Micro-Policies: A Framework for Verified, Tag-Based Security Monitors

Advisor: Cătălin Hriţcu ⟨catalin.hritcu@gmail.com⟩

Institition: INRIA Paris-Rocquencourt, Prosecco Team

Location: 23 Avenue d'Italie, Paris, France

Language: English

Existing skills or strong desire to learn:

- formal verification in the Coq proof assistant,
- functional programming (e.g. OCaml or Haskell)
- optional: security, compiler construction, programming languages theory

Research Context

Today's computer systems are distressingly insecure, but many of their vulnerabilities can be avoided if low-level code is constrained to obey sensible safety and security properties. Ideally, such properties might be enforced statically, but for obtaining pervasive guarantees all the way to the level of running machine code it is often more practical to detect and prevent violations dynamically using a *reference monitor* (Anderson, 1972; Erlingsson and Schneider, 2000; Schneider, 2000). Decades of work have produced a large body of knowledge about monitoring techniques that can reduce the vulnerability of computers and software to malicious subversion by providing sanity checks that current hardware lacks (e.g., "array accesses should always be in-bounds," "this instruction should never be the target of a *Jump*,"). Many simply enforce language abstractions that current software and hardware omit, e.g., memory safety or control-flow integrity (CFI); these policies can be automatically applied to all code with no additional programmer burden. Others are policies a system architect might like to apply to security-critical components without auditing or rewriting every line of code in a larger system (e.g., "un-sanitized data from the network should not be treated as a pointer," "private data should be encrypted before flowing to I/O devices,"). Still others offer primitives that system designers can use to protect critical data, track confidentiality and integrity, or isolate untrusted components.

Reference monitors are sometimes implemented in software, but this can significantly degrade performance and/or cause designers to settle for rough approximations of the intended policy that are potentially vulnerable to attack (Davi et al., 2014; Göktaş et al., 2014). Hardware acceleration is thus an attractive alternative, especially in an era of cheap transistors. Many designs for hardware monitors have been proposed, with early designs focusing on enforcing single hard-wired security policies (Suh et al., 2004) and later ones evolving toward more programmable mechanisms that allow quicker adaptation to a shifting attack landscape.

The motivating insight underlying our work[1] is that a wide range of policies can be efficiently enforced using a *generic* mechanism that associates each piece of data in the system with a *metadata tag* describing its provenance or purpose (e.g., "this is an instruction," "this came from the network," "this is secret," "this is sealed with key k"), propagates this metadata as instructions are executed, and checks that policy rules are obeyed throughout the computation. This fully programmable hardware/software architecture provides great flexibility for defining policies and removes limitations on the size of the metadata and the number of policies supported. We use the term *micro-policies* for instruction-level security-monitoring mechanisms based on fine-grained metadata. The

[1] This project is a collaboration between (a) Cătălin Hriţcu from INRIA Paris-Rocquencourt, (b) Benjamin Pierce, André DeHon, Arthur Azevedo de Amorim, and Antal Spector-Zabusky from University of Pennsylvania, (c) Nick Giannarakis from ENS Cachan, (d) Andrew Tolmach from Portland State University, and (e) in the past several other students and researchers.

goal of this project is to develop and demonstrate a framework for defining, reasoning about, verifying, and efficiently implementing micro-policies. For this we use our team's expertise in security, formal verification, and hardware design.

When viewed abstractly, each micro-policy consists of (i) sets of *metadata tags* that are used to label every piece of data in the machine's memory and registers (including the *pc*); (ii) a *transfer function* that, given the current opcode and the tags on the current instruction, the *pc*, and the instruction operands, specifies how the *pc* and the instruction's result should be tagged in the next machine state; and (iii) a set of *monitor services* that can be invoked by user code. For example, in a micro-policy for "dynamic sealing" (a language-based protection mechanism in the style of perfect symmetric encryption (Morris, 1973)) the set of tags used for registers and memory might be $\{Data, Key(k), Sealed(k)\}$, where *Data* is used to tag ordinary data values, *Sealed(k)* is used to tag values sealed with the key k, and a value tagged *Key(k)* denotes a key that can be used for sealing and unsealing values. The transfer function for this micro-policy would allow, for example, arithmetic operations on values tagged *Data* but deny them on data tagged *Sealed* or *Key*. Monitor services are provided to allow user programs to create new keys and to seal and unseal data values with given keys.

