Formalizing Information Flow in a Haskell Hypervisor

Rebekah Leslie
Portland State University

Levent Erkök and Flemming Andersen
Intel Corporation

***Abstract*— Separation kernels are the holy grail of secure systems, remaining elusive despite years of research into their design, implementation, and analysis. Though separation kernel research has achieved many successes, the disconnect between information flow theory and system implementation is a significant barrier to further progress. In this paper, we show how a particular branch of information flow theory, noninterference, can be utilized to formulate correctness and security properties of a microkernel-style hypervisor. Thus, we not only provide a first step towards a formally verified separation kernel, but also reduce the gap between information flow theory and operating systems practice.**

I. Introduction

Noninterference provides a concise and general way to formalize the information flow relationships between components of a system. A noninterference security policy specifies which components, or domains, may not *interfere* with each other, where a domain *u* interferes with a domain *v* if *v* can observe the effects of *u*'s execution [1]. The generality of such policies makes them useful for capturing a wide variety of security requirements, including separation. A key benefit of noninterference is that there are existing frameworks for reasoning about systems governed by a noninterference policy [1], [2], thus reducing the barriers to verifying such systems formally.

Our interest in noninterference stems from our efforts to develop a hypervisor with formally verified separation between user-level processes. The design of our hypervisor is similar to secure microkernel APIs such as seL4 [3] and L4sec [4], but we use the term hypervisor to emphasize our intent to employ our system as a platform for secure, separate execution. Such an execution environment is an essential part of many high assurance systems, and is increasingly important in light of recent hardware developments, such as multi-core platforms based on Intel® Virtualization Technology and Intel® Trusted Execution Technology [5].

Following the approach of the Programatica [6] and seL4 [3] projects, we are writing a model of our hypervisor, called HHV, in the functional language Haskell [7]. The mathematical semantics and strong type system of Haskell make our model easier to reason about than a low-level implementation. By combining the use of a high-level functional language with the application of an existing reasoning framework, proving separation for HHV becomes a more tenable goal than with other techniques.

In this paper, we concentrate on the formulation of correctness and security properties of the communication mechanisms in HHV, because these are the source of all legal information flow in our design. We formally characterize the information flow relationships induced by these communication primitives using a notation based on higher-order logic, similar to the P-logic programming logic for Haskell [8]. More importantly, we develop specification patterns for expressing correctness properties in terms of noninterference concepts. In defining these patterns, we take advantage of higher-order functions to extract common aspects of noninterference-style properties. For instance, we abstract over the kernel state and kernel operations so that we can instantiate the patterns in different contexts and formulate assertions about particular operations in a generic way. Hence, our work provides a link between the theory of noninterference and the actual practice of building secure microkernels.

We organize the remainder of the paper as follows. Section II describes the foundational concepts of HHV and outlines the communication mechanisms. Section III presents the system model used by our hypervisor. Section IV introduces the noninterference specification patterns that are the basis of our property formulations. Section V defines the desired information flow behavior of HHV, both as an informal specification and as a set of formal properties. We discuss related work in Section VI and present our conclusions in Section VII.

II. Communication in HHV

The fundamental abstractions in HHV are *protection domains*—the basic unit of resource protection—and *execution contexts*—the unit of execution. A protection domain (PD) corresponds to an address space in other systems; an execution context (EC) corresponds to a thread. HHV is a migrating thread system [9], so there is a single execution context per processor that moves between protection domains with the logical flow of control.

Each EC contains a representation of the processor state—such as the general purpose registers and the instruction pointer—for the running domain. We store this hardware context using a record type, called `Context`, which contains a field for each hardware register. When a domain is not running, HHV preserves the processor state in a region of memory called the save/restore area (SRA).

Levent Erkök is now at Galois Connections, Inc.

Protection domains communicate through uni-directional channels called *portals*. A *portal traversal* causes a context switch from the initiator of the traversal (the source domain) to the target of the traversal (the destination domain), potentially transferring a message from the source to the destination in the process. There are two modes of communication in HHV: direct data transfer via registers and indirect data transfer through shared resources, such as memory pages.[1] In this paper, we focus on the information flow properties of direct data transfer, although we have also formally characterized the aspects of portal traversal that deal with resource sharing.

