Formulaire 4 — PROGRAMME DE RECHERCHE
Form 4 — RESEARCH PROGRAM

Efficient Formally Secure Compilation

Abstract 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 is an ERC-funded project aimed at leveraging emerging hardware capabilities for fine-grained protection to build the first efficient secure compilation chains for realistic low-level programming languages (the C language, and Low⋆ [110] a safe subset of C embedded in F⋆ [119] for verification). These compilation chains 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 compilation chains target a *tagged architecture* [22, 54], which associates a metadata tag to each word and efficiently propagates and checks tags according to a software-defined micro-policy. We use property-based testing and formal verification to provide high confidence that our compilers are indeed secure. Formally, we construct machine-checked proofs in Coq of various new security criteria, which are defined as the preservation of property classes even against an adversarial context. These strong criteria complement compiler correctness and ensure that no machine-code attacker can do more harm to securely compiled components than a source-level attacker already could with respect to a secure semantics for the source language.

Context: Low-level attacks on computer systems are a severe security problem. Today's computer systems are distressingly insecure. This affects the foundation upon which today's information society is built and makes everyone potentially vulnerable. Visiting a website, opening an email, or serving a client request is often enough to cause a computer to be compromised by a cyber-attack giving remote attackers full control. This often results in the disclosure or destruction of information and the use of the machine in further cyber-attacks. Hundreds of thousands of compromised computers are hoarded into "botnets" that are used to send spam, mount distributed denial of service attacks, or mine cryptocurrency. Given their cyber-attack power, previously unknown ("0-day") exploitable low-level vulnerabilities in widely-used software are often sold to intelligence agencies or botnet "controllers" for tens to hundreds of thousands of dollars.

The causes for this dissatisfying state of affairs are complex, but at this point mostly historical: our programming languages, compilers, and architectures were designed in an era of scarce hardware resources and far too often trade off security for efficiency. Today's mainstream low-level languages, C and C++, give up even on the most basic safety checks for the sake of efficiency, which leaves programmers bearing all the burden for security: the smallest mistake in widely-deployed C and C++ code can cause security vulnerabilities with disastrous consequences [58]. Over the last 12 years, around 70 percent of all patched security bugs at Microsoft were memory safety issues [96, 42]. The C and C++ languages do not guarantee memory safety because currently deployed hardware provides no good support for it and software checks would incur 70-80% overhead on average [100, 99]. Instead, much weaker low-overhead mitigation techniques are deployed and routinely circumvented by practical attacks [120, 60, 45]. Unfortunately, just ensuring memory safety would in fact not be enough to make C and C++ safe, as the standards and compilers for these languages call out a much larger number of undefined behaviors [68, 83], for which compilers produce code that behaves arbitrarily, often leading to security vulnerabilities, including invalid unchecked type casts [57, 67], data races, and sometimes even integer overflows.

Safer languages such as Java, C#, ML, Haskell, or Rust provide type and memory safety by default as well as many useful abstractions for writing more secure code (e.g., modules, interfaces, parametric polymorphism, etc). Unfortunately though, these languages are still not immune to low-level attacks. All the safety guarantees of these languages are lost when interacting with low-level code, for instance when using low-level libraries. This interaction is useful but dangerous because the low-level code can be malicious or compromised (e.g., by a buffer overflow). Currently, not only is the low-level code trusted to be safe, but also to preserve all the complex abstractions and internal invariants of the high-level language semantics, compiler, and runtime system. So even if some critical code is secure with respect to the semantics of a high-level language, any low-level code with which it interacts can break its security.

Verification languages such as Coq and F⋆ [119] provide additional abstractions, such as dependent types, logical pre- and postconditions, and tracking of side effects, e.g., distinguishing pure from stateful computations. Such abstractions are crucial for making the verification effort more tractable in practice, but they also make the final verification result only valid in these very abstract languages. In order for a Coq or F⋆ program to be executed it is first compiled all the way down to machine code. Even if such compilation is correct [91, 84], this is usually not enough to ensure the security of the verified code, since usually not all the code can be written and verified in the abstract verification language.