Our proposed hardware support for micro-policies, called the *Programmable Unit for Metadata Processing (PUMP)* (Dhawan et al., 2015), is designed as an add-on to a conventional RISC processor. Conceptually, every word of data on the machine is associated with a *tag*, a full machine word that can, in particular, hold a pointer to an arbitrary data structure in memory. The interpretation of tags is left entirely to software; the hardware simply propagates tags from operands to results according to software-defined *rules*. To propagate tags efficiently, the processor is augmented with a *rule cache* that operates in parallel with instruction execution. On a rule cache miss, control is transferred to a trusted *miss handler* which, given the tags of the instruction's arguments, decides whether the current operation should be allowed and, if so, computes appropriate tags for its results. It adds this set of argument and result tags to the rule cache so that, when the same situation is encountered in the future, the rule can be applied without slowing down the processor.

This flexible hardware/software mechanism can efficiently implement a wide range of different micro-policies, singly or in combination. It achieves the flexibility and adaptability of software with performance comparable to dedicated hardware. Hardware simulations (Dhawan et al., 2015) using a simple RISC processor (an Alpha) show that a composite micro-policy simultaneously enforcing memory safety, CFI, and taint tracking can be used to monitor a standard benchmark suite with modest impact on runtime (typically under 10%) and power ceiling (less than 10%), in return for some increase in energy usage (typically under 60%) and chip area (110%).

However, encoding desired properties as micro-policies that perform well in practice can be nontrivial. While a higher-level rule-based programming model helps in writing down micro-policies, it is still easy to get them wrong: only formal verification can give complete confidence. This observation motivated two recent papers. In the first we give a formal definition and correctness proof for an information-flow control (IFC) micro-policy (Azevedo de Amorim et al., 2014a). In the second, recently submitted, paper (Azevedo de Amorim et al., 2014b) we introduce a generic framework for defining and formally reasoning about arbitrary micro-policies. The framework is entirely implemented (and checked) in Coq and the code is available online.[2] We use this framework to define and verify micro-policies for dynamic sealing, control-flow integrity, compartmentalization, and memory safety; in addition, we show how to use the tagging mechanism to protect its own integrity. For each micro-policy, we prove by refinement that the hardware running a correctly implemented monitor embodies a high-level specification characterizing a useful security property.

Topics

Express and Verify New Micro-Policies and Extend Some Existing Ones

The existing policies should give a useful template for studying new micro-policies and further extending some of the existing ones. We will also study *composite* policies that enforce several of these together (see below).

Linearity We would like to develop a novel micro-policy for tracking *linear use of resources* (Nöcker et al., 1991; Wadler, 1990). Resources such as memory regions and channel endpoints can be marked (e.g., at creation time) as linear, resulting in a pointer (or channel identifier, or any other resource) that is marked as non-copyable; this ensures that, for example, any piece of code that has access to a linear pointer may `free` it with no danger

[2] `https://github.com/micro-policies`

of creating a dangling reference elsewhere. Instructions such as register moves, loads, and stores will have associated policy rules that, when the source register (or memory location) contains a linear value, *retag* that value as unusable once the instruction completes. This requires one (nontrivial, but we believe feasible) extension to the PUMP hardware: allowing the result of a rule cache lookup not only to determine the tag on the current instruction's result, but also to set the tag on its source.