HHV uses a dynamically configurable set of message registers. This approach results in an extremely flexible communication mechanism, and also allows us to overcome the security issues induced by the use of a migrating-thread model [9], [10]. Specifically, we must guarantee that a portal traversal does not leak information that the source domain wishes to keep private and that a portal traversal does not overwrite data that the destination wishes to preserve. By allowing both the source and destination domains to control which registers are part of a message, we enable the domains to protect their state against unwanted observation and modification. To this end, we introduce the concept of *portal masks*, which are fine-grained guards used by the source and destination domains to control information flow.

A traversal potentially affects every register in the execution context, so a mask (represented by the type `Mask`) contains a flag for every register in the hardware context. There are two masks involved in each portal traversal: the source domain uses a *transfer mask* to control which registers are sent during a traversal and the destination uses a *pass mask* to control which registers are modified. HHV only copies a register value from the source to the destination if both masks permit the transfer.

We define a context switch primitive that encapsulates the direct data transfer aspect of portal traversal. The exact behavior of a switch depends on the portal mask settings, but the fundamental algorithm consists of three basic steps.

1) Filter the running EC to remove any data that the source does not wish to send to the destination, as specified in the transfer mask, and save this data to the source SRA.
2) Perform a context switch to the destination.
3) Load the saved state from the destination's SRA into the EC, as dictated by the pass mask settings.

We implement these steps using several state-manipulation functions, the type signatures for which are shown in Figure 1: `saveContext` writes the registers that are not part of the message to the source SRA, `computeTransfer` builds a new context containing only the message registers, `loadContext` restores the state of the destination from its SRA, and `computePass` combines the restored registers with those sent by the source domain. The top-level context switch operation, `switch`, weaves together these state-manipulation functions to achieve the data transfer and preservation behavior specified by the portal masks.

[1]Portal traversal is the only mechanism for sharing resources, hence we do not consider shared resources to be a separate source of information flow.

```
saveContext     :: Mask → Context → PDID → Kernel ()
computeTransfer :: Mask → Context → Context
loadContext     :: Portal → SRA → Context
computePass     :: Mask → Context → Context → Context
```

Fig. 1. Functions for preserving domain state and performing a message transfer. These functions are used by the context switch operation to implement the dynamic message-transfer semantics specified by the portal masks.

The implementation of `switch` depends on the kernel state, information about the source and destination domains, and the portal configuration (which includes the mask settings). These dependencies are reflected in the type signature of `switch`:

```
switch :: PDID → PDID → Portal → Kernel ()
```

The parameters are the source domain (of type `PDID`), the destination domain (of type `PDID`), and the portal configuration (of type `Portal`). The operation runs in a monad called `Kernel`, which encapsulates the kernel state. We will examine the details of this monad in the next section; for now it is only important to note that `Kernel` contains a state component for the running EC.

The first step of `switch` is to perform the source-side state-manipulations; this involves calling `saveContext` to store the registers that are not being sent and calling `computeTransfer` to clear the value of those registers in the running EC. The variable `ec` corresponds to the running EC (read from the kernel state by the `get` function).

```
switch src dst port
  = do ec ← get
       saveContext (transfer port) ec src
       let ec' = computeTransfer (transfer port) ec
```

At this point, `ec'` contains only the data that the source wants to send to the destination, so we perform a context switch to the destination by setting the active domain to be the destination. Next, we perform the destination-side state-manipulations. We invoke `loadContext` to restore the registers that the destination does not wish to receive from the source. The final register values are obtained by merging the sent context with the restored context using `computePass`.

```
setActiveDomain dst
dstSRA ← getSRA dst
let rc = loadContext port dstSRA
set (computePass (pass port) ec' rc)
```

We omit the details hidden by the sub-computations of `switch`, because they are not essential for formalizing the behavior of HHV. The important idea is that each of these steps should respect the intended information-flow semantics of portal traversal, which we define in Section V.

