International Journal of Computer and Internet Security.
ISSN 0974-2247 Volume 3, Number 1 (2011), pp. 9-16

http://www.irphouse.com

Achieving Concurrent Information Security through Monadic Control Technique of Effects

Geetika and Neelam Kumari

Research Scholar, Singharni University,
Pacheri Bari, Rajasthan, India

Abstract

In this paper the controlling information flow and maintaining integrity via monadic encapsulation of effects. This approach is constructive, relying on properties of monads and monad transformers to build, verify, and extend secure software systems. We illustrate this approach by construction of abstract operating systems called separation kernels. Starting from a mathematical model of shared-state concurrency based on monads of resumptions and state, we outline the development by stepwise refinements of separation kernels supporting Unix-like system calls, inter domain communication, and a formally verified security policy. Because monads may be easily and safely represented within any pure, higher-order, typed functional language, the resulting system models may be directly realized within a language.

Introduction

Integrity and Confidentiality concerns within the setting of shared-state concurrency are primarily addressed by controlling interference and interaction between threads. Several investigators have attempted to achieve control of interference through language mechanisms that systematically separate information. Most of these approaches have been security-specific extensions to type systems for existing languages. In this investigation we take a different approach. We do not use a domain-specific extension to the type system. We use a standard pure functional language, with its existing type system, as our base language. Within that language and type system we characterize the effects that are at play in an operating system kernel using

the semantic technique of monadic encoding of effects. Most importantly, we construct the effect model in a modular manner using constructions called monad transformers. This modularity clearly distinguishes within the type system those facets of the global effect system on which a program fragment acts.

The development proceeds by developing three model kernels, the complete code of which may be downloaded from our website. These kernels build on one another. The first provides the reference point for thread behavior in isolation – the model of integrity of thread execution. The second and third kernels provide more sophisticated concurrency and communication primitives with sufficient power to be vulnerable to exploitation if separation is not achieved.

Precise control of effects

This approach supports an "abstract data type approach" to language definition, capturing distinct computational paradigms as algebras. A helpful metaphor is that a monad is a programming language with sequencing and "no-op" (skip) constructs where (;) is associative and has skip as its right and left unit. Monad "languages" may contain other language features corresponding to their computational paradigms: the state monad, for example, has assignment (:=) and resumption monads define concurrent execution (||) and, in some formulations, "reactive" programming features such as message passing, synchronization, etc. Monad transformers are monad language "constructors" that add new features to a monad language with each monad transformer application; such modularly-constructed monads are referred to as layered monads. This metaphor is not a precise characterization of monads or monadic semantics; it is imprecise in the following sense.

We demonstrate this approach through the construction of abstract operating systems called separation kernels. Separation kernels enforce a non-interference-based security property by partitioning the state into separate user.

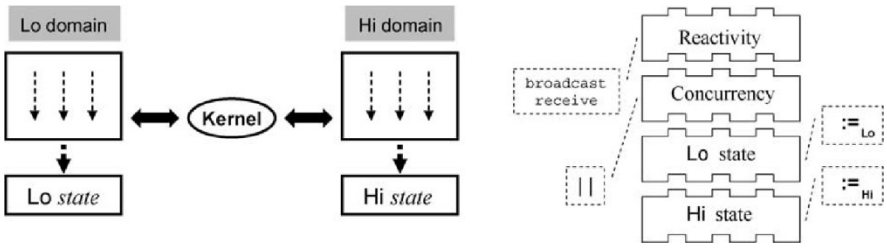

Figure 1: (Left) Separation kernel: threads within each domain can only access their own state, and all inter-domain communication is mediated by the kernel. The kernel enforces a "no write down" security policy. (Right) Layering monads for separation: combining fine control of stateful effects with concurrency into Layered Monads have important properties "by construction".