For a concrete example, consider the miTLS⋆ implementation of the TLS standard, the most widely-used security protocol framework on the Internet. miTLS⋆ is being written and formally verified in Low⋆, a safe subset of C embedded in F⋆ [110, 37, 128]. miTLS⋆ includes tens of thousands lines of Low⋆ code, and even when all this code will be formally verified, it will

be just a tiny library linked from large unverified applications such as web browsers, web servers, and operating systems, which have millions of lines of C, C++, and assembly code. Not only are these applications not verified and can thus break the verified security properties of the Low⋆ code, but these applications are not even memory safe, and any error can allow remote attackers to take complete control, disclose the memory of the process stealing the TLS private keys, etc. A correct compilation chain is not enough in this case, since (1) a correct compilation chain [91] for an insecure language like C still produces insecure code and leaves the burden of avoiding undefined behaviors to the programmer, and (2) a correct compilation chain does not protect the interaction between high-level and low-level code and does not enforce the abstractions of each language against faulty or malicious code written in the lower-level languages. In order for miTLS⋆ to be secure in practice we don't need only correct compilation, but also *secure compilation*.

Secure compilation We call a compilation chain secure when it prevents or soundly mitigates low-level attacks on compiled code and it allows sound reasoning about security in the source language. Secure compilation (1) gives all programs in the source language a secure semantics (e.g., ensuring memory safety, component isolation, defined behavior, etc.) and (2) protects compiled programs from their interaction with unsafe low-level code.

SECOMP grand challenge: *build the first efficient formally secure compilers for realistic programming languages.* For this we will devise secure compiler chains from Low⋆ [110] to C, and from C to RISC-V [18] assembly. These compiler chains will provide security for programs written in a combination of Low⋆, C, and assembly, by providing more secure semantics to C programs and by protecting the abstractions of each language against attacks from the lower-level ones. To achieve this level of security without sacrificing efficiency, our compiler chains will target our micro-policies tagged architecture [22]. For measuring and optimizing efficiency we will use standard benchmark suites [69] and realistic source programs, with miTLS⋆ as the main end-to-end case study. In order to ensure high confidence in the security of our compiler chains, we will thoroughly test them using property-based testing [104] and then formally verify their security using Coq.

Further Studying Secure Compilation Criteria As discussed in Form 3, Fiche 2, so far most the work on this project has been focused on devising secure compilation criteria based on preserving classes of properties against adversarial contexts [81, 8] and extending these criteria to unsafe languages [7, 72]. Several interesting open problems remain in this space, including extending our component model [7] with *dynamic component creation*. This would make crucial use of our current dynamic compromise model, since components would no longer be statically known, and thus static compromise would not apply at all. This extension could allow us to move from our current "code-based" compartmentalization model to a "data-based" one [66], e.g., one compartment per incoming network connection or per browser tab.

Formally Secure Compartmentalization for C We plan to base our compartmentalizing compilation chain on the CompCert C compiler. Scaling up to the whole of C will certainly entail challenges such as defining a variant of C with components and efficiently enforcing compartmentalization all the way down using micro-policies. To achieve this, we will build on the solid foundation of our recent work [7]: the formal security criterion, the scalable proof technique, and the proof-of-concept secure compilation chain. As a first step, we plan to extend our prototype to allow sharing memory between components. Since we already allow arbitrary reads at the lowest level, it seems appealing to also allow external reads from some of the components' memory in the source. The simplest would be to allow certain static buffers to be shared with all other components, or only with some if we also extend the interfaces. More ambitious would be to allow pointers to dynamically allocated memory to be passed to other components, as a form of read capabilities. This would make pointers appear in traces and we will need to take into account that these pointers will vary at the different levels in our compilation chain. Moreover, each component produced by the back-translation would need to record all the read capabilities it receives for later use. Finally, to safety allow write capabilities we could combine compartmentalization with memory safety. This will give us a fine-grained object-capability model [123] on a fully generic tagged architecture and will enable compartmentalized applications that are much more granular and secure than using currently-deployed isolation techniques [113, 66, 126].