III. HHV System Model

The implementation of portal traversal—and other HHV operations—requires access to the running execution context and to the state of the protection domains. The domain state

resides in memory, so we introduce a simple virtual memory model consisting of a memory datatype, `VirtMem`, and basic operations on this type (such as read and write). In addition to the user state stored in the EC and memory, HHV keeps track of certain internal state, represented by the type `HHV`, that the kernel operations potentially modify. We incorporate these state components into our model using a layered monadic approach that is similar to the techniques previously applied in the construction of modular interpreters [11], [12].

Using monad transformers, we construct a monad with `EC`, `VirtMem`, and `HHV` state components (`ST s m` adds a state component of type `s` to monad `m`, resulting in a new monad). Every computation that runs in this monad will share a single EC, memory, and HHV structure; the operations available to computations in this monad include reading and writing to each of the state components.

```
type Kernel = ST EC (ST VirtMem (ST HHV Id))
```

We often reference values for the three state components of the `Kernel` monad together, so we introduce a new type, `KS`, to represent the kernel state.

```
data KS = KS { ec :: EC, mem :: VirtMem, hhv :: HHV }
```

Running a `Kernel` computation with an initial value for the kernel state allows us to examine the result of the computation and the new state that the computation produces. We are primarily concerned with the state effects of HHV operations, so we define a run function that returns the state produced by executing a kernel computation (throwing away the result of the computation).

```
runKs :: Kernel a → KS → KS
runKs k s
  = let (((_, c), m), h) = unId (k `runST` (ec s)
                                   `runST` (mem s)
                                   `runST` (hhv s))
    in KS c m h
```

The library function `runST` peels off a single layer from our monad transformer stack. Thus, we use three applications of `runST` in the definition of `runKs`, one for each state component of the `Kernel` monad. We will use `runKs` frequently in our property formulations to compare the impact that two distinct monadic computations have on the state.

IV. Noninterference

The properties we wish to formulate of the context switch operation all deal with the absence of information flow between domains. Noninterference [1] is a mathematical formalism for expressing exactly this kind of property. Noninterference captures the idea that a given operation is not allowed to change the state of a particular domain. For example, "`switch` does not change the state of any domain except the source and destination" can be thought of as a noninterference property.

By employing noninterference to formalize our system, we can take advantage of the existing mathematical frameworks for reasoning about noninterference security policies [1], [2], [13], [14]. Noninterference frameworks establish a correspondence between properties of individual operations, called unwinding conditions, and the information flow that occurs during the execution of a system. In this section, we define two generic noninterference-style properties, which we will later instantiate to capture specific information flow properties of communication in HHV. These properties are analogous to unwinding conditions, so noninterference frameworks provide us with confidence that the properties will also hold for sequences of HHV operations.

The property *NoStateEffect* (see Figure 2) captures the notion that some operation, `k`, does not modify the kernel state. However, we do not wish to say that the kernel state does not change at all, rather, we want to express that a certain component of the state does not change (such as a particular domain's SRA or the page-table memory). For this reason, *NoStateEffect* takes a function argument, `extract`, whose role is to extract a portion of the kernel state. This extraction function is polymorphic in its return type and can perform arbitrary (pure) computation. We say that an operation has *NoStateEffect* if the extraction function produces the same result in the initial state and in the state that results from running the operation.

Note that we do not expect *NoStateEffect* to be a general property of HHV, rather, we will use *NoStateEffect* to describe the effects of particular operations on particular components of the kernel state. As a simple example, consider the `setActiveDomain` operation, which changes the running domain. Performing this operation should not change the instruction pointer stored in the running EC (among other things). In this case, the extraction function simply projects the `eip` field from the `EC` component of the state: $(\lambda s \rightarrow eip\ (ec\ s))$.[2] Having determined the extraction function, we can formalize the assertion as: $\forall s, dom.\ NoStateEffect(s, \lambda s \rightarrow eip\ (ec\ s), setActiveDomain\ dom)$.

