Verifying Security Invariants in ExpressOS

Haohui Mai Edgar Pek Hui Xue
Samuel T. King P. Madhusudan
University of Illinois at Urbana-Champaign
{mai4,pek1,huixue2,kingst,madhu}@illinois.edu

Abstract

Security for applications running on mobile devices is important. In this paper we present ExpressOS, a new OS for enabling high-assurance applications to run on commodity mobile devices securely. Our main contributions are a new OS architecture and our use of formal methods for proving key security invariants about our implementation. In our use of formal methods, we focus solely on proving that our OS implements our security invariants correctly, rather than striving for full functional correctness, requiring significantly less verification effort while still proving the security relevant aspects of our system.

We built ExpressOS, analyzed its security, and tested its performance. Our evaluation shows that the performance of ExpressOS is comparable to an Android-based system. In one test, we ran the same web browser on ExpressOS and on an Android-based system, and found that ExpressOS adds 16% overhead on average to the page load latency time for nine popular web sites.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection; D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Security, Verification

Keywords mobile security; microkernel; programming by contracts; automatic theorem proving

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 and exploits in their software stacks pose a real threat to the security and privacy of modern mobile systems [7, 18].

Current work on secure operating systems has focused on formalizing UNIX implementations [14, 36] and on verifying microkernel abstractions [17, 20]. Although these projects approach or achieve full functional correctness, they require a large verification effort (the seL4 paper claims that it took 20 man-years to build and prove [20]) and detailed knowledge of low-level theorem-proving expertise.

This paper presents ExpressOS, a new OS we designed to provide high assurance security mechanisms for application security policies, and a verified implementation of the mechanisms that ExpressOS uses to enforce security policies. Our design includes an OS architecture for coping with legacy hardware safely, a programming language and run-time system for building our operating system, and a set of proofs on our kernel implementation that provide high assurance that our system can uphold the security invariants we define.

A key novelty of our verification approach is to focus on a set of security invariants of the code *without attempting full-blown functional correctness*. For example, we have verified that the private storage areas for different applications are isolated in ExpressOS. We *do not* derive this security invariant by showing the every aspect of all involved components, like the file systems and the device drivers, is correct. Instead, ExpressOS isolates the above components as untrusted system services. The proofs show that ExpressOS encrypts all private data of applications before sending it out to the system services, and ExpressOS places security checks correctly so that only the application itself can access its private data.

The proofs focus on seven security invariants covering secure storage, memory isolation, user interface (UI) isolation, and secure IPC. By proving these invariants, ExpressOS enables sensitive applications to provably isolate their state (mostly ensuring integrity of state; confidentiality is also ensured to a certain degree, but side-channel attacks are not provably prevented), and run-time events from malicious applications running on the same device.

We achieve the proofs of these invariants by annotating source code with formal specifications written using mathematical abstractions of the properties, using code contracts [23] and BOOGIE-based tools [8, 22], writing ghost-code annotations that track and update these mathematical abstractions according to the code's progress, and by discharging verification using either abstract interpretation or automatic theorem provers (mainly SMT solvers [12]). A thorough verification of the invariants above requires only a modest annotation effort ($\sim 2.8\%$ annotation overhead).

To evaluate the security and the performance of ExpressOS, we have built a prototype system that runs on x86 hardware and exports a subset of the Android/Linux system call interfaces. To evaluate security, we have examined 383 recent vulnerabilities listed in CVE [9]. The ExpressOS architecture successfully prevents 364 of them. To evaluate performance, we have used an ASUS Eee PC and run a web browser on top of ExpressOS. Our experiments show that the performance of ExpressOS is comparable to Android: ExpressOS shows 16% overhead for the web browsing benchmark.

Our contributions are:

- ExpressOS is the first OS architecture that provides verifiable, high-level abstractions for building mobile applications.

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

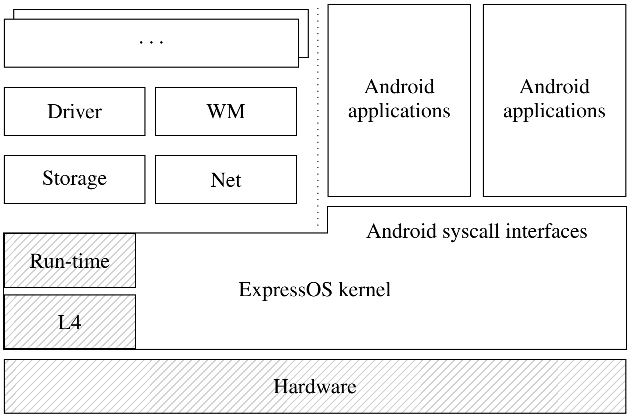

Figure 1: Overall architecture of ExpressOS. Android applications run directly on top of the ExpressOS kernel. Boxes that are left to the vertical dotted line represent the system services in ExpressOS. Shaded regions show the trusted computing base (TCB) of the system.

- ExpressOS shows that verifying security invariants is feasible with the help of new programming languages and sparse source code annotations.
- We have built ExpressOS and our experiments show that this type of OS can be practical and improve the security of software systems running on mobile devices.

2. ExpressOS overview

The primary goal of ExpressOS is to be a practical, high assurance operating system. As such, ExpressOS should support diverse hardware and legacy applications. Also, security invariants of ExpressOS should be formally verified.

This section provides an overview to the architecture, the verification approach, and system components of ExpressOS. Section 5 discusses the security implications of our architecture and the detailed design of the ExpressOS kernel.

2.1 Architecture for verification

Figure 1 describes the architecture of ExpressOS. The architecture includes the ExpressOS kernel, system services, and abstractions for applications.

ExpressOS uses four main techniques to simplify verification effort. First, ExpressOS pushes functionality into microkernel services, just like traditional microkernels, reducing the amount of code that needs to be verified. Second, 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. Third, ExpressOS relies on programming language type-safety to isolate the control and data flows within the kernel. Fourth, ExpressOS makes minor changes to the IPC system-call interface to expose explicitly IPC security policy data to the kernel. 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.