Call-stack protection Our fine-grained CFI micro-policy (Azevedo de Amorim et al., 2014b) enforces that each indirect jump reaches a possible target according to the program's CFG. In particular each return can only reach one of the possible call sites. While useful at preventing attacks, this is not enough for enforcing a stack discipline on calls and returns, and more generally enforcing that an arbitrary stack-based calling convention is respected. We plan to implement, verify, and experiment with a micro-policy enforcing call-stack protection and stack-based calling conventions. A naive implementation of this would simply prohibit user code from executing call and return instructions, and instead require it to perform these operations using monitor calls that perform additional checks. This would, however, have large overhead, and would require the monitor code to access user memory without itself faulting in the rule cache. Existing mechanisms (Azevedo de Amorim et al., 2014b; Dhawan et al., 2015) —ground rules together with "don't cares" and "copy-throughs"—can solve this second problem. For low-overhead enforcement we would not use monitor calls, but instead enforce that a certain sequence of instructions corresponding to the calling convention is executed before each call or return instruction. Doing this for actual calling conventions for a stock architecture (e.g., ARM) could prove challenging and might require extending the micro-policies framework in non-trivial ways. These extensions will probably be generally useful for expressing inline-reference monitors (Erlingsson and Schneider, 2000; Erlingsson et al., 2006) as micro-policies.

Data-race detection We plan to study a micro-policy enforcing *data race detection* of concurrent programs. Many algorithms have been developed to detect data races in running programs (Elmas et al., 2007; Flanagan and Freund, 2009; Itzkovitz et al., 1999; O'Callahan and Choi, 2003; Pozniansky and Schuster, 2003; Savage et al., 1997); they differ in the tradeoffs they make between soundness (with respect to data-race freedom), completeness, efficiency, and simplicity. We plain to target sound and complete algorithms such as Itzkovitz et al. (1999), Pozniansky and Schuster (2003) and FastTrack (Flanagan and Freund, 2009). The format of micro-policies meshes well with the common presentation of these algorithms as instrumenting all data in the program, along with a monitor containing global state. The PUMP gives us the facilities to do this directly: instrumentation data for specific variables can be represented as tags on the memory, per-thread information can be stored in the *pc* tag, and each micro-policy is permitted to have arbitrary internal state for any shared monitor data (e.g., synchronization information). Prior studies on synchronization frequency in benchmark workloads (Bienia, 2011; Flanagan and Freund, 2009) suggest that our scheme should add only modest runtime overhead to typical concurrent programs.

Others More speculatively, we have a number of other ideas for micro-policies that should fit well in our framework: (i) refining pointers into a more expressive form of "capabilities" by tagging them with subsets of $\{readable, writeable, jumpable, callable\}$; (ii) closures (i.e., represented as code pointers together with protected local environments, tagged so they can be called but not examined); (iii) higher-order contracts (Findler and Felleisen, 2002) (we conjecture that recently proposed mechanisms for tracking components and assigning blame (Dimoulas et al., 2011) can be encoded as tags); and (iv) dynamic type tags.

Extensions Finally, we would also like to work on extensions of our current micro-policies. For instance, we would like to enforce memory safety not only for boxed heap-allocated data (Azevedo de Amorim et al., 2014b), but also for stack-allocated data and C-like unboxed `structs`.

A Formal Language for Defining Micro-Policies

To streamline both implementing and reasoning about micro-policies, we need a suitably high-level notation—similar in flavor to the informal symbolic rule notation used in Azevedo de Amorim et al. (2014b); Dhawan et al. (2015), but with a detailed, formal definition and a verified compiler. Our LAnguage for Micro-Policies (LAMP) should (i) be *expressive* enough to support a wide range of micro-policies; (ii) *support verification* of the expressed micro-policies; (iii) compile to *efficient* low-level code, in order to write transfer functions and monitor services that may be invoked very often and thus have a huge impact on overall system performance;

(iv) *provide high-assurance* in the correctness of the generated machine code, ideally in the form of a verified compiler; (v) be implementable with *reasonable effort*, because compiler construction is not the focus of this project; and (vi) support *composition* of micro-policies to get multiple guarantees at once.

