Domains in Java. In *Proc. of the 1998 USENIX Annual Technical Conf.*, New Orleans, LA, 1998.

[14] T. Jaeger, J. Liedtke, and N. Islam. Operating System Protection for Fine-Grained Programs. In *Seventh USENIX Security Symp. Proc.*, pages 143–157, Jan. 1998.

[15] J. Lepreau, B. Ford, and M. Hibler. The Persistent Relevance of the Local Operating System to Worldwide Applications. In *Proc. of the Seventh ACM SIGOPS European Workshop*, pages 133–140, Sept. 1996.

[16] D. Malkhi, M. Reiter, and A. Rubin. The Design and Implementation of a Java Playground. Technical Report 97-32, AT&T Labs-Research, Oct. 1997.

[17] MCI. networkMCI DoS Tracker: Denial of Service Tracker. http://www.security.mci.net/dostracker.

[18] A. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, pages 129–142, St. Malo, France, Oct. 1997.

[19] L. L. Peterson et al. Joust: A Platform for Liquid Software. Technical Report 97–16, Univ. of Arizona, CS Dept., Nov. 1997.

[20] T. Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, Queens' College, University of Cambridge, Apr. 1995.

[21] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. The Chorus Distributed Operating System. *Computing Systems*, 1(4):287–338, Dec. 1989.

[22] E. G. Sirer, A. J. Gregory, and B. Bershad. Kimera: A Java System Architecture. http://kimera.cs.washington.edu/.

[23] S. Smalley. Flask: Flux Advanced Security Kernel. http://-www.cs.utah.edu/projects/flux/fluke/html/flask.html, Jan. 1997.

[24] O. Spatscheck and L. L. Peterson. Escort: A Path-Based OS Security Architecture. Technical Report 97-17, Department of Computer Science, University of Arizona, Nov. 1997.

[25] Sun Microsystems, Inc. JavaOS: A Standalone Java Environment, Feb. 1997. http://www.javasoft.com/products/-javaos/javaos.white.html.

[26] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, Jan. 1997.

[27] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architectures for Java. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, pages 116–128, Oct. 1997.

[28] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and its Operating System*. North Holland, NY, 1979.

[29] T. Wilkinson. Kaffe—A virtual machine to compile and interperet Java(tm) bytecodes. http://www.kaffe.org/.

[30] N. Wirth and J. Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, 1992.