Formally Secure Compilation

Last updated: October 31, 2017

General Information

Advisor: Cătălin Hriţcu ⟨catalin.hritcu@inria.fr⟩

Institition: Inria Paris, Prosecco Team

Location: 2 rue Simone Iff, 75012 Paris, France

Language: English

Keywords:

- state-of-the-art research in programming languages;
- security foundations and/or building secure systems;
- secure compilation, full abstraction, components;
- preserving hyperproperties, safety properties;
- dynamic enforcement, tag-based monitoring;
- formal verification in the Coq proof assistant;
- property-based testing (e.g., with QuickChick).

Context

Severe low-level vulnerabilities abound in today's computer systems, allowing cyber-attackers to remotely gain full control. This happens in big part because our programming languages, compilers, and architectures were designed in an era of scarce hardware resources and too often trade off security for efficiency. The semantics of mainstream low-level languages like C is inherently insecure, and even for safer languages, establishing security with respect to a high-level semantics does not guarantee the absence of low-level attacks. *Secure compilation* using the coarse-grained protection mechanisms provided by mainstream hardware architectures would be too inefficient for most practical scenarios.

SECOMP[1] is a new ERC-funded project aimed at leveraging emerging hardware capabilities for fine-grained protection to build the first, efficient secure compilers for realistic low-level programming languages (the C language, and Low* [27] a safe subset of C embedded in F* [2, 30] for verification). These compilers will provide a more secure semantics for source programs and will ensure that high-level abstractions cannot be violated even when interacting with untrusted low-level code. To achieve this level of security without sacrificing efficiency, our secure compilers target a *tagged architecture* [4, 13], which associates a metadata tag to each word and efficiently propagates and checks tags according to software-defined rules. We are using property-based testing and formal verification to provide high confidence that our compilers are indeed secure. Formally, we are constructing machine-checked proofs in Coq of fully abstract compilation and of a new property we call robust compilation, which implies the preservation of trace properties even against an adversarial context. These strong properties complement compiler correctness and ensure that no machine-code attacker can do more harm to securely compiled components than a component already could with respect to a secure source-level semantics.

This document is aimed at briefly presenting various topics from the SECOMP project to which an excellent student or young researcher could contribute (during a research internship, MSc or PhD thesis, PostDoc, etc, see `https://secure-compilation.github.io/#positions` for details). The list is not exhaustive, and if you are interested you should get in contact and we will together try to find a topic that is in sync with your interests and expertise.

[1]`http://secure-compilation.github.io/`

Contents

1 Robust Hyperproperty Preservation for Secure Compilation

Secure compilation is an emerging field that puts together advances in programming languages, verification, compilers, and security enforcement mechanisms to devise secure compiler chains that eliminate many of today's devastating low-level vulnerabilities. One class of low-level vulnerabilities arises when code written in a safe language is compiled and interacts with unsafe code written in a lower-level language, e.g., when linking with libraries. While currently all the guarantees of the source code are generally lost in such cases, we would like to devise secure compilers that protect some of the security guarantees established in the source language even against adversarial low-level contexts.

What is a good soundness criterion for a compiler that attains this? Fully abstract compilation [1] is a criterion that provides one potential answer to this question: a fully abstract compiler preserves the observational equivalence of partial source programs.[2] In more detail, a compiler is fully abstract when any two partial source programs that are observationally indistinguishable by all compatible adversarial source contexts get compiled to two low-level programs that are indistinguishable by all *low-level* adversarial contexts. While fully abstract compilation has received significant attention in the literature, the indistinguishability of partial programs in all contexts is not the only security property one might be interested in preserving. In this work we set out to explore a much larger space of security properties that can be preserved even against adversarial low-level contexts.

