Internship Topics on SECOMP: Formally Secure Compilation of Compartmentalized C Programs

CĂTĂLIN HRIȚCU, MPI-SP, Germany

1 BACKGROUND

Undefined behavior is endemic in the C programming language: buffer overflows, use after frees, double frees, signed integer overflows, invalid type casts, various concurrency bugs, etc., cause mainstream C compilers to produce code that can behave completely arbitrarily. This leads to devastating security vulnerabilities that are often remotely exploitable, and both Microsoft and Chrome report that around 70% of their high severity security bugs are caused by undefined behavior due to memory safety violations alone [15, 33, 45].

A strong practical mitigation against such vulnerabilities is *compartmentalization* [12, 23, 28, 49], which allows developers to structure large programs into mutually distrustful compartments that have clearly specified privileges and that can only interact via well-defined interfaces. This way, the compromise of some compartments has a limited impact on the security of the whole program. This intuitive increase in security has made compartmentalization and the compartment isolation technologies used to enforce it become widely deployed in practice; e.g., all major web browsers today use both process-level privilege separation [12, 23, 28] to isolate tabs and plugins [43], and software fault isolation (SFI) [34, 44, 50, 54, 54, 55] to sandbox WebAssembly modules [24, 26, 29, 48].

In this paper, we investigate how one can provide strong *formal guarantees* for compartmentalized C source code by making the C compiler aware of compartments. We follow Abate *et al.* [5], who argue that a compartment-aware compiler for an unsafe language can restrict the scope of undefined behavior both (a) spatially to just the compartments that encounter undefined behavior [27], and (b) temporally by still providing protection to each compartment up to the point in time when it encounters undefined behavior. Abate *et al.* formalize this intuition as a variant of a general *secure compilation* criterion called *Robust Safety Preservation (RSP)* [6, 7, 41, 42]. Their RSP variant ensures that any low-level attack against a compiled program's safety properties mounted by compartments dynamically compromised by undefined behavior could also have been mounted at the source level by arbitrary compartments with the same interface and privileges, while staying in the secure fragment of the source semantics, without undefined behavior. This strong formal guarantee allows source-level security reasoning about compartmentalized programs that have undefined behavior, and thus for which the C standard and the usual C compilers would provide no guarantees whatsoever.

Such strong formal guarantees are, however, notoriously challenging to achieve in practice and to prove mathematically. RSP [6, 7, 41, 42] belongs to the same class of secure compilation criteria as full abstraction [2, 38], for which simple and intuitive but wrong conjectures have sometimes survived for decades [16], and for which careful paper proofs can take hundreds of pages even for very simple languages and compilers [19, 27]. Such proofs are generally so challenging that no compiler for a mainstream programming language that is guaranteed to achieve any such secure compilation criterion has ever been built. Moreover, such secure compilation proofs are at the moment often only done on paper [2–4, 7–9, 14, 19, 20, 25, 27, 36–42], even though at the scale of a realistic compiler, paper proofs would be impossible to trust, construct, and maintain. All this stands in stark contrast to *compiler correctness*, for which CompCert [31, 32]—a realistic C compiler

Author's address: Cătălin Hriţcu, MPI-SP, Bochum, Germany, catalin.hritcu@mpi-sp.org.

that comes with a machine-checked correctness proof in the Coq proof assistant—has already existed for more than a decade and is used in practice in highly safety-critical applications [30].

In our current work we take an important step towards bridging this gap by devising SECOMP, a formally secure compiler for compartmentalized C code. For this we extend the CompCert compiler and its correctness proof with isolated compartments that can only interact via procedure calls and returns. While compiler correctness is definitely not enough to ensure secure compilation, since on its own it gives up on programs with undefined behavior, we make good use of compiler correctness for proving secure compilation. For this we adopt the high-level proof structure proposed by Abate *et al.* [5], who showed how proving their RSP variant can be reduced to showing compiler correctness together with three security-related properties: *back-translation*, *recomposition*, and *blame*. Proving these properties at scale and achieving formally secure compilation for a compiler for a mainstream programming language were open research challenges, which we solve in this work by bringing the following *novel contributions*:

- We devise the SECOMP compiler for compartmentalized C programs by extending the syntax and semantics of all the languages of CompCert from Clight all the way down to RISC-V assembly with the abstraction of isolated compartments that can only interact via procedure calls, as specified by cross-compartment interfaces. This extension is particularly interesting for CompCert's RISC-V assembly language, for which we propose a low-level, enforcement-independent characterization of C compartments that relies on a new shadow stack to ensure the well-bracketedness of cross-compartment control flow. We adapt all passes and optimizations of CompCert to this extension, except cross-compartment inlining and tail-calls, which we purposefully disallow.
- In addition to passing scalar values to each other on calls and returns, our compartments can also perform input and output (IO), which was not the case in the very simple languages studied by Abate *et al.* [5]. Our IO model allows pointers to global buffers of scalars to be passed to the external calls implementing IO and also allows these buffers to be changed nondeterministically by these external calls, which goes beyond what was previously possible in CompCert's IO model.
- We extend the large-scale CompCert compiler correctness proof to account for these changes so that we can use it to show secure compilation. Our extension of the correctness proof is elegant and relatively small, even though two of our changes to the semantics of the CompCert languages are substantial and have a non-trivial impact on the proofs: (1) we extend the CompCert memory model with compartments, and (2) we extend the CompCert trace model with events recording cross-compartment calls and returns, as needed for the secure compilation proof.
- We develop a machine-checked secure compilation proof for SECOMP in the Coq proof assistant. This proof shows the RSP variant of Abate *et al.* [5], which captures the secure compilation of multiple mutually distrustful C compartments that can be dynamically compromised by undefined behavior. We are the first to prove such a strong secure compilation criterion for a compiler for a mainstream programming language, which makes this a milestone for secure compilation.
- In order to scale up the secure compilation proofs to SECOMP, we introduce several novelties in proof engineering.

Our formally secure compiler is available at https://github.com/secure-compilation/CompCert.

2 POTENTIAL INTERNSHIP TOPICS

Lower-level backends, in particular targetting CHERI RISC-V. SECOMP targets CompCert's RISC-V assembly language that we extended with the abstraction of isolated compartments, which formally defines *what* compartment isolation enforcement should do, but which leaves the *how* to lower-level enforcement mechanisms. Various enforcement mechanisms should be possible, including SFI [26, 29, 44, 50, 54] (for instance by going via WebAssembly [13, 24, 29, 53]) and

tagged architectures [11, 17], as shown in a much simpler setting by Abate *et al.* [5]. There is also ongoing work on designing and implementing two such lower-level backends for SECOMP targetting the CHERI RISC-V architecture [46, 51] providing hardware capabilities, which are unforgeable pointers with base and bounds that cannot be circumvented. Moreover, CHERI's entry and sealed capabilities are used to implement protected wrappers that mediate the compartment interaction, for instance cross-compartment calls and returns. One of the backend designs only uses these existing CHERI features, inspired by the original work of Watson *et al.* [52]. This backend, however, requires an unusual stack layout, which is allowed by the C standard and the CompCert memory model, but which changes the RISC-V calling convention. The other backend uses a more usual contiguous stack layout, but instead relies on recently proposed CHERI extensions with uninitialized capabilities [21] and directed capabilities [22].

At the moment these lower-level backends are not only unfinished [46], worked out only in a simpler setting [5], or just hypothetical designs, but they are also all unverified. Extending the secure compilation proofs all the way down to cover them is a formidable research challenge that we leave as future work. All existing secure compilation proof techniques in this space [7, 38], including the one we use in the current paper [5], have their origin in proof techniques for full abstraction [38]. Once the memory layout becomes concrete though, we can no longer hide all information about the compartments' code, as would be needed for full abstraction (or in our case for recomposition), so new proof techniques will be needed for proving these lower level backends secure.