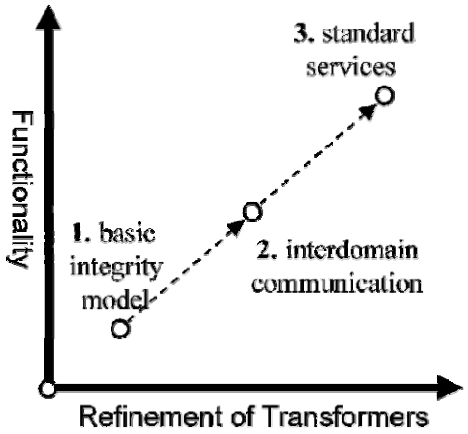

Figure 2: Scalability: Kernel specifications based on fine control of effects achieve a significant level of scalability in two important respects: they are easily extended and modified and the impact of such extensions on the security verification is minimized.

Layered state monads and separation

This section considers the representation and construction of separated domains. The principle tool applied to these tasks is the *layered state monad*. It is shown how "by construction" properties of layered state monads give rise naturally to an algebraic foundation for domain separation. Separation semantics for the process and event languages will be given partly in terms of layered state monads. Monads and their uses in the denotation semantics of languages with effects are essential to this work, and we assume of necessity that the reader possesses familiarity with them. This section begins with a quick review of the basic concepts of monads and monad transformers, and readers requiring more should consult the references for further background.

Definition 1: A state monad structure is a quintuple $\langle M, \eta, \beta, u, g, s\rangle$ where $\langle M, \eta, \beta\rangle$ is a monad, and the *update* and *get* operations on s are: $u : (s \to s) \to M()$ and $g : M\,s$.

We will refer to a state monad structure $\langle M, \eta, \beta, u, g, s\rangle$ simply as M if the associated operations and state type are clear from context.

Definition 2: A state monad is a state monad structure $\langle M, \eta, \beta, u, g, s\rangle$ such that the following equations hold for any f, $f' : s \to s$ and $\varphi : Ma$, $u\,f \gg u\,f' = u\,(f' \circ f)$ (sequencing) $g \gg \varphi = \varphi$ (cancellation)

The (sequencing) axiom shows how updating by f and then updating by f' is the same as updating by their composition $(f' \circ f)$. The (cancellation) axiom specifies

that g operations whose results are ignored have no effect on the rest of the computation.

Definition 3: A mask for state monad $\langle M, \eta, \beta, u, g, s \rangle$ is defined as: $mask_s = u(\lambda\_.\ \sigma_0)$ for any arbitrary fixed state σ_0. The (clobber) rule captures the defining property of mask operators: $uf \gg mask_s = mask_s$ (clobber)

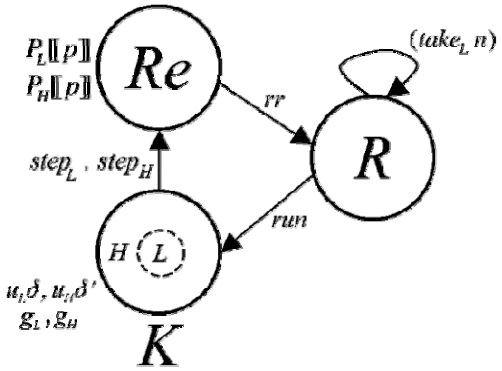

Figure 3: Monadic architecture.

The basic model of integrity

The confidentiality policies seek to eliminate inappropriate disclosure of information, integrity policies seek to eliminate inappropriate modification of data. This section demonstrates how monadic fine control of effects addresses integrity concerns. To do so, we present the basic model of integrity. In this kernel, threads in different domains are totally separate – they cannot modify storage in an-other domain. This complete separation is a direct consequence of the properties of layered state monads developed. Before the basic integrity model may be described, however, we must formulate its concurrency model, and for this, we use monads of resumptions.

Allowing secure inter domain interaction

This section extends the basic integrity model to include primitives for inter domain interaction – in this case, asynchronous message broadcast and blocking receive events – that introduce the possibility of insecure information flow. Inter domain interactions are mediated entirely through the separation kernel as in Rushby's original conception and it is in the kernel that the "no write down" security policy is enforced. This extension follows the pattern of modular language definitions as well in that the text of the basic integrity model remains almost entirely intact within the enhanced kernel; the increased system functionality comes about through refinements to the underlying monads. The encapsulation of the new reactive features (i.e., message-passing primitives) by a monad transformer aids the security verification by isolating them from the other kernel building blocks.

