AFRL-IF-RS-TR-2000-43
Final Technical Report
April 2000

THE FLUKE SECURITY PROJECT

University of Utah

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. D781

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2000-43 has been reviewed and is approved for publication.

APPROVED: Kevin A. Kwiat

KEVIN A. KWIAT
Project Engineer

FOR THE DIRECTOR:

WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFGA, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

THE FLUKE SECURITY PROJECT

Jay Lepreau

Contractor: University of Utah
Contract Number: F30602-96-2-0269
Effective Date of Contract: 01 August 1996
Contract Expiration Date: 31 July 1999
Program Code Number: D7810001
Short Title of Work: The Fluke Security Project

Period of Work Covered: Aug 96 – Jul 99

Principal Investigator: Jay Lepreau
Phone: (801) 581-4285
AFRL Project Engineer: Kevin Kwiat
Phone: (315) 330-1692

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Kevin Kwiat, AFRL/IFGA, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE APRIL 2000	3. REPORT TYPE AND DATES COVERED Final Aug 96 - Jul 99

4. TITLE AND SUBTITLE
THE FLUKE SECURITY PROJECT

5. FUNDING NUMBERS
C - F30602-96-2-0269
PE - 62301E
PR - D781
TA - 00
WU - 01

6. AUTHOR(S)
Jay Lepreau

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Utah
Department of Computer Science
50 Central Campus Drive, RM. 3190
Salt Lake City, UT 84112-9205

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

Air Force Research Laboratory/IFGA
525 Brooks Road
Rome NY 13441-4514

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2000-43

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Kevin Kwiat/IFGA/(315) 330-1692

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This document presents the final report of the University of Utah-based project officially titled "Mach 4 Kernel and IDL Infrastructure for Security" but often referred to as the "Fluke Security" Project. This project is noteworthy for the number diversity, scale, robustness, and documentation of the six new software systems that were designed and developed. Although the project's primary focus was security and resource management in local operating systems, independently valuable results were also achieved in diverse technical areas conceived in support of the primary goals. These included compilation of interface definition languages, component-based software and software reuse, typesafe language technology, and distributed object systems.

14. SUBJECT TERMS
Operating System Kernel. Secure Operating System

15. NUMBER OF PAGES
40

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

1 Overview 3

2 Results 5

2.1 The Fluke/Flask Kernel and Operating System 5

2.1.1 Fluke/Flask OS Components 6

2.1.2 Recursive Virtual Machine Architecture 7

2.1.3 Flexible Checkpointing 7

2.1.4 Atomicity of Kernel API and Kernel Operations 7

2.1.5 CPU Inheritance Scheduling 8

2.1.6 The Fluke Device Driver Framework 8

2.1.7 Extending Fluke IPC for Transparent Remote Communication 9

2.1.8 The Flask Security Architecture: System Support for Diverse Security Policies . . . 10

2.1.9 Implementing Mandatory Network Security in a Policy-flexible System 10

2.1.10 Utah Assurance Work: Applying Formal Methods to Fluke IPC 10

2.1.11 External Assurance Work: Formal Analysis of Fluke/Flask 11

2.1.12 Kernel Performance 11

2.2 The OSKit: Reusable Components for OS Implementation 12

2.2.1 OSKit Components 13

2.3 The Flick Interface Definition Language Compiler 15

2.3.1 General, Flexible, and Optimizing IDL Compilation 15

2.3.2 Flexible IDL Compilation for Complex Communication Patterns 16

2.4 MOM: The Mini Object Model 17

2.5 Java Operating Systems 17

2.5.1 Making the Case: OS Structure for Mobile Code 17

2.5.2 The Alta Operating System 17

2.5.3 Java Operating Systems: Comparative Design and Implementation 18

2.5.4 Drawing the Red Line in Java 18

2.5.5 Memory Management: The Need for Predictable Garbage Collection 19

2.6 The Khazana/KOLA Infrastructure for Building Distributed Services 19

3 Technology Transfer 21

3.1 Software 21

3.2 Personnel 21

3.3 Relationships 22

4 Publications 23

1 Overview

This project focused on developing advanced security technology in the areas of operating systems, software components, distributed shared memory, and language types. The security emphasis of much of the work was on *resource management*. As originally conceived and proposed in April 1995, the software base for the OS work was to be Utah's version of the *Mach* kernel and operating system, the component work was to build on the *Flick* IDL compiler and Utah's *OMOS* object server, and the distributed shared memory work was to be an extension of Utah's nascent *Quarks* system, oriented to closely coupled cluster environments.

However, the grant did not actually begin until over a year later, in August 1996. By then we had made progress that caused some of our research infrastructure and our research emphases to change.

With DARPA's approval, in late 1995 we moved away from the Mach kernel as an OS development base to *Fluke*, a new operating system architecture and microkernel that we developed. For component work we moved from OMOS to the *OSKit*; the Flick IDL compiler remained an important research base, as originally proposed. We also moved our distributed shared memory focus from the local area to the more general and, we believe, more important problem of providing infrastructure to a wide variety of distributed applications and over a wider geographical area than a cluster. Out of this grew the *Khazana/KOLA* system. Finally, we applied language type-based protection to developing *Java operating systems*, applying our OS structuring techniques to Java virtual machines.

These changes set the context for all of the work performed under this grant. We now summarize original and new approaches in each of the areas of work in the original proposal, plus three entirely new areas.

- Mach kernel and OS: resource management and information control
  New Fluke kernel and OS: new infrastructure, same topic

- Adaptation
  Replaced by adaptive handling of network bandwidth: OSKit hpfq component

- Flick IDL Compiler
  Same

- Secure local DSM
  Khazana/KOLA: Wide-area caching infrastructure

- Language type-based security
  Language type-based (Java) OS implementations: Alta, GVM

- Application to test cases
  Secure environments (wrappers), plus design of sophisticated applications (NRL pump)

- Official collaboration
  NSA: very close; WUSTL, BBN, CMU: moderate

- *Entirely new areas, not originally proposed:*
  - *Flask high-security kernel and OS*
  - *User-level device drivers*
  - *Formal methods applied to Fluke IPC*

Scope: Approximately 2.5 years of this 3 year project overlapped with another DARPA-funded project of ours, "Fast and Flexible Mach-Based Systems," AO #B799, monitored by the Department of the Army. As discussed with our Rome Air Force Research Lab grant monitors and DARPA Program Manager during their visit to Utah on 18 March 1998, the extremely intertwined nature of these projects made it impossible to separate and allocate tasks entirely to one contract or the other. Therefore they approved having our quarterly reports partially overlap in content. We follow the same procedure in this report: results that were achieved before or after the overlapping 2.5 years are reported only in the appropriate report; during the overlap results are attributed to both grants. A few of the papers show a publication date beyond the end of the respective grant period; in these cases the reported work and all or nearly all of the writing was actually accomplished during the grant period. Normal latency in the publishing process accounts for the apparent discrepancy.

Administrative notes: We had an unusually low rate of expenditures during the early years of this project. The most important reason for this was the project's technical overlap with the DARPA-funded project mentioned above, combined with the scarcity of "systems" students and qualified staff due to high demand by industry, compounded, for staff, by University-mandated pay scales. (With the current Silicon Valley industrial boom, this problem will just get more severe.) However, much of the work was still carried out under the other DARPA-funded project. That project could not find enough staff either, so the pace of progress in both projects was slower than planned. Since no extension of this grant's duration was possible, the low expenditure rate in the grant's early years caused the total funding received under this grant to be only $1,742,987, not the originally planned amount of $2,542,987. This 31% reduction in funding did cause some reduction in our planned accomplishments, but nevertheless, we did deliver a great number of results, both in software and publications.