Current techniques to isolate components and manage the complexity of a kernel include software-fault isolation [6, 30, 35], isolating application state through virtualization [2, 10, 37], and microkernel architectures [14, 17, 20, 36]. These techniques alone, however, are insufficient for verifying the implementation of a system's security policies – the correctness of the security policies still relies on the correctness of individual components. For example, an isolated but compromised file system might store private data with world-readable permissions, compromising the confidentiality of the data.

2.2 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. More importantly, the security of these abstractions should be formally verifiable to support secure applications.
- *Reuse existing components*. ExpressOS should be able to reuse existing components to reduce the engineering effort to support production environments. 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*. Fully verifying a practical system requires significant effort, thus the verification of ExpressOS focuses *only on security invariants*. Also, this principle enables combining lightweight techniques like code contracts with heavywieght techniques like Dafny annotations [22] to further reduce verification effort.

2.3 Verification approach

Our modular design limits the scope of verification down to the ExpressOS kernel. The ExpressOS kernel is implemented in C# and Dafny [22], both of which are type-safe languages. Dafny is a research programming language from Microsoft Research that researchers use to teach people about formal methods. Like SPIN [4] and Singularity [17], type-safety ensures that the ExpressOS kernel is free of memory errors, and provides fine-grain isolation between components within the kernel. A static compiler compiles both C# and Dafny programs to object code for native execution.

We verify security invariants in ExpressOS with both code contracts and Dafny annotations. Code contracts are verified using abstract interpretation techniques, which have low annotation overhead but are unable to reason about complicated properties, like properties about linked lists. In contrast, Dafny annotations are verified using logical constraint solvers, which are capable of handling complicated properties, but require a heavy annotation burden and deep expertise in formal methods to use. Based on their charactertics, we verified simpler security invariants using code contracts, and more complex ones (like manipulation of linked lists) in Dafny, where we use ghost variables (i.e., variables that aid verification) to connect the proofs of both techniques. The combination enables high productivity for verification using code contracts, yet still retaining the the full expressive power of Dafny to verify complicated properties needed to prove security invariants.

We verify security invariants that involve asynchronous execution contexts by reducing them into the object invariants of relevant data structures (Please see Section 4 for more about object invariants). Verification in asynchronous execution contexts is challenging due to the separation of the control and data flows. Verifying only the security invariants, however, has a simple solution. In particular, we express security invariants and the object invariants of relevant data structures in terms of ghost variables, and the object invariants imply the original security invariants. The main advantage is that these object invariants can be reasoned about locally; therefore it is easier to verify them rather than reason about asynchronous execution contexts.

2.4 The ExpressOS kernel

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). L4 resides in the TCB of ExpressOS, although a variant of L4, seL4 [20], has been fully formally verified.

2.5 System services

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 [21] to implement these services.

All these services are untrusted components in ExpressOS. They are isolated from the ExpressOS kernel. The isolation combined with other techniques (discussed in Section 5) ensures that the verified security invariants remain valid even if a system service is compromised.

2.6 Application abstractions

The ExpressOS kernel exports a subset of the Android system call interfaces directly to applications. These abstractions include processes, files, sockets, and IPC. ExpressOS supports unmodified Android applications like a Web Browser to run on top of it directly.

The ExpressOS kernel provides an alternative set of IPC interfaces that is more amenable to formal verification, but still enables applications to perform Android-like IPC operations. First, it exposes all IPC interfaces as system calls rather than using `ioctl` calls. Second, it requires all IPC channels to be annotated with permissions, so that the ExpressOS kernel can perform access control on IPC operations. This design choice enables us to verify IPC operations in ExpressOS.

3. Threat Model

An untrusted Android application runs directly on ExpressOS. Such an application contains arbitrary code and data, along with a manifest listing all its required permissions. The user grants all permissions listed on the manifest to the application once he or she agrees to install it into the system. ExpressOS must be able to confirm that all activities of the application conform to its permissions, and all security invariants defined by ExpressOS (discussed in Section 5) must hold during the lifetime of the system.

ExpressOS should be able to isolate multiple applications. An application must not be able to compromise any security invariants of other applications.

The TCB of ExpressOS includes the hardware, the L4 microkernel, the compilers, and the language run-time. All system services in ExpressOS, however, are untrusted. ExpressOS should be able to maintain its security invariants for all applications even if the system services execute arbitrary code.

This paper focuses only on confidentiality and integrity; availability is out of the scope of this paper.

4. Formal methods background

In this section, we describe a few basic verification related concepts for readers who are not familiar with formal techniques. Experienced readers could skip this section.

The idea of verification is to use a verifier to reason about the implementation of a program: whether its behavior matches programmer's intentions, which are formalized and referred to as the *specification*.

The Floyd-Hoare style approach to verification achieves modular verification by annotating each function/method with pre- and post-conditions, and annotating the rest of the code with appropriate assertions that capture the specification. In this paper we focus on proving *safety properties*, which models properties that fail in finite time (availability is not a safety specification, typically). In particular, a *pre-condition* (*post-condition*) specifies certain assertions that have to hold before (after) executing a function. *Object invariants* refer to assertions that have to hold both before and after executing any methods in the object. Figure 4 shows an example of pre- and post-conditions used in ExpressOS.

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.

Ghost variables can range over mathematical types that are not supported in the programming language, including abstract sets, lists, etc., and can be used for abstracting the data contained in the program using mathematical objects, and thus encode specifications naturally (for example, a ghost sequence variable can track the sequence of keys stored in a linked list and we can then assert that this sequence is sorted). Apart from the ghost code for mathematical abstractions of the specification, the code is also annotated with additional information to prove a property of a correct program (typical examples include updating ghost states to reflect how the code meets the specification, inductive invariants on loops to facilitate automated reasoning for recursion, etc.). One good example of ghost variables in use is the `CurrentState` variable shown in Figure 4, which help maintain information during verification about the current state of the `Page` object.

The verification system (ours are based mostly on the suite of BOOGIE-based verification tools, including Dafny) proceeds to verify the code by taking straight-line fragments flanked by pre- and post-annotations, and derives verification conditions, which capture the semantics of the program, and are purely logical statements whose validity implies correctness of the code segments. These logical statements are, thanks to ghost code, often expressible in quantifier-free first-order theories combining arithmetic, arrays, recursive data-structures, sets, sequences, etc., and can be discharged using automatic constraint solvers, especially the emerging powerful class of SMT solvers.