Reactive concurrency and separation

A reactive program is one that interacts continually with its environment and may be designed to not terminate (e.g., an operating system). We coin the term reactivity to mean the notion of computation given by reactive programs. We now consider a refinement to the concurrency model that allows computations to signal requests to the kernel and receive responses from it; we coin the term reactive resumption monad to distinguish this structure from the previous one. Although the reactive resumption monadic structure is mentioned in passing in the literature, it was never named to the authors' best knowledge. A reactive resumption monad has constructors for pausing computations just as basic resumption monads do, and it also extends the basic resumption structure with Unix-like system requests and responses.

Reactive resumption monad transformer

Reactive resumption monads have two non-proper morphisms. The first of these, step, is defined analogously to its definition in ResT. The definition of step shows why we require that Req and Rsp have a particular shape including Cont and Ack, respectively; namely, there must be at least one request/response pair for the definition of step. Another non-proper morphism provided by ReactT allows a computation to raise a signal; its definition is given below. Furthermore, there are certain cases where the response to a signal is intentionally ignored, for which we define signull:

$$Re = ReactT\ Req\ Rsp\ M,$$

step : $Ma \rightarrow \mathrm{Re}\ a$,

step $x = P$ (Cont, λ Ack. $x \gg_M (\eta_M \circ D)$), signal : Req $\rightarrow$ Re Rsp,

signal $q = P\ (q, \eta_M \circ \eta_{\mathrm{Re}})$, signull : Req $\rightarrow$ Re (),

signull $q = P(q, \eta_M \circ \eta_{Re} \circ (\lambda\_.\ ()))$.

In the definition of (*signal q*) above, the effect of the composition, $\eta_M \circ \eta_{Re}$, is that the system response ultimately passed to this function will be returned as the value of the computation. The pre-composition of (λ_.()) in the definition of (*signull q*), in contrast, will replace this system response by the nil value, ().

Different versions of *step* with *R* and *Re* are used in this paper, but without ambiguity as the version of *step* is determined by the type context of its use. Within the atom (*step x*), a *Cont* request by a user thread expects the response *Ack* and this expectation is encoded by the pattern "λ *Ack*". This imposes constraint on the kernel to always respond to a *Cont* request with an *Ack* acknowledgment. Each kernel in this article obeys this simple, sensible constraint. It is a simple matter to design a semantics in which processes handle any response from the kernel, al-though, for the purposes of the present article; this was seen as a needless distraction.

Security-conscious reactive resumption monad transformer

We make the monad transformer (*ReactT q r*) security-conscious as before by including a high and low security pause; this is the version used hereafter:

```
data ReactT q r M a = D a
P_L(q, r → (M (ReactT q r M a)))
P_H (q, r → (M (ReactT q r M a))).
```

Inter-domain communication

This section considers point 2 in Fig. 2: the extension of the basic model of integrity of Section 4.1 to express inter-domain communication. Any such extension requires demonstration that Hi domain threads cannot affect Lo threads – in this case that the system obeys a "no write down" security policy.

The Event language is extended with two new events, bcast(l) and recv(l), and accommodating them requires the introduction of reactivity. To this end, Req is extended with broadcast and receive request tags (Bcst Int and Rcv, respectively) and Rsp is extended with the response to a receive request, (Msg Int):

```
type Re = ReactT Req Rsp K, data Req = Cont | Bcst Int | Rcv, data Rsp = Ack | Msg Int.
```

Achieving scalability

How are typical operating system behaviors (e.g., process creation, preemption, synchronization, etc.) achieved in this layered monadic setting and what impact, if any, do such enhancements to functionality have on the security verification? These are questions to which it is difficult to give final, definitive answers; however, by considering an example, one can get some indication as to what the relevant concerns are. This section considers such an extension – a process creation primitive called dupl – to the inter-domain communication kernel of the previous section.

