Research Statement

Marco Patrignani

My research goal is to enable the development of secure compilers, i.e., compilers that generate code that adheres to certain security properties.

Compilers are a crucial part of modern software development. They perform various tasks, including that of translating code written in some source, high-level language eventually down to target, low-level code. The language translation phase has primarily been studied in the literature with one goal in mind: preservation of source-level behaviour in the target code [15, 29, 37, 41]. However, more can be achieved by compilation, e.g., security.

Preservation of security-relevant properties through compilation aims at generating compiled code that can withstand target-level attacks. In fact, while source languages often have abstractions to rule out source-level attacks, the same may not hold for target languages [1, 33]. It is thus realistic to assume that attackers can interact with a program in the target language, e.g., by exploiting some system bug such as buffer overflows. Depending on the target language this can enable attacks such as improper stack manipulation, reading/writing private memory, breaking the intended control flow etc.

By definition, a compiler that preserves security is provably resistant to these kinds of attacks. The field of secure compilation aims at developing such secure compilers as well as criteria that lets one prove that the compiler in question is indeed secure.

Concerning the development of secure compilers, two main approaches exist, depending on the target language. If the target language is typed, as is the case in many intermediate languages used by compilers, one may rely on types to rule out attacks [8, 9, 13, 42]. If the target language is untyped, some secure compilers rely on security architectures that operate at the target language level to provide e.g., memory isolation, capabilities, secure tags, etc. [5, 25, 30, 31, 46, 51].

Concerning the criteria for proving a compiler secure, the de-facto standard is proving compiler full-abstraction [1]. This amounts to proving that a compiler preserves and reflects a form of observational equivalence, often contextual equivalence. Different techniques exist for proving compiler full abstraction, based on target-level traces, logical relations, bisimulations etc [9, 19, 25, 30, 31, 46].

Secure Compiler Development

My research was initially motivated by an interest in using the Protected Module Architecture (PMA) security architecture as a target for secure compilation. PMA provides isolation primitives for machine code, effectively creating encapsulated modules at the level of untyped assembly language [56]. The recent Intel SGX is an example of a PMA [10, 38].

We have developed a secure compilation scheme for an object-oriented language as well as for an ML-like language to untyped assembly extended with PMA and proved that it is fully-abstract [7, 35, 45, 46, 50]. Then I extended the object-oriented compiler to handle separate compilation and multiple target modules [51] and we have studied how to devise secure abstract machines

for PMA [36]. In order to reason about target code, I developed a trace-based formal model of PMA [48, 49] as well as a high-level model of an attacker for PMA [34].

PMA, in the Intel SGX implementation, is a security architecture that supports secure compilation and that is closest to mass adoption [10, 38]. To proceed further towards this goal, a challenging research direction is to investigate concurrency support in PMA and how that can be used by a secure compiler for concurrent source-level languages. Secure compilation of concurrent code has been studied both in the distributed setting [2, 3, 4, 14] and for typed assembly languages [12]. However, no concurrent untyped assembly language has been used by secure compilers, yet that is the most widespread kind of target language.

Capability Machines PMA is not the only interesting emerging security architecture. Recent developments in capability machines [28, 39, 60, 61, 62], indicate that they are a feasible target for secure compilation, generating code with a smaller overhead than PMA. Capability machines embody the capability paradigm of Dennis and Van Horn [17], though a vast literature on the subject exists. The idea revolves around the concept that *subjects perform operations on objects*, though expressed at the machine level: an instruction can operate on a memory area if it provides a capability (a permission) to do so. For example, a write to address A succeeds only if the instruction specifies a register containing a capability that allows it to write a memory region comprising A.

While it seems that capability machines could indeed support secure compilation, no such results exist. We have started this line of research by formalising the most promising instance of capability machines: the CHERI architecture [22]. We can already see that capability machines provide better support for secure compilation than existing architectures on two fronts, and this is due to their fine-grained memory control. First, capability machines seem suitable for compiling programs that are verified in separation logic in a more efficient way than existing approaches [6]. Second, we believe capability machines support secure compilation of program-specific fine-grained security policies such as the object capability paradigm[1] and of dynamic security policies. Finally, this research aims to understand the limitations of current capability machines so to propose improvements that are better suited for efficient secure compilation.