Specifically, we look at preserving classes of *hyperproperties* despite adversarial contexts. Hyperproperties [11] are a generalization of trace properties that can express important security policies such as noninterference. While trace properties are formally expressed as sets of (potentially infinite) traces, hyperproperties are sets of sets of traces. Concretely, these traces are built over events such as inputs from and outputs to the environment [21]. We say that a complete program P satisfies a hyperproperty H when the set of traces of P is a member of H, or formally $\{t \mid P \Downarrow t\} \in H$. We say that a partial program P *robustly satisfies* [20] a hyperproperty H when P linked with any (adversarial) context satisfies H. Armed with this notion of robust satisfaction of hyperproperties, we define secure compilation as preserving the robust satisfaction of a class of hyperproperties $\mathcal{H}$, so if a partial source program P robustly satisfies a hyperproperty $H \in \mathcal{H}$ (wrt. all source contexts) then its compilation $P{\downarrow}$ must also robustly satisfy H (wrt. all low-level contexts).

We study the preservation of robust satisfaction for various classes of hyperproperties, many of which are mentioned in Figure 1, and which include all hyperproperties, subset-closed hyperproperties, safety hyperproperties, trace properties, and safety properties. For each such class we propose an equivalent "property-free" characterization of secure compilation that is generally better suited for proofs. For instance, we prove that preserving all hyperproperties robustly can be equivalently stated as the following criterion we call *hyper-robust compilation*:

$$\forall P. \forall C_T. \exists C_S. \forall t.\ C_T[P{\downarrow}] \Downarrow t \iff C_S[P] \Downarrow t$$

This requires that, given a program P, each target context C_T can be mapped to a source context C_S in a way that perfectly preserves the set of traces produced when linking with P and $P{\downarrow}$ respectively. On the other hand, preserving all trace properties robustly is equivalent to the following *robust compilation* criterion:

$$\forall P. \forall C_T. \forall t. \exists C_S.\ C_T[P{\downarrow}] \Downarrow t \Rightarrow C_S[P] \Downarrow t$$

Compared to the previous definition, the $\exists C_S$ and $\forall t$ quantifiers in this definition are swapped and the implication is in just one direction: Each (bad) trace in the target can be emulated using a different source context C_S. As a final example, preserving only safety properties robustly [15] is equivalent to the following *robustly safe compilation* criterion:

$$\forall P. \forall C_T. \forall t.\ C_T[P{\downarrow}] \Downarrow t \Rightarrow \forall m \leq t. \exists C_S.\ C_S[P] \Downarrow m$$

Here only the (bad) finite prefixes m of a potentially infinite trace t in the target need to be back-simulated in the source.

Even the strongest of our secure compilation criteria, hyper-robust compilation, which is, as explained above, equivalent to the robust preservation of all hyperproperties, seems achievable. We plan to demonstrate this by adapting a recent fully abstract translation of a simply typed λ-calculus into the untyped λ-calculus [12]. For this to be interesting, we first extend the two λ-calculi with a notion of trace by adding inputs from and outputs to the environment. For achieving hyper-robust compilation we also extend the source language with recursive types. This allows us to encode the unitype of untyped λ-calculus values using recursive, product, and sum types, allowing for a precise back-translation of contexts. We expect that the logical relation proof technique of Devriese et al. [12] can be adapted to prove the hyper-robust compilation of their translation. Moreover, if we drop recursive types from the source we expect to still be able to use the approximate back-translation of Devriese et al. [12] to show robust hypersafety preservation, a weaker security criterion.

While preserving hyperproperties that are not safety seems to require powerful context back-translation techniques, for preserving safety hyperproperties translating each finite trace prefix individually back to a source context is also possible. This could potentially be simpler as it can benefit from proof techniques that are based on trace semantics [17], which was