The function `readPage` is a more realistic example of the kind of extraction function we will use in our property formulations. Given a domain and a virtual-page number, `vi`, `readPage` looks up the corresponding frame number in the virtual address space of that domain.

```
readPage :: PDID → VirtIdx → KS → Page
readPage did vi s
  = let m = mem s
        (pi,_) = pageTables m did vi
    in lookupPage m pi
```

We define extraction functions for reading page-table entries (`readPageTable`) and for reading the save/restore area of a domain (`readSRA`) in a similar fashion.

Clearly, the polymorphic nature of *NoStateEffect* makes it a powerful tool for expressing a range of noninterference properties. We will use *NoStateEffect* to write many of the information flow properties of portal traversal, but in some cases we need something even more general. In particular, for successful portal traversals, we need to express that a computation may change the state, but only in a controlled

[2] In Haskell syntax, $\lambda x \rightarrow E$ defines an anonymous function, which will evaluate the expression E with the argument x.

```
NoStateEffect(s :: KS, extract :: KS → a, k :: Kernel b) =
  extract (runKs k s) === extract s

ControlledStateEffect(s :: KS, extract :: KS → a, k1 :: Kernel b, k2 :: Kernel b) =
  extract (runKs k1 s) === extract (runKs k2 s)
```

Fig. 2. Patterns for formulating noninterference properties of monadic Haskell programs. *NoStateEffect* describes kernel operations that do not modify a particular region of the kernel state. *ControlledStateEffect* describes kernel operations that may modify the kernel state, but only in a constrained manner. The first argument to both patterns corresponds to the kernel state. The argument called *extract* is a function that extracts a particular region of the kernel state from one or more of the kernel state components (such as the page-table memory). In the case of *NoStateEffect*, k is the kernel operation of interest. For *ControlledStateEffect*, $k1$ is the kernel operation and $k2$ is the reference computation. Note that the relation $===$ captures the denotational equivalence of two terms, rather than structural equality.

way. *ControlledStateEffect* is a generalization of *NoStateEffect* that captures this idea.

The definition of *ControlledStateEffect* (see Figure 2) is very similar to *NoStateEffect*, except that it compares the state produced by a given operation to the state produced by a reference computation. The intended use of this property is that the reference computation will perform only the allowed effect. Thus, if the resulting states are compatible, then the effects of the first computation must be limited to those effects that are explicitly allowed. The state-dependent nature of the reference computation makes *ControlledStateEffect* an instance of a dynamic noninterference property [2].

V. Properties of Portal Traversal

Portal masks provide fine-grained control over the data transfer that occurs during a portal traversal. The correct implementation of this control mechanism is the fundamental property of `switch`; there should not be any information flow that is not expressly permitted by the portal masks. We divide this high-level requirement into three categories:

- **Destination State Preservation:** The value of a field only passes from the source domain to the destination domain if both the transfer mask and the pass mask allow the transfer. Only the source domain can leak information to the destination.
- **Source State Preservation:** No information flows into the source domain. The SRA of the source only changes as specified by the transfer mask of the portal.
- **Memory Preservation:** Executing `switch` does not modify the memory pages or the page-table of any domain in the system. For every domain except the source and destination, the SRA pages do not change.

We will formulate these properties using the noninterference specification patterns defined in the previous section.

A. Destination state preservation

Portal masks are the principal mechanism afforded to domains for protecting their state during a portal traversal. If either the source or the destination wishes to protect a field, then HHV must ensure that the value of that field in the source EC does not leak to the destination EC. To formalize this property, we need a mechanism for identifying protected fields and a formal way to capture that a field value is not leaked. The predicate `noLeakage` (see Figure 3) determines if a particular field is protected; it is true if either mask blocks the transfer of that field.