Dynamic Privilege Notions Our prototype compilation chain [7] uses a very simple notion of interface to statically restrict the privileges of components. This could, however, be extended with dynamic notions of privilege such as capabilities and history-based access control [1]. In one of its simplest forms, allowing pointers to be passed between components and then used to write data, as discussed above, would already constitute a dynamic notion of privilege, that is not captured by the static interfaces, but nevertheless needs to be enforced, in this case using some form of memory safety.

Memory Safety for C We also plan to enforce memory safety for C and its interactions with untrusted RISC-V assembly. This will protect buggy programs from malformed inputs that would normally trigger undefined behavior. Enforcing memory safety requires changes to the C compiler and a sophisticated micro-policy, which extends our simple heap memory safety policy [23, 54, 19] to additionally deal with unboxed structs, stack allocation [115], byte addressing, unaligned memory accesses, custom allocators, etc. We plan to build an extension of CompCert [91] that is memory safe. To verify security we will target both properties describing the absence of spatial (e.g., buffer overflows) and temporal (e.g., use after free, double free) memory safety violations [100] and our higher-level reasoning principles enabled by memory safety [24].

Verifying Compartmentalized Applications. It would also be interesting to build verification tools based on the source reasoning principles provided by our secure compilation criteria [7] and to use these tools to analyze the security of practical compartmentalized applications. Effective verification, however, require good ways for reasoning about the exponential number of dynamic compromise scenarios. One idea is to do our source reasoning with respect to a variant of our partial semantics, which would use nondeterminism to capture the compromise of components and their possible successive

actions. Correctly designing such a partial semantics for a complex language is, however, challenging. Fortunately, our secure compilation criteria provide a more basic, low-TCB definition against which to validate any fancier reasoning tools, like partial semantics [78], program logics [79], logical relations [52], etc.

Secure compilation of Low* to C using Components, Contracts, and Sealing We also plan to devise a secure compilation chain from Low* to C. Low* programs are verified with respect to Hoare-style pre- and post- conditions to achieve correctness and use the F* module system (i.e., data abstraction and parametricity) to achieve confidentiality of secret data, even against certain side-channels. These high-level abstractions will have to the protected at the C level, and while compartmentalization will offer a first barrier of defense, more work will be needed. We plan to enforce specifications by turning them into dynamic contracts and parametricity by relying on dynamic sealing. We hope that micro-policies can help us implement both contracts and sealing efficiently.

Micro-policies for C Micro-policies operate at the lowest machine-code level. While this is appropriate for devising secure C compilers, we also want our secure Low* to C compiler to directly make use of micro-policies in order to efficiently enforce the high-level abstractions of Low*. Moreover, we want a general solution that is not tied to our compilation chain, but instead allows arbitrary programs in C to benefit from efficient programmable tag-based monitoring. Exposing micro-policies in C and then translating them down is challenging, because the structure of programs in these languages is different than that of machine code. We will extend the semantics of C with support for tag-based reference monitoring. These 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 [20, 23, 54]. 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. Finally, we want to automatically translate C micro-policies to machine-code ones.