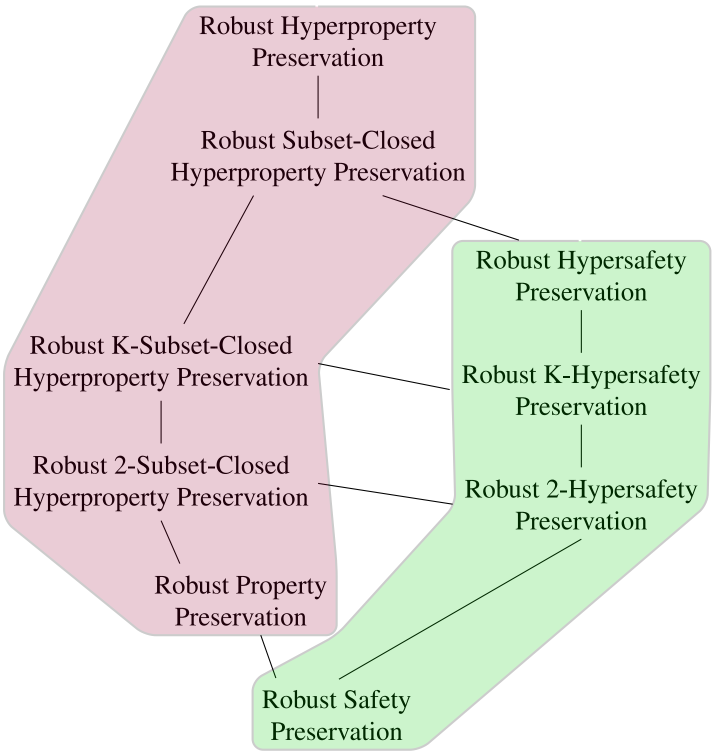
Robust Hyperproperty Preservation
Robust Subset-Closed Hyperproperty Preservation
Robust K-Subset-Closed Hyperproperty Preservation
Robust 2-Subset-Closed Hyperproperty Preservation
Robust Property Preservation
Robust Hypersafety Preservation
Robust K-Hypersafety Preservation
Robust 2-Hypersafety Preservation
Robust Safety Preservation

Figure 1: Different notions of robust hyperproperty preservation. Notions higher in the figure are stronger.

[2]While often full abstraction is defined to also include the reflection of observational equivalence, we choose to ignore that direction here and focus solely on the preservation direction.

also used in the context of full abstraction proofs [18,25]. In Figure 1 we mark in green the secure compilation criteria for which mapping finite trace prefixes is possible, and in light purple the ones for which it is not.

Finally, the property-free characterizations of all classes of hyperproperties from Figure 1 have quantifier alternation of the form $\forall P. \forall C_T \ldots \exists C_S \ldots$, so a (constructive) proof that a compiler satisfies such a characterization can define C_S as a function of the source program P and the target context C_T. While the dependence of C_S on C_T is essential in most cases, the dependence of C_S on P is necessary only when the low-level target language allows the context to make observations that the source does not allow. For example, the target language may have reflection but the source language may not have it. However, in many cases, this kind of an abstraction mismatch does not exist and, in fact, many existing proof techniques [24], including the aforementioned context back-translation techniques, construct C_S only from C_T, independent of the source program P. This begs the question of what kinds of properties are actually preserved by a compiler that satisfies a *stronger* criterion of the form $\forall C_T. \exists C_S. \forall P \ldots$. For example, what properties of source programs are preserved by a compiler that satisfies the following stronger variant of hyper-robust compilation?

$$\forall C_T. \exists C_S. \forall P. \forall t.\ C_T[P{\downarrow}] \Downarrow t \iff C_S[P] \Downarrow t$$

We conjecture that such strong soundness criteria correspond to the robust preservation of *relational* properties of programs. In particular, the criterion listed above implies full abstraction, which is the robust preservation of a specific relational property, namely, observational equivalence. In fact, we expect that even the strong soundness criterion corresponding to robust 2-hypersafety preservation will imply full abstraction.

2 Formally Secure Compilation of Unsafe Low-Level Components

Computer systems are distressingly insecure. Visiting a website, opening an email, or serving a client request often suffice to open the door to control-hijacking attacks. These devastating low-level attacks typically exploit memory-safety vulnerabilities such as buffer overflows, use-after-frees, or double frees, which are abundant in large software systems.