Implementation, Optimisation and Benchmarking A number of research trajectories can be explored independently of what security architecture is adopted to support secure compilation. One of these is implementing and benchmarking the compiler. Once implemented, compilers need to be (i) fast and (ii) secure. Concerning (i), compilers often achieve performance through optimisation. Investigating the interaction between code optimisation and security has been recently proposed [21] in a call-to-arms paper but has not been thoroughly carried out. We foresee that studying which optimisations violate which security properties will be a necessary and interesting research trajectory. Concerning (ii), while there exist a number of standard benchmarks to test code performance, no specific benchmarks for secure compilation exist. Specifically, we would like to have security benchmarks that provide a crucial evaluation metric in terms of applicability of secure compilation results by mainstream developers. Secure compilers are studied as formal models, which must abstract from implementation detail and risk missing out on details of the actual implementation. The goal of such security benchmarks would be to provide a suite of tests to highlight weaknesses that

[1] Caja is a JavaScript-based version of object capabilities and Joe-e brings object capabilities to Java. Available respectively here: https://developers.google.com/caja/ and here https://code.google.com/archive/p/joe-e/.

the model abstracted away in order to withstand implementation-specific vulnerabilities. Research in this direction will set a new standard for future secure compiler developers to uphold.

Garbage Collection To develop secure compilers for real-world languages, they have to be integrated with language runtime mechanisms that programmers rely upon. One such mechanism is Garbage Collection (GC). GC is a reality in many mainstream programming languages, though no existing work considers how a secure compiler interacts with a GC. By integrating secure compilation and GC we aim to show that GC can be used in concert with a secure compiler without changing the latter. To achieve this, we have to (i) pick a secure compilation scheme, (ii) choose a viable garbage collection algorithm and (iii) select a proof technique to present the results. Finally, the secure GC must be implemented to test its computational overhead. Concerning (i), object-oriented languages use GC extensively, so we can start from our secure compiler for such a language to PMA [51]. This would also make PMA more interesting from an industrial perspective. Concerning (ii), distributed GC faces challenges similar to those faced by the secure garbage collector we want to study, namely it is unknown when some object can be safely deallocated [45]. To the best of our knowledge, distributed GCs have not been integrated with secure compilers. Concerning (iii), the behaviour of the GC must be captured in a different semantics of the source-level language and compiler security can be expressed as full abstraction wrt these semantics.

Secure Compilation of Advanced Type Systems We envision additional research directions regarding secure compilation of advanced source language features that no existing work has investigated. We discuss only two but anticipate that several such features can be of interest for secure compilation, e.g., enforcing specific properties such as Rust-like ownership.

Many programming languages provide a form of polymorphism (e.g., generics Java), which lets programmers both abstract from the details of specific types and achieve information hiding. A conjecture by Sumii and Pierce [53, 57] stated that (parametric) polymorphism can be securely compiled using encryption. We have recently disproved that conjecture, showing that encryption is not sufficient to securely compile polymorphic code [20]. Thus, an interesting research direction is to study how to strengthen the target language to support secure compilation of polymorphism.

Venturing forth in supporting complex source-level abstractions, secure compilation of dependently-typed languages seems necessary. Dependently-typed intermediate languages seem a necessity for compilers as intermediate languages often need to accommodate the widest array of source-level abstractions [59]. For a compiler using a dependently-typed language it is necessary to be able to compile those type abstractions securely. This further exploration of the lambda-cube (after standard and polymorphic programs) poses a challenging research question: which security mechanism will enable such a secure compiler to be devised?

Integration of Secure Compilation in Software Systems A more long-term research trajectory is integrating the acquired and developed secure compilation notions into existing software systems. Two such examples come to mind: web browsers and wireless sensor networks (WSNs). In web browsers, a great deal of research is devoted to providing more secure browser extensions [24, 58]. Integrating secure compilation to browser (or browser plugin) development seems necessary as a step towards safe internet programming. In WSN, the concern to address is making secure compilation resource efficient and specific to certain security properties. One such example is minimising the overhead introduced by the secure compiler to make it feasible for securely compiled code to be

deployed in small WSN sensors. Additionally, the secure compiler could be tailored only against specific attacks that are WSN pertinent, to limit the overhead of checking extensively for any possible attack (as is currently done in many secure compilers). These are but two examples of how secure compilation research can be merged with different research trajectories to produce interesting results; I expect more to follow with more research experience.