Secure micro-policy composition Our secure compilation chains require composing many different micro-policies. For instance, we need to simultaneously enforce isolation of mutual distrustful components and memory safety for some of the components. Recent microarchitectural optimizations enable us to efficiently enforce multiple micro-policies simultaneously [54], by taking tags to be tuples, where each tag component is handled by a different sub-policy. Yet composing isolation and memory safety is non-trivial, since each of them has its own view on memory, and a naive composition would be dysfunctional, for instance dynamically allocating in the memory of the wrong component. While this problem can be fixed by changing the code of the composed micro-policies, the bigger conceptual difficulty is in composing specifications and security proofs. Secure composition principles are badly needed, since verifying each composed micro-policy from scratch does not scale. Secure composition is, however, very difficult to achieve in our setting, because micro-policies can directly influence the monitored code by answering to direct calls and by raising catchable exceptions, and thus one micro-policy's observable behavior can break the other micro-policy's guarantees. We will study several techniques for composing micro-policy specifications and proofs, with the composite policies needed by our secure compilation chains as motivating examples. First, we will investigate layering micro-policies, by choosing an order among them and constructing a sequence of abstract machines, each of which "virtualizes" the tagging mechanisms in the hardware so that further micro-policies can be implemented on top. We will then use ideas from monad transformers and algebraic effects to allow the micro-policies to be verified separately and layered in any order. Finally, we will investigate forms of composition in which each micro-policy specifies how its tags should be affected by the interactions of the other policies with the monitored code.

Preserving Confidentiality and Hypersafety It would be interesting to extend our secure compartmentalizing compilation criterion and enforcement mechanisms [7] from robustly preserving safety to confidentiality and hypersafety [8]. For this we need to control the flow of information at the target level—e.g., by restricting direct reads and read capabilities, cleaning registers, etc.This becomes very challenging though, in a realistic attacker model in which low-level contexts can observe time. While at first we could assume that low-level contexts cannot exploit such side-channels, an interesting challenge will be to try to extend our enforcement to also protect against timing side-channels. In this context, we could investigate preserving various K-Safety Hyperproperties such as nonmalleable information flow control [41], timing-sensitive noninterference [111], and cryptographic "constant time" [34] (i.e. secret independent timing).

Research Group, Collaborators, and Community A problem of this size cannot be solved by a single person. I will continue to use my ERC and other grants to fund PhD students and PostDocs to work with me on various pieces. I will further rely on my current collaborations (described in Form 1) and try to establish new ones where meaningful. For instance, I will try to establish a network of French researchers interested in various approaches to making compilers like CompCert more secure, including for instance Xavier Leroy (Collège de France); Frederic Besson, Sandrine Blazy, Thomas Jensen, and David Pichardie (Inria Rennes); as well as Benjamin Grégoire and Tamara Tezk (Inria Sophia Antipolis). Finally, I will continue my effort to build a broader community around secure compilation, by organizing international workshops (like PriSC) and meetings (like Dagstuhl Seminar 20201), and by teaching and trying to build reusable teaching materials.

Bibliographie / *Bibliography*

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*. The Internet Society, 2003.

[2] M. Abadi. Protection in programming-language translations. In *Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*. 1999.

[3] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *J. ACM*, 52(1):102–146, 2005.

[4] M. Abadi and L. Cardelli. *A theory of objects*. Springer, 1996.

[5] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1):37–83, 2002.

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

[7] C. Abate, A. Azevedo de Amorim, R. Blanco, A. N. Evans, G. Fachini, **C. Hriţcu**, T. Laurent, B. C. Pierce, M. Stronati, and A. Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. In *25th ACM Conference on Computer and Communications Security (CCS)*. October 2018.

[8] C. Abate, R. Blanco, D. Garg, **C. Hriţcu**, M. Patrignani, and J. Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. arXiv:1807.04603, July 2018.

[9] A. Aguirre, **C. Hriţcu**, C. Keller, and N. Swamy. From F* to SMT (extended abstract). Talk at 1st International Workshop on Hammers for Type Theories (HaTT), July 2016.

[10] D. Ahman, C. Fournet, **C. Hriţcu**, K. Maillard, A. Rastogi, and N. Swamy. Recalling a witness: Foundations and applications of monotonic state. *PACMPL*, 2(POPL):65:1–65:30, January 2018.

[11] D. Ahman, **C. Hriţcu**, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. Dijkstra monads for free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. January 2017.