Various techniques have been proposed for guaranteeing memory safety [5,23], but the challenges of efficiency, precision, scalability, backwards compatibility, and effective deployment have hampered their widespread adoption. Meanwhile, new mitigation techniques aim at dealing with the most onerous consequences of memory unsafeness. In particular, *compartmentalization* offers a strong, practical defense against low-level attacks exploiting memory unsafeness [8,16,32]. At least three compartmentalization technologies are widely deployed: process-level privilege separation [8,16,19] (used, e.g., in OpenSSH [28] and for sandboxing plugins and tabs in modern web browsers [29]), software fault isolation [31] (provided, e.g., by Google Native Client [33]), and hardware enclaves (e.g., Intel SGX); many more are on the drawing boards [5,10,26,32].

Such low-level compartmentalization mechanisms are well suited for building more secure compiler chains. In particular, they can be exposed in unsafe low-level languages like C and be targeted by their compiler chains to enable efficiently breaking up large applications into mutually distrustful components that run with minimal privileges and that can interact only via well-defined interfaces. Intuitively, protecting each component from all the others should have strong security benefits: the compromise of some components should not compromise the security of the whole application.

What, exactly, *are* the formal security guarantees one can obtain from such secure compiler chains? To answer this question, we start from *robust compilation* [14], a recently proposed formal criterion for secure compilation, which implies the preservation of all trace properties even against adversarial contexts. These traces are normally built over events such as inputs from and outputs to the environment [21]. We write $P \Downarrow t$ to mean that the complete program P can produce trace t with respect to some operational semantics. Armed with this, robust compilation is formally stated as:

$$\forall P\ C_T\ t.\ C_T[P{\downarrow}] \Downarrow t \Rightarrow \exists C_S.\ C_S[P] \Downarrow t$$

For any partial source program P and any (adversarial) target context C_T where C_T linked with the compiled variant of P can produce a (bad) trace t in the target language (written $C_T[P{\downarrow}] \Downarrow t$), we can construct a(n adversarial) source-level context C_S that can produce trace t in the source language when linked with P (i.e., $C_S[P] \Downarrow t$). Intuitively, any attack trace t that context C_T can mount against $P{\downarrow}$ can already be mounted against P by some source language context C_S. Conversely, any trace property that holds of P when linked with any arbitrary source context will still hold for $P{\downarrow}$ when linked with an arbitrary target context.

In this work, we propose a new formal criterion for secure compilation that extends robust compilation to protecting mutually distrustful components against each other in an unsafe low-level language with C-style undefined behavior. The characterization of robust compilation above does not directly apply in this setting, since it assumes the source language is safe and P cannot have undefined behavior. The natural way to adapt robust compilation to an unsafe source language is the following:

$$\forall P\ C_T\ t.\ C_T[P{\downarrow}] \Downarrow t \Rightarrow \exists C_S\ t'.\ C_S[P] \Downarrow t' \wedge t' \preccurlyeq_P t$$

Instead of requiring that $C_S[P]$ perform the entire trace t, we also allow it to produce a finite prefix t' that ends with an undefined behavior in P (which we write as $t' \preccurlyeq_P t$). Intuitively, since we want to reason only in terms of safe source contexts we do not allow C_S to exhibit undefined behaviors. However, even a safe context can sometimes trigger an undefined behavior in the protected program P, in which case there is no way to keep protecting P going forward. However, P is fully protected until it receives an input that causes undefined behavior. This is a good step towards a model of *dynamic compromise*.

We show that this can be extended to support *mutual distrustful components*. We start by taking both partial programs and contexts to be sets of components and plugging a program in a context to be linking. We compile sets of components by separately compiling each component. We start with all components being uncompromised and incrementally replace any component that exhibits undefined behavior in the source with an arbitrary safe component that will now attack the remaining uncompromised components. Formally, this is captured by the following property:

$$\begin{array}{l}
\{C_1, ..., C_n\}{\downarrow}{\Downarrow}\, t \Rightarrow \exists A_{i_1}, ..., A_{i_m}. \\
\quad (1)\ (\{C_1, ..., C_n\} \backslash \{C_{i_1}, ..., C_{i_m}\} \cup \{A_{i_1}, ..., A_{i_m}\}) \Downarrow t \\
\quad (2)\ \forall j \in 1...m.\ \exists t' \prec_{C_{i_j}} t. \\
\quad (\{C_1, ..., C_n\} \backslash \{C_{i_1}, ..., C_{i_j - 1}\} \cup \{A_{i_1}, ..., A_{i_j - 1}\}){\downarrow}{\Downarrow}\, t'
\end{array}$$