For some of the simpler specifications, a completely automatic analysis based on *abstract interpretation* is feasible. We use the code contracts framework to achieve this in ExpressOS.

5. Proving security invariants

This section describes that how we apply the techniques described in Section 2 to verify security invariants on the implementation of storage, memory management, user interface, and IPC in ExpressOS.

5.1 Secure storage

The storage system of ExpressOS provides guarantees of access control, confidentiality, and integrity for applications, yet still offers the same sets of storage APIs as Linux. The storage system is implemented as an additional layer on top of the basic storage APIs (e.g., `open()`, `read()`, and `write()`), which are provided by the untrusted storage service.

```
static SecureFSInode Create(Thread current,
                            ByteBufferRef metadata,
                            ...)
{
  ...
  var ret = InitializeAndVerifyMetadata(...);
  ...

  var metadata_verified = ret == 0;
  ...

  var access_permission_checked
    = SecurityManager.CanAccessFile(current, metadata);
  ...

  // Verify both Property 1 and Property 2
  Contract.Assert(metadata_verified
                  && access_permission_checked);

  return ...;
}
```

Figure 2: Relevant code snippets in C# for Property 1 and Property 2.

The first security invariant for secure storage that the ExpressOS kernel enforces is:

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

An application can implement its security policy directly on top of SI 1. For example, an application might restrict the permission of a file so that only the application itself has access to it.

Since other compromised components such as the storage service and device drivers can affect SI 1, the first step of verifying SI 1 is to isolate the effects of these services. The ExpressOS kernel uses the HMAC algorithm [3] to achieve this goal. The ExpressOS kernel prepends several pages to all files managed by the secure storage system. These pages store metadata such as permissions, the size of the file, as well as an HMAC signature of these pages. The ExpressOS kernel checks the HMAC signature to ensure the integrity of the metadata when loading the file into the memory.

Now SI 1 can be reduced to the following two lower level properties:

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

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

Figure 2 shows relevant code of Property 1 and Property 2. The `Create()` function calls `InitializeAndVerifyMetadata()` to verify the HMAC signature of the metadata. Then it calls `SecurityManager.CanAccessFile()` to determine whether the current process has appropriate permissions to open the file. Finally, the annotation of `Contract.Assert()` instructs the verifier to prove that both the integrity of the metadata, and the access permission of the file have been checked.

The reduction is a trade-off between formal verification and practicality. We argue that pragmatically the conjunction of Property 1 and Property 2 implies SI 1. The reduction captures the important fact that ExpressOS misses no security checks in its access control logic, which is the main point of verifying ExpressOS. It does assume the implementation of relevant libraries like AES / SHA-1, and the one of `InitializeAndVerifyMetadata()` and `SecurityManager.CanAccessFile()` is correct. These components can be verified independently, and a verified implementation can be plugged into the system to further strengthen the proof.[1]

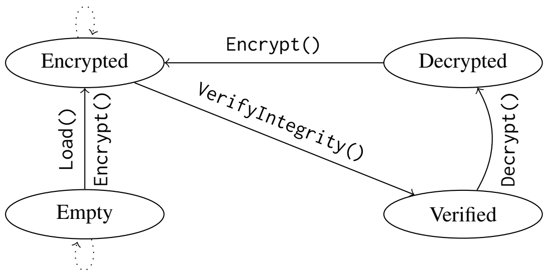

Figure 3: State transition diagram for the `Page` object. Ellipses represent the states. Solid and dotted arrows represent the successful and failed transitions.

The ExpressOS kernel also enforces integrity and confidentiality of its secure storage system:

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

Similar to SI 1, the ExpressOS kernel uses encryption to defend against compromised system services. From a high level, it partitions a file into multiple pages, and then it encrypts each page with AES for confidentiality. To ensure integrity, it signs each encrypted page with the HMAC algorithm, and packs the results to the metadata of the file. The overall implementation is similar to Cryptfs [39].

ExpressOS assigns each application a different private key during installation; therefore SI 2 can be reduced to the following two properties:

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.*

The idea behind verifying Property 3 and Property 4 is to use the ghost variable `CurrentState` to record the current state of the page. Figure 3 shows the state transition diagram for a `Page` object. 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.

Figure 4 shows the relevant code snippets. Notice that Property 3 can be specified as a pre-condition of the function `Flush()`: the function can only be called if the page is in the `Encrypted` state.

[1] The `InitializeAndVerifyMetadata()` function parses binary data read from the disk and verifies its integrity, which is easier implemented in C. To demonstrate the feasibility of this approach, we have implemented `InitializeAndVerifyMetadata()` in C, and verified its correctness with VCC [8].

```
class CachePage {
  enum State { Empty, Verified, Decrypted, Encrypted }
  [Ghost] State CurrentState;

  void Encrypt(...) {
    Contract.Requires(CurrentState == State.Empty
                   || CurrentState == State.Decrypted);
    Contract.Ensures(CurrentState == State.Encrypted);
    ...
  }

  void Decrypt(...) {
    Contract.Requires(CurrentState == State.Verified);
    Contract.Ensures(CurrentState == State.Decrypted);
    ...
  }

  // Verify the integrity of the page,
  // Returns true if the page is authentic.
  bool VerifyIntegrity(...) {
    Contract.Requires(CurrentState == State.Encrypted);
    Contract.Ensures(!Contract.Result<bool>()
                  || CurrentState == State.Verified);
    ...
  }

  // Load the content of the page from
  // the storage service.
  bool Load(...) {
    Contract.Requires(CurrentState == State.Empty);
    Contract.Ensures(!Contract.Result<bool>()
                  || CurrentState == State.Encrypted);
    ...
  }

  void Flush(...) {
    Contract.Requires(CurrentState == State.Encrypted);
    ...
  }
}
```