In light of these requirements, our plan is to build LAMP as a domain-specific language within Coq, comprising two integrated layers: a *specification* layer, where tags, symbolic rules and monitor services are described directly in a high-level declarative language; and an *implementation* layer, where those definitions are implemented directly by machine code that runs on PUMP-enhanced hardware (see Fig. 1). The symbolic PUMP provides an abstract execution model for tag propagation that elides low-level details, making it more suitable for proofs. This separation will allow micro-policy designers to prove security properties of a system by *refinement*, as the composition of two theorems: one showing that tags as given by the abstract specification of a micro-policy are capable of enforcing some desired security property, and another one showing that the concrete micro-policy code is indeed an implementation of that specification. Refinement is a standard and effective technique for proving facts about low-level systems (Azevedo de Amorim et al., 2014a; Klein et al., 2009; Liang et al., 2013). Furthermore, since LAMP is just an embedded DSL in Coq, it will be much easier to integrate different definitions used for specifying a micro-policy. For instance, it will be possible to refer directly to the high-level definitions of transfer functions and system services when proving the correctness of the corresponding machine-code implementations. This separation will also make it easier to split the work of micro-policy design, proof, and implementation between different people.

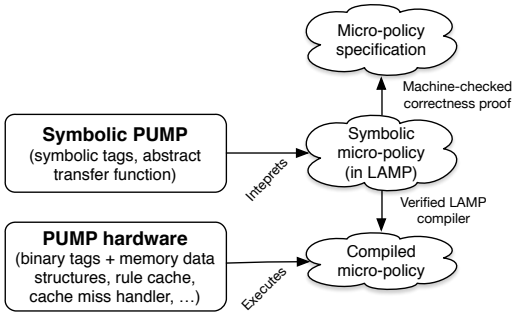

Figure 1: Overview

We plan to base LAMP's implementation layer on Bedrock (Chlipala, 2013), an extensible separation-logic-based Coq framework for verifying machine code generated using structured macros. Bedrock is already tuned for formal verification and generation of efficient code with precise control over memory layout.

Formally Study Micro-Policy Composition

Recent hardware optimizations enable us to efficiently enforce multiple policies simultaneously (Dhawan et al., 2015): under 10% overhead for memory safety, CFI, and taint tracking simultaneously. The major difficulty is indeed not in composing micro-policies (that is for the most part easy: just take tags to be tuples of subtags and dispatch to the transfer functions of the individual policies), but in composing specifications and correctness proofs. One subclass for which we think composition should be relatively easy is micro-policies *without* monitor services: all micro-policies in this class are orthogonal, and one is in the realm of execution monitoring / safety properties where composition is well studied (Schneider, 2000). While this class is restrictive, some of the micro-policies we study in this work are already in this class (CFI doesn't itself use monitor services) or could plausibly be translated to this class by replacing monitor services by new primitives like "one-shot" rules for the allocation of fresh names (memory safety, taint tracking). Our ASPLOS paper (Dhawan et al., 2015) did in fact use one-shot rules to implement and evaluate the overhead of such orthogonal combinations. One downside of this solution is that the monitor service code lives in user mode, so monitor services are no longer protected from potential abuse; this might be OK security-wise if the composed micro-policies include memory safety and CFI, but it greatly complicates formal reasoning (which runs the danger of becoming circular).

For policies with monitor services, composition is a bigger problem, since one policy's monitor services can break another policy's guarantees. For example, if we wish to enforce IFC together with other policies, observing the tags of these other policies via monitor operations can reveal sensitive information and thus break the noninterference property established for the IFC micro-policy in isolation. There are several ways one could approach this problem. The first way builds on the idea of *vertical sequential composition*. In this case a linear order is carefully chosen in advance between the micro-policies to be composed. One starts with the lowest-level micro-policy and the bare hardware and proves that the policy is correct with respect to a higher-level abstract machine. This higher-level abstract machine virtualizes the tagging mechanisms in the hardware so that the second micro-policy can be implemented on top of this first abstract machine, instead of the bare hardware. We then prove the correctness of this second micro-policy with respect to a second abstract machine, and so on. While this technique is likely to work well for one-off proofs, it is dependent on the set of composed policies

and the initially chosen order between them. If we want to add or remove policies we can only do that at the top of the stack; otherwise we have to redo a large number of proofs. One can try to use ideas from monad transformers to lift some of these limitations. One can view the micro-policies verified in (Azevedo de Amorim et al., 2014b) as examples of vertical sequential compositions of a low-level monitor-self protection policy and the considered higher-level policy.