What this is saying is that each low-level trace t of a compiled set of components $\{C_1, ..., C_n\}{\downarrow}$ can be reproduced in the source language after replacing the compromised components $\{C_{i_1}, ..., C_{i_m}\}$ with source components $\{A_{i_1}, ..., A_{i_m}\}$ (part 1). Moreover (part 2), $C_{i_1}, ..., C_{i_m}$ precisely characterizes a compromise sequence, in which each component C_{i_j} is taken over by the already compromised components at that point in time $\{A_{i_1}, ..., A_{i_j - 1}\}$. Above we write $t' \prec_{C_{i_j}} t$ for expressing that trace t' is a prefix of t ending with an undefined behavior in C_{i_j}.

This new definition allows us to play an iterative game in which each component is protected until it receives an input that triggers an undefined behavior, causing it to become compromised and to attack the remaining uncompromised components. This is the first security definition in this space to support both dynamic compromise and mutual distrust, whose interaction is subtle and has eluded previous attempts at characterizing the security guarantees of compartmentalizing compilation as extensions of fully abstract compilation [18]. We further show that this new security definition can be obtained by repeatedly applying an instance of the simpler robust compilation definition above that is phrased only in terms of a program and a context.

3 Micro-policies for C

We will extend the semantics of C with support for tag-based reference monitoring. These tag-based monitors–i.e., high-level micro-policies–will be written in rule-based domain-specific languages (DSLs) inspired by our rule format for micro-policies monitoring machine code [3, 4, 13]. Some parts of the micro-policy DSLs for C and machine code will be similar: for instance, we want a simple way to define the structure of tags using algebraic datatypes, sets, and maps. The kinds of tags differs from level to level though: at the machine code level we have register, program counter, and memory tags, while in C we could replace register tags with value and procedure tags. The way tags are checked and propagated also differs significantly between levels. At the machine-code level, propagation is done via rules that are invoked on each instruction, while in C we have many different operations that can be monitored, e.g., primitive operations, function calls and returns etc. Moreover, the tags of C values could be propagated automatically as values are copied around, without needing to write explicit rules for that. A continuation of this task would be to translate micro-policies from the C to the machine-code level.

4 Secure micro-policy composition

While very useful in practice, secure composition of micro-policies [4] is difficult, as one policy's interaction with the code can break another policy's guarantees. For example, if we wish to enforce information-flow control (IFC) together with other policies, observing the tags of these other policies can reveal sensitive information and thus break the noninterference property established for the IFC micro-policy in isolation. There are several ways in which we plan to approach this problem.

The first builds on the idea of *vertical sequential composition*: 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. We plan to use ideas from monad transformers [22] and algebraic effects [6] to lift this limitation.

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 services and error handlers. Basically, all other micro-policies' routines 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 returning a value's tag in another micro-policy is as classified as the original value. For this more flexible kind of composition there is additional verification effort when composing policies, and this effort scales quadratically with the number of composed policies.

5 Verified Low*-Vale interoperability

In another thread of work, we would like to investigate the interoperability between Low* [27], a shallow embedding of a safe subset of C in F*, and Vale [9], a deep embedding of assembly for various architectures in F* [2, 30]. Low* and Vale are targeted at verification of security critical code (e.g., an implementation of the TLS protocol [7]). As a first step we want to give a formal semantics to programs mixing Low* and Vale (i.e., C and assembly) and defining what it means for a program combining Low* parts and Vale parts to satisfy an F* specification. This interoperability model will capture in particular the C calling conventions and data representations, as well as various other invariants the Vale code should verifiably respect.