Figure 4: Relevant code snippets in C# for Property 3 and Property 4. `Contract.Requires()` and `Contract.Ensures()` specify the pre- and post-conditions of the functions.

```
static uint HandlePageFault(Process proc,
                            uint faultType,
                            Pointer faultAddress,
                            Pointer faultIP)
{
  Contract.Requires(proc.ObjectInvariant());

  ...
  AddressSpace space = proc.Space;
  var region = space.Regions.Find(faultAddress);

  if (region == null || (faultType & region.Access) == 0)
    return 0;

  ...
  var shared_memory_region = IsSharedRegion(region);
  var ghost_page_from_fresh_memory = false;

  if (shared_memory_region)
  {
    ....
  }
  else
  {
    page = Globals.PageAllocator.AllocPage();
    ghost_page_from_fresh_memory = true;
    ...

    if (region.File != null)
    {
      // Assertion of Property 5
      Contract.Assert(region.File.GhostOwner == proc);
      var r = region.File.Read(...);
      ...
    }
    ...
  }

  Contract.Assert(shared_memory_region
                ^ ghost_page_from_fresh_memory);
  ...
}
```

Figure 5: Code snippets for the page fault handler.

For compatibility reasons, ExpressOS does allow an application to create unencrypted, public readable files. ExpressOS, however, does not provide additional security invariants for these files.

5.2 Memory isolation

The ExpressOS kernel enforces proper access control and isolation for all memory of the applications:

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.*

The challenge of verifying SI 3 is that there is insufficient information available for verification at the point of assertions (i.e., in the pager). This is because the security checks are executed in different contexts, where both the control and data flows are separated in these two contexts.

ExpressOS addresses this challenge by connecting the information indirectly through the *object invariants* of relevant data structures. It strengthens these object invariants to contain information about the security checks, so that the object invariants can derive the desired security invariants.

Figure 5 and Figure 6 show the relevant implementation of the pagers. From a high level, the ExpressOS kernel organizes the virtual memory of a process with a series of `MemoryRegion` objects. A `MemoryRegion` represents a continuous region of the virtual memory, which has information on its access permissions and location. In addition, if the region is mapped to a file, it contains a reference to the mapped file (i.e., the `File` field in the `MemoryRegion` class). Since we have verified that the ExpressOS kernel properly checks access to files in Section 5.1, SI 3 can be reduced to the following property:

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.*

The key of verifying Property 5 is to use a ghost variable to record the *ownership* of the relevant objects, and to specify object invariants based on the ownership. For example, the first assertion in Figure 5 specifies Property 5, which gets verified through a series of object invariants.

First, the `AddressSpace` class represents the virtual address space of a process by a sequence of `MemoryRegion` objects. Intuitively the `AddressSpace` object "owns" all `MemoryRegion` objects

```
class MemoryRegion {
  ...
  var File: File;
  var GhostOwner: Process;

  function ObjectInvariant() : bool ... {
    File != null ==> File.GhostOwner == GhostOwner
    ...
  }
}

class Process {
  var space: AddressSpace;

  function ObjectInvariant() : bool ... {
    space != null && space.GhostOwner == this
    ...
  }
}

class AddressSpace {
  var GhostOwner: Process;
  var Head: MemoryRegion;

  // Ghost variable to record the sequence of
  // MemoryRegions owned by this AddressSpace
  ghost var Contents: seq<MemoryRegion>;

  function ObjectInvariant() : bool ... {
    forall x :: x in Contents ==>
      x != null && x.ObjectInvariant()
      && x.GhostOwner == GhostOwner;
    ...
  }

  method Find(address: Pointer) returns (ret: MemoryRegion)
  requires ObjectInvariant();
  ensures ObjectInvariant();
  ensures ret != null ==>
    ret.ObjectInvariant() && ret.GhostOwner == GhostOwner;

  method Insert(r: MemoryRegion)
  requires r != null && r.GhostOwner == GhostOwner;
  requires ObjectInvariant() && r.ObjectInvariant();
  ...
  ensures ObjectInvariant();
}

class File { ... var GhostOwner: Process; }
```

Figure 6: Reduced code snippets in Dafny for the MemoryRegion and the AddressSpace class.

in the sequence, which is specified in the object invariant of the AddressSpace object:

$$\forall \texttt{x},\ \texttt{x} \in \texttt{Contents} \rightarrow \quad \texttt{x} \neq \texttt{null} \wedge \texttt{x.ObjectInvariant()} \\ \wedge \texttt{x.GhostOwner == GhostOwner}$$

The Find() method looks up and returns the corresponding MemoryRegion object for the virtual address, therefore it only returns MemoryRegion objects that are owned by the AddressSpace object. Combining it with the object invariant above can lead to the post-condition of Find():

$$\texttt{ret != null} \rightarrow \quad \texttt{ret.ObjectInvariant()} \wedge \\ \texttt{ret.GhostOwner == GhostOwner}$$

At the first assertion in Figure 5, this is simplified down to:

$$\begin{aligned} \texttt{region.File.GhostOwner} \quad &== \quad \texttt{region.GhostOwner} \\ &== \quad \texttt{space.GhostOwner} \end{aligned}$$

The object invariant of the proc object ensures that

$$\texttt{proc.space.GhostOwner == proc}$$

This leads to the assertion of Property 5.

Property 5 is strictly weaker than the property that ensures full functional correctness. For example, Property 5 does not enforce that the file used in page fault handler has to be the exact same file that was requested by the user. The property, however, shows that the file must be opened by the same process. It maintains isolation since the access to the file has been properly checked. Moreover, it can be verified through object invariants.

We combine code contracts and Dafny to verify this property. Dafny verifies the MemoryRegion and AddressSpace class, because Dafny is able to reason about the linked lists in their implementation. The verification results from Dafny are expressed as properties of ghost variables (i.e., the GhostOwner fields). These properties are exported to code contracts as ground facts with the Contract.Assume() statements. Using Contarct.Assume() is a simple way let code contracts know about proofs about Dafny code. Additionally, we annotate the GhostOwner fields as read-only fields in C# to ensure soundness.

SI 4 can be expressed in a slightly different way to ease the verification.

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

The idea is to ensure that the page fault handler always allocates a fresh memory page, i.e., a memory page that is not overlapped with any allocated pages. ExpressOS adopted the verified memory allocator from seL4 [34] for this purpose. ExpressOS dedicates a fixed region to this allocator in order to implement this property. The reduction allows specifying this property as the second assertion in Figure 5.