[12] A. Ahmed. Verified compilers for a multi-language world. In *1st Summit on Advances in Programming Languages*, volume 32 of *LIPIcs*. 2015.

[13] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. In *16th International Conference on Functional Programming*. 2011.

[14] A. J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[15] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[16] M. Amy, M. Roetteler, and K. M. Svore. Verified compilation of space-efficient reversible circuits. In R. Majumdar and V. Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*. 2017.

[17] A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.

[18] K. Asanović and D. A. Patterson. Instruction sets should be free: The case for RISC-V. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, Aug 2014.

[19] A. Azevedo de Amorim. *A methodology for micro-policies*. PhD thesis, University of Pennsylvania, 2017.

[20] 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)*. January 2014.

[21] 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. *Journal of Computer Security (JCS); Special Issue on Verified Information Flow Security*, 24(6):689–734, December 2016.

[22] 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. In *36th IEEE Symposium on Security and Privacy (Oakland S&P)*. May 2015.

[23] 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. In *36th IEEE Symposium on Security and Privacy (Oakland S&P)*. May 2015.

[24] A. Azevedo de Amorim, **C. Hriţcu**, and B. C. Pierce. The meaning of memory safety. In *7th International Conference on Principles of Security and Trust (POST)*, April 2018.

[25] M. Backes, A. Busenius, and **C. Hriţcu**. On the development and formalization of an extensible code generator for real life security protocols. In *4th NASA Formal Methods Symposium (NFM)*. April 2012.

[26] M. Backes, M. P. Grochulla, **C. Hriţcu**, and M. Maffei. Achieving security despite compromise using zero-knowledge. In *22th IEEE Symposium on Computer Security Foundations (CSF 2009)*. July 2009.

[27] M. Backes, **C. Hriţcu**, and M. Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *21th IEEE Symposium on Computer Security Foundations (CSF 2008)*. June 2008.

[28] M. Backes, **C. Hriţcu**, and M. Maffei. Type-checking zero-knowledge. In *15th ACM Conference on Computer and Communications Security (CCS 2008)*. October 2008.

[29] M. Backes, **C. Hriţcu**, and M. Maffei. Union and intersection types for secure protocol implementations. In *Theory of Security and Applications (TOSCA 2011; part of ETAPS and the precursor of POST)*. March 2011. Invited paper.

[30] M. Backes, **C. Hriţcu**, and M. Maffei. Union, intersection, and refinement types and reasoning about type disjointness for secure protocol implementations. *Journal of Computer Security (JCS); Special Issue on Foundational Aspects of Security*, 22(2):301–353, February 2014.

[31] M. Backes, **C. Hriţcu**, and T. Tarrach. Automatically verifying typing constraints for a data processing language. In *First International Conference on Certified Programs and Proofs (CPP 2011)*. December 2011.

[32] M. Backes, M. Maffei, and E. Mohammadi. Computationally sound abstraction and verification of secure multi-party computations. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*, volume 8, 2010.

[33] D. Baelde, S. Delaune, and L. Hirschi. A reduced semantics for deciding trace equivalence. *Logical Methods in Computer Science*, 13(2), 2017.

[34] G. Barthe, B. Grégoire, and V. Laporte. Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In *31st IEEE Computer Security Foundations Symposium (CSF)*. 2018.

[35] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *Proc. 21th IEEE Symposium on Computer Security Foundations (CSF)*, 2008.

[36] 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. In *2nd Summit on Advances in Programming Languages (SNAPL)*, May 2017.

[37] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella Béguelin, and J. K. Zinzindohoue. Implementing and proving the TLS 1.3 record layer. *IEEE Security & Privacy*, 2017.

[38] G. M. Bierman, A. D. Gordon, **C. Hriţcu**, and D. Langworthy. Semantic subtyping with an SMT solver. In *15th ACM SIGPLAN International Conference on Functional programming (ICFP 2010)*. September 2010.