We would like to use *NoStateEffect* to express the idea that the execution of `switch` does not affect the value of protected registers. Thus, we need to determine the value that a register should have when no leakage occurs. The function `loadField` performs this task, loading a single field value from a given SRA. *FieldNotLeaked*, defined in Figure 3, formalizes the desired register protection property by instantiating *NoStateEffect* with `loadField` as the extraction function and `switch` as the computation of interest.

The first argument to *FieldNotLeaked* is a function that projects a field from the context of the running EC. The second argument is the corresponding projection function for the `Mask` type (e.g., `bEip` returns the mask value that corresponds to the `eip` field of a context). Parameterizing *FieldNotLeaked* in this way allows us to write a single property that can be instantiated for each field of the context. Figure 3 shows an example of one such property, *EIPNotLeaked*, which specifies that HHV does not transfer the instruction pointer register unless both the source and destination indicate that the transfer should occur.

The field-preservation properties capture information flow into the destination through the EC, which is the only component of the destination state that may legally change during a traversal. We expect the remaining components of the destination state—the saved context and non-SRA memory—to be unaffected by the execution of `switch`. We define *DestSavedContextUnchanged* to express that the saved context does not change (Figure 3), we again use *NoStateEffect*. In this case, the extraction function is simply the SRA-projection function that returns the saved context (`savedContext`). We address the memory preservation properties in Section V-C.

B. Source state preservation

A portal traversal does not permit any information flow into the source domain. However, the source state is not entirely unaffected by the traversal, because HHV will potentially save register values to the source SRA. In other words, executing `switch` modifies the source SRA in the same way as `saveContext`. We use *ControlledStateEffect* to express this property, as shown in Figure 4; `readSRA` is the extraction function, `switch` is the computation of interest, and `saveContext` is the reference computation.

```
noLeakage :: Mask → Mask → (Mask → Bool) → Bool
noLeakage transfer pass p = not (p transfer && p pass)

FieldNotLeaked(f :: Context → a, p :: Mask → Bool, src :: PDID, dst :: PDID, port :: Portal) =
  ∀s. NoStateEffect(s, λs → loadField (pass port) f p (readSRA dst s), switch src dst port)

EIPNotLeaked =
  ∀src, dst, port. noLeakage (transfer port) (pass port) bEip ⟹ FieldNotLeaked(eip, bEip, src, dst, port)

DestSavedContextUnchanged =
  ∀s, src, dst, port. NoStateEffect(s, λs → savedContext (readSRA dst s), switch src dst port)
```

Fig. 3. Destination state preservation properties. *EIPNotLeaked* is an example of a field-preservation property built from the predicate *noLeakage* and the abstract property *FieldNotLeaked*. *DestSavedContextUnchanged* captures the property that a context switch does not change the destination's SRA. We quantify over the kernel state (*s*), the source and destination domains (*src* and *dst*), the portal being traversed (*port*).

```
NoFlowIntoSource =
  ∀s, src, dst, port. ControlledStateEffect(s, readSRA src, switch src dst port,
                                            saveContext (transfer port) (ec s) src)
```

Fig. 4. Source state preservation property: executing a context switch potentially modifies the saved context of the source domain, but all other components of the source state are unchanged. The quantified variables have the same meaning as in Figure 3.

The page-table and non-SRA memory pages of the source should also be preserved by `switch`, as we shall see in the next section.

C. Memory preservation

The only memory regions that a context switch will modify are the source and destination SRAs. We capture this notion using three properties: when `switch` executes, the page-table of every domain remains the same, the mapped memory of every domain remains the same, and the SRA memory of every domain except the source and destination remains the same. The definitions of these properties are presented in Figure 5.

Recall from Section IV that `readPageTable` projects the page-table component of the kernel state. We use *NoStateEffect* with `readPageTable` as the extraction function to express that `switch` does not modify the page-tables in the property *PageTablesUnchanged*.

Similarly, `readPage` projects a specified memory page from the kernel state. We quantify over all virtual-page numbers to express that none of the memory pages visible to user-level domains are affected by a context switch.

For the SRA memory pages, we again use *NoStateEffect*, but in this case the extraction function is `readSRA`. We quantify over domain identifiers, but exclude the source and destination because the SRA of these domains might in fact change.