Pointer passing and memory sharing. As for the mainstream compartment isolation mechanisms (e.g., SFI or OS processes), we assume that compartments can only communicate via scalar values, but cannot pass each other pointers to share memory. While secure pointer passing between compartments seems possible to implement efficiently on a capability machine like CHERI [51] or on the micro-policies tagged architecture [11] and this would allow a more efficient interaction model that is also natural for C programmers, the main challenge one still has to overcome is *proving* secure compilation at scale in the presence of such fine-grained, dynamic memory sharing.

Recent work by El-Korashy *et al.* [18] in a much simpler setting shows that it is indeed possible to prove in Coq the security of a compiler that allows passing secure pointers (e.g., capabilities) between compartments. With such fine-grained memory sharing, however, proofs become more challenging and the proof technique of El-Korashy *et al.* [18] led to much larger proofs and still has conceptual limitations that one would need to overcome for it to work for CompCert, in particular for supporting CompCert's sophisticated memory injections. In fact, even extending CompCert's compiler correctness proof to passing arbitrary pointers seems a challenge, since it would imply a significant change to CompCert's trace model. In the nearer future we will try to allow more limited forms of memory sharing between compartments, for instance of statically allocated buffers, which could be passed without significantly changing CompCert's trace model.

From safety to hypersafety. Another interesting future direction is extending SECOMP to stronger criteria beyond robust preservation of safety properties, in particular to safety hyperproperties [7], such as data confidentiality. We expect that SECOMP can be easily adapted to these stronger criteria, by for instance always clearing registers before changing compartments, and also that our proof technique can still apply, by only extending the back-translation step to take finite sets of trace prefixes as input [7, 47]. The more challenging problem is actually enforcing robust preservation of safety hyperproperties in the lower-level backends, especially with respect to side-channel attacks, including devastating micro-architectural attacks like Spectre.

Dynamic compartment creation and dynamic privileges. SECOMP uses a static notion of compartments and static interfaces to restrict their privileges. SECOMP compartments are defined statically by the source program, so a form of code-based compartmentalization. In the future one

could also explore dynamic compartment creation, which would allow for data-based compartmentalization [23], e.g., one compartment per incoming network connection or one compartment per web browser tab or plugin [43]. It would also be interesting to investigate dynamic privileges for compartments, e.g., dynamically sharing memory by passing secure pointers (as discussed above), dynamically changing the compartment interfaces [35], or history-based access control [1, 10].

REFERENCES

[1] M. Abadi and C. Fournet. Access control based on execution history. *NDSS*. The Internet Society, 2003.
[2] M. Abadi. Protection in programming-language translations. *Secure Internet Programming*. 1999.
[3] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1):37–83, 2002.
[4] M. Abadi and G. D. Plotkin. On protection by layout randomization. *ACM TISSEC*, 15(2):8, 2012.
[5] 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. *CCS*. 2018.
[6] C. Abate, R. Blanco, Ş. Ciobâcă, A. Durier, D. Garg, C. Hriţcu, M. Patrignani, É. Tanter, and J. Thibault. An extended account of trace-relating compiler correctness and secure compilation, 2021.
[7] C. Abate, R. Blanco, D. Garg, C. Hriţcu, M. Patrignani, and J. Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. *CSF*, 2019.
[8] A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. *ICFP*. 2008.
[9] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. *ICFP*. 2011.
[10] C.-C. Andrici, Ştefan Ciobâcă, C. Hriţcu, G. Martínez, E. Rivas, Éric Tanter, and T. Winterhalter. Securing verified IO programs against unverified code in F*. To appear at POPL, 2024.
[11] 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.
[12] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. *USENIX NSDI*, 2008.
[13] J. Bosamiya, W. S. Lim, and B. Parno. Provably-safe multilingual software sandboxing using WebAssembly. *USENIX Security*. 2022.
[14] M. Busi, J. Noorman, J. V. Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens. Securing interruptible enclaved execution on small microprocessors. *ACM Trans. Program. Lang. Syst.*, 43(3):12:1–12:77, 2021.
[15] C. Cimpanu. Chrome: 70% of all security bugs are memory safety issues. ZDNet, 2020.
[16] D. Devriese, M. Patrignani, and F. Piessens. Parametricity versus the universal type