Secure Compilation and Verification A second long-term research trajectory is devising verification techniques for securely-compiled code. Verification of compiled code can: (i) reduce the trusted computing base (TCB) on which the security guarantees rest and (ii) reduce the overhead of securely-compiled code. Concerning (i), presently the compiler is part of the TCB; to remove it, we would like to devise verification techniques that the compiled code adheres to a source-level security specification. Concerning (ii), most of the interaction that occurs with compiled code does not come from a malicious attacker but just from some third-party code. Nevertheless, secure compilers assume anything but the compiled code is malicious and thus insert runtime checks that are always executed. By verifying that some external module complies with secure compilation specifications, these checks can be then skipped with no security breach and no overhead. Proof-carrying code techniques [40] can be used to equip a compiled component with some proof of compliance to a source-level specification. The (secure) linker will then be responsible for exchanging these proofs so that compiled code can attest the trustworthiness of code it links against. Then, compiled code can use this new information to determine whether or not to run the dynamic checks.

Correctness Criteria for Secure Compilation

Formal criteria for secure compilation and proving that a compiler is correct is another challenging research field that we have explored and plan to explore further. As stated, the most common formal definition for secure compilation is full abstraction [1, 27, 43, 44].

Proving full abstraction relies on the concept of back-translation, i.e., the expression of target-level behaviour – or some abstraction of it – in the source language. Currently full abstraction is mainly proven in two ways, either by relying on target-level trace semantics [5, 7, 30, 31, 46] or using logical relations [8, 9, 19, 42]. I studied both techniques in different settings. Traces are often used when the source and target languages are very different (e.g., object-oriented and assembly) to reason about target-level behaviour by abstractions that are expressible in the source-level (e.g., call-returns) [7, 31, 46, 51]. Concerning logical relations, we demonstrated that logical relations between typed and untyped lambda calculi provide a scalable proof technique for full abstraction [19] that can be fully mechanised in Coq [18]. Other techniques for back-translation exist [9, 13, 42]; they are applicable in settings where the target language has constructs that do not seem expressible in the source (e.g., control-flow manipulation constructs such as exceptions).

Improving Compiler Full Abstraction Improving proof techniques for compiler full abstraction is needed on various fronts. First, traces have been used in various settings but a more general approach, more amenable to mechanisation, is necessary, since traces provide a simple bridge between different languages. Second, the logical relation-based technique can be made to interoperate to study full abstraction of source languages into target ones whose types are more expressive than the source and that also have richer primitives. Third, in order to eliminate the lack of mechanisation in these results, new formal tools can be developed. Recently, the Iris [32] framework has

been proposed for reasoning about languages (in a concurrent setting) using logical relations, and it automatically deals with a number of tedious proof constructions. However, it is still unknown if any additional machinery is needed for Iris to support reasoning about back-translation, so an extension of Iris might be necessary.

New Criteria for Secure Compilation Unfortunately, full abstraction also has a number of shortcomings for secure compilation and it is desirable to devise new soundness criteria for secure compilation [31, 51, 54]. I investigated one such criterion in the form of trace-preserving compilation (TPC) which provides a uniform criterion for compiler security with a clear identification of the security properties it preserves [52]. Specifically, we proved that TPC preserves hypersafety properties (which include safety properties and non-interference [16]) under an intuitive cross-language translation of the meaning of hypersafety. However, more study on the subject is needed. For example, a fully abstract compiler is inefficient, as it must perform checks that are only needed for contextual equivalence-preservation and have no security relevance. We plan to investigate compilers that only aim at preserving specific security properties (e.g., robust safety [11, 23, 26]), and study how this affects compiler efficiency. Another example of the limitations of full abstraction is in side channels, since they are, by definition, means for an attacker to observe program behaviour outside of what the language expresses. For example, by timing the execution of some code, the input to some cryptographic operations can be deduced, thus leaking secret inputs [55]. We believe that novel definitions of compiler security should account for some channels such as timing. This can then be used to harden compiled code from a spectrum of attacks, both expressible and not expressible directly in the target language.

The Secure Compilation Research Field

Finally, the secure compilation field is relatively young and it is attracting more and more interest from the rest of the community. For it to grow, together with other researchers involved in this area we have organised (and plan to keep organising) a number of events to ensure more researchers contribute to secure compilation. We started with only informal meetings among few research groups that have evolved into a workshop: the Secure Compilation Meeting (SCM) co-located with POPL in 2017.[2] We plan to hold this workshop regularly at top venues to capture the attention of more experienced researchers and as a medium to keep the community up-to-date with the latest developments.