5.3 UI isolation

The ExpressOS kernel enforces the following UI invariant:

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.*

To write to the screen, an application requests shared memory from the window manager, and writes screen contents onto the shared memory region. In ExpressOS, this can only be done through an explicit API so that the ExpressOS kernel knows exactly which memory region is the screen buffer.

The ExpressOS kernel enforces SI 5 by explicitly enabling and disabling the write access of screen buffers when changing the currently active application.

The object invariant of UIManager in Figure 7 specifies SI 5. The boolean ghost variable ScreenEnabled (which is not shown in the paper) denotes whether the application has write access to the screen. The initial value of ScreenEnabled is set to false for each application. Since it is the only API to manipulate the screen, the object invariant implies that only the current application has write access to the screen buffer.

5.4 Secure IPC

To simplify verification, ExpressOS provides an alternative, secure IPC (SIPC) interface over the Android's IPC interface. First, SIPC exposes all IPC functionality explicitly through system calls to eliminate the implementation and verification efforts on complex logic in ioctl() of Android's IPC. Second, the ExpressOS kernel enforces proper access controls for SIPC, compared to relying on

```
class UIManager
{
  Process ActiveProcess;

  [ContractInvariantMethod]
  void ObjectInvariantMethod() {
    Contract.Invariant(ActiveProcess == null
                       || ActiveProcess.ScreenEnabled);
  }

  void OnActiveProcessChanged(Process next) {
    Contract.Requires(next != null);
    Contract.Ensures(ActiveProcess == next);
    ...
  }

  void DisableScreen() {
    Contract.Ensures(ActiveProcess == null);
    Contract.Ensures(
      !Contract.OldValue(ActiveProcess).ScreenEnabled);
    ...
  }

  void EnableScreen(Process proc) {
    ...
    Contract.Ensures(ActiveProcess == proc);
    Contract.Ensures(ActiveProcess.ScreenEnabled);
    ...
  }
}
```

Figure 7: Relevant code snippets for SI 5. `[ContractInvariantMethod]` annotates the method that specifies the object invariants.

the receiver of Android's IPC for proper access control. This design moves the access control logic of SIPC into the ExpressOS kernel.

SIPC provides basic functionality to the applications, including creating SIPC channels, connecting to SIPC channels, and sending and receiving messages over the channel. Applications can still perform Android-like IPC operations using the SIPC interface.

The ExpressOS kernel enforces the following security invariants for SIPC:

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

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

SI 6 and SI 7 can be verified with similar approaches described above.

SI 6 is an access control invariant, thus the strategy of proving SI 6 is similar to the one of SI 1. We use a ghost variable to indicate whether the process has properly checked the permissions when opening a new SIPC channel.

We follow the proving strategy of Property 5 to verify SI 7. The idea is to create a `SIPCMessage` object for each IPC message, and to introduce a ghost variable `target` to record the target process of the message, which provides sufficient information to verify SI 7.

5.5 Verification experience

Overall, we found the verification effort in terms of annotations practical for the properties that were proven correct. While we developed the system and wrote code, we came up with the relevant security properties at the level of the module we were writing, and formalized it using appropriate annotations. Combining code contracts and Dafny reduced the code to annotation ratio down to about 2.8% (implementing the specification defined by the annotations). There were some instances where the code we wrote was not actually correct, and using the verification tools to prove the property led us to find the error. Equally importantly, formalizing the specification at the level of the code crystallized the vague properties we had in mind, and helped us write better code as well.

One lesson that we had in ExpressOS was to refine the shapes and aliasing information of the objects through redesigning data structures, and implementing ownership using ghost variables. Simpler shapes eased the verification. For example, we have reimplemented the the `MemoryRegion` object as a singly-linked list instead of a doubly-linked one, since verifying the manipulations of the linked list in the latter case requires specifying reachability predicates, which are difficult to reason about within SMT-based frameworks. The verification of heap structure properties in Dafny was achieved sometimes using further ghost annotations in the style of natural proofs [24, 27].

In simpler cases we used ownership to constrain the effect of aliasing. For example, the ghost field `GhostOwner` in the `AddressSpace` object specified which `Process` had created the object. The information was used in proving that each process creates its own `AddressSpace` object, effectively forbidding aliasing between `AddressSpace` objects of different processes.

One potential drawback we found during verification was that the specification using ghost code is sometimes too intimately interleaved with the implementation. Consequently, the specification gets strewn all across the code, and it is our responsibility that this actually is correct. Though the mathematical abstractions do help to some extent to distance the specification from the code, ghost updates to these abstractions are still intimately related. To illustrate this, consider a programmer implementing the code `Encrypt()` in Figure 4, and consider the scenario where the actual encryption fails for some reason, and yet the programmer puts the page into the `Encrypted` state. The verifier will go through even though the implementation is incorrect with respect to what the developer wanted; the onus of writing the correct specification is on the developer.

While we did not encounter any case where we noticed we made errors in inadvertently formulating too weak a specification, we did spend time double-checking that our *specifications* were indeed correct. We think an alternate mechanism for writing specifications that are a bit more independent from the code, resilient to code changes, and yet facilitates automated proving would make the developer's work more robust and productive.

6. Implementing ExpressOS

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.

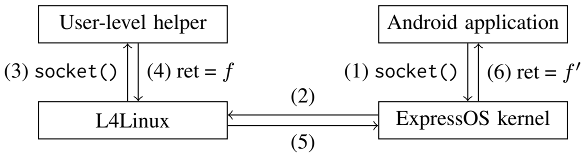

Figure 8: Work flow of handling the `socket()` system call in ExpressOS. The arrows represent the sequences of the work flow.

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. The kernel contains about 15K lines of code. The source code is available at `https://github.com/ExpressOS`.

The ExpressOS kernel delegates the implementation of system calls to ExpressOS services whenever it does not affect the soundness of the verification. These services include file systems, networks, device drivers, as well as Android's user-level services like window manager and service manager. ExpressOS reuses L4Android to implement these services. L4Android is a port of Android to a Linux kernel that runs on top of L4::Fiasco (i.e., L4Linux).