Outline: The bulk of the rest of this report is structured around the many software systems developed under this project. They fall into six categories:

1. The Fluke/Flask Kernel and Operating System
2. The OSKit: Reusable Components for OS Implementation
3. The Flick Interface Definition Language Compiler
4. MOM: The Mini Object Model
5. Java Operating Systems
6. The Khazana/KOLA Infrastructure for Building Distributed Services

This is followed by some comments on technology transfer in Section 3. The report concludes with a categorized list of publications.

2 Results

2.1 The Fluke/Flask Kernel and Operating System

Background: In mid-December 1995 we met with Gary Koob, our DARPA program manager for the related contract (AO #B799), and discussed a change in strategy to achieve that project's original goals, as well as to attempt more ambitious ones. In the next quarter that proposed direction crystallized and work accelerated in that direction. Teresa Lunt (at the time our DARPA program manager for this grant), Gary Koob, Bob Parker, and Jeff Turner (NCSC/NSA), all seemed pleased with our new strategy, and we were gratified both by the response and our own progress.

In summary, before this grant began, we moved away from the Mach kernel to *Fluke*, a novel operating system structure designed for high assurance and efficient support of recursive virtual machines.[1] We conceived the new structure, developed the requirements for the new kernel, and designed the kernel architecture.

During the term of this grant, we worked closely with the National Security Agency and Secure Computing Corporation to design and implement additions to Fluke to form a high-security system called *Flask*. We implemented a prototype Fluke/Flask kernel on the Intel x86, implemented ten virtual machine monitors and servers, including a virtual memory manager, a checkpointer, a process manager, a file server, a network server, a secure network server, and a server providing transparent distributed IPC. Several libraries to support these services were also designed and implemented. The overall Fluke operating system includes major additional components which we developed and separately packaged and distributed: the OSKit, the Flick IDL compiler, and the Mini Object Model and runtime. These are described in later sections.

Our Fluke/Flask kernel implementation was designed to be relatively simple in order to facilitate experimentation, to be as portable as possible, to provide maximum error checking and reporting, to conform to strict code modularity and layering disciplines in anticipation of high-security implementations, and to reuse as much existing code as possible. For all of these reasons this implementation of the Fluke kernel architecture is not optimized, and conclusions regarding the performance limits of the Fluke architecture should not be drawn from it. Application of the well-documented design and implementation techniques demonstrated by kernels such as L4 would markedly improve Fluke's performance.

Fluke Motivation: the Persistent Relevance of the Local Operating System: The growth and popularity of loosely-coupled distributed systems such as the World Wide Web and the touting of Java-based systems as the solution to the issues of software maintenance, flexibility, and security are changing the research emphasis away from traditional single node operating system issues. Apparently, the view is that traditional OS issues are either solved problems or minor problems. By contrast, we believed that building such vast distributed systems upon the fragile infrastructure provided by today's operating systems is analogous to building castles on sand. In a paper [1] at a high-visibility international workshop we outlined the supporting arguments for these views and described the Fluke OS design that supports secure encapsulation of the foreign processes that will be increasingly prevalent in tomorrow's distributed systems.

[1] It is important to note that Fluke's "virtual machines" have nothing inherently to do with Java. They are not byte code interpreters, but are more akin to the "virtual machine monitors" of classical virtual machine architectures of the 1970's.

2.1.1 Fluke/Flask OS Components

The Fluke/Flask operating system is highly modular and highly structured, as is appropriate for a high-security or safety-critical system. Besides the kernel, the Fluke OS includes a large number of servers, libraries, and IDL-specified interfaces between processes. Most of the major servers providing POSIX-like function (e.g., network, filesystems) are themselves highly modular, containing unchanged OSKit components.

- "Common protocols": IPC-based interfaces between the virtual machine layers (between the "nesters"), specified in CORBA IDL. For example, the common protocols include interfaces for memory allocation, networking, file access, and device access.
- Fluke libraries
  - *libsac:* Provides a POSIX-like interface to client applications, including optimized access to Fluke IPC-based services.
  - *libnest:* Common infrastructure for Fluke servers (nesters), such as functions to create and initialize new virtual environments (new levels of nesting).
  - *libmm:* Memory management. Used by both the kernel and the virtual memory manager.
  - *libwrap:* Used to interpose on the common protocols, to provide sandboxing or logging.
  - *libmemdebug:* Memory allocation debugging aid.
  - *libyfs:* Simple filesystem, useful for development.
- Fluke servers
  - *kernel server:* The "root" nester; this is a normal Fluke nester, except that it has the same memory mappings as the kernel. The source of resources (e.g. memory, device access).
  - *virtual memory manager:* Obtains physical memory from an ancestor (normally the kernel server) and provides virtual memory to all descendent processes. Normally the first nester started outside the kernel, except in Flask, when the security policy server is first.
  - *checkpointer:* Provides checkpointing of descendent processes.
  - *process manager:* Provides process management for its immediate child processes; e.g., fork/kill/signal.
  - *branch nester:* Forks a branch of the virtual machine hierarchy into two branches, without changing anything else (unlike the process manager, above).
  - *logging nester:* When interposed into a branch of the virtual machine hierarchy, logs all communication between levels.
  - *network server:* Provides TCP/IP and sockets.
  - *network (distributed) IPC server:* Extends Fluke IPC across a network.
  - *real file server:* Provides full BSD filesystem functionality.
  - *memory file server:* RAM disk filesystem.
  - *trivial file server:* A simple file server.
- Flask-specific libraries and servers
  - *security policy server:* Based on a policy database, provides decisions on access, object labeling, and polyinstantiation. Normally the first Flask server to be started.
  - *libavc:* Access Vector Cache: minimizes and optimizes communication to the security policy server, while providing coherence.

2.1.2 Recursive Virtual Machine Architecture

Fluke brings together two lines of research and two heretofore disparate architectures: *microkernels* and *recursive virtual machines* [2].

Fluke represents a novel approach to providing modular and extensible operating system functionality and encapsulated environments based on a synthesis of microkernel and virtual machine concepts. We developed a software-based *virtualizable architecture* called Fluke that allows recursive virtual machines (virtual machines running on other virtual machines) to be implemented efficiently by a microkernel running on generic hardware. A complete virtual machine interface is provided at each level; efficiency derives from needing to implement only *new* functionality at each level. This infrastructure allows common OS functionality, such as process management, demand paging, fault tolerance, and debugging support, to be provided by cleanly modularized, independent, stackable virtual machine monitors, implemented as user processes. It can also provide uncommon or unique OS features, including the above features specialized for particular applications' needs, virtual machines transparently distributed cross-node, or security monitors that allow arbitrary untrusted binaries to be executed safely. Our prototype implementation of this model indicates that it is practical to modularize operating systems this way. Some types of virtual machine layers impose almost no overhead at all, while other impose some overhead (typically 0–35%), but only on certain classes of applications.

2.1.3 Flexible Checkpointing

One of the novel features provided by this fully "nestable" OS architecture is the ability for an otherwise ordinary user process to checkpoint (or migrate) the state of *unchanged, arbitrary* child processes. Indeed, any set of programs may be checkpointed. Transparent checkpointing of unchanged programs is itself highly unusual. As far as we know only three systems have ever provided that feature, and they are either old or experimental. L3 , KeyKOS , and KeyKOS's contemporary re-implementation, EROS . However, in those OS's the entire computer system is persistent. Fluke is novel in the *flexible scope* one can choose for its persistence feature: anywhere on the spectrum from one (or zero) processes, to the whole machine.

Such checkpointing requires complete access to the entire state of a process, and Fluke is novel in that it can provide such access, in a secure manner, to any ordinary user process [3]. Checkpointing, process migration, and similar services need to have access not only to the memory of the constituent processes, but also to the complete state of all kernel provided objects (e.g., threads and ports) involved. Traditionally, a major stumbling block in these operations is acquiring and re-creating the state in the operating system.