VI. Related Work

OS verification has a long history filled with mixed successes, an overview of which is provided by Tuch, et al. [15]. More recent verification efforts can be divided into two categories: attempts to verify low-level source code written in C/C++ and attempts to verify functional language models/implementations. The VFiasco [16] and L4Verified [17] projects both pursue the former option in their effort to prove properties of L4 [18] implementations. The results of these projects are limited to proofs about sub-systems of L4, rather than top-level separation properties, due to the inherent complexities involved in reasoning about C and security issues in the L4 design. Furthermore, these projects do not employ a general information flow framework, which we consider to be an important contribution of our work.

The use of functional languages together with formal property specifications is a relatively recent development, but there is already a large body of work on the subject. Elphinstone, et al. are using Haskell models in the the development of seL4 [19], [20], a secure redesign of the L4 microkernel [18], and are already making progress towards verifying properties of this model. They have translated their model into Isabelle/HOL, thus guaranteeing termination, but have not yet formalized any separation properties.

The Programatica project formally verified a virtual memory model written in Haskell [21]. They have also specified an axiomatic semantics for an implementation of a safe, Haskell interface to hardware [22]. We view this work as highly compatible with ours because proving our separation properties will depend on guarantees about the behavior of the underlying hardware.

There is also existing work on the instantiation of noninterference frameworks for concrete systems. In his foundational noninterference paper [1], Rushby applied noninterference to a simple access control mechanism. Subsequently, Schellhorn et al. applied Rushby's work to prove security properties of their generic formal model of operating systems for multiapplicative smart cards [23]. Von Oheimb used noninterference to analyze the security of the Infineon SLE66 smart card processor [13]. The key difference between our work and these earlier efforts is that the complexities of general-purpose operating systems make them more difficult to integrate with a theoretical framework. Also, the access control and smart card work did not

```
PageTablesUnchanged =
  ∀ s, src, dst, port, did, vi. NoStateEffect(s, readPageTable did vi, switch src dst port)

UserMemoryUnchanged =
  ∀ s, src, dst, port, did, vi. NoStateEffect(s, readPage did vi, switch src dst port)

SRAMemoryUnchanged =
  ∀ s, src, dst, port, did. (did /= src) && (did /= dst) ⟹ NoStateEffect(s, readSRA did, switch src dst port)
```

Fig. 5. Properties describing the absence of information flow via memory during a context switch. As with the previous properties, we quantify over the kernel state (*s*), the domains involved in the portal traversal (*src* and *dst*), and the portal being traversed (*port*). We introduce two new quantified variables: *did* is the identifier of the domain whose memory we are interested in and *vi* is a particular page in that domain's virtual address space.

utilize the notion of dynamic noninterference [2], which is essential for reasoning about HHV.

VII. Conclusions and Future Work

In this paper, we formalized the information flow behavior of direct communication in a microkernel-style hypervisor using property-specification patterns derived from general noninterference frameworks. The development of these patterns is a significant advance in the realm of operating system verification because it illustrates how to apply information flow theory to a practical system.

The next step is to port our model to a theorem proving environment, such as Isabelle/HOL, and to prove the properties specified in this paper. We anticipate that the use of Haskell will make this task easier because Haskell has a well-defined semantics and there are established connections between Haskell and Isabelle. We also plan to formalize the information flow behavior of the remaining primitives of HHV using the same techniques we applied to communication.

We based the correctness and security properties of our hypervisor on a Haskell model, rather than a low-level implementation, so that we could focus on the conceptual elements of our design. Our model is based on the same specification as the actual C++ implementation, but we have yet to establish a formal connection between the two. Ideally, we would like to derive the model directly from the implementation or refine the model into an implementation. However, our immediate plans focus on the specification and verification of the critical components using the model alone.

Acknowledgments

Sebastian Schönberg made significant contributions to this work by sharing his expertise in hypervisor design and implementation. We would also like to thank Mark P. Jones and Iavor S. Diatchki for their helpful suggestions regarding the presentation of this paper. This work was done during the first author's internship at Intel Research Labs.