To introduce newcomers to the field of secure compilation, I compiled a survey on the subject that presents what work has been done and what techniques have been employed [47]. To attract PhD students and instruct them in all the techniques which are relevant for this field, a Daghstul seminar is being organised for 2018. However, we envision that more of these initiatives are necessary. Advanced courses in compiler security must be held at local universities and summer schools can help young researchers to meet, discuss and foster growth. I plan to participate in these initiatives in the future.

[2] http://popl17.sigplan.org/track/SCM-2017

References

[1] Martín Abadi. Protection in programming-language translations. In *Secure Internet programming*, pages 19–34. Springer-Verlag, 1999.

[2] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, LICS '98, pages 105–, Washington, DC, USA, 1998. IEEE Computer Society.

[3] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure communications processing for distributed languages. In *IEEE Symposium on Security and Privacy*, pages 74–88, 1999.

[4] Martín Abadi, Cédric Fournet, and Georges Gonthier. Authentication primitives and their compilation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 302–315, New York, NY, USA, 2000. ACM.

[5] Martín Abadi and Gordon Plotkin. On protection by layout randomization. In *CSF '10*, pages 337–351. IEEE, 2010.

[6] Pieter Agten, Bart Jacobs, and Frank Piessens. Sound Modular Verification of C Code Executing in an Unverified Context. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, 2015.

[7] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. Secure compilation to modern processors. In *CSF '12*, pages 171 – 185. IEEE, 2012.

[8] Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. *SIGPLAN Not.*, 43(9):157–168, September 2008.

[9] Amal Ahmed and Matthias Blume. An equivalence-preserving CPS translation via multi-language semantics. *SIGPLAN Not.*, 46(9):431–444, September 2011.

[10] Ittai Anati, Shay Gueron, S Johnson, and V Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *HASP'13*, volume 13. ACM, 2013.

[11] Michael Backes, Catalin Hritcu, and Matteo Maffei. Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations. *Journal of Computer Security*, 22(2):301–353, 2014.

[12] Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. *ACM Trans. Inf. Syst. Secur.*, 13(3):21:1–21:32, 2010.

[13] William J. Bowman and Amal Ahmed. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP '15, New York, NY, USA, 2015. ACM.

[14] Michele Bugliesi and Marco Giunti. Secure implementations of typed channel abstractions. In *POPL*, pages 251–262, 2007.

[15] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. *SIGPLAN Not.*, 42(6):54–65, June 2007.

[16] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010.

[17] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, March 1966.

[18] Dominique Devriese, Marco Patrignani, Steven Keuchel, and Frank Piessens. Modular, Fully-Abstract Compilation by Approximate Back-Translation. In *To appear.*, 2017.

[19] Dominique Devriese, Marco Patrignani, and Frank Piessens. Fully-abstract compilation by approximate back-translation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 164–177, 2016.

[20] Dominique Devriese, Marco Patrignani, and Frank Piessens. Fully-Abstract Compilation of Parametric Polymorphism into Dynamic Sealing. In *To apppear*, 2017.

[21] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 27:1165–1178, 2008.

[22] Akram El-Korashy. A Formal Model for Capability Machines – An Illustrative Case Study towards Secure Compilation to CHERI. Master's thesis, 2016.

[23] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007.

[24] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Ben Livshits. Fully abstract compilation to JavaScript. Technical report, MSR, 2012.

[25] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to javascript. In *POPL '13*, pages 371–384, New York, NY, USA, 2013. ACM.

[26] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. *J. Comput. Secur.*, 11(4):451–519, July 2003.

[27] Daniele Gorla and Uwe Nestman. Full abstraction for expressiveness: History, myths and facts. *Math Struct Comp Science*, 2014.

[28] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean application compartmentalization with soaap. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, Denver, CO, USA, October 2015.

[29] Chung-Kil Hur and Derek Dreyer. A Kripke logical relation between ML and Assembly. *SIGPLAN Not.*, 46(1):133–146, January 2011.

[30] Radha Jagadeesan, Corin Pitcher, Julian Rathke, and James Riely. Local memory via layout randomization. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 161–174, Washington, DC, USA, 2011. IEEE Computer Society.