[39] G. M. Bierman, A. D. Gordon, **C. Hriţcu**, and D. Langworthy. Semantic subtyping with an SMT solver. *Journal of Functional Programming (JFP)*, 22(1):31–105, March 2012.

[40] E. F. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proc. 11th ACM Conference on Computer and Communications Security*. 2004.

[41] E. Cecchetti, A. C. Myers, and O. Arden. Nonmalleable information flow control. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2017.

[42] C. Cimpanu. Microsoft: 70 percent of all security bugs are memory safety issues. ZDNet, February 2019.

[43] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2000.

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

[45] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015.

[46] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of TACAS*, 2008.

[47] A. DeHon, E. Boling, and **C. Hriţcu**. Techniques for metadata processing, 2017.

[48] A. DeHon, B. Karel, T. F. Knight, Jr., G. Malecha, B. Montagu, R. Morisset, G. Morrisett, B. C. Pierce, R. Pollack, S. Ray, O. Shivers, J. M. Smith, and G. Sullivan. Preliminary design of the SAFE platform. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, PLOS, October 2011.

[49] S. Delaune and L. Hirschi. A survey of symbolic methods for establishing equivalence-based properties in cryptographic protocols. *J. Log. Algebr. Meth. Program.*, 87:127–144, 2017.

[50] M. Dénès, **C. Hriţcu**, L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. QuickChick: Property-based testing for Coq. The Coq Workshop, July 2014.

[51] D. Devriese, M. Patrignani, and F. Piessens. Fully-abstract compilation by approximate back-translation. In *43nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[52] D. Devriese, F. Piessens, and L. Birkedal. Reasoning about object capabilities with logical relations and effect parametricity. In *1st IEEE European Symposium on Security and Privacy*. 2016.

[53] U. Dhawan and A. DeHon. Area-efficient near-associative memories on FPGAs. In *International Symposium on Field-Programmable Gate Arrays, (FPGA2013)*, February 2013.

[54] 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. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. March 2015.

[55] U. Dhawan, A. Kwon, E. Kadric, **C. Hriţcu**, B. C. Pierce, J. M. Smith, G. Malecha, G. Morrisett, T. F. Knight, Jr., A. Sutherland, T. Hawkins, A. Zyxnfryx, D. Wittenberg, P. Trei, S. Ray, G. Sullivan, and A. DeHon. Hardware support for safety interlocks and introspection. In *SASO Workshop on Adaptive Host and Network Security*, September 2012.

[56] C. Dubois, A. Giorgetti, and R. Genestier. Tests and proofs for enumerative combinatorics. In B. K. Aichernig and C. A. Furia, editors, *Tests and Proofs - 10th International Conference, TAP 2016, Held as Part of STAF 2016, Vienna, Austria, July 5-7, 2016, Proceedings*, volume 9762 of *Lecture Notes in Computer Science*. 2016.

[57] G. J. Duck and R. H. C. Yap. EffectiveSan: Type and memory error detection using dynamically typed C/C++. 2018.

[58] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of Heartbleed. In C. Williamson, A. Akella, and N. Taft, editors, *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*. 2014.

[59] F. Eigner. Type-based verification of electronic voting systems. Master's thesis, Saarland University, September 2009.

[60] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. To appear at CCS, 2015.

[61] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*. March 2013.

[62] C. Fournet, N. Swamy, J. Chen, P. Dagand, P. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *Symposium on Principles of Programming Languages, POPL*. 2013.

[63] D. Gallois-Wong. Formalising Luck: Improved probabilistic semantics for property-based generators. Inria Internship Report, August 2016.

[64] J. A. Goguen and J. Meseguer. Security policies and security models. In *Symposium on Security and Privacy*, 1982.