As it turns out, this additional functionality requires *no* change to the existing resumption monadic framework and has little impact on the security verification. The impact it does have on the verification is as limited as one could reasonably hope for and boils down simply to considering an extra case corresponding to the added functionality in each of the various proofs and definitions of Section 5.3. That is to say, the verified properties in Section 5.3 required no re-verification at all. This modularity and scalability in the verification would seem to arise from the fact that the new functionality is an orthogonal concern to the security property. That is, because the added functionality does not result in inter-domain information flow, it has no impact on the system-wide security.

Conclusion

Type constructions and their properties are the foundation of this approach to language-based security; this is fundamentally different from approaches based on information flow control via type checking. The approach reflects the semantic foundations of effects and effect interaction into a pure functional language in which provably separable computations can be constructed. At the same time, it allows explicit regions of the program in which the type system does not, by itself, guarantee separation. In the monadic approach it is clear from the type construction when information flow separation is established and when it is established by reasoning about program behavior.

This approach can be used either for direct implementation or as a modeling language. As a modeling language, these techniques can explain the effect separation provided by unprivileged execution modes in hardware, while at the same time modeling the potential interference of privileged execution. As an implementation language it provides, through the type constructions, ways to construct programs that achieve information flow separation. In this sense this work is similar to language-based security mechanisms based on type checking. However, such approaches are domain-specific extensions of type systems to express information flow properties; the monadic approach uses concepts easily expressed in existing type systems for pure higher-order languages.

We have not explored the formal relationship between domain-specific type systems for information flow and monads. We suspect that in some cases it may be possible to prove the soundness of information flow extensions to other languages by embedding them into the monadic type systems presented here. This may be of particular interest when applied to recent enhancements to information flow type systems that allow for policy enabled downgrading functions to be defined.

Confidentiality and integrity concerns within the setting of shared-state concurrency are really about controlling interference and interaction between threads. It is a natural and compelling idea, therefore, to apply the mathematics of effects – monads – to this problem as monads provide precise control of such effects. In fact, layering monads – i.e., modularly constructing monads with monad transformers – yields fine-grained control of effects and their interactions. This paper demonstrates how the fine-grained tailoring of effects possible with monad transformers promotes integrity and information security concerns. As a proof of concept, we showed that a classic design in computer security (the separation kernel of Rushby can be realized and verified in a straightforward manner.

References

[1] M. Abadi, A. Banerjee, N. Heintze and J. Riecke, A core calculus of dependency, in: Proceedings of the Twenty-Sixth ACM Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 1999, pp. 147–160.

[2] D. Alexander, W. Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. Moore, C. Gunder, S. Nettles and J. Smith, The switchware active network architecture, IEEE Network (1998).
[3] M. Archer and C. Heitmeyer, TAME: A specialized specification and verification system for timed automata, in: Work In Progress (WIP) Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96), A. Bestavros, ed., Washington, DC, USA, 1996, pp. 3–6.
[4] M. Barr and C. Wells, Category Theory for Computing Science, Prentice Hall, 1990.
[5] R. Bird, Introduction to Functional Programming using Haskell, 2nd edn, Prentice-Hall Series in Computer Science, Prentice-Hall Europe, London, UK, 1998.
[6] K. Birman, R. Constable, M. Hayden, C. Kreitz, O. Rodeh, R. van Renesse and W. Vogels, The Horus and Ensemble projects: Accomplishments and limitations, in: Proceedings of the DARPA Informa-tion Survivability Conference & Exposition (DISCEX'00), 2000.
[7] K. Claessen, A poor man's concurrency monad, Journal of Functional Programming 9(3) (1999), 313–323.
[8] D. Espinosa, Semantic Lego, PhD thesis, Columbia University, 1995.
[9] R. Giacobazzi and I. Mastroeni, Adjoining declassification and attack models by abstract interpretation, in: European Symposium on Programming (ESOP'05), LNCS, Vol. 3444, Springer-Verlag, 2005, pp. 295–310.
[10] D. Greve, R. Richards and M. Wilding, A summary of intrinsic partitioning verification, in: Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004), November 2004.