The second approach we will consider can be called *parallel composition* or *cross-product composition*. The idea is to require each micro-policy to specify what should happen with its own tags on all other micro-policies' monitor operations. Basically, all other micro-policies' monitor operations become "virtual instructions" for each micro-policy machine (thus the name cross-product composition). For instance, the IFC micro-policy could specify that the result of a monitor call that returns a value's tag in another micro-policy is as classified as the original value. For this kind of composition there is additional proof effort when composing policies, and this effort scales quadratically with the number of composed policies.

Efficient Secure Compilation Using Micro-Policies

Micro-policies are very well suited for efficiently enforcing all high-level language abstractions against all low-level attackers. Formally, this is captured by a property called *full abstraction* (Abadi, 1999; Abadi and Plotkin, 2012; Fournet et al., 2013) or *secure compilation* (Agten et al., 2012); because of the presence of an attacker this is a much stronger property than the usual compiler correctness (Leroy, 2009). Efficiently enforcing full abstraction all the way to the machine code level is, however, very hard. While some fully abstract compilers have been built, they often have very large overhead (Fournet et al., 2013), large TCB (Fournet et al., 2013), or only provide weak probabilistic guarantees against specific classes of attacks (Abadi and Plotkin, 2012). We will explore the feasibility of building an efficient fully abstract compiler targeting a micro-policy machine.

We will start by investigating simple languages: an imperative language with first-order procedures, a simple object-oriented language with private fields (Agten et al., 2012), the untyped and the simply typed λ-calculus, etc. In each case we will identify a set of micro-policies that can be composed to enforce the higher-level abstractions with low overhead, devise a simple compiler targeting this composite policy, and prove full abstraction in Coq for this compiler. To obtain efficient compiled code, it may be interesting to consider hybrid static / dynamic enforcement strategies. Even for such simple languages there are several expected challenges, which we group into three classes: (i) We need to formally define the class of contexts we want to protect against. While this class will be very broad, it will still need to distinguish the untrusted context from the compiled program. Furthermore protecting against certain kinds of attacks (e.g., timing attacks, in which a low-level context counts instructions to obtain information a high-level context cannot obtain) might be out of the reach for micro-policies, in which case they need to be carefully excluded from our security notion. (ii) We need to secure the runtime system used by the code we produce. High-level languages usually rely on a garbage collector, programs in all languages rely on library code, concurrent programs rely on a scheduler, and all these components also have to be protected from the untrusted context. (iii) Obtaining end-to-end full-abstraction proofs will be challenging, and will require changes to our proof methodology (Azevedo de Amorim et al., 2014a), which is currently based on the weaker notion of refinement. Previous work on contextual refinement (Liang et al., 2013; Shao et al., 2013) could be useful at bridging this gap.

An Architecture-Independent Language for Micro-Policies

Our previous work has targeted specific hardware architectures: MIPS, Alpha (Dhawan et al., 2015), or an idealized RISC (Azevedo de Amorim et al., 2014a,b; Hriţcu et al., 2014). Writing micro-policies and formally verifying them is challenging;[3] so we would like to avoid repeating this effort for each architecture. Moreover, we want to scale our techniques up to more realistic and complex architectures (like ARM or even x86). We will try to achieve all this by expressing and verifying micro-policies at the RTL (register transfer list) level (Dias and Ramsey, 2010; Lim, 2011; Ramsey and Davidson, 1998; Ramsey and Fernandez, 1997). Any instruction set architecture (ISA) that can be specified in our RTL will get for free verified monitors for standard properties. We will take inspiration in the RockSalt project (Morrisett et al., 2012) that introduced a dependently-typed monadic RTL embedded into Coq, and used this RTL to encode and reason about a fragment of the x86 ISA.

[3] Our previous verification efforts (Azevedo de Amorim et al., 2014a,b) add up to more than 40,000 LOC (`https://github.com/micro-policies`).