[65] N. Grimm, K. Maillard, C. Fournet, **C. Hriţcu**, M. Maffei, J. Protzenko, T. Ramananandro, A. Rastogi, N. Swamy, and S. Zanella-Béguelin. A monadic framework for relational verification: Applied to information security, program equivalence, and optimizations. In *7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. January 2018.

[66] 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. In *22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015.

[67] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe. TypeSan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[68] C. Hathhorn, C. Ellison, and G. Rosu. Defining the undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 2015.

[69] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[70] **C. Hriţcu**. A step-indexed semantic model of types for the functional object calculus. Master's thesis, Saarland University, May 2007.

[71] **C. Hriţcu**. *Union, Intersection, and Refinement Types and Reasoning About Type Disjointness for Security Protocol Analysis*. PhD thesis, Saarland University, January 2012.

[72] **C. Hriţcu**. *The Quest for Formally Secure Compartmentalizing Compilation*. Habilitation thesis, ENS Paris; PSL Research University, January 2019.

[73] **C. Hriţcu**, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCException are belong to us. In *34th IEEE Symposium on Security and Privacy (Oakland S&P)*. May 2013.

[74] **C. Hriţcu**, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. In *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. September 2013.

[75] **C. Hriţcu**, L. Lampropoulos, A. Spector-Zabusky, A. Azevedo de Amorim, M. Dénès, J. Hughes, B. C. Pierce, and D. Vytiniotis. Testing noninterference, quickly. *Journal of Functional Programming (JFP); Special issue for ICFP 2013*, 26:e4 (62 pages), April 2016.

[76] **C. Hriţcu** and J. Schwinghammer. A step-indexed semantics of imperative objects. *Logical Methods in Computer Science (LMCS)*, 5(4:2):1–48, December 2009.

[77] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. In *24th IEEE Computer Security Foundations Symposium*. 2011.

[78] A. Jeffrey and J. Rathke. Java Jr: Fully abstract trace semantics for a core Java language. In *14th European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*. 2005.

[79] L. Jia, S. Sen, D. Garg, and A. Datta. A logic of programs with interface-confined code. In *IEEE 28th Computer Security Foundations Symposium, CSF*. 2015.

[80] Y. Juglaret, **C. Hriţcu**, A. Azevedo de Amorim, B. Eng, and B. C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In *29th IEEE Symposium on Computer Security Foundations (CSF)*. July 2016.

[81] Y. Juglaret, **C. Hriţcu**, A. Azevedo de Amorim, B. Eng, and B. C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016*, 2016.

[82] Y. Juglaret, **C. Hriţcu**, A. Azevedo de Amorim, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Towards a fully abstract compiler using micro-policies: Secure compilation for mutually distrustful components. *CoRR*, abs/1510.00697, 2015.

[83] R. Krebbers. *The C Standard Formalized in Coq*. PhD thesis, Radboud University Nijmegen, December 2015.

[84] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. 2014.

[85] L. Lamport and F. B. Schneider. Formal foundation for specification and verification. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, 1984.

[86] L. Lampropoulos, D. Gallois-Wong, **C. Hriţcu**, J. Hughes, B. C. Pierce, and L. Xia. *Luck: A Probabilistic Language for Testing*. To appear in Foundations of Probabilistic Programming.

[87] L. Lampropoulos, D. Gallois-Wong, **C. Hriţcu**, J. Hughes, B. C. Pierce, and L. Xia. Beginner's Luck: A language for random generators. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. January 2017.

[88] L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. Generating good generators for inductive relations. *PACMPL*, 2(POPL):45:1–45:30, 2018.

[89] L. Lampropoulos and B. C. Pierce. *Software Foundations: QuickChick: Property-Based Testing in Coq*. Electronic textbook, October 2018.

[90] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, Berlin, Heidelberg, 2010.

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

[92] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[93] M. Maffei and K. Pecina. Privacy-aware proof-carrying authorization. In *Fifth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2011)*, April 2011.

