Lecture Note
Verifying Security Invariants in ExpressOS

Chun-Kun ”Amos” Wang

1 Introduction

Modern mobile devices have put a wealth of information and ever-increasing opportunities for social interaction at the fingertips of users. At the center of this revolution are smart phones and tablet computers, which give people a nearly constant connection to the Internet. Applications running on these devices provide users with a wide range of functionality, but vulnerabilities pose a real threat to the security and privacy of modern mobile systems. Formalizing UNIX implementations and verifying microkernel abstractions require a large verification effort (the seL4 paper claims that it took 20 man-years to build and prove) and detailed knowledge of low-level theorem-proving expertise.

2 ExpressOS

ExpressOS is the first OS architecture that provides verifiable, high-level abstractions for building mobile applications. ExpressOS shows that verifying security invariants is feasible with the help of new programming languages and sparse source code annotations.

Figure 2.1 shows the architecture of ExpressOS including kernel, system services, and abstractions for applications. Android applications run directly on top of the ExpressOS kernel. Boxes that are left to the vertical dotted line represent the system services in ExpressOS. Shaded regions show the trusted computing base (TCB) of the system. The ExpressOS kernel is responsible for managing the underlying hardware resources and providing abstractions to applications running above. The ExpressOS kernel uses L4 to access the underlying hardware. L4 provides abstractions for various hardware-related activities, such as context switching, address space manipulation, and inter-process communication (IPC).

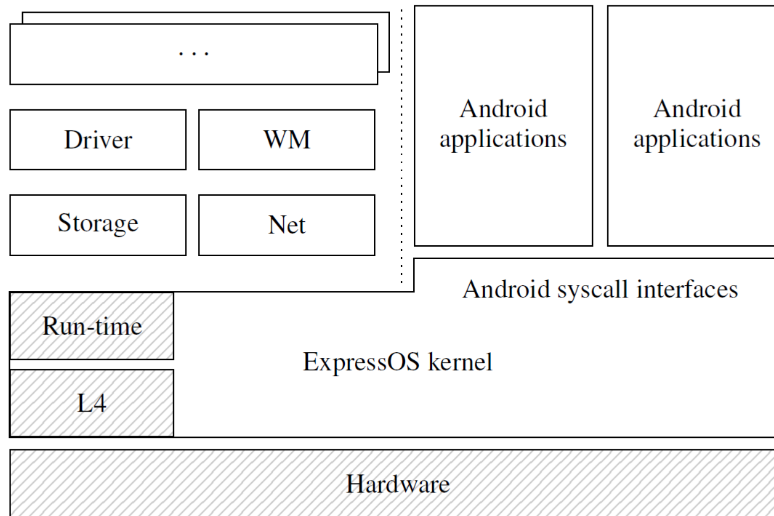

Figure 2.1: Overall architecture of ExpressOS

ExpressOS separates subsystems as system services running on top of the ExpressOS kernel. These services include persistent storage, device drivers, networking protocols, and a window manager for displaying application GUIs and handling input. ExpressOS reuses the existing implementation from L4Android to implement these services. All these services are untrusted components in ExpressOS. They are isolated from the ExpressOS kernel.

2.1 Design principles

In order to meet the overall goal of ExpressOS, three design principles guide our design:

- *Provide high-level, compatible, and verifiable abstractions.* ExpressOS should provide high-level, compatible abstractions (e.g., files) rather than low-level abstractions (e.g., disk blocks), so that existing applications can run on top of ExpressOS, and developers can express their security policies through familiar abstractions.
- *Reuse existing components.* Some of the components might have vulnerabilities. ExpressOS isolates these vulnerabilities to ensure they will not affect the security of the full system.
- *Minimize verification effort.* The verification of ExpressOS focuses only on security invariants.

ExpressOS uses four main techniques to simplify verification effort. By using these techniques, ExpressOS isolates large components of the kernel while still being able to prove security properties about the abstractions they operate on.

1. ExpressOS pushes functionality into microkernel services, just like traditional microkernels, reducing the amount of code that needs to be verified.

2. It deploys end-to-end mechanisms in the kernel to defend against compromised services. For example, the ExpressOS kernel encrypts all data before sending it to the file system service.

3. ExpressOS relies on programming language type-safety to isolate the control and data flows within the kernel.

4. ExpressOS makes minor changes to the IPC system-call interface to expose explicitly IPC security policy data to the kernel.

2.2 Implementation

The implementation of ExpressOS consists of two parts: the ExpressOS kernel and ExpressOS services. The ExpressOS kernel is a single-thread, event-driven kernel built on top of L4::Fiasco. We have implemented the kernel in C# and Dafny, which is compiled to native X86 code using a static compiler.

The implementation of ExpressOS kernel includes processes, threads, synchronization, memory management (e.g. mmap()), secure storage, and secure IPC. The kernel also implements a subset of Linux system calls to support Android applications like the Android Web browser.

1. *Dispatching a system call to ExpressOS services.* The ExpressOS kernel forwards system calls with IPC calls to L4Android. Figure 2.2 shows the workflow of handling the socket() system call in the ExpressOS kernel. When (1) the application issues a socket() system call to the ExpressOS kernel, (2) the kernel wraps it as an IPC call to the L4Linux kernel. The L4Linux kernel executes the system call (which might involve a user-level helper like the step (3) & (4)), and (5) returns the result back to the ExpressOS kernel. The ExpressOS kernel (6) interprets the result and returns it to the application. It is important for the ExpressOS kernel to maintain proper mappings between the file descriptors (fd) of the user-level helper and those of the application. It maps between the fd f and f0 so that subsequent calls like send() and recv() can be handled correctly.

2. *Bridging AndroidâĂŹs binder IPC.* Android applications communicate to Android system services (e.g., the window manager) through the AndroidâĂŹs binder IPC interface. The ExpressOS kernel extends the mechanism in Figure 2.2 to bridge the binder IPC. The user-level helper in Figure 2.2 acts as a proxy between the Android application and Android system services. Both the user-level helper and the ExpressOS kernel transparently rewrite the IPC messages to support advanced features like exchanging file descriptors.

3. *Supporting shared memory.* The ExpressOS kernel maps all physical memory of L4Linux into its virtual address space. It maps the corresponding pages to the address space of the

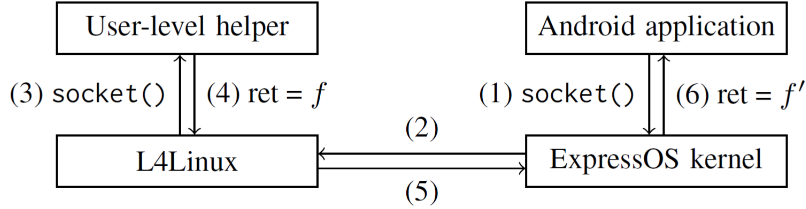

Figure 2.2: Work flow of handling the socket() system call in ExpressOS.

application when sharing occurs. We have modified L4Linux to expose its page allocation tables so that the ExpressOS kernel is able to compute the address.

3 Formal methods background

In this paper we focus on proving safety properties, which models properties that fail in finite time (availability is not a safety specification, typically). Complex specifications, such as the security specifications we prove in our code, cannot always be stated in terms of assertions on the program state that is in scope at the point of assertion. This leads us to use ghost code in the verification. Ghost code is code added to the original program to aid formal reasoning during verification but is not needed at run-time. Ghost code changes only the ghost states associated with ghost variables, which are annotated with the keyword ghost. The ghost state can never change the semantics of the original code (for instance, a conditional on a ghost variable followed by an update of a real program variable is disallowed, syntactically). Furthermore, ghost code must always be provably terminating.

4 Security invariants

The proofs focus on seven security invariants covering secure storage, memory isolation, user interface (UI) isolation, and secure IPC.

4.1 Secure storage

SI 1. *An application can access a file only if it has appropriate permissions. The permissions cannot be tampered with by the storage service.*

Property 1 (Integrity-Metadata). *The signature of a fileâĂŹs metadata is always checked before an application can perform any operations on it.*

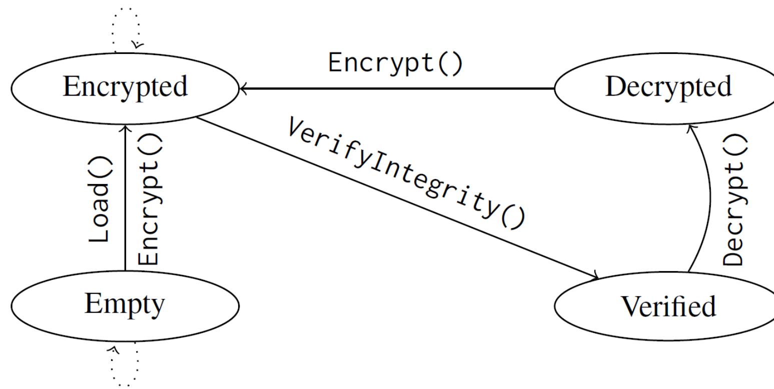

Figure 4.1: State transition diagram for the Page object.

Property 2 (Access Control). *An application can only access a file when it has appropriate permissions.*

SI 2. *Only the application itself can access its private data. Neither other applications nor system services can access or tamper with the data.*

Property 3 (Confidentiality). *Every page is encrypted before it is sent to the storage service.*

Property 4 (Integrity). *Each page loaded from the storage service has the appropriate integrity signature.*

Figure 4.1 shows the state transition diagram for a Page object. Ellipses represent the states. Solid and dotted arrows represent the successful and failed transitions. A page can be in the state of Empty, Verified, Decrypted, and Encrypted, meaning that (1) the page is empty, (2) its integrity has been verified, (3) its contents have been decrypted, and (4) its contents have been encrypted. To verify these properties, we specify the valid state transitions as the pre- and post-conditions of the relevant functions. For example, the specifications of Decrypt() state that the page should has its integrity verified before entering the function, and its contents are decrypted afterwards.

4.2 Memory isolation

SI 3. *If a memory page of an application is backed by a file, the pager can map it in if and only if the application has proper access to the file.*

SI 4. *An application cannot access the memory of other applications, unless they explicitly share the memory.*

Location	Example	Num.	Prevented
The core of the kernel	Logic errors in the futex implementation allow local users to gain privileges.	9	9 (100%)
Libraries of applications	Buffer overflow in libpng 1.5.x allows attackers to execute arbitrary code.	102	102 (100%)
System services	Missing checks in the vold daemon allows local users to execute arbitrary code.	240	226 (93%)
Sensitive applications	The BoA application stores a security question's answer in clear text which allows attackers to obtain the sensitive information.	32	27 (84%)
Total		383	364 (95%)

Figure 5.1: Categorization on 383 relevant vulnerabilities listed in CVE. It shows the number of vulnerabilities that ExpressOS prevents.

Property 5. *When the page fault handler serves a file-backed page for a process, the file has to be opened by the same process.*

Property 6 (Freshness). *The page fault handler maps in a fresh memory page when the page fault happens in non-shared memory.*

4.3 UI ISOLATION

SI 5. *There is at most one currently active (i.e., foreground) application in ExpressOS. An application can write to the screen buffer only if it is currently active.*

4.4 SECURE IPC

SI 6. *An application can only connect to secure IPC (SIPC) channels when it has appropriate permissions.*

SI 7. *An SIPC message will be sent only to its desired target.*

5 EVALUATION

To evaluate to what extent that the ExpressOS architecture can prevent attacks, we examined 383 relevant real-world vulnerabilities to analyze the security of the system. Figure 5.1 summarizes our analysis of 383 vulnerabilities. ExpressOS is able to prevent 364 (95%) of them.