[31] Yannis Juglaret, Cătălin Hriţcu, Arthur Azevedo de Amorim, and Benjamin C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In *29th IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, July 2016. To appear.

[32] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *SIGPLAN Not.*, 50(1):637–650, January 2015.

[33] Andrew Kennedy. Securing the .NET programming model. *Theor. Comput. Sci.*, 364(3):311–317, November 2006.

[34] Adriaan Larmuseau, Marco Patrignani, and Dave Clarke. A high-level model for an assembly language attacker by means of reflection. In *Dependable Software Engineering: Theories, Tools, and Applications - First International Symposium, SETTA 2015, Nanjing, China, November 4-6, 2015, Proceedings*, pages 168–182, 2015.

[35] Adriaan Larmuseau, Marco Patrignani, and Dave Clarke. A secure compiler for ML modules. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, pages 29–48, 2015.

[36] Adriaan Larmuseau, Marco Patrignani, and Dave Clarke. Implementing a Secure Abstract Machine. . In *Proceedings of the 31th Annual ACM Symposium on Applied Computing*, SAC '16. ACM, 2016.

[37] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54, 2006.

[38] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP '13*, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[39] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of c: Elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2016, pages 1–15, New York, NY, USA, 2016. ACM.

[40] George C. Necula. Proof-carrying code. In *POPL*, pages 106–119, 1997.

[41] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 166–178, New York, NY, USA, 2015. ACM.

[42] Max S. New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 103–116, New York, NY, USA, 2016. ACM.

[43] Joachim Parrow. Expressiveness of process algebras. *Electronic Notes in Theoretical Computer Science*, 209(0):173 – 186, 2008. Proceedings of the {LIX} Colloquium on Emerging Trends in Concurrency Theory (LIX 2006).

[44] Joachim Parrow. General conditions for full abstraction. *Math Struct Comp Science*, 2014.

[45] Marco Patrignani. *The Tome of Secure Compilation: Fully Abstract Compilation to Protected Modules Architectures*. PhD thesis, KU Leuven, Leuven, Belgium, May 2015.

[46] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.*, 37(2):6:1–6:50, April 2015.

[47] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal Approaches to Secure Compilation – A Survey. In *To appear.*

[48] Marco Patrignani and Dave Clarke. Fully Abstract Trace Semantics of Low-level Isolation Mechanisms. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1562–1569. ACM, 2014.

[49] Marco Patrignani and Dave Clarke. Fully abstract trace semantics for protected module architectures. *Computer Languages, Systems & Structures*, 42(0):22 – 45, 2015. Special issue on the Programming Languages track at the 29th {ACM} Symposium on Applied Computing.

[50] Marco Patrignani, Dave Clarke, and Frank Piessens. Secure Compilation of Object-Oriented Components to Protected Module Architectures. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS'13)*, volume 8301 of *LNCS*, pages 176–191, 2013.

[51] Marco Patrignani, Dominique Devriese, and Frank Piessens. On Modular and Fully-Abstract Compilation. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium CSF 2016, Lisbon, Portugal*, CSF 2016, 2016.

[52] Marco Patrignani and Deepak Garg. Secure Compilation as Hyperproperties Preservation. In *to appear.*

[53] Benjamin Pierce and Eijiro Sumii. Relating cryptography and polymorphism. manuscript.

[54] Frank Piessens, Dominique Devriese, Jan Tobias Muhlberg, and Raoul Strackx. Security guarantees for the execution infrastructure of software applications. In *IEEE SecDev 2016*, 2016.

[55] Jean-Jacques Quisquater and David Samyde. Side Channel Cryptanalysis. In *Invited talk in SEcuritè de la Communication sur Internet (SECI 02). Tunis, Tunisia*, 9 2002.

[56] Raoul Strackx. *Security Primitives for Protected-Module Architectures Based on Program-Counter-Based Memory Access Control*. PhD thesis, KU Leuven, December 2014.

[57] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. *SIGPLAN Not.*, 39(1):161–172, January 2004.

[58] Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in javascript. *SIGPLAN Not.*, 49(1):425–437, January 2014.

[59] David Walker. A type system for expressive security policies. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 254–267, New York, NY, USA, 2000. ACM.

[60] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. 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 *2015 IEEE Symposium on Security and Privacy*, pages 20–37, May 2015.

[61] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *USENIX Security Symposium*, pages 29–46. USENIX Association, 2010.

[62] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.