[94] K. Maillard, D. Ahman, R. Atkey, G. Martínez, **C. Hriţcu**, E. Rivas, and E. Tanter. Dijkstra monads for all. Draft, March 2019.

[95] G. Martínez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, **C. Hriţcu**, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ramananandro, A. Rastogi, and N. Swamy. Meta-F*: Proof automation with SMT, tactics, and metaprograms. In *28th European Symposium on Programming (ESOP)*, 2019. To appear.

[96] M. Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. BlueHat IL, 2019.

[97] B. Montagu, B. C. Pierce, and R. Pollack. A theory of information-flow labels. In *26th IEEE Computer Security Foundations Symposium (CSF)*. 2013.

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

[99] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Everything you want to know about pointer-based checking. In *1st Summit on Advances in Programming Languages*, volume 32 of *LIPIcs*. 2015.

[100] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In *9th International Symposium on Memory Management*. 2010.

[101] M. S. New, W. J. Bowman, and A. Ahmed. Fully abstract compilation via universal embedding. In *21st ACM SIGPLAN International Conference on Functional Programming, ICFP*, 2016.

[102] Z. Paraskevopoulou and **C. Hriţcu**. A Coq framework for verified property-based testing. Internship Report, Inria Paris-Rocquencourt, September 2014.

[103] Z. Paraskevopoulou, **C. Hriţcu**, M. Dénès, L. Lampropoulos, and B. C. Pierce. A Coq framework for verified property-based testing. Workshop on Coq for PL, January 2015.

[104] Z. Paraskevopoulou, **C. Hriţcu**, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. In *6th International Conference on Interactive Theorem Proving (ITP)*, volume 9236 of *Lecture Notes in Computer Science*. 2015.

[105] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems*, 2015.

[106] M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys*, 2019.

[107] B. C. Pierce and L. Lampropoulos. Property-based random testing with QuickChick. Course at DeepSpec Summer School, July 2017.

[108] B. C. Pierce and D. N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

[109] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan. Formally verified cryptographic web applications in WebAssembly. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[110] 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*. *PACMPL*, 1(ICFP):17:1–17:29, September 2017.

[111] W. Rafnsson, L. Jia, and L. Bauer. Timing-sensitive noninterference through composition. In M. Maffei and M. Ryan, editors, *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10204 of *Lecture Notes in Computer Science*. 2017.

[112] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *In Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland) 2014*, 2014.

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

[114] A. Riazanov and A. Voronkov. Vampire. In *Proc. 16th International Conference on Automated Deduction (CADE)*, 1999.

[115] N. Roessler and A. DeHon. Protecting the stack with metadata policies and tagged hardware. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 2018.

[116] B. Sattarzadeh and M. S. Fallah. Typing secure implementation of authentication protocols in environments with compromised principals. *Security and Communication Networks*, 7(11):1815–1830, 2014.

[117] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[118] G. T. Sullivan, S. Chiricescu, A. DeHon, D. Demange, S. Iyer, A. Kliger, G. Morrisett, B. C. Pierce, H. Reubenstein, J. M. Smith, A. Thomas, J. Tov, C. M. White, and D. Wittenberg. SAFE: A clean-slate architecture for secure systems. In *Proceedings of the IEEE International Conference on Technologies for Homeland Security*, November 2013.

[119] 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*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. January 2016.

[120] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy*. 2013.

[121] The Coq development team. *The Coq proof assistant*.

[122] E. Torlak and R. Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2014.

[123] 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. In *IEEE Symposium on Security and Privacy*. 2015.

[124] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In *Automated Deduction – CADE-22 : 22nd International Conference on Automated Deduction*, 2009.

[125] L. Xia and **C. Hriţcu**. Integrating functional logic programming with constraint solving for random generation of structured data. Inria Internship Report, September 2015.

[126] 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. *Communications of the ACM*, 53(1):91–99, 2010.

[127] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. In *ACM Conference on Computer and Communications Security*. 2017.

[128] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. In *CCS*, 2017.