As a second step we want to prove a joint security theorem about a combined program with: (1) Low* parts that achieve secrecy using type abstraction and (2) Vale parts achieves secrecy using taint tracking and that can be given relational specifications in F*. This is a combination of separate theorems we previously proved for complete Low* programs and complete Vale programs.

As a third step we would like to prove that replacing Low* code with Vale code for efficiency is semantically justified. In particular we want to prove that replacing a Low* function with a Vale function preserves correctness and security, provided we prove that the Vale function has the reified version of the Low* function as a specification (for correctness) or as a relational specification (for security).

References

[1] M. Abadi. Protection in programming-language translations. *Secure Internet Programming*. 1999.

[2] D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. Dijkstra monads for free. *POPL*. 2017.

[3] 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. *POPL*. 2014.

[4] A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-Policies: Formally verified, tag-based security monitors. *Oakland S&P*. 2015.

[5] A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: Formally verified, tag-based security monitors. *Oakland S&P*. 2015.

[6] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *JLAMP*, 84(1):108–123, 2015.

[7] K. Bhargavan, B. Bond, A. Delignat-Lavaud, C. Fournet, C. Hawblitzel, C. Hriţcu, S. Ishtiaq, M. Kohlweiss, R. Leino, J. Lorch, K. Maillard, J. Pang, B. Parno, J. Protzenko, T. Ramananandro, A. Rane, A. Rastogi, N. Swamy, L. Thompson, P. Wang, S. Zanella-Béguelin, and J.-K. Zinzindohoué. Everest: Towards a verified, drop-in replacement of HTTPS. *SNAPL*, 2017.

[8] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. *USENIX NSDI*, 2008.

[9] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. In *Proceedings of the USENIX Security Symposium*, 2017.

[10] D. Chisnall, C. Rothwell, R. N. M. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. *ASPLOS*. 2015.

[11] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[12] D. Devriese, M. Patrignani, and F. Piessens. Fully-abstract compilation by approximate back-translation. *POPL*, 2016.

[13] 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. *ASPLOS*. 2015.

[14] D. Garg, C. Hriţcu, M. Patrignani, M. Stronati, and D. Swasey. Robust hyperproperty preservation for secure compilation (extended abstract). arXiv:1710.07309, 2017.

[15] A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *JCS*, 12(3-4):435–483, 2004.

[16] K. Gudka, R. N. M. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson. Clean application compartmentalization with SOAAP. *CCS*. 2015.

[17] A. Jeffrey and J. Rathke. Java Jr: Fully abstract trace semantics for a core Java language. *ESOP*. 2005.

[18] Y. Juglaret, C. Hritcu, A. Azevedo de Amorim, B. Eng, and B. C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. *CSF*, 2016.

[19] D. Kilpatrick. Privman: A library for partitioning applications. *USENIX FREENIX*. 2003.

[20] O. Kupferman and M. Y. Vardi. Robust satisfaction. *CONCUR*. 1999.

[21] X. Leroy. A formally verified compiler back-end. *JAR*, 43(4):363–446, 2009.

[22] S. Liang, P. Hudak, and M. P. Jones. Monad transformers and modular interpreters. *POPL*. 1995.

[23] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Everything you want to know about pointer-based checking. *SNAPL*. 2015.

[24] M. S. New, W. J. Bowman, and A. Ahmed. Fully abstract compilation via universal embedding. *ICFP*. 2016.

[25] M. Patrignani and D. Clarke. Fully abstract trace semantics for protected module architectures. *Computer Languages, Systems & Structures*, 42:22–45, 2015.

[26] M. Patrignani, D. Devriese, and F. Piessens. On modular and fully-abstract compilation. *CSF*, 2016.

[27] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in F*. *ICFP*, 2017. To appear.

[28] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*. 2003.

[29] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. *EuroSys*. 2009.

[30] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *POPL*. 2016.

[31] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SOSP*, 1993.

[32] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. *IEEE S&P*, 2015.

[33] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: a sandbox for portable, untrusted x86 native code. *CACM*, 53(1):91–99, 2010.