We implemented a transparent user-mode checkpointer as an application on the Fluke microkernel. The microkernel consistently and cleanly supports the importing and exporting of fundamental kernel state safely to and from user applications. Implementing a transparent checkpointing facility with this sort of kernel support simplifies the implementation, and expands its flexibility and power.

2.1.4 Atomicity of Kernel API and Kernel Operations

In order to provide completely exportable kernel state a novel kernel property is required. This property relates to the *"atomic" nature of the kernel's application programming interface*—its API [4]. In the Fluke kernel we defined and implemented a kernel API that makes every exported operation fully interruptible and restartable, thereby appearing atomic to the user. To achieve interruptibility, all possible kernel states in which a thread may become blocked for a "long" time are represented as kernel system calls, without requiring the kernel to retain any unexposable internal state.

Since all kernel operations appear atomic, services such as transparent checkpointing and process migration that need access to the complete and consistent state of a process can be implemented by ordinary user-mode processes. Atomic operations also enable applications to provide reliability in a more straightforward manner.

This API also allowed us to explore novel kernel implementation techniques and to evaluate existing techniques. The Fluke kernel's single source implements either the "process" or the "interrupt" execution model on both uniprocessors and multiprocessors, depending on a configuration option affecting a small amount of code.

We reported measurements comparing fully, partially and non-preemptible configurations of both process and interrupt model implementations. We found that the interrupt model has a modest performance advantage in some benchmarks, maximum preemption latency varies nearly three orders of magnitude, average preemption latency varies by a factor of six, and memory use favors the interrupt model as expected, but not by a large amount. We found that the overhead for restarting the most costly kernel operation ranges from 2–8% of the cost of the operation.

This atomicity of the kernel interface provides an important building block for the high-security variant of Fluke, called "Flask." *Revocation* of permissions is a key challenge for secure systems, and as we'll see later, Fluke's "atomic" API makes revocation significantly more tractable.

2.1.5 CPU Inheritance Scheduling

Fluke's recursive architecture is a good match for hierarchical resource management schemes, which themselves are a good match to the natural administrative hierarchy found in the real world. In designing resource management frameworks for Fluke, we developed a novel hierarchical processor scheduling framework called *CPU inheritance scheduling* [5]. This is a framework for scheduling policies, not a policy itself, and is applicable far beyond the Fluke kernel.

Traditional processor scheduling mechanisms in operating systems are fairly rigid, often supporting only one fixed scheduling policy, or, at most, a few "scheduling classes" whose implementations are closely tied together in the OS kernel. We invented *CPU inheritance scheduling*, a novel processor scheduling framework in which arbitrary threads can act as schedulers for other threads. Widely different scheduling policies can be implemented under the framework, and many different policies can coexist in a single system, providing much greater scheduling flexibility. Modular, hierarchical control can be provided over the processor utilization of arbitrary administrative domains, such as processes, jobs, users, and groups, and the CPU resources consumed can be accounted for and attributed accurately. Applications, as well as the OS, can implement customized local scheduling policies; the framework ensures that all the different policies work together logically and predictably. As a side effect, the framework also cleanly addresses priority inversion by providing a generalized form of priority inheritance that automatically works within and among diverse scheduling policies. CPU inheritance scheduling extends naturally to multiprocessors, and supports processor management techniques such as processor affinity and scheduler activations. In our prototype implemented we showed that this flexibility can be provided with acceptable overhead in typical environments, depending on factors such as context switch speed and frequency.

2.1.6 The Fluke Device Driver Framework

Providing efficient device driver support in the Fluke operating system presents novel challenges, which stem from two conflicting factors: (i) for maintainance and economic reasons, a requirement to reuse unmodified legacy device drivers, as encapsulated in the OSKit, discussed later in Section 2.2, and (ii) the mismatch between the Fluke kernel's internal execution environment and the execution environment expected by these legacy device drivers. In this work, which was documented in a thesis [6], we developed a solution to this conflict: a framework whose design is based on running device drivers as user-mode servers, which resolves the fundamental execution environment mismatch.

This approach introduces new problems and issues, of which the most important are synchronization, interrupt delivery, physical memory allocation, access to shared resources, and performance. We successfully addressed the functional issues, as demonstrated by the fact that the majority of device drivers execute successfully without change and are routinely used by Fluke developers. Based on our experience with the minority of drivers that did require changes, and our experience developing the framework, we proposed guidelines for improving device drivers' portability across different execution environments.

Running device drivers in user mode raises serious performance issues but on the whole they were successfully mitigated. We compared the driver performance in Fluke with that in the original legacy systems, in terms of latency, bandwidth, and processor utilization. We found that reasonable performance (between 88–93% of the best-performing Unix systems in a realistic workload) and acceptable processor overhead (between 0–100%) are achievable. The limiting factor is the IPC performance of the underlying Fluke OS layers.

2.1.7 Extending Fluke IPC for Transparent Remote Communication

Fluke not only provides a base for the POSIX-like environment that we implemented, it provides a base for experimental OS features. One such feature is the checkpointer described above. In a major effort we explored the issues of providing transparently distributed interprocess communication [7] for Fluke. We had several motivations for this work.

Distributed systems such as client-server applications and cluster-based parallel computation are an important part of modern computing. Distributed computing allows the balancing of processing load, increases program modularity, isolates functionality, and can provide an element of fault tolerance. In these environments, systems must be able to synchronize and share data through some mechanism for remote interprocess communication (IPC). Although distributed systems have many advantages, they also pose several challenges. One important challenge is transparency. It is desirable that applications can be written to a communication interface that hides the details of distribution.

One way to achieve transparency is through the extension of local communication mechanisms over a network for remote communication. The ability to transparently extend local communication depends on the semantics of the local IPC mechanisms. Unfortunately, those semantics are often driven by other architectural goals of the system and may not necessarily be best suited for remote communication.

In this work we developed a remote IPC implementation for the Fluke operating system and documented our findings in a thesis [7]. We performed an extensive analysis of the Fluke architecture, IPC system, and IPC semantics, with regard to the extension of local IPC for transparent remote communication. We showed that the overall complexity of both the kernel IPC subsystem and the network IPC implementation is considerably less than similar operating systems' IPC mechanisms, and that the Fluke IPC architecture is generally well-suited for transparent remote IPC. However, our work also showed that the lack of kernel-provided reference counting caused more problems than it solved, and that the generality of an important Fluke kernel object, the "reference," makes it impossible for the network IPC system to provide completely transparent remote IPC without extensive additional services.

2.1.8 The Flask Security Architecture: System Support for Diverse Security Policies

For a secure operating system to be practical and cost effective to build and maintain, it must be useful to a wide variety of market segments and application domains. Therefore a single OS must provide sufficient underlying mechanisms to support the wide variety of real-world security policies, which need to be separable from the OS.

Such flexibility requires controlling the propagation of access rights, enforcing fine-grained access rights and supporting the revocation of previously granted access rights. Previous systems are lacking in at least one of these areas. In a technical report and later a published paper [8], jointly written with our NSA and SCC collaborators, we presented and evaluated the Flask operating system security architecture that solves these problems. Control over propagation is provided by ensuring that the security policy is consulted for every security decision. This control is achieved without significant performance degradation through the use of a security decision caching mechanism that ensures a consistent view of policy decisions. Both fine-grained access rights and revocation support are provided by mechanisms that are directly integrated into the service-providing components of the system. The architecture was described through its prototype implementation in the Flask microkernel-based operating system, and the policy flexibility of the prototype was evaluated. We presented initial evidence that the architecture's impact on both performance and code complexity is modest. Moreover, our architecture is applicable to many other types of operating systems and environments.