References

[1] J. Rushby, "Noninterference, transitivity, and channel-control security policies," SRI International, Tech. Rep. CSL-92-02, December 1992.
[2] R. Leslie, "Dynamic intransitive noninterference," in *First IEEE International Symposium on Secure Software Engineering*, 2006.
[3] "seL4 web site," http://www.ertos.nicta.com.au/research/sel4.
[4] B. Kauer, "L4.sec implementation: Kernel memory management," Diploma Thesis, TU Dresden, 2005.
[5] D. Grawrock, *The Intel Safer Computing Initiative*. Intel Press, 2006.
[6] "Programatica web site," http://programatica.cs.pdx.edu, 2006.
[7] S. Peyton Jones, Ed., *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.
[8] W. L. Harrison and R. B. Kieburtz, "The logic of demand in Haskell," *J. Funct. Program.*, vol. 15, no. 6, pp. 837–891, 2005.
[9] B. Ford and J. Lepreau, "Evolving Mach 3.0 to a migrating thread model," in *Proceedings of the Winter 1994 USENIX Technical Conference and Exhibition*, 1994, pp. 97–114.
[10] J. S. Shapiro, "Vulnerabilities in synchronous IPC designs," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003, pp. 251–262.
[11] M. P. Jones, "Functional programming with overloading and higher-order polymorphism," in *Advanced Functional Programming, 1st Int. Spring School on Advanced Functional Programming Techniques-Tutorial Text*. London, UK: Springer-Verlag, 1995, pp. 97–136.
[12] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *POPL '95: Proc. 22nd ACM Symp. on Principles of programming languages*, 1995, pp. 333–343.
[13] D. von Oheimb, "Information flow control revisited: Noninfluence = Noninterference + Nonleakage," in *Computer Security – ESORICS 2004*, ser. LNCS, vol. 3193. Springer, 2004, pp. 225–243.
[14] J. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.
[15] H. Tuch, G. Klein, and G. Heiser, "OS verification — now!" in *Proc. 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, M. Seltzer, Ed., 2005.
[16] M. Hohmuth, H. Tews, and S. G. Stephens, "Applying source-code verification to a microkernel – The VFiasco project," Technische Universität Dresden, Tech. Rep. TUD-FI02-03, March 2002.
[17] H. Tuch and G. Klein, "Verifying the L4 virtual memory subsystem," in *Proc. NICTA Formal Methods Workshop on Operating Systems Verification*, G. Klein, Ed., NICTA Technical Report 0401005T-1. National ICT Australia, 2004, pp. 73–97.
[18] L4ka Team, *L4 eXperimental Kernel Reference Manual*, January 2005. [Online]. Available: http://l4hq.org/docs/manuals/l4-x2-20041209.pdf
[19] K. Elphinstone, G. Klein, and R. Kolanski, "Formalising a high-performance microkernel," in *Workshop on Verified Software: Theories, Tools, and Experiments (VSTTE 06)*, ser. Microsoft Research Technical Report MSR-TR-2006-117, R. Leino, Ed., Seattle, USA, 2006, pp. 1–7.
[20] P. Derrin, K. Elphinstone, G. Klein, D. Cock, and M. M. T. Chakravarty, "Running the manual: an approach to high-assurance microkernel development," in *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*. New York, NY, USA: ACM Press, 2006.
[21] M. P. Jones, "Bare Metal: A Programatica model of hardware," in *High Confidence Software and Systems Conference*, Baltimore, MD, March 2005.
[22] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach, "A principled approach to operating system construction in Haskell," in *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM Press, 2005.
[23] G. Schellhorn, W. Reif, A. Schairer, P. A. Karger, V. Austel, and D. Toll, "Verification of a formal security model for multiapplicative smart cards," in *ESORICS '00: Proceedings of the 6th European Symposium on Research in Computer Security*. London, UK: Springer-Verlag, 2000, pp. 17–36.