Study Expressive Power of Micro-Policies

Formally characterizing the class of properties that can be expressed as micro-policies and efficiently enforced by the PUMP is an interesting open problem. We know for sure that this class includes interesting security properties: IFC, CFI, compartmentalization, and memory safety, but precisely characterizing this class is challenging. One can take inspiration in the work done by Schneider et al. (Hamlen et al., 2006; Schneider, 2000) for execution monitors and program rewriting, but one has to also capture the working sets and caching behavior of micro-policies. If we restrict our attention to specific programs, we can try to use static analysis to predict caching behavior (Ferdinand et al., 1999; Wilhelm et al., 2008).

References

M. Abadi. Protection in programming-language translations. In *Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 1999.

M. Abadi and G. D. Plotkin. On protection by layout randomization. *ACM Transactions on Information and System Security*, 15(2):8, 2012.

P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *25th IEEE Computer Security Foundations Symposium*, pages 171–185. IEEE, 2012.

J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, Oct. 1972.

A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 165–178. ACM, Jan. 2014a.

A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: A framework for verified, tag-based security monitors. Under Review, Nov. 2014b.

C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

A. Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 391–402. ACM, 2013.

L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium*, pages 401–416, 2014.

U. Dhawan, C. Hriţcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon. Architectural support for software-defined metadata processing. To appear in ASPLOS, 2015.

J. Dias and N. Ramsey. Automatically generating instruction selectors using declarative machine descriptions. In *37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 403–416. ACM, 2010.

C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. In *38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 215–226. ACM, 2011.

T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 42, pages 245–255, 2007.

Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255. IEEE Computer Society, 2000.

Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *7th Symposium on Operating Systems Design and Implementation*, pages 75–88. USENIX Association, 2006.

C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2):163–189, 1999.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th International Conference on Functional Programming*, ICFP, pages 48–59. ACM, 2002.

C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 44, pages 121–133, 2009.

C. Fournet, N. Swamy, J. Chen, P.-É. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 371–384. ACM, 2013.

E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, 2014.

K. W. Hamlen, J. G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, 2006.

C. Hriţcu, L. Lampropoulos, A. Spector-Zabusky, A. A. de Amorim, M. Dénès, J. Hughes, B. C. Pierce, and D. Vytiniotis. Testing noninterference, quickly. arXiv:1409.0393; Submitted to Special Issue of Journal of Functional Programming for ICFP 2013, Sept. 2014.

A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai. Towards integration of data race detection in DSM systems. *Journal of Parallel and Distributed Computing*, pages 180–203, 1999.

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 Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009.

X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

H. Liang, J. Hoffmann, X. Feng, and Z. Shao. Characterizing progress properties of concurrent objects via contextual refinements. In *24th International Conference on Concurrency Theory*, volume 8052 of *Lecture Notes in Computer Science*, pages 227–241. Springer, 2013.

J. Lim. *Transformer Specification Language: A System for Generating Analyzers and its Applications*. Ph.d dissertation and tech. rep. tr-1689, University of Wisconsin-Madison, May 2011.

J. H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.

G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: better, faster, stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 395–404. ACM, 2012.

E. G. M. H. Nöcker, S. E. W. Smetsers, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Concurrent clean. In *Symposium on Parallel Architectures and Languages Europe, Volume I: Parallel Architectures and Algorithms (PARLE), Eindhoven, The Netherlands*, volume 505 of *Lecture Notes in Computer Science*, pages 202–219. Springer-Verlag, June 1991.

R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 38, pages 167–178, Oct. 2003.

E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency Computation Practice And Experience*, 38(10):179–190, 2003.

N. Ramsey and J. W. Davidson. Machine descriptions to build tools for embedded systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *Lecture Notes in Computer Science*, pages 176–192. Springer, 1998.

N. Ramsey and M. F. Fernandez. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, 1997.

S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

F. B. Schneider. Enforceable security policies. *ACM Transactions of Information Systems Security*, 3(1):30–50, 2000.

Z. Shao, R. Gu, T. Ramananandro, N. Wu, H. Zhang, Y. Guo, and J. Koenig. Compositional layered specification and verification of a hypervisor OS kernel. Talk at Layered Assurance Workshop, Dec. 2013.

G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2004.

P. Wadler. Linear types can change the world. In *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 546–566, Apr. 1990.

R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions in Embedded Computing Systems*, 7(3), 2008.