2.1.9 Implementing Mandatory Network Security in a Policy-flexible System

The use of networks is growing continuously, constantly increasing the vulnerability of the computer systems that use them. Current solutions for network security, such as firewalls, cannot support sophisticated trust relationships with external entities and lack a comprehensive approach to security. Research in security has shown the usefulness of mandatory security mechanisms for supporting sophisticated trust relationships and secure endpoints in addition to secure communication channels. Other efforts at incorporating mandatory security mechanisms into the network stack have a limited notion of access control policies. This work, documented in a thesis [9], dealt with the design and implementation of a more comprehensive and flexible network security architecture that enforces a mandatory access control policy on network-related operations and a mandatory cryptographic policy on network traffic.

The implementation involved modifying the FreeBSD TCP/IP stack within the Flask secure operating system. Access control decisions are made in a policy-flexible manner by consulting a security server and security attributes are interpreted only by the security server. The access control design maps access control requirements to checks made at different layers in the network stack according to the functionality provided by the layer. This approach has several advantages, which include less time spent on illegal packets and the ability to specify policy in a fine-grain manner. Network cryptographic protection was provided using the IPsec protocol for cryptographic support and the ISAKMP protocol for key management.

2.1.10 Utah Assurance Work: Applying Formal Methods to Fluke IPC

The formal methods community has long known about the need to formally analyze concurrent software, but the OS community has been slow to adopt such methods. The foremost reasons for this are the cultural and knowledge gaps between formalists and OS hackers, fostered by three beliefs: inaccessibility of the tools, the disabling gap between the validated model and actual implementation, and the intractable size of operating systems. In a paper at a prestigious and high visibility OS workshop [10], we showed these beliefs to be untrue for appropriately structured operating systems.

We applied formal methods to verify properties of the implementation of the Fluke microkernel's IPC subsystem, a major component of the kernel—over 20%. In particular, we verified, in many scenarios, the lack of deadlock and certain liveness properties, with results that apply to both SMP and uniprocessor environments. The SPIN model checker developed by Holzmann provided an exhaustive concurrency analysis of the IPC subsystem, unattainable through traditional OS testing methods. In a novel result, we found that the stylized coding used in Fluke made it relatively straightforward to generate the model directly from the implementation, as opposed to from the design, as is commonly done— giving stronger assurance guarantees. SPIN proved to be easily accessible to programmers inexperienced with formal methods. We presented our results as a starting point for a more comprehensive inclusion of formal methods in practical OS development. In addition, our software artifact—Fluke IPC implemented in a model checking language—was publically distributed.

2.1.11 External Assurance Work: Formal Analysis of Fluke/Flask

The Secure Computing Corporation did extensive analysis of the Flask security architecture and its design and implementation in our Fluke microkernel-based system. At times both we and our NSA collaborators worked closely with SCC, sometimes leading to changes in the Fluke design and implementation, and always leading to a clearer understanding by all parties. SCC's efforts culminated in three SCC-authored reports, which provide a solid analysis of many aspects of Fluke/Flask, and a base for future efforts. (We list these reports in Section 4, but of course do not intend to imply that we produced them.)

Assurance in the Fluke Microkernel: Formal Security Policy Model [11] presents the formal model of the security control requirements for the Fluke microkernel, including a formalization in the PVS specification language. More generally, it provides a security model for the Fluke and Flask systems, discussing the overall security architecture, security requirements on the different elements of the architecture, and the range of policies supportable within the architecture.

Assurance in the Fluke Microkernel: Formal Top-Level Specification [12] contains formal specifications of the access vector cache and prototype security server for the Flask system, including a formalization in the PVS specification language. It also presents a model for reasoning about the ability of a computing system to support dynamic security policies, and in particular changes to the security policy which restrict operations that were previously allowed.

Assurance in the Fluke Microkernel: Final Report [13] contains the final report for SCC's "Fluke Assurance" program. It includes a summary of the objectives, accomplishments, and outputs from the program, and overviews of the major technical areas: the security architecture, the Fluke implementation of the security architecture, the dynamic security policy study, innovations in the security policy server, results of the composition study, and suggestions for future work. Updates to the previously published formal security policy model and top level specification are also included.

2.1.12 Kernel Performance

Major performance work in Fluke was pursued during the entire project. Some of the work had interesting and sometimes novel research aspects. For example, a number of Fluke kernel object optimizations have been implemented which improve "critical path" operations. In a novel design, these optimizations allow certain kernel operations to be performed in user mode, i.e., without trapping to the kernel, and without loss of integrity or atomicity. Optimized operations include locking and unlocking of uncontested mutexes as well as certain state gathering operations on Fluke references (capabilities). In early 1998 we increased our efforts to improve performance in the base system. In addition to "gprof" (profiling) support for the kernel, we added gprof support for user mode applications. Together, this support provided a vital tool for understanding and analyzing the behavior of Fluke and its applications. Profiling identified a number of problem areas which have been corrected including inefficient memory and object allocation in Flick stubs and the need for a specialized memory allocator for Fluke kernel objects.

As another example, we later designed and prototyped two IPC extensions (atomic calls to perform "receive-message-and-acquire-mutex" and "receive-message-with-sender-identification") that enable a more efficient implementation of the higher-level MOM (see Section 2.4) client/server infrastructure. Though simple in concept, the extensions required careful analysis to ensure that they meet the atomicity requirements of the Fluke system as well as the security requirements of the Flask architecture. Preliminary results showed improvements of up to 30% on MOM operations that pass object references.

2.2 The OSKit: Reusable Components for OS Implementation

We gradually isolated, formalized, and generalized the infrastructure that we had originally developed only to support Fluke. The OSKit—reusable components for low-level systems—went from a rough prototype to a robust set of over 30 components with carefully designed interfaces. The OSKit has proven to be highly useful to a wide variety of external research, development, and even commercial projects. It currently supports two architectures, the *Intel x86*, important because of its wide use, and the *StrongARM*, important because of its growing popularity in embedded systems, due to its combination of low power and high performance.

Implementing new operating systems is tedious, costly, and often impractical except for large projects. The Flux OSKit addresses this problem in a novel way by providing clean, well-documented OS components designed to be reused in a wide variety of *other* environments, rather than defining a new OS structure. The OSKit uses unconventional techniques to maximize its usefulness, such as intentionally exposing implementation details and platform-specific facilities. Further, the OSKit demonstrates a technique that allows unmodified code from existing mature operating systems to be incorporated quickly and updated regularly, by wrapping it with a small amount of carefully designed "glue" code to isolate its dependencies and export well-defined interfaces. The OSKit uses this technique to incorporate over 230,000 lines of stable code including device drivers, file systems, and network protocols. Our experience demonstrates that this approach to component software structure and reuse has a surprisingly large impact in the OS implementation domain. In two papers [14, 15] we described the OSKit along with four real-world examples that showed how the OSKit is catalyzing research and development in operating systems and programming languages.

We wrote a 500 page detailed manual [16] for the OSKit, targeted primarily at users but also containing information on OSKit internals. It includes much expository text (rationale, extended descriptions), which is appropriate as overview, background, and introductory information. The more concise but extensive "man pages" constitute an API reference manual.

We made several public releases, which were downloaded at a rate of about 1000/month, including downloads by a number of Web mirror sites, so the actual number of distributed copies will be higher. We have over 470 users on the OSKit mailing list.

Along with the OSKit, we released a set of changes that can be applied to the Kaffe OpenVM (a popular, commercial open source Java virtual machine), enabling Kaffe to link with the OSKit and run on "bare hardware." Our changes were integrated into the Kaffe code base.

As a step toward making Flask technology more accessible, we started to integrate Flask-enhanced OSKit components into Linux. In particular, we glued a simple OSKit filesystem component into Linux via a dynamically loadable kernel module (LKM).

2.2.1 OSKit Components

The facilities provided by the OSKit are organized into two main categories, *interfaces* and *libraries*.

Interfaces

The OSKit's interfaces are a set of clean, object-oriented interfaces specified in the framework of Microsoft's Component Object Model (COM). For example, the OSKit provides a "block I/O" interface for communication between file systems and disk device drivers, a "network I/O" interface for communication between network device drivers and protocol stacks, and a file system interface similar to the "VFS" interface in BSD. These interfaces are used and shared by the various OSKit components in order to provide consistency.

Libraries

Following is a summary of the 34 libraries currently provided by the OSKit.

- **Function libraries**
  - *c*: A simple, minimal C library designed to work in a restricted OS environment.
  - *kern*: Low-level kernel support code of all kinds.
  - *smp*: Kernel code providing symmetric multiprocessor support.
  - *com*: Utility functions and wrappers for handling COM interfaces.
  - *osenv*: Default implementations of the "glue" functions required by "large" encapsulated components (e.g., device drivers, networking, filesystems) imported from other operating systems.
- **POSIX emulation and libraries**
  - *posix*: Support for what a POSIX conformant system would typically implement as system calls.
  - *freebsd_c*: Complete POSIX-like C library derived from FreeBSD, providing both single- and multithreaded configurations. Together with the above `posix` library, provides a very large subset of the POSIX API.
  - *freebsd_m*: Complete standard math library, taken from FreeBSD's `libm`.
  - *fsnamespace*: Provides file naming translation ("namei"-style), as well as high level mount and unmount capabilities.
  - *rtld*: *Runtime Linker/Loader* provides dynamic linking, loading, and shared library facilities.
- **Memory management**
  - *lmm*: A flexible memory management library that can manage either physical or virtual memory. This library supports many special features needed by OS-level code, such as multiple memory types, allocation priorities, and arbitrary alignment and placement constraints.
  - *amm*: The *Address Map Manager* library manages collections of resources where each element has a name (address) and some set of attributes. Examples of resources include swap space and process virtual address space.

  - *svm*: The *Simple Virtual Memory* library uses the AMM library to define a simple virtual-memory interface for a single address space that can provide memory protection and paging to a block device.

- **Threads, synchronization, and scheduling**

  - *threads*: This library provides support for multithreaded kernels, including POSIX threads, synchronization, scheduling, and stack guards. Scheduling policies are the standard POSIX Round-Robin and and FIFO, as well as the CPU inheritance scheduling framework with provides rate-monotonic, stride (WFQ), and lottery scheduling policies.

- **Development aids**

  - *memdebug*: Provides debugging versions of `malloc` et al.
  - *gprof*: Run-time profiling of kernels.

- **Simple disk/file reading and loading**

  - *diskpart*: Recognizes various common disk partitioning schemes and produces a complete "map" of the organization of any disk.
  - *fsread*: A simple read-only file system interpretation library supporting various common types of file systems; typically used for bootstrapping.
  - *exec*: A generic executable interpreter and loader that supports popular executable formats.

- **Filesystem implementations**

  - *linux_fs*: Encapsulated Linux filesystem code. Includes support for `ext2`, the primary Linux filesystem, as well as numerous other PC filesystems supported under Linux.
  - *netbsd_fs*: Encapsulated NetBSD filesystem code.
  - *memfs*: A trivial memory-based filesystem (i.e., a RAM disk), exporting the standard OSKit filesystem interfaces.

- **Networking implementations**

  - *freebsd_net*: Encapsulated FreeBSD networking code, including sockets.
  - *bootp*: BOOTP protocol (RFC 1048/1533) support.
  - *hpfq*: Hierarchical proportional-share control of outgoing network link bandwidth.
  - *dpf*: A packet dispatching mechanism using packet filter technology.

- **Device driver implementations**

  - *linux_dev*: Encapsulated Linux device drivers. Currently includes over 50 block (SCSI, IDE) and network drivers.
  - *freebsd_dev*: Encapsulated FreeBSD device driver set. Currently includes eight TTY (virtual console and serial line, including mouse) drivers.

- **Video and window manager implementations**

  - *wimpi*: Simple hierarchical windowing system.

  - **video*: Basic video support, with two implementations: one encapsulating all of SVGALIB with broad device support, and one based on XFree86, but with only the S3 driver currently supported. We also provide support for X11 clients.

- **Miscellaneous**

  The following four libraries are are not as well documented but are still very useful, especially the first two.

  - *unix*: Support to debug and run many OSKit components on FreeBSD and Linux in user-mode. Very useful for debugging.
  - *startup*: Contains functions to start up and initialize various OSKit components.
  - *fudp*: Provides a "Fake UDP" implementation: a restricted send-only no-fragmenting version of UDP.
  - *unsupp*: Contains various unsupported hacks and utilities.

2.3 The Flick Interface Definition Language Compiler

In our work to optimize Mach and, later, Fluke IPC, we found it essential to improve the user-space costs involved, as well as to achieve more flexibility. A major source of costs proved to be the generation of stub code from the interface specification in IDL.

2.3.1 General, Flexible, and Optimizing IDL Compilation

An interface definition language (IDL) is a nontraditional language for describing interfaces between software components. IDL compilers generate "stubs" that provide separate communicating processes with the abstraction of local object invocation or procedure call. High-quality stub generation is essential for applications to benefit from component-based designs, whether the components reside on a single computer or on multiple networked hosts. Typical IDL compilers, however, do little code optimization, incorrectly assuming that interprocess communication is always the primary bottleneck. (As networks and operating systems become faster, the bottleneck for structured communication (RPC, object invocation) will move to the presentation layer.) More generally, typical IDL compilers are "rigid" and limited to supporting only a single IDL, a fixed mapping onto a target language, and a narrow range of data encodings and transport mechanisms.

Flick, our IDL compiler, is based on the insight that IDLs are true languages amenable to modern compilation techniques. Flick exploits concepts from traditional programming language compilers to bring both flexibility and optimization to the domain of IDL compilation. Through the use of carefully chosen intermediate representations, Flick supports multiple IDLs, diverse data encodings, multiple transport mechanisms, and applies numerous optimizations to all of the code it generates. Our experiments showed that Flick-generated stubs marshal data between 2 and 17 times faster than stubs produced by traditional IDL compilers, and on today's generic operating systems, increased end-to-end throughput by factors between 1.2 and 3.7.

Our primary paper on Flick [17] was published in the most prestigious and competitive venue for compiler research, the ACM SIGPLAN Conference on Programming Language Design and Implementation. However, Flick was not merely a research advance; it became a robust, well supported, well documented tool. During the course of the project we made several formal, public releases of Flick. The releases included a user's manual [18] that describes how to build Flick, how to run the various compiler passes, and how to use the generated stubs. The manual presents a simple client/server phonebook, implemented in three different ways: as an ONC RPC (Sun RPC) application, as a CORBA C application, and as a CORBA C++ application. The examples are presented in detail and illustrate how users can make use of Flick-generated stubs in their own programs.

We also produced a Flick internals manual [19], which describes its implementation details. People interested in studying, modifying, or extending Flick should consult this manual as a starting point. Those who simply want to *use* Flick do not need to read this document: use of the Flick compiler tools is described in the separate user's manual.

Some of these releases contained dramatic improvements in function, completeness, and performance. New releases supported new transports and encodings, including Fluke IPC, CORBA IIOP, Sun ONC, and Trapeze (a gigabit Myrinet-based communication protocol from Duke University). Flick supported new IDLs and message formats, including almost complete support for Mach MIG, and support for security annotations on interfaces. New releases supported new platforms, including Windows 95/NT. Support for CORBA improved significantly: we supported CORBA on Mach and provided a greatly improved CORBA runtime during the course of the project.

Because Flick provides a unique combination of flexibility and optimization, Flick was chosen to be part of the Quorum Distributed Objects Integration (DOI) project. The initial integration efforts went well: for instance, in 1998 Joe Loyall of BBN successfully modified BBN's QuO "delegate generator" to work with Flick's CORBA IDL front end. We worked with the Quorum distributed object integration team, in particular BBN and Washington University at St. Louis, to evolve Flick to support their needs, in particular adding complete C++ support and support for WUSTL's TAO system. Our implementation of the CORBA C++ mapping resulted in many additions to Flick, because the CORBA C++ mapping is significantly more complex than any of the other (C language) mappings we had previously implemented.

2.3.2 Flexible IDL Compilation for Complex Communication Patterns

We continued to develop Flick for other uses as well. In particular, we developed a strategy for using Flick to generate stubs for Khazana, our global memory service requiring asynchronous messages between nodes, described in Section 2.6. Flick now produces specially "decomposed" CORBA stubs, achieving flexible IDL compilation for complex communication patterns.

Distributed applications are complex by nature, so it is essential that there be effective software development tools to aid in the construction of these programs. Commonplace "middleware" tools, however, often impose a tradeoff between programmer productivity and application performance. For instance, many CORBA IDL compilers generate code that is too slow for high-performance systems. More importantly, these compilers provide inadequate support for sophisticated patterns of communication. We believe that these problems can be overcome, thus making IDL compilers and similar middleware tools useful for a broader range of systems.

To this end we extended Flick to produce specialized high-performance code for complex distributed applications. Flick can produce specially "decomposed" stubs that encapsulate different aspects of communication in separate functions, thus providing application programmers with fine-grain control over all messages. The design of our decomposed stubs was inspired by the requirements of a particular distributed application called Khazana. In two papers [20, 21] we described our experience in refitting Khazana with Flick-generated stubs. We believe that the special IDL compilation techniques developed for Khazana will be useful in other applications with similar communication requirements.

2.4 MOM: The Mini Object Model

We designed, specified [22], and implemented MOM, the "Mini-Object Model": a layer above Fluke IPC that presents a more client/server-oriented interface to applications. MOM specifies a simple object invocation model and language-specific bindings to the facilities of that model. The purpose of this specification is to facilitate the portability of modules written for message-passing interfaces to a variety of operating environments. Many components implemented in client-server systems using microkernel IPC facilities, in complex client-server infrastructure systems such as COM and CORBA, and in subsystems of monolithic operating system kernels, in fact assume only a conceptually simple model of object invocation and rely on relatively few other incidental operating system facilities. This specification hopes to provide an interface for object management and invocation that can be implemented in a variety of operating environments and impose little or no run-time overhead on modules using this interface instead of the specific interface native to a particular IPC system, OS kernel, or complex object system.

During our analysis and performance improvements to the Fluke Operating System a number of experiments identified our MOM infrastructure as a problem area. To address this problem, we made significant algorithmic improvements to the reference counting component of MOM, in addition to making other MOM performance enhancements. We created a standalone benchmark to assist in tracking the effect of future changes to MOM.

2.5 Java Operating Systems

In the latter parts of the contract periods we explored the implications of applying our OS structuring ideas to operating systems that use the type-safe properties of the Java programming language to provide memory safety, instead of using the hardware MMU to do so. We explored many aspects of the issues in two experimental prototypes, with an emphasis on *resource control*. In our largest effort in this domain, we designed and implemented "Alta," an implementation of the Fluke nested process model in a Java virtual machine.

2.5.1 Making the Case: OS Structure for Mobile Code

In a highly competitive and visible OS workshop, we presented and published early results from our Alta prototype [23], focusing on making the case for such a comprehensive approach. 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 argued 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 argued that real solutions will come only if an OS model is uniformly applied to these environments. We presented *Alta*, our prototype Java-based system patterned on Fluke, a highly structured, hardware-based OS, and reported on its features appropriate to controlling mobile code.

2.5.2 The Alta Operating System

Many modern systems, including web servers, database engines, and operating system kernels, are using language-based protection mechanisms to provide the safety and integrity traditionally supplied by hardware. As these language-based systems become used in more demanding situations, they are faced with the same problems that traditional operating systems have solved—namely shared resource management, process separation, and per-process resource accounting. While many incremental changes to language-based, extensible systems have been proposed, we demonstrated in an implementation and a thesis [24] that comprehensive solutions used in traditional operating systems are applicable and appropriate.

The thesis gives a detailed description of Alta, an implementation of the Fluke operating system's nested process model in a Java virtual machine. The nested process model is a hierarchical operating system process model designed to provide a consistent approach to user-level, per-process resource accounting and control. This model accounts for CPU usage, memory, and other resources through a combination of system primitives and a flexible, capability-based mechanism.

Alta supports nested processes and interprocess communication. Java applications running on Alta can create child processes and regulate the resources—the environment—of those processes. Alta demonstrates that the Java environment is sufficient for hosting traditional operating system abstractions. Alta extends the nested process model to encompass Java-specific resources such as class files, modifies the model to leverage Java's type safety, and extends the Java type system to support safe fine-grained sharing between different applications. Existing Java applications work without modification on Alta.

Alta was compared in terms of structure, implementation and performance to Fluke and traditional hardware-based operating systems. A small set of test applications demonstrated flexible, application-level control over memory usage and file access.

2.5.3 Java Operating Systems: Comparative Design and Implementation

Language-based extensible systems such as Java use type safety to provide memory safety in a single address space. Memory safety alone, however, is not sufficient to protect different applications from each other. Such systems must support a *process model* that enables the control and management of computational resources. In particular, language-based extensible systems must support resource control mechanisms analogous to those in standard operating systems. They must support the separation of processes and limit their use of resources, but still support safe and efficient interprocess communication.

In our prototypes and in a paper [25] we demonstrated how this challenge can be addressed in Java operating systems. First, we described the technical issues that arise when implementing a process model in Java. In particular, we laid out the design choices for managing resources. Second, we described the solutions that we explored in two complementary projects, *Alta* and *GVM*. GVM is similar to a traditional monolithic kernel, whereas Alta closely models the Fluke operating system. Features of our prototypes include flexible control of processor time using CPU inheritance scheduling, per-process memory controls, fair allocation of network bandwidth, and execution directly on hardware using the OSKit. Finally, we compared our prototypes with other language-based operating systems, in particular Cornell's "J-Kernel," and explored the tradeoffs between the various designs.

2.5.4 Drawing the Red Line in Java

Software-based protection has become a viable alternative to hardware-based protection in systems based on languages such as Java, but the absence of hardware mechanisms for protection has been coupled with an absence of a user/kernel boundary. In a well-received paper [26] we showed why such a "red line" must be present in order for a Java virtual machine to be as effective and as reliable as an operating system. We discussed how the red line can be implemented using software mechanisms, and explained the ones we use in the Java system that we are building.

2.5.5 Memory Management: The Need for Predictable Garbage Collection

Modern programming languages such as Java are increasingly being used to write systems programs. By "systems programs," we mean programs that provide critical services (compilers), are long-running (Web servers), or have time-critical aspects (databases or query engines). One of the requirements of such programs is predictable behavior. Unfortunately, predictability is often compromised by the presence of garbage collection. Various researchers have examined the feasibility of replacing garbage collection with forms of stack allocation that are more predictable than GC, but the applicability of such research to systems programs had not previously been studied or measured. A particularly promising approach allocates objects in the nth stack frame (instead of just the topmost frame): we call this *deep stack allocation*. In a detailed paper [27] we presented dynamic profiling results for several Java programs to show that deep stack allocation should benefit systems programs, and we described the approach that we are developing to perform deep stack allocation in Java.

2.6 The Khazana/KOLA Infrastructure for Building Distributed Services

In 1998 we transitioned our DSM effort away from LAN-based parallel numeric applications to a broader and more prevalent class of applications: local and wide-area *distributed* applications. The base layer of our new architecture is called *Khazana*; above that are two layers that comprise *KOLA*, the Khazana Object Layer Architecture.

Description of Khazana: Essentially all distributed systems, applications, and services at some level boil down to the problem of managing distributed shared state. Unfortunately, while the problem of managing distributed shared state is shared by many applications, there is no common means of managing the data—every application devises its own solution. We developed *Khazana*, a distributed service exporting the abstraction of a distributed persistent globally shared store that applications can use to store their shared state, and described it in a paper [28]. Khazana is responsible for performing many of the common operations needed by distributed applications, including replication, consistency management, fault recovery, access control, and location management. Using Khazana as a form of middleware, distributed applications can be quickly developed from corresponding uniprocessor applications through the insertion of Khazana data access and synchronization operations.

Experience with Khazana-based Applications: We evaluated the effectiveness of basing distributed systems on a persistent globally shared address space abstraction, as implemented by *Khazana*, and documented our results in a paper [29]. Khazana provides shared state management services to distributed application developers, including consistent caching, automated replication and migration of data, location management, access control, and (limited) fault tolerance. We reported on our experience porting three applications to Khazana: a distributed file system (KFS) based on Linux local filesystem code, a distributed name service (KNS), and a cooperative drawing program based on `xfig` (KFIG). The basic Khazana abstraction of persistent shared memory made it easy to create fairly efficient distributed services with flat data abstractions (KFS and KNS), but our port of `xfig` made it clear that a higher level abstraction is preferable for applications with more structured state. As a result, we extended Khazana to support a limited set of object-like functions (reference swizzling, event upcalls, and update propagation). We present herein the current Khazana design and implementation, and discuss the lessons learned from our initial evaluation of it.

KOLA: the Khazana Object Layer Architecture: Based on our experience with Khazana, we designed a form of three-tier middleware for distributed applications and services based on Khazana. The lowest layer, Khazana proper, provides basic distributed storage, consistency, availability, locking, and security mechanisms for unstructured page-oriented data. The middle layer provides language independent support for "objects," including object caching, persistent heap management, basic reference swizzling, and an extension of the basic "lock intents" mechanism that supports independent locking of multiple objects within a page. The top layer is a set of language specific libraries that support method invocation on remotely instantiated objects, run time type inference, language-specific reference swizzling, and (optional) garbage collection. The middle and top layer are referred to collectively as KOLA (Khazana Object Layer Architecture).

KOLA allows us to support reference-filled, pointer-rich applications while allowing the base "region-based" Khazana to co-exist independently. Our initial version of KOLA demonstrated the feasibility of extracting the necessary programming language-level information to support persistent distributed objects on top of Khazana.

A Language-Specific Layer: Distributed Persistent C++ Objects: We designed and implemented a C++ object layer for Khazana. The C++ layer described herein lets programmers use familiar C++ idioms to allocate, manipulate, and deallocate persistent shared data structures. It handles the tedious details involved in accessing this shared data, replicating it, maintaining consistency, converting data representations between persistent and in-memory representations, associating type information including methods with objects, etc. To support the C++ object layer on top of Khazana's flat storage abstraction, we developed a language-specific preprocessor that generates support code to manage the user-specified persistent C++ structures. In a paper [30] we described the design of the C++ object layer and the compiler and runtime mechanisms needed to support it.

3 Technology Transfer

Most technology transfer was mentioned under each software component in the "Results" section. A few aspects will be mentioned here.

3.1 Software

The OSKit: The releases were picked up by hundreds of sites and have many users, ranging from the DARPA-funded Express project at MIT, which is using the OSKit to develop an operating system focused around the ML implementation language, to Network Storage Solutions, a company using the OSKit inside network-attached "file service appliances." Using the OSKit, MIT's Express project has gotten CMU's Foxnet system (TCP/IP in ML) to run on the bare hardware. Other users include the DARPA-funded exokernel project at MIT/CMU, the L4-clone-based Real-time Operating Systems project at TU-Dresden, and the Rice University Programming Languages group.

Flick IDL Compiler Kit: robust releases were downloaded by over 650 sites in 1998 alone. Flick was used for the DARPA-funded "Porcupine" distributed, scalable mail service at the University of Washington. The real-time OS project at Technische Universität Dresden ported Flick to L4 IPC, and are using it regularly. Flick was part of BBN's DARPA-sponsored QuOIN 2.0 software release in 1999, under the Quorum distributed object integration project.

Various commercial companies have also expressed interest in using Flick. The most promising is Apple. We have been talking regularly with Apple about the possibilities for using Flick to generate code for their Mac OS X operating system, because Flick offers the opportunity to help bridge the many disparate object models and implementations in their system. From this interaction we have been learning that Flick may have much to offer legacy systems through its flexibility. In December 1998, BBN included Flick in their (unfunded) proposal for DARPA BAA 99–03, "Data Intensive Computing."

Fluke/Flask: We publically released the Fluke/Flask kernel source in February 1999, timed to coincide with our OSDI paper on Fluke "atomicity." Portions of the Flask servers were released as part of regular OSKit releases during 1999. We plan to release the rest of the Fluke/Flask operating system later this year, timed to coincide with the release of Fluke's Java virtual machine incarnation, "Alta." Our NSA collaborators are integrating the Flask architecture and existing components into Linux.

3.2 Personnel

National Security Agency: two section R23 (Information Security Research) researchers, Jeff Turner and Steve Smalley, worked on-site at the University of Utah for one year from August 1996 through July 1997. They worked on Fluke/Flask and other security and assurance-related aspects of the Flux project. Close interaction with these and other R23 personnel continued for the duration of the project, and in general, continues to this day. Besides regular email and phone meetings, this includes joint paper writing and read/write access to shared source repositories,

Greg Benson, a Ph.D. student in the CS department at the University of California at Davis, joined us as a visiting Research Assistant from January–May 1997. His thesis topic was "run time support for advanced programming languages" and while with us he ported the SR parallel language to the OSKit. He co-authored a paper with us on the OSKit.

Khazana: many of the ideas and mechanisms have involved two-way tech transfer between Utah and MangoSoft Corporation (http://www.mango.com/), due to the co-PI's consulting involvement.

Our many changes to Kaffe, a popular commercial open-source Java Virtual Machine, to support flexible multithreading, flexible scheduling, improve robustness, and to support the OSKit, have been incorporated into the base version of Kaffe.

3.3 Relationships

During the course of the grant periods, we forged relationships with many companies who were interested in the DARPA-sponsored technology we were developing. For two companies, this interest was great enough that it was manifested in grants. To help support the research reported herein, during the grant period we received several cash grants from Novell and an equipment grant from Compaq/DEC. We had numerous technical meetings with Novell, the Open Software Foundation, Hewlett-Packard, Sun Microsystems, Apple Computer, IBM, Secure Computing Corporation, Trusted Information Systems, and BBN, to discuss our research and its relevance to their needs.

4 Publications

The Fluke/Flask Kernel and Operating System

[1] **The Persistent Relevance of the Local Operating System to Worldwide Applications.** Jay Lepreau, Bryan Ford, and Mike Hibler. In *Proc. of the Seventh ACM SIGOPS European Workshop*, pages 133–140, September 1996.

[2] **Microkernels Meet Recursive Virtual Machines.** Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 137–151. USENIX Association, October 1996.

[3] **User-level Checkpointing Through Exportable Kernel State.** Patrick Tullmann, Jay Lepreau, Bryan Ford, and Mike Hibler. In *Proc. Fifth International Workshop on Object Orientation in Operating Systems*, pages 85–88, Seattle, WA, October 1996. IEEE Computer Society.

[4] **Interface and Execution Models in the Fluke Kernel.** Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. In *Proc. of the Third Symposium on Operating Systems Design and Implementation*, pages 101–115, New Orleans, LA, February 1999. USENIX Association.

[5] **CPU Inheritance Scheduling.** Bryan Ford and Sai Susarla. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 91–105, Seattle, WA, October 1996. USENIX Association.

[6] **The Fluke Device Driver Framework.** Kevin T. Van Maren. Master's thesis, University of Utah, 1999. 101 pages.

[7] **Extending Fluke IPC for Transparent Remote Communication.** Linus Peter Kamb. Master's thesis, University of Utah, 1998. 79 pages.

[8] **The Flask Security Architecture: System Support for Diverse Security Policies.** Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. In *Proc. of the Eighth USENIX Security Symposium*, pages 123–139, August 1999.

[9] **Implementing Mandatory Network Security in a Policy-flexible System.** Ajaya Chitturi. Master's thesis, University of Utah, 1998. 70 pages.

[10] **Formal Methods: A Practical Tool for OS Implementors.** Patrick Tullmann, Jeff Turner, John McCorquodale, Jay Lepreau, Ajay Chitturi, and Godmar Back. In *Proc. of the Sixth Workshop on Hot Topics in Operating Systems*, pages 20–25, Cape Cod, MA, May 1997. IEEE Computer Society.

[11] **Assurance in the Fluke Microkernel: Formal Security Policy Model.** Secure Computing Corp. CDRL A003, 2675 Long Lake Rd, Roseville, MN 55113, February 1999. 102 pages. http://www.cs.utah.edu/flux/flask/.

[12] **Assurance in the Fluke Microkernel: Formal Top-Level Specification.** Secure Computing Corp. CDRL A004, 2675 Long Lake Rd, Roseville, MN 55113, February 1999. 121 pages. http://www.cs.utah.edu/flux/flask/.

[13] **Assurance in the Fluke Microkernel: Final Report**. Secure Computing Corp. CDRL A002, 2675 Long Lake Rd, Roseville, MN 55113, April 1999. 45 pages. http://www.cs.utah.edu/flux/flask/.

The OSKit: Reusable Components for OS Implementation

[14] **The Flux OS Toolkit: Reusable Components for OS Implementation**. Bryan Ford, Kevin Van Maren, Jay Lepreau, Stephen Clawson, Bart Robinson, and Jeff Turner. In *Proc. of the Sixth Workshop on Hot Topics in Operating Systems*, pages 14–19, Cape Cod, MA, May 1997. IEEE Computer Society.

[15] **The Flux OSKit: A Substrate for OS and Language Research**. Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St. Malo, France, October 1997.

[16] **The OSKit: a Manual (version 0.97)**. Flux Research Group. University of Utah. Postscript and HTML available at http://www.cs.utah.edu/flux/oskit/doc/. 504 pages, January 1999.

The Flick Interface Definition Language Compiler

[17] **Flick: A Flexible, Optimizing IDL Compiler**. Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 44–56, Las Vegas, NV, June 1997.

[18] **Flick: The Flexible IDL Compiler Kit Version 1.2c User's Manual**. Flux Research Group. University of Utah. Part of the Flick 1.2c software distribution, available at http://www.cs.utah.edu/flux/flick/. 48 pages, May 1999.

[19] **Flick Internals**. Flux Research Group. University of Utah. Part of the Flick 2.0 software distribution, available at http://www.cs.utah.edu/flux/flick/, November 1999.

[20] **Flexible and Optimized IDL Compilation for Distributed Applications**. Eric Eide, Jay Lepreau, and James L. Simister. In David O'Hallaron, editor, *Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR '98)*, volume 1511 of *Lecture Notes in Computer Science*, pages 288–302. Springer, May 1998.

[21] **Flexible IDL Compilation for Complex Communication Patterns**. Eric Eide, James L. Simister, Tim Stack, and Jay Lepreau. *Scientific Programming*, 7(3, 4):275–287, 1999.

MOM: The Mini Object Model

[22] **MOM, The Mini Object Model: Specification (Draft)**. Roland McGrath. July 1998. Unpublished report, available at http://www.cs.utah.edu/flux/docs/mom.ps.gz.

Java Operating Systems

[23] **Nested Java Processes: OS Structure for Mobile Code**. Patrick Tullmann and Jay Lepreau. In *Proc. of the Eighth ACM SIGOPS European Workshop*, pages 111–117, Sintra, Portugal, September 1998.

[24] **The Alta Operating System.** Patrick A. Tullmann. Master's thesis, University of Utah, 1999. 104 pages. Also available at http://www.cs.utah.edu/flux/papers/tullmann-thesis-abs.html.

[25] **Java Operating Systems: Design and Implementation.** Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hsieh, and Jay Lepreau. Technical Report UUCS-98-015, University of Utah, August 1998.

[26] **Drawing the Red Line in Java.** Godmar V. Back and Wilson C. Hsieh. In *Proc. of the Seventh Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, March 1999. IEEE Computer Society.

[27] **The Need for Predictable Garbage Collection.** Alastair Reid, John McCorquodale, Jason Baker, Wilson Hsieh, and Joseph Zachary. In *Second ACM SIGPLAN Workshop on Compiler Support for System Software*, Atlanta, GA, May 1999.

The Khazana/KOLA Infrastructure for Building Distributed Services

[28] **Khazana: An Infrastructure for Building Distributed Services.** John Carter, Anand Ranganathan, and Sai Susarla. In *Proc. of the Eighteenth IEEE International Conf. on Distributed Computing Systems*, pages 562–571, May 1998.

[29] **Experience Using a Globally Shared State Abstraction to Support Distributed Applications.** Sai Susarla, Anand Ranganathan, and John Carter. Technical Report UUCS-98-016, University of Utah, August 1998.

[30] **Supporting Persistent C++ Objects in a Distributed Storage System.** Anand Ranganathan, Yury Izrailevsky, Sai Susarla, John Carter, and Gary Lindstrom. In *Second ACM SIGPLAN Workshop on Compiler Support for System Software*, Atlanta, GA, May 1999.

DISTRIBUTION LIST

addresses	number of copies
AFRL/IFGA ATTN: KEVIN KWIAT 525 BROOKS ROAD ROME, NEW YORK 13441-4505	20
JAY LEPREAU DEPT OF COMPUTER SCIENCE UNIVERSITY OF UTAH SALT LAKE CITY, UT 84112-9205	2
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	1
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
ATTN: NAN PFRIMMER IIT RESEARCH INSTITUTE 201 MILL ST. ROME, NY 13440	1
AFIT ACADEMIC LIBRARY AFIT/LDR, 2950 P.STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1
AFRL/HESC-TDC 2698 G STREET, BLDG 190 WRIGHT-PATTERSON AFB OH 45433-7604	1

ATTN: SMDC IM PL 1
US ARMY SPACE & MISSILE DEF CMD
P.O. BOX 1500
HUNTSVILLE AL 35807-3801

COMMANDER, CODE 4TL000D 1
TECHNICAL LIBRARY, NAWC-WD
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6100

CDR, US ARMY AVIATION & MISSILE CMD 2
REDSTONE SCIENTIFIC INFORMATION CTR
ATTN: AMSAM-RD-OB-R, (DOCUMENTS)
REDSTONE ARSENAL AL 35898-5000

REPORT LIBRARY 1
MS P364
LOS ALAMOS NATIONAL LABORATORY
LOS ALAMOS NM 87545

ATTN: D'BORAH HART 1
AVIATION BRANCH SVC 122.10
FOB10A, RM 931
800 INDEPENDENCE AVE, SW
WASHINGTON DC 20591

AFIWC/MSY 1
102 HALL BLVD, STE 315
SAN ANTONIO TX 78243-7016

ATTN: KAROLA M. YOURISON 1
SOFTWARE ENGINEERING INSTITUTE
4500 FIFTH AVENUE
PITTSBURGH PA 15213

USAF/AIR FORCE RESEARCH LABORATORY 1
AFRL/VSOSA(LIBRARY-BLDG 1103)
5 WRIGHT DRIVE
HANSCOM AFB MA 01731-3004

ATTN: EILEEN LADUKE/D460 1
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730

OUSD(P)/DTSA/DUTD 1
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.