The rest of the section describes (i) how to dispatch a system call to ExpressOS services, (ii) how to bridge Android's binder IPCs between ExpressOS and Android's system services, and (iii) how to support shared memory between an application and ExpressOS services.

Dispatching a system call to ExpressOS services. The ExpressOS kernel forwards system calls with IPC calls to L4Android. Figure 8 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. In Figure 8, it maps between the fd f and f' so that subsequent calls like `send()` and `recv()` can be handled correctly. This workflow mirrors the implementation of the Coda file system inside Linux [19].

Bridging Android's binder IPC. Android applications communicate to Android system services (e.g., the window manager) through the Android's binder IPC interface. The ExpressOS kernel extends the mechanism in Figure 8 to bridge the binder IPC. The user-level helper in Figure 8 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.

Supporting shared memory. To support shared memory between Android applications and Android services, 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 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. Both the ExpressOS kernel and L4Linux use the L4 Runtime Environment (L4Re) to facilitate this process.

7. Evaluation

This section describes our evaluation of ExpressOS. 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. Then, we present the performance measurements of ExpressOS.

7.1 Vulnerability study

To understand to what extent ExpressOS is able to withstand attacks, we studied 742 real-world vulnerabilities (from Jun, 2011 to Jun, 2012) listed in CVE. 383 out of 742 are valid vulnerabilities, and they affect different components used in Android. We manually examined each of them, and classified it into one of the four categories based on its location:

In the core of the kernel. The vulnerability exists in the core of the Linux kernel, which means that the same functionality is implemented in the ExpressOS kernel. The proofs of ExpressOS ensure that such a vulnerability cannot affect any security invariants discussed in Section 5. If the vulnerability is irrelevant to the security invariants, the language run-time ensures it cannot subvert the control flow and data flow integrity to circumvent the proofs.

In the libraries used by applications. The vulnerability exists in the libraries used by the applications, like the Adobe Flash Player and libpng. In the worst case, the attacker can gain full control of applications by exploiting the vulnerability. ExpressOS ensures that the compromised application must adhere to its permissions, which prevents it from accessing other applications' private data, effectively protecting sensitive applications from compromised applications.

In system services. The vulnerability exists in the system services of ExpressOS, including the file system, the networking stack, device drivers, and Android user-level services. ExpressOS combines three techniques to contain the vulnerability.

First, ExpressOS uses end-to-end security mechanisms and the protections provided by the ExpressOS kernel to protect file system and network data. For example, both the confidentiality and integrity of any private data remain intact when the storage service is compromised, because SI 2 ensures that the attacker cannot access or tamper with private data. Similarly, the attacker cannot eavesdrop or tamper with any TLS/SSL/HTTPS connections, even if he or she compromises the networking service.

Second, ExpressOS isolates applications and system services, and restricts updates to the screen (i.e., SI 5) to contain compromises of the window manager service. An attacker with a compromised window manager takes full control of the physical screen. Successful attacks, however, still require information about UI widgets in the targeted application, and the ability to provide timely visual feedback to the user. For example, the attacker might steal the user's input by overlaying a malicious application running in background on top of the targeted application. The isolation mechanism prevents the window manager from accessing the memory of the targeted application to retrieve the exact locations of UI widgets, and SI 5 prevents the malicious application running in background updating the screen to timely react to the user's input.

Third, the L4 layer isolates the system services and the ExpressOS kernel. An attacker can potentially compromise the L4Android kernel with a vulnerability. However, the ExpressOS kernel remains intact because the L4 layer manages allocation of physical memory and the IOMMU, ensuring that the ExpressOS kernel's memory is isolated from all system services.

ExpressOS currently does not prevent compromises where a service acts as a privileged deputy that allows the attacker to use its permissions to attack the system. For example, "the Bluetooth service in Android 2.3 before 2.3.6 allows remote attackers within Bluetooth range to obtain contact data via an AT phone book transfer." (CVE-2011-4276), and "the HTC IQRD service for Android ... does not restrict localhost access to TCP port 2479, which allows remote attackers to send SMS messages." (CVE-2012-2217).

In sensitive applications. If an application is exploited, there is not much that ExpressOS can do for that application. Although we proved our implementation of several security policies in ExpressOS, if an application configures its policy incorrectly or its application logic leads to a security compromise, there is little the ExpressOS kernel can do to protect it.

Figure 9 summarizes our analysis of 383 vulnerabilities. ExpressOS is able to prevent 364 (95%) of them.

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

A large portion of vulnerabilities are due to memory errors in application libraries or in the system services. The verified security invariants and the protection from the ExpressOS kernel protect the data of sensitive applications from being compromised.

Out of the 383 vulnerabilities, there are two vulnerabilities related to covert channels. For example, the Linux kernel before 3.1 allows local users to obtain sensitive I/O statistics to discover the length of another user's password (CVE-2011-2494). These types of vulnerabilities are beyond the scope of our verification efforts and something ExpressOS is unable to prevent.

7.2 Performance

We evaluate the performance of ExpressOS by measuring the execution time of a variety set of benchmarks. We compared the performance of benchmarks running on ExpressOS, unmodified L4-Android, and Android-x86. All experiments run on an ASUS Eee PC 1005HA with an Intel Atom N270 CPU running at 1.60 GHz, 1GB of DDR2 memory, and a Seagate ST9160314AS 5, 400 RPM, 160G hard drive. Our Eee PC connects to our campus network through its built-in Atheros AR8132 Fast Ethernet NIC.

Both ExpressOS and L4Android run the L4Linux 3.0.0 kernel. The Android-x86 runs on top of Linux 2.6.39. All three systems run the same Android 2.3.7 (Gingerbread) binaries in user spaces.

We evaluate the performance of the Android web browser on real network, and microbenchmarks evaluating different aspects of system performance, including IPC, the file system, the graphics subsystem, and the networking stack. All numbers reported in this section are the mean of five runs.

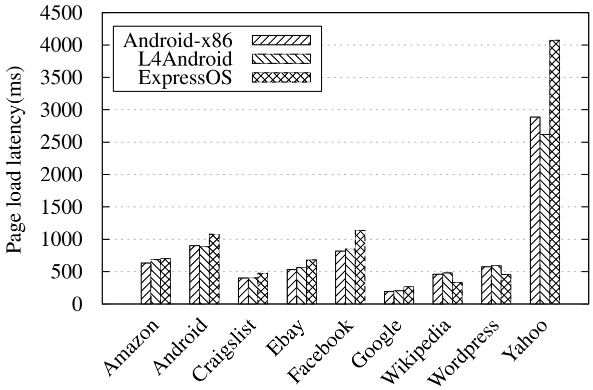

Figure 10: Page load latency in milliseconds for nine web sites for the same browser under Android-x86, L4Android, ExpressOS over a real network connection.

Page load latency in web browsing. We measure the page load latency for nine popular web sites to characterize the overall performance of ExpressOS compared to L4Android and Android-x86. The page load latency for a web site is the latency from initial URL request to the time when the browser fires the DOM `onload` event. The app clears all caches between each run.

Figure 10 shows the page load latency for nine web sites of all three systems. L4Android has 2% overhead on average, suggesting that the microkernel layer added by ExpressOS adds little overhead in real-world web browsing.

ExpressOS shows 14% and 16% overhead on average compared to L4Android and Android-x86.

IPC performance. We uses a simple ping-pong IPC benchmark to compare the performance of the SIPC mechanism in ExpressOS against the Android's Binder IPC mechanism. There are two entities (the server and the client) in this benchmark. For each round, the client sends a fixed-size IPC message to the server. The server receives the message and sends an IPC message back to the client which has the same content. Then the client receives the reply and continues to the next round.

We measured the total execution time for 10, 000 rounds of this benchmark on Android-x86, L4Android and ExpressOS. We measured the performance with different message sizes, including four bytes, 1KB, 4KB, 8KB, and 16KB. Figure 11 describes the results of this benchmark. These numbers show that it is possible to implement a verifiable IPC mechanism in the ExpressOS architecture without sacrificing efficiency.

Other microbenchmarks. We further evaluate the performance of ExpressOS with three microbenchmarks, including (1) a SQLite benchmark (SQLite) which creates a new database, then inserts 25, 000 records in one transaction, and writes all data back to the disk. (2) A network benchmark (Netcat), which receives a 32M file from local network. (3) A graphics benchmark (Bootanim), showing a PNG image, and adding light effects with OpenGL, which is derived from the boot animation program from Android.

These microbenchmarks help to categorize different aspects of the ExpressOS's performance. First, the SQLite benchmark manipulates the heap heavily, thus it is used to evaluate the performance of memory subsystem. Second, the Netcat benchmark helps quantify the effects of microkernel servers, because ExpressOS delegates all networking operations to L4Android. Finally, the Bootanim benchmark helps quantify the cost of using user-level helpers. Manipulating the screen heavily relies on Android's Binder IPC and shared semaphores, both of which are forwarded back and forth between the ExpressOS kernel and the user-level helpers in L4Android.

Figure 12 describes the results of all three microbenchmarks above. For the SQLite benchmark, both ExpressOS and L4Android are about 10% slower than Android-x86. For the Netcat benchmark, Android-x86, L4Android, and ExpressOS perform almost the same. ExpressOS is about 10% slower than L4Android in the Bootanim benchmark, but surprisingly, Android-x86 performs significantly worse than both L4Android and ExpressOS. We suspect that it might be due to some subtle differences between the kernel of L4Android and Android-x86, since all three systems are using the same user-level binaries.

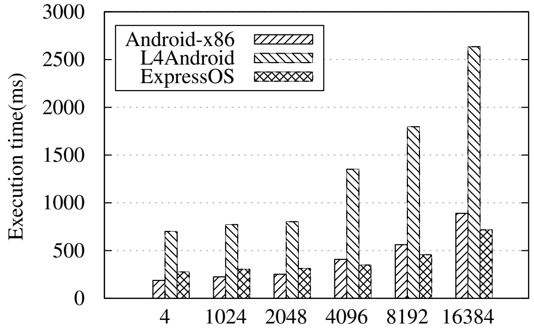

Figure 11: Performance of the ping-pong IPC benchmark of Android-x86, L4Android, and ExpressOS. X axis shows the size of IPC messages, Y axis shows the total execution time of running $10,000$ rounds of ping-pong IPC.

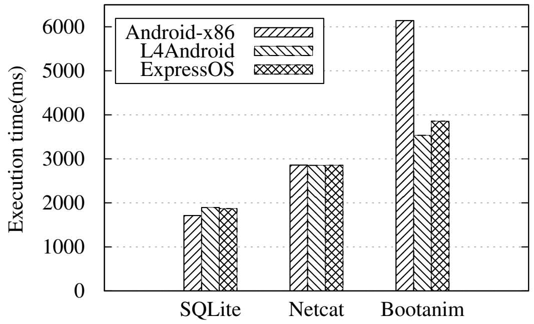

Figure 12: Performance results of the SQLite, Netcat and the Bootanim microbenchmark. X axis shows the type of the benchmark. Y axis shows their total execution time in milliseconds.

8. Related work

Attempts to eliminate defects of operating systems with full formal verification date back to the late 1970s. Dealing with all details of real-world operating systems has been a challenge for heavy-weight full formal verification methods. Verifying OSes that provide UNIX abstractions has been cumbersome (e.g., UCLA Secure Unix [36], PSOS [14], and KSOS [26] provide partially verified UNIX abstractions). Hypervisors and microkernels have lower-level abstractions that are more amenable for verification [5, 8, 16, 20, 28], but they provide lower-level abstractions such as IPC, interrupts and context switches, which are not immediately meaningful to applications.

The key difference between ExpressOS and the above work is that the verification of ExpressOS only focuses on security invariants rather than achieving full functional correctness. ExpressOS ensures that defects in unverified parts of the system cannot subvert the security invariants. As a result, ExpressOS provides high-level abstractions (e.g., files) with verified security invariants, and verifying these security invariants requires only $\sim 2.8\%$ annotation overhead.

Alternative approaches to improve security of operating systems include controlling information flows in operating systems [13, 40], separating application state through virtualization [2, 10, 37], intercepting security decisions in reference monitors [1, 29], and exposing browser abstractions at lowest software layer [32]. These techniques reduce the TCB dramatically down to the implementation of themselves. There are two differences between them and ExpressOS. First, ExpressOS does not require the applications to pervasively adopt new APIs, instead it provides Android/Linux system calls so that it can run legacy applications directly. Second, ExpressOS provides formally verified abstractions to the applications, where other techniques trust their implementation.

Implementing OS in safe languages has several benefits, such as avoiding memory errors, and isolating control and data flows in a finer granularity [4, 11, 15, 17, 25]. ExpressOS inherits these benefits, and further verifies that the security invariants in it always hold using code contracts and Dafny.

The current implementation of ExpressOS trusts the language run-time and the L4 microkernel. Verve [38] and seL4 [20] have verified the language run-time and the L4 microkernel. They are complementary to ExpressOS: ExpressOS can plug them in to further reduce the size of TCB. ExpressOS might also benefit from potential hardware support [31, 33].

9. Conclusions

This paper has presented ExpressOS, a new OS architecture that provides formally verified security invariants to mobile applications.

The verified security invariants covers various high-level abstractions, including secure storage, memory isolation, UI isolation, and secure IPC. By proving these invariants, ExpressOS provides a secure foundation for sensitive applications to isolate their state and run-time events from malicious applications running on the same device.

The verification effort on ExpressOS focuses on the most important properties from a system builder's perspective rather than full functional correctness. Also, ExpressOS combines several verification techniques to further reduce the verification effort. The approach is relatively lightweight and has about $\sim 2.8\%$ annotation overhead.

Our evaluation shows that ExpressOS is effective in preventing existing vulnerabilities from different attack surfaces. Besides its strong security guarantees, ExpressOS is a practical system with performance comparable to native Android.

Our experience suggests that the verification technique we pursue is mature enough to be broadly used by systems developers in order to obtain lightweight proofs of safety and security by focusing on a small but crucial subset of properties.

Acknowledgments

We thank Xi Wang, and our anonymous reviewers for their valuable feedback on this paper. We thank Rustan Leino for encouraging us to use Dafny for our verification tasks. We also thank Shuo Tang for implementing an earlier version of the system. This research was funded in part by AFOSR MURI grant FA9550-09-01-0539, ONR grant N000141210552, NSF grant CCF-1018182, and supported by Intel through the ISTC for Secure Computing.

References

[1] J. P. Anderson. Computer security technology planning study. Technical report, HQ Electronic Systems Division (AFSC), Oct. 1972. ESD-TR-73-51.

[2] J. Andrus, C. Dall, A. Van't Hof, O. Laadan, and J. Nieh. Cells: A virtual mobile smartphone architecture. In *Proceedings of the 23rd ACM Symposium on Operating System Principles*, Cascais, Portugal, Oct. 2011.

[3] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, Santa Barbara, CA, Aug. 1996.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, Dec. 1995.

[5] W. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15:1382–1396, 1989.

[6] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, Big Sky, MT, Oct. 2009.

[7] B. X. Chen and N. Bilton. Et tu, google? android apps can also secretly copy photos. `http://bits.blogs.nytimes.com/2012/03/01/android-photos`.

[8] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, Munich, Germany, Aug. 2009.

[9] Common Vulnerabilities and Exposures (CVE). `http://cve.mitre.org`.

[10] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A safety-oriented platform for web applications. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.

[11] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: a safe execution environment for commodity operating systems. In *Proceedings of the 21st ACM Symposium on Operating System Principles*, Stevenson, WA, Oct. 2007.

[12] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, Budapest, Hungary, Mar. 2008.

[13] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, Canada, Oct. 2010.

[14] R. J. Feiertag and P. G. Neumann. The foundations of a provably secure operating system (PSOS). In *Proceedings of the 1979 National Computer Conference*, New York City, NY, June 1979.

[15] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX operating system. In *Proceedings of the USENIX 2002 Annual Technical Conference*, Monterey, CA, June 2002.

[16] M. Hohmuth and H. Tews. The VFiasco approach for a verified operating system. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, Glasgow, UK, July 2005.

[17] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, Apr. 2007.

[18] G. Keizer. Google pulls 22 more malicious android apps from market. `http://www.computerworld.com/s/article/9222595/Google_pulls_22_more_malicious_Android_apps_from_Market`.

[19] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, Feb. 1992.

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

[21] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4Android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile devices*, Chicago, IL, Oct. 2011.

[22] K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, Dakar, Senegal, Apr. 2010.

[23] F. Logozzo. Practical verification for the working programmer with codecontracts and abstract interpretation. In *Proceedings of the 12th Conference on Verification, Model Checking and Abstract Interpretation*, Austin, TX, Jan. 2011.

[24] P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive proofs for inductive tree data-structures. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Philadelphia, PA, Jan. 2012.

[25] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, Saint Malo, France, Oct. 1997.

[26] T. Perrine, J. Codd, and B. Hardy. An overview of the kernelized secure operating system (KSOS). In *Proceedings of the 7th DoD/NBS Computer Security Initiative Conference*, Gaithersburg, MD, Sept. 1984.

[27] X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. Unpublished manuscript, 2012.

[28] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, Charleston, SC, Dec. 1999.

[29] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, Oct. 2011.

[30] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct. 2003.

[31] J. Szefer and R. B. Lee. Architectural support for hypervisor-secure virtualization. In *Proceedings of 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, London, England, UK, Mar. 2012.

[32] S. Tang, H. Mai, and S. T. King. Trust and protection in the Illinois browser operating system. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, Canada, Oct. 2010.

[33] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *Proceedings of 38th International Symposium on Computer Architecture*, San Jose, CA, June 2011.

[34] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Nice, France, Jan. 2007.

[35] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM symposium on Operating Systems Principles*, Asheville, NC, Dec. 1993.

[36] B. J. Walker, R. A. Kemmerer, and G. J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, Feb. 1980.

[37] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[38] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, June 2010.

[39] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical report, Columbia University, June 1998. CUCS-021-98.

[40] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, Nov. 2006.