AFRL-IF-RS-TR-2006-210
Final Technical Report
June 2006

EROS-BASED CONFINED CAPABILITY CLIENT

Johns Hopkins University

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

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

STINFO FINAL REPORT

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-2006-210 has been reviewed and is approved for publication.

APPROVED: /s/

JAMES L. SIDORAN
Project Engineer

FOR THE DIRECTOR: /s/

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

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JUNE 2006	Final	Jul 01 – Feb 05

4. TITLE AND SUBTITLE
EROS-BASED CONFINED CAPABILITY CLIENT

5a. CONTRACT NUMBER
F33615-01-C-1972

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
69199F

6. AUTHOR(S)
Jonathan S. Shapiro

5d. PROJECT NUMBER
ARPS

5e. TASK NUMBER
NZ

5f. WORK UNIT NUMBER
OM

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Johns Hopkins University
3400 N. Charles Street
Baltimore Maryland 21218-2680

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/IFGB
525 Brooks Rd
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2006-210

12. DISTRIBUTION AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 06-429

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Objective of EROS-Based Confined Capability Client Project is to demonstrate that systems constructed using capability-based protection can be organized in a way that makes it defensible from hostile internet content. This was accomplished by constructing o a single exemplar application, a web browser using capability-based structuring techniques, and determining whether this application can defend itself against hostile content. The specific test employed demonstrated that the browser always renders the URL being displayed accurately, and that this rendering cannot be altered by any means available to the page author. Means of attack available to the page author include exploiting flaws in the browser implementation, such as buffer overrun vulnerabilities. The work performed produced four specific results: a secure network protocol stack, a trusted window system, a browser prototype, and an assessment of the cost and complexity of porting existing applications to capability-based foundations as currently implemented. One important conclusion is that it is possible to build high-performance, defensible systems using capability-based protection and confinement to provide defense in depth that is difficult for either hostile content or well-intentioned misconfiguration to compromise.

15. SUBJECT TERMS
Capability-based protection techniques, defense in depth, web-based applications

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			James L. Sidoran
U	U	U	UL	28	19b. TELEPONE NUMBER (*Include area code*)

Table of Contents

List of Figures

1 Introduction and Overview

This document is the final report for EROS-Based Confined Capability Client project (CCCP), a project in the DARPA Fault Tolerant Networks (FTN) program. The project was funded through the Air Force Research Laboratory (AFRL) Advanced Technology for Information Assurance and Survivability (ATIAS) program. This project was proposed under ATIAS Focus Research Topic (FRT) #5, which specifically addressed defensibility against hostile content in capability-based systems.

1.1 Statement of the Problem

Reading web content is frought with peril. There are problems at several levels: (1) current commodity operating systems are not secure enough to safely connect to the web, (2) all current browser rendering systems contain exploitable bugs, even in the absence of hostile scripting, and (3) web content contains scripting code, which may be hostile. These problems are not limited to web content; they arise for all of the currently available mechanisms of network-based content dissemination. Adobe's "Portable Document Format," for example, is similarly vulnerable, as is Microsoft Word, Macromedia's Flash Player, and other scriptable systems.

The objective of CCCP is to construct a web browser prototype in which hostile content cannot obtain control of the viewing machine. As a specific, concrete test of control, the solicitation required that the browser should display the URL of the web page being viewed, and that it should be impossible for the hostile content to alter this display.

The creation of a web browser prototype was chosen as a proxy for a much larger problem. An indirect goal of the solicitation was to evaluate the complexity of porting general-purpose applications to capability-based operating systems.

1.2 Proposed Solution

Our proposal is that solution to these problems should be structural. Since we cannot humanly eliminate 100% of the flaws in the content dissemination system, we must contrive to execute them in an environment that imposes control from the outside. The difficulty is that some of the needed controls cannot easily be established at the operating system level — they require knowledge of application-specific behavior.

Our proposed solution is to implement a browser prototype on top of the EROS operating system [13], using confinement [10] and capabilities as the fundamental structuring tools for our implementation. By confining the hostile code within a controllable boundary, and surrounding it with a "cocoon" of small, simple components that restrict the application's interaction with the outside world, it should be possible in many cases to render the hostile code harmless.

EROS is a research operating system built entirely around capability-based protection. The method of creating new processes in the EROS system guarantees that nearly all processes are confined. At the time of the solicitation, EROS was a minimal research system lacking a network stack, a windowing system, or interactive applications. Therefore, the work plan divided into four main tasks:

1. Implement a prototype defensible network stack.

2. Implement a prototype secure window system.

3. Construct an early browser-like exemplar to satisfy the concrete requirements of the solicitation.

4. Port an existing full-featured browser, thereby evaluating the cost and complexity of porting and containing generalized applications.

The first two tasks are non-trivial. While many network stacks exist, none provide any degree of defense in depth. All include flaws by which one application can compromise the network communications of a second. Current window systems provide essentially no controls over application interactions. In a browser designed to manage hostile content, a critical issue is to ensure that tight control is maintained over where hostile content may appear on the display. To achieve this, a new window system was needed.

To simplify the problem, we chose in several respects to generalize the requirements of the solicitation. For example, we generalized "must not be able to tamper with the displayed URL" to "must not be able to draw anything outside of its designated rendering area." We also chose to extend the notion of a web page with hostile content to include the network connection over which the content is delivered. That is, we consider the possibility that a hostile web server may attempt to compromise the browser by compromising the underlying network stack and operating system.

1.3 Contributions of this Project

The project produced four specific results:

- Design, implementation, and evaluation of a network stack that exploits capability-based protection to provide high-performance while substantially reducing the number of vulnerable lines of code in the overall network subsystem.

- Design and implementation of a trusted window system that imposes confined display subsessions on applications. This window system is significantly more flexible than previous trusted window systems (notably Trusted X [5]), several orders of magnitude smaller than conventional window systems, and potentially offers higher performance than current window system software.

- A browser prototype that demonstrates confinement of content. This prototype is limited in that it does not fully implement browser-style rendering. It also fetches content using TFTP rather that HTTP. However, neither of these simplifying differences changes the confinement outcome: the browser is confined in spite of the fact that both the browser implementation and the content it displays are presumed hostile.

- An evaluation of the cost and complexity of porting general-purpose software and ensuring that it runs in contained form. While we were not successful in porting a production browser within the timeframe of the contract, we now know exactly what needs to be done in order to perform such ports quickly, and we understand what sorts of application changes are required to isolate such applications.

An indirect contribution of this project comes from the extension and evaluation of the existing EROS system structure. In particular, our work on porting large bodies of existing software has prompted a small number of key architectural changes that are being adopted in the EROS successor and will make such ports cost effective in future work. Our success in the network and window system subtasks is a strong indicator that the key elements of the EROS architecture are extremely effective at providing high-performance security.

1.4 Report Organization

There are six sections to this report. Section 2 covers technology background. Section 3 discusses our technical approach. Section 4 provides a summary of our results and their significance. Section 5 discusses why the combination of confinement and capabilities was uniquely powerful in this project. Section 6 covers conclusions and recommendations for future work.

1.5 Summary Conclusion

It is possible to build high-performance, defensible systems using confinement and capabilities. While CCCP did not succeed in porting existing commodity software to the new platform, the problems encountered are straightforwardly fixable. The CCCP result demonstrates that complex subsystems can be restructured using capability-based protection and confinement to provide defense in depth that is difficult for either hostile content or well-intentioned misconfiguration to compromise.

2 Technology Background

This section provides definitions, background, and prior context for the CCCP effort.

2.1 Fundamentals of Hostile Program Control

A capability is a protected data structure consisting of an object reference and a set of permissions. Conceptually, it is similar to a car key:

- If you have the key (capability), you are in control of the car (object). If you don't, you can't make the car (object) do anything at all. Unlike a car, the objects that capabilities named cannot be "hot wired." No capability, no access.

- Like a car key, a capability cannot be obtained unless you already have one. You can copy a capability, but you cannot forge one.

- In fancier cars, you may have several types of keys: the driver key, the valet key, the glove compartment key, and so forth. Each key permits different actions on the car. In a similar way, you can have multiple capabilities to the same object, each permitting different operations on that object.

- As with car keys, the essence of capability security is to avoid giving the capabilities to the wrong party. In the case of capabilities, we want to avoid giving them to programs that may misuse them.

In a capability-based operating system (or programming language), the only way to perform any action is to invoke some capability that you have. A consequence is that the set of actions that a program might perform are entirely defined by the capabilities that it holds. Control the capabilities and you control the program.

EROS is a capability-based operating system. It was used as the foundation for the CCCP effort.

Confinement is a compartmenting mechanism for programs. A program is confined if it can only send information to other programs using authorized channels. In a capability system, this can be restated as: an application is confined if the only writable capabilities that it holds are authorized (capabilities to read-only objects do not violate confinement). EROS provides a formally verified confinement mechanism known as the constructor. The constructor is the standard mechanism for creating new processes in the EROS system. In consequence, every process in the EROS system is initially confined, and its controlling process must explicitly permit access to other processes and components. This is in contrast to UNIX, where every process initially has very broad access rights, and the controlling process must somehow restrict these rights.

Concept: Principle of Least Privilege

If you want to build a secure system, every component should start with no privileges and you should then grant only those privileges that the component requires. Trying to take away privilege after the fact empirically doesn't work. This is analogous to the "need to know" principle.

If you want to restrict what a program can do, two things are necessary:

1. You need to know exactly what it can do. In a capability system, this is accomplished by controlling what capabilities the program is given.

2. You need to know that that is all it can do. This is accomplished by ensuring that the program is confined.

   Concept: Control = Confinement + Least Privilege

   Confinement plus control over what capabilities are granted constitutes complete and total control over the actions of a program even if the program is actively hostile.

The "actively hostile" part deserves emphasis. If you completely control the interactions that a program can have with the surrounding system, and you can determine that all of these actions are non-threatening, then it does not matter which of these interactions actually occur. Controlling the operational environment of the program renders it harmless.

2.1.1 Practical Limits on Effective Control

In practice, there are two limits on the control that can be achieved using confinement and least privilege:

1. The control provided does not address covert channels. Covert channel controls were not required by the FRT, but it is important to recognize that the control provided by this technique is not total control. The control provided is sufficient to prevent penetration, but it is not sufficient to prevent leakage of information that may be disclosed to a hostile program.

2. There is always some point where the potentially hostile component must communicate with the surrounding system. For example, a user may click on a web link, and the hostile web page must now communicate with the web browser to request that a new page should be loaded. This is both good news and bad news:
   - The bad news is that any communication between an untrusted program and a program that has the power to do something damaging presents an opportunity for attack.

- The good news is that these are the only points of vulnerability to attack, there aren't very many of them, the developer knows where they are (and can therefore apply paranoid design in a small number of pivotal places), and each interaction point requires a differently specialized attack.

The combination of confinement and least privilege restores the advantages of position, terrain, and control over terms of engagement to the defender. A weak lead software developer can still lose this battle, but it is conceivable that a merely competent software developer might successfully defend themselves.

2.1.2 Defense in Depth

The description above describes the restoration of control that is achieved when the system design provides a defensible bottleneck and confinement and least privilege are applied at this bottleneck.

The obvious extension of this idea is to design a system in which there are a series of defensible bottlenecks in the system design, each of which must be independently attacked in order to penetrate. This is comparable to designing a mechanical system in which three or four related parts must fail in sequence to cause an overall system failure.

I should emphasize that the "bottlenecks" discussed here are not performance bottlenecks. A better term might be "checkpoint." The flow of information across these checkpoints can be quite fast. The key issue is that these are points that information and requests must cross, where it is natural and efficient to check the requests and information that is flowing.

2.2 EROS

EROS is a high-performance research operating system built entirely around capability-based protection as a foundational mechanism. It provides a basic utility for confinement that has been formally verified [14]. EROS runs on commodity Pentium-family machines, requiring no unusual hardware to provide security. If run on a system providing hardware-supported tamper-resistant bootstrap, EROS is potentially capable of resisting compromise by operating system replacement and is further potentially capable of establishing secure distributed systems sharing a common trusted computing base. DARPA elected not to fund the optional work to demonstrate this.

At the start of CCCP, EROS consisted of a kernel and a very small number of low level operating system utilities. Functionallly, the EROS system was comparable to a Linux kernel with no networking, no graphics system, no file system, and no applications.[1]

[1] Actually, we had a "hello world" program, and it was quite secure, but it wasn't terribly useful for CCCP.

There was extremely limited language support (a C compiler, but essentially no C library).

In addition, EROS had no simple mechanisms to let developers hook together programs that we might write. In Linux, the closest mechanism is the pipe, which lets one program send a byte stream to a second. In EROS, the system is designed to communicate by messages. Message-based systems are more expressive, more powerful, and more flexible than stream-based designs, but they require a greater degree of effort to specify what the interface (the messages) between two programs should be. While we knew approximately what the interface definition tool should look like, we did not have a concrete design the interface description language or a tool that managed the code generation for these interfaces.

2.3 Alternatives to EROS

In spite of the relative incompleteness of EROS at the start of CCCP, there are no OS-based alternatives. Only one other system (KeyKOS [7]) has ever been constructed with comparable architectural support for confinement and least privilege. KeyKOS (still) does not run on the most important processor architectures, and no license for the KeyKOS technology was available at the time of project start. In any case, the state of the KeyKOS system was roughly equal to that of EROS,[2] with the additional complication that much of KeyKOS is implemented in PL/1, a language with limited compiler support.

Mandatory access controls, such as the multilevel security required by TCSEC and/or the more general controls implemented by SELINUX are not sufficient to address this problem. In the case of TCSEC, the issue is that this isn't a mandatory access control problem. The problem here is discretionary — allowing an interactive application to defend itself from hostile content in spite of faulty implementation. In the case of SELINUX, the problem is that the control requirements cannot be specified at the operating system level. For example, isolation requires a one-way network communication channel. The hostile page should not be able to communicate outwards over the TCP/IP connection that it is using to read the page. At the TCP level of abstraction, however, various control messages must flow to the server machine to implement the TCP protocol. The issue is that the connection must be read/write at the lower level of abstraction and read-only at the higher level of abstraction. Expressing and implementing this requires application-specific knowledge, which is beyond the scope of what SELINUX and similar mechanisms can handle.

One viable alternative approach to the one we proposed is a language-based solution. The architecture of this approach is fundamentally comparable to the one we adopted, but is implemented at a different level of the system. This approach was taken by Combex, Inc., who was the other contract awardee under this FRT.

[2] EROS initially started as a reverse-engineering effort intended to rebuild the KeyKOS system using modern tools. It later became a research effort in its own right.

If the high-level objective is to build a single application solving one particular problem (e.g. a web browser), the language approach is clearly preferred. By building on a carefully stripped-down operating system and running only one application (the browser), a tremendous amount of control can be obtained at low cost. The limitations of this approach are:

- It doesn't scale. In the absence of operating system controls like the ones we apply in EROS, the second and subsequent applications cannot be protected from the first.

- It doesn't generalize. Language-based solutions are wonderful when all of the code you need can be found in a safe language (e.g. Java). The overwhelming majority of code out there isn't written in safe languages, and we need to be able to re-use it.

This statement of limitations is endorsed by the Combex team. Combex was able to build a very successful capability-based confined browser, but they acknowledge that the underlying Linux system is a vulnerability in their approach. Modestly larger applications than the one required for this FRT would force them to introduce code written in C using the Java Native Interface mechanism, and that this code would be vulnerable and not controlled by their approach.

The EROS OS-based approach was intended to work up from the bottom to provide a platform on which generalized, scalable solutions would be possible.

3 Technical Approach

This section describes the approach and design used in CCCP.

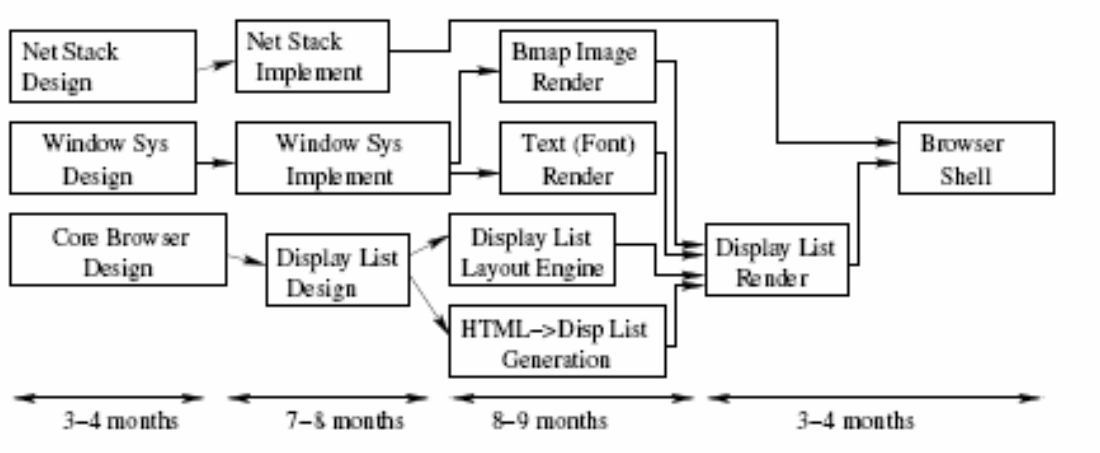

Figure 1: Originally proposed task workflow for CCCP.

The original proposal called for a three-pronged development path with late integration (Figure 1). The general idea was to implement a network stack and a window system while pursuing the application portability problem, relying on the assumption that most Linux applications — including the browser we intended to port — use a low-level graphics toolkit, and that the graphics toolkit already had a highly portable implementation that would mate up with our window system when it was ready.

Accordingly, we split the effort into three teams. The networking team set out to port the lwIP network stack to EROS. The windowing team set out to architect and implement a trusted window system, and the application team set to work porting the GTK toolkit, the Pango rendering system, and ultimately the khtml browser to EROS.

3.1 Defensible Network Stack

The networking stack effort needed to address two challenges:

- The EROS software environment doesn't look anything like other kernel software environments. How could we create a compatibility environment that would allow us to quickly adapt an existing network stack?

- Because it is so large, the network stack is a significant source of systemic vulnerability. How could we reduce this, both by reducing the amount of trusted code (limiting global vulnerability) and by achieving better separation across applications (localizing vulnerability to a single victim).

Because it is highly portable and has run successfully in a number of embedded environments, we chose to port the lwIP [4] network stack to EROS. Our strategy was to break the stack into separated ethernet driver and protocol portions, establish suspicious interfaces between these components, and polyinstantiate the protocol stack to provide isolation between applications.

To achieve separation of concerns, we chose to implement our own low-level network driver. This allowed us to establish a multi-level defensible network stack (Figure 2). In the long term it should be possible to build an encapsulation environment for Linux ethernet drivers in general, as has been done in OSKit <citer></citer>.

From a research perspective, the primary innovation in this networking stack is the successful combination of defensible layer boundaries and high performance. An exploitable flaw in the ethernet driver can compromise network traffic, but cannot tie up other system resources. Compromise of the ethernet allows the attacker to send bad data packets into the TCP/IP subsystem, but does not allow the attacker to directly communicate with the application. The TCP/IP stack, for its part, checks incoming packets to validate that they are well-formed. Due to operating-system enforced interface boundaries, the ethernet driver is unable to make arbitrary calls into the network stack code. This means that a successful upwards attack must contrive to get the TCP/IP stack

to compromise itself by mis-processing well-formed input. Finally, each connection has a private TCP/IP stack. An exploitable flaw in one TCP/IP stack does not impact other applications.

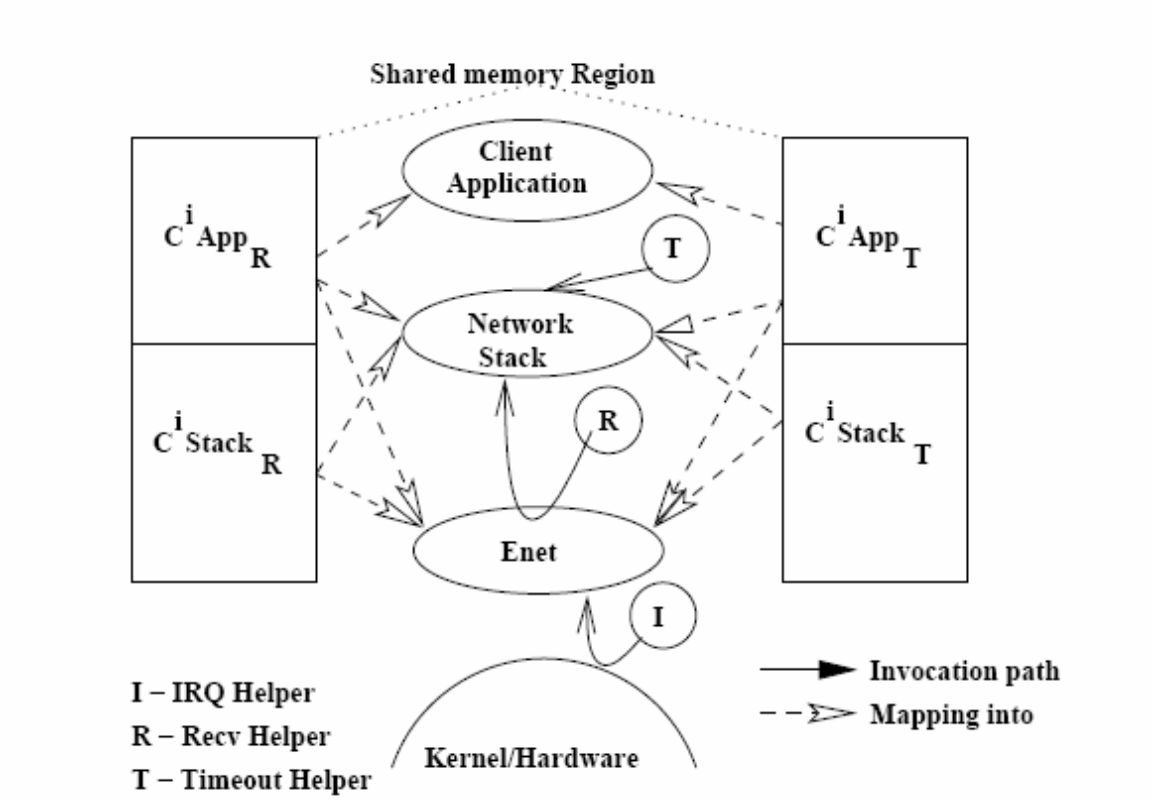

Figure 2: Network stack with ethernet driver, protocol stack, and application in separated protection domains.

Surprisingly, this division does not substantially alter network performance.

3.2 Secure Window System

The window system presented more fundamental challenges than the network stack. Current window systems fail to enforce security in two regards:

- They implement no effective isolation between applications. One application may draw in the window of another.

- They provide a general-purpose communication system through cut&paste. This allows arbitrary applications to influence one another, and to establish long-running bidirectional communication channels (a potential tool for Trojan horse exploits).

In order to meet the requirements of CCCP, we needed to solve these problems and go an additional step further: ensure that two components of a single application could be isolated from each other.

After a significant false start, we arrived at a window system design providing a hierarchical session structure that guarantees isolation between applications as well as between an application "shell" and a specific rendering subsystem. Further, the window system provides an authenticated cut&paste mechanism enforcing a purely unidirectional channel.

3.3 Application Porting

This portion of the effort was designed to evaluate the cost and feasibility of porting existing general-purpose software to a capability-based platform.

The application porting team had three major sub-tasks:

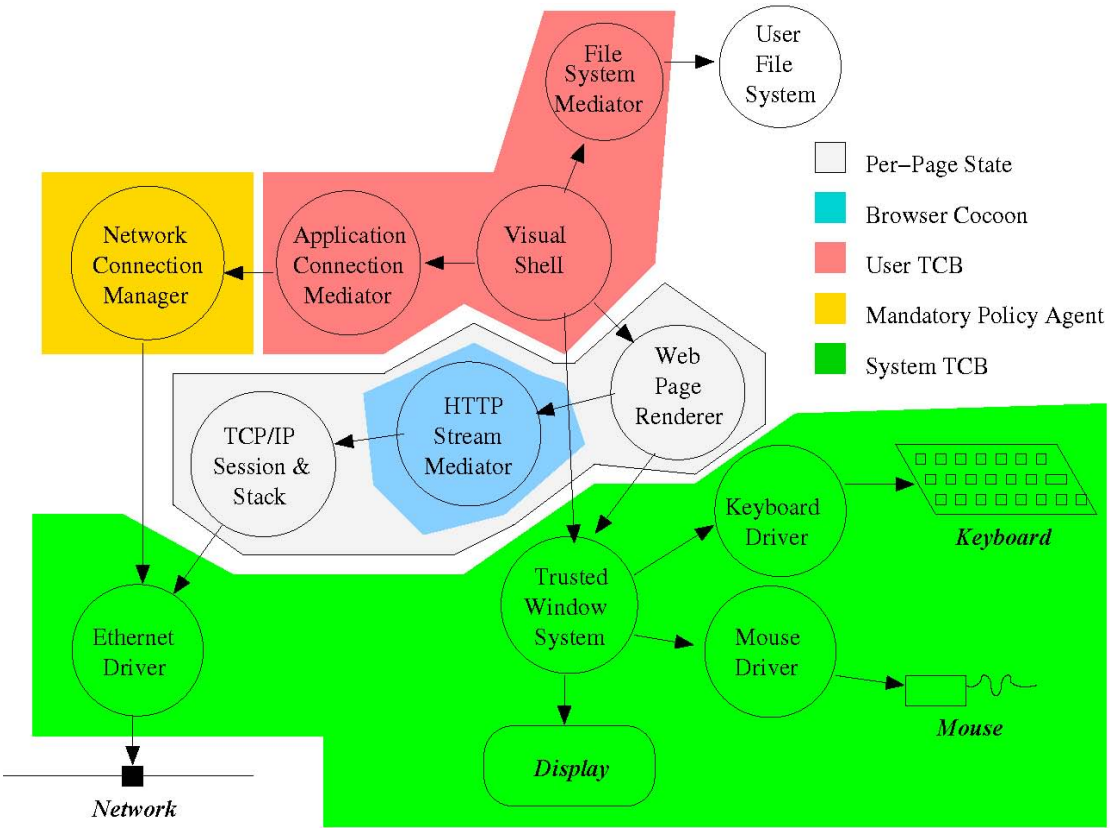

Figure 3: Structure of the capability confined browser and surrounding system.

- Port the GTK library, which provides a platform-neutral toolkit for many applications.
- Port the Pango rendering system, used to draw styled, anti-aliased text in multiple fonts, faces, and styles.
- Port the KHTML web browser to the EROS environment and adapt it to use the previous two libraries.

The application porting effort involved no fundamental invention. Our naive expectation was that this would simply be a matter of porting some UNIX libraries to run in the native EROS environment. This assumption was grossly mistaken, and for reasons discussed in the evaluation section below, our porting efforts failed.

3.4 Overall Design

The overall structure of the final CCCP environment is shown in Figure 3. Each circle indicates a separate process running in its own address space. Arrows indicate capabilities between processes. If process A can invoke services from process B, there is an arrow from A to B.

A capability grants the right to request services. It does not provide the holder with any ability to tamper with, alter, or examine the state of the service provider. The processes shown divide approximately into four groups:

- Processes that are part of the system services layer (green background). These processes are part of the systemwide trusted computing base, and share in the responsibility for isolation. For example, the Trusted Window System implements isolated sessions.

- Processes that implement mandatory policy (yellow). These processes are responsible for implementing administrator-defined policy, such as administrative restrictions on what sites may be contacted.

- Processes that implement per-user policy (pink). These processes serve as the agents of the user, and are responsible for guarding the user's interests and information by imposing interaction restrictions on behalf of the user.

- Non-sensitive processes (uncolored). These processes are non-sensitive either because they do not provide an opportunity for compromise (the TCP/IP stack) or because they are entirely untrusted and presumed hostile.

The key requirement of the FRT is the ability to impose application-specific mandtory policy on the subject application. One of the unfortunate legacies of the Orange Book (TCSEC) [3] is a fundamental confusion about the roles of mandatory and discretionary policy. Before explaining the confusion, let us state the definitions:

- **Discretionary Policy** is policy that is imposed by a program.

- **Mandatory Policy** is policy that is imposed on a program.

The confusion of the Orange Book and Common Criteria [9] standards is to assume that mandatory policy can only be imposed by the operating system. In fact, the distinction between mandatory and discretionary policy is purely a difference of point of view. If you are a process doing the enforcing, the policy you enforce is discretionary. If you are a process getting enforced, the policy is mandatory. The essential point about a mandatory policy is that it cannot be circumvented by the restricted process. While implementing a mandatory policy in the operating system certainly can ensure this, other arrangements can also do so.

For example, suppose we wish to impose the mandatory policy that no web page should be able to communicate outward to its server. This policy is actually quite hard to implement. We tend to think of HTTP (the protocol that serves web pages) as an application-level protocol, but several published results have shown that the HTTP protocol is powerful enough to use as a transport protocol — you can build a stream protocol on top of HTTP, and you can transmit arbitrary information over that stream.

It is tempting to imagine that a stateful firewall could be used to check that the HTTP protocol is not being abused, but this does not work. Using HTTP as a low-level transport protocol does not require misuse of the protocol! In fact, the first version of TCP on top of HTTP was implemented specifically to circumvent corporate firewalls in support of an entirely legitemate conferencing application.

In CCCP, the solution we chose was to interpose an HTTP Stream Mediator between the web page renderer and the network. The renderer never receives direct access to the TCP/IP layer at all. The HTTP stream mediator is responsible for establishing the HTTP connection and reading the content from the server. Internally, it makes the resulting bytes available to the web page renderer. This gives us an implementation of the policy, but why is it mandatory?

In order to be mandatory, we must ensure that this policy cannot be circumvented. This is accomplished by a series of steps:

1. When opening a page, the first thing that happens is that the visual shell creates a new HTTP stream mediator for the requested URL. The web page renderer never has access to the network or ability to create a new network connection.

2. The renderer holds a capability to the HTTP stream mediator. In theory, the stream mediator has the ability to send arbitrary data to the server, but it will not do so. Furthermore, the capability held by the web page renderer does not provide any write operation.

3. The visual shell holds a capability to the application connection manager that identifies it as the visual shell. The application connection manager will only open HTTP connections when requested to do so by the visual shell.

4. The application connection manager in turn holds the only capability to the network connection manager, which in turn holds the only capability to the ethernet. Ultimately, this means that the network connection manager is the only way one can establish an ethernet connection.

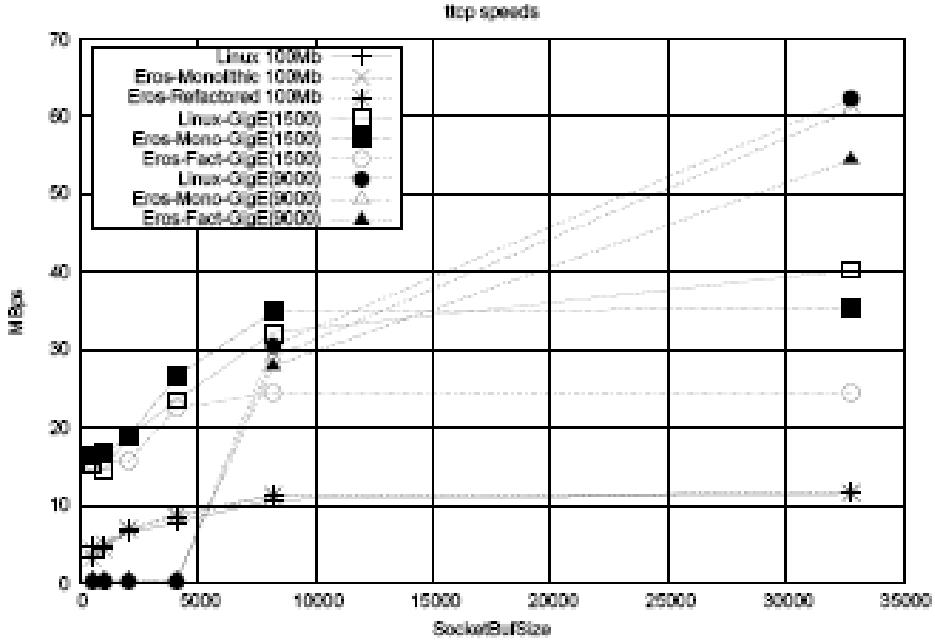

Figure 4: TTCP throughput measurements.

5. The network connection manager, when asked for a connection, will only respond by providing either (a) a failure result, or (b) a capability to a freshly instantiated TCP/IP stack.

6. Because each TCP/IP stack is a private stack, there is no possibility of cross-talk between two TCP/IP connections.

If the only application of interest is a web browser, this structure of reliance is excessively complicated. However, as other applications are added to the required set, this structure places enforcement responsibility at the most natural choke point for each individual requirement.

Note that even if the system provides a compiler, the user cannot construct an alternative browser that would compromise this chain of controls. While an alternative visual shell — let's call it alternate shell — could be constructed, the user does not have and therefore cannot give it the necessary capability that would cause the application connection manager to agree to open an HTTP connection for the alternate shell.

We used a combination of this sort of mediator pattern and isolation implemented by the trusted computing base to guard against the presumptively hostile web content and renderer.

4 Evaluation

This section discusses the results obtained in the CCCP effort.

4.1 Network Stack

The network stack implementation achieved all of our objectives. TCP throughput is shown in Figure 4. While latency is very slightly higher (Figure 5) than the Linux network stack, throughput is directly comparable to Linux.

Details of this result are given in [12], but the essential result for CCCP is that we were able to provide a high-performance, low-latency network stack split across several independently defensible protection domains without

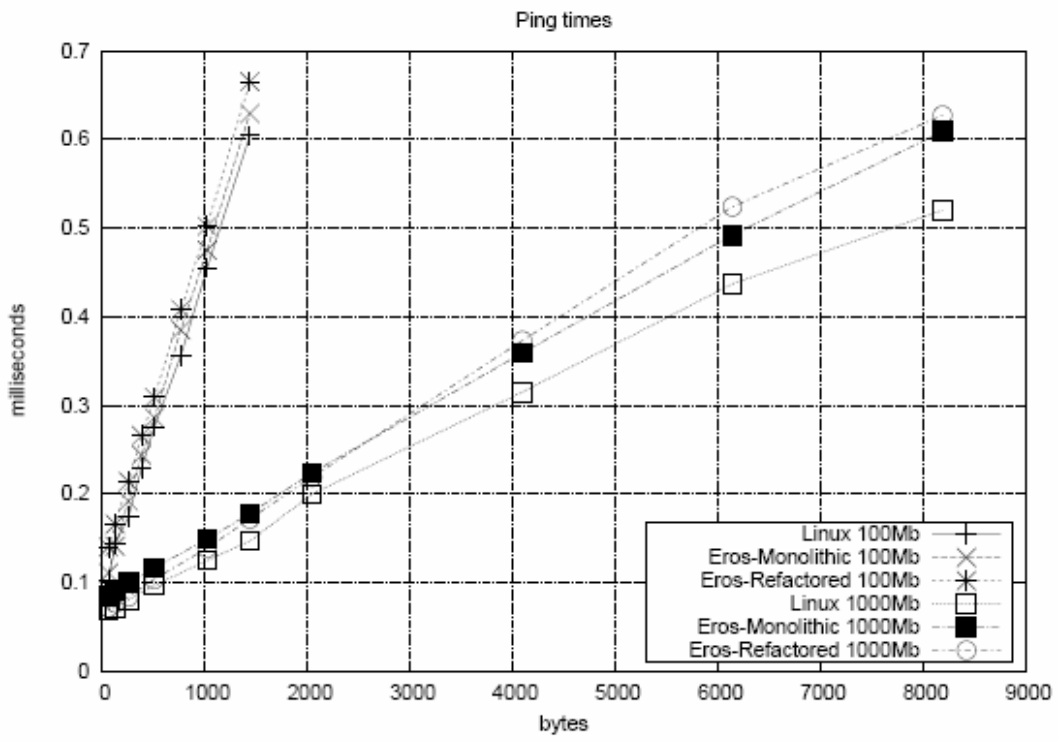

Figure 5: Ping latency compared to Linux.

any substantial loss of network throughput. This suggests that the arguments against layered multiplexing made by Tennenhouse [17] were somewhat misframed. The issue is not layering per se, but the additional data copy costs that are commonly incurred in layered designs. Because the EROS network subsystem is able to use an efficient shared-memory interface across mutual suspicion boundaries, it is not necessary for the implementation to explicitly copy packets across protection boundaries. The result is that

we achieve performance comparable to that of the IoLite design [11] while preserving separate protection domains.

A second key innovation in this network stack lies in the accountability for resources. A common problem in general-purpose network stacks is that memory and CPU resources are not properly accounted to their associated application [1]. In the EROS network stack, both CPU and memory resources associated with a particular TCP connection are provided by the client application. No action by a client will cause the network subsystem to consume memory that is accounted to another application.

The combined effect of this is to make successful attacks through the network stack substantially more difficult than in conventional systems. In addition, the utility of successful penetration is reduced to a single application, and cannot by promulgated into complete control of the target system.

4.2 Window System

The two key challenges of the window system effort were to substantially reduce overall code complexity and imple ment inter-application isolation. Detailed discussion of the design and its results may be found in [16].

The trusted window system reduces complexity by moving all responsibility for rendering into the client. The window system and the client share a memory buffer that is allocated using per-client resource. This provides the client with a double-buffered display system. By having the window system copy bits from the client buffer to the frame buffer at will, update-related covert channels are eliminated, and the total code size of the window system is reduced to approximately 5,000 lines. Figure 6 shows our earliest test application using the double-buffered secure display system. The artist is the son of one of our project members.

Isolation is achieved through the implementation of a session abstraction in the window system. An application can only modify windows associated with that application’s sessions. Session and window capabilities are implemented by the window system to enforce this restriction. We also invented a secure cut & paste design by exploiting confinement to achieve a purely unidirectional cut & paste operation without giving up the flexibility of content style and format negotiation.

Figure 6: Work of a proto-Picaso using ErosPaint under the secure window system.

The session isolation mechanism allows the visual shell to implement a “container” around the web browser that the web browser cannot modify. The container displays menus and the active URL. The browser itself (that is, the portion that draws the page) runs in a subordinate session.

The primary metric of success for the window system is size and functionality. The prototype visualizers shown in Figure 7 demonstrate that rich rendering is possible within the prototype design, including all of the elements required for a full-service browser. Each of these visualizers is implemented as a confined subsystem using capability-based confinement.

4.3 Application Porting

Our attempts to port large-scale application software from the Linux environment largely failed. In hindsight, the reasons for this are painfully obvious:

1. Modern interactive application toolkits are multithreaded and tend to rely internally on the non-blocking notification primitives of the underlying operating system. EROS was specifically designed to discourage this type of usage pattern, and does not support it well.

2. While the libraries and applications themselves would have ported easily, the build-time configuration tools for these applications have become entirely POSIX

dependent. Nearly every open source application in the world relies on the GNU autoconf utility, which in turn relies on having a rich, compliant POSIX environment.

Taken together, these impediments prevented us from performing any substantial port within the timeframe of the project. The application team put in overwhelming effort trying to hand-configure these subsystems without success. Fortunately, both issues are conceptually easy to solve.

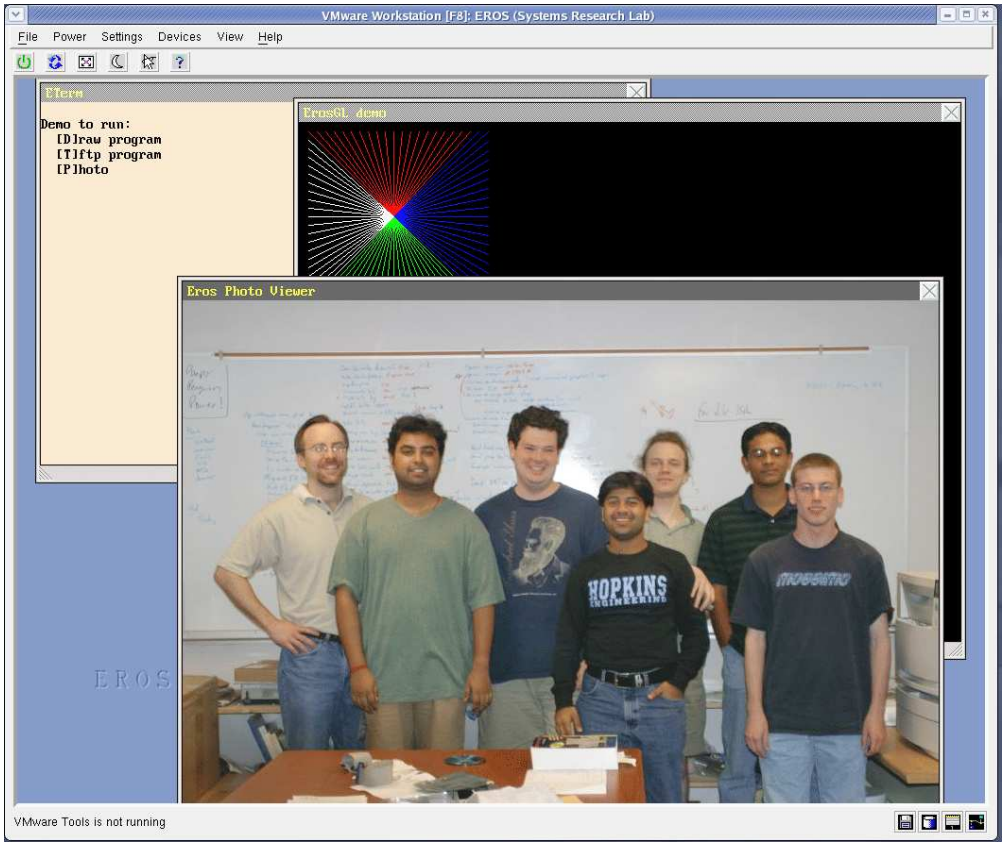

Figure 7: Various prototype visualizers under the secure window system. Foreground picture shows the CCCP contributors.

The first issue is EROS-specific, and relates to the design of the EROS interprocess communication system. EROS provides no mechanism for non-blocking communication. One has been added in the specification of the EROS successor. Similarly, EROS does not provide any straightforward mechanism for polling or multithreading. We have revised the successor design to address this.

With the introduction of these new features in the EROS successor, it will be straightforward to bring up a POSIX-compatible environment. At that point we will be able to directly support application-level multithreading and also to run autoconf -based configuration scripts.

5 Discussion

While the CCCP results show that capability-based protection is feasible and efficient, a key question that needs to be answered is: Is it necessary? That is, is there anything in the work we have done that was fundamentally enabled by capabilities and confinement and could not have been done as easily in a more conventional system design? Now that we know how to do this, could we perhaps simulate these results using the Linux Security Module feature set or similar controls?

The technical work done here exploits capability-based protection and confinement in three ways:

1. To narrowly restrict the set of operations (system calls and communication paths) accessible to each component.

2. To provide each component with tamper-resistance that prevents bypassing the policy implemented by that component.

3. To facilitate certain "suspiciously shared memory" designs that are critical to high performance across asymmetric trust boundaries.

Restricting Operations Restrictions on operations can be imposed in any operating system by using system call filtering. In principle, it is possible to build a policy module that narrows the operations accessible to each process in the way described here. In practice, the complexity of the engineering required is considerable, and highly prone to mistakes of omission.

The critical problem with filtering-based solutions is that they tend to be "fail open" rather than "fail closed." As new features are introduced into the underlying system, new paths of communication are introduced as well. There is a lag before application-specific filters are updated, and during this lag there exist exploitable vulnerabilities. This is essentially the same lag that we currently experience with antivirus and intrusion detection systems. The filtering approach is fundamentally reactive rather than proactive.

Capability designs, in contrast, tend to be "fail closed." Adding new features to the system does not alter the capabilities held by a given application, and does not increase its authority. As a result, the application tends to remain as safely restricted as it was before the feature introduction. One possible exception to this is the addition of new features to some object that is part of the existing application structure. In contrast to general system features, object-specific features are introduced in a locally defined requirements context, and are usually introduced without expanding the existing security vulnerabilities of the object being revised. Because of this "locality of modification effect" property, capability-based protection offers a kind of robustness under engineering, maintenance and evolution that we do not understand how to achieve in non-capability designs.

Tamper Resistance A key element in the defensive structure that we have created is the ability to construct services that can be invoked (called) without being examined or modified. For example, the HTTP stream mediator is an ordinary application. Its utility lies in the fact that it is always present (therefore always mediating) and cannot be bypassed.

In a non-capability operating system, this simple set of constraints is surprisingly difficult to achieve. If the browser runs on behalf of some user fred , and a mediating process also runs on behalf of a user fred , then the browser is in a position to modify the state of the mediating process by virtue of running within the same access control domain (i.e. the same user). In addition, the (human) user of the system is in a position to modify the mediating programs. Because of this, mediating applications are hard to construct and defend in non-capability systems. The customary solution is to make such mediating applications privileged, which leads to other problems such as the “Confused Deputy” scenario [8].

In the CCCP arrangement, these applications are trustworthy because neither the user nor the application can bypass them, but they are not in any way privileged.

Asymmetric Trust The essential enabler to performance in the defensible network stack and the trusted window system is the ability to establish high-performance shared memory regions between processes that do not fully trust each other. The tricky problem in doing this comes from the combination of three requirements that are generic to all robust and secure systems:

1. Resources should be accounted to the application that they serve.

2. He who pays for storage that is allocated must be able to deallocate it.

3. Shared, trusted subsystems must never be stopped or caused to fail by non-trusted subsystems.

The first requirement means that buffer space in the network stack and the window system should be allocated from a client-supplied resource pool. The second implies that the client should be able to reclaim (destroy) that storage on demand (thereby revoking access). The third implies that the client cannot be given control over fault-handling policy for the shared storage (which has a variety of structuring implications, none pleasant). In summary, there is a need to maintain trusted control flow in a condition of untrusted allocation.

We are not aware of any operating system other than EROS that can currently address these sumultaneous requirements, and it would not be a simple change to the POSIX environment to fix this issue. It is also an issue in every other microkernel we know about, because the same issue appears in disguise in any synchronous interprocess communication primitive that cannot separate these concerns. [15].

Each of these insights was strongly reinforced by our engineering experiences in the current project. Base on this, our conclusion is that capabilities and confinement provide unique support for asymmetric trust.

6 Conclusion

The CCCP outcome is a success. We have demonstrated a browser prototype that is able to successfully restrict hostile content using capability-based protection and confinement. More broadly, we have demonstrated that two key system components — the window system and the networking stack — can be reformulated to leverage capabilities and confinement to enforce isolation and provide defense in depth.

In the larger sense, the success of CCCP is more qualified. We were unable to quickly port conventional application software efficiently because of impedance matching problems between the EROS operating environment and the UNIX operating environment. In spite of this failure, we have established three useful outcomes in the effort:

- We have identified the impediments to low-cost porting success, and generated concrete designs for how to resolve them in the EROS successor.

- Based on previous work performed by Key Logic, Inc. [2], we know that a POSIX compatibility environment can be brought up with a bounded amount of effort and reasonable performance. It is now an engineering problem.

- We were able to confirm that a modest, relatively non-invasive “cocoon” wrapped around a complex and actively hostile application is sufficient to render it largely harmless, and that the emplacement of these cocoons can be entirely automated by a correctly designed user environment.

From the perspective of the EROS effort, CCCP also identified some aspects of the EROS system that needed to be eliminated — in particular support for transparent persistence. As our experience porting existing applications grew, we came to recognize that persistence was simply getting in the way, and would actually complicate matters at application level. Removing persistence would significantly simplify the EROS kernel. Between this and the interprocess communication changes that were identified during CCCP, we have stopped work on the EROS system in favor of its successor: Coyotos (http://www.coyotos.org). It appears likely that will be able to formally verify the Coyotos implementation.

In summary, further work is needed before it will be possible to do rapid deployment of general-purpose applications on a capability-based operating system. The results of CCCP show that capability-and confinement-based defenses are possible and efficient. Further engineering investment is required to turn this work into a directly deployable alternative to current commodity designs.

References

[1] G. Banga, P. Druschel, and J. Mogul. “Resource Containers: A New Facility for Resource Management in Server Systems.” Operating Systems Design and Implementation, pp. 45–58, 1999.

[2] A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. “The KeyKOS NanoKernel Architecture.” Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures, pp. 95–112, Apr 1992.

[3] U.S. Department of Defense, U.S. Department of Defense Trusted Computer System Evaluation Criteria, Document Number DoD 5200.28-STD, 1985.

[4] A. Dunkels. lwIP -a lightweight TCP/IP stack. Oct 2002. <tt>http://www.sics.se/˜adam/lwip/</tt>

[5] J. Epstein and M. Shugerman. “A Trusted X Window System Server for Trusted Mach.” Proceedings of the USENIX Mach Conference, Oct. 1990.

[6] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin and O. Shivers, “The Flux OSKit: A Substrate for Kernel and Language Research”, Proc. 16th ACM Symposium on Operating Systems Principles, pp 38–51, Oct. 1997

[7] Hardy, N.: The KeyKOS Architecture. Operating Systems Review 4(19), Oct. 1985, pp. 8–25.

[8] N. Hardy. “The Confused Deputy,” Operating Systems Review, 22(4), Oct. 1988.

[9] —: Common Criteria for Information Technology Security, International Standards Organization. International Standard ISO/IS 15408, Final Committee Draft, version 2.0, 1998

[10] Lampson, B. W.: A Note on the Confinement Problem. Comm. ACM. 16(10), 1973, pp. 613–615.

[11] V. Pai, P. Druschel, and W. Zwaenepoel. “IO-Lite: A Unified I/O Buffering and Caching System.” Proc. Third USENIX Symposium on Operating Systems Design and Implementation, pp. 22–35, Feb 1999.

[12] Sinha, A., Sarat, S, Shapiro, J. S.: Network Subsystems Reloaded. Proc. 2004 USENIX Annual Technical Conference. Dec. 2004

[13] Shapiro, J. S., Smith, J. M., Farber, D. J.: EROS, A Fast Capability System. Proc. 17th ACM Symposium on Operating Systems Principles. Dec 1999, pp. 170–185. Kiawah Island Resort, SC, USA.

[14] Shapiro, J. S., Weber, S.: Verifying the EROS Confinement Mechanism. Proc. 2000 IEEE Symposium on Security and Privacy. May 2000. pp. 166–176. Oakland, CA, USA

[15] J. S. Shapiro. “Vulnerabilities in Synchronous {IPC} Design,” Proc. 2003 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 2003

[16] Shapiro, J., Vanderburgh, J. Northup, E, Chizmadia, D: Design of the EROS Trusted Window System. Proc. 13th USENIX Security Symposium. 2004

[17] David Tennenhouse, Layered Multiplexing Considered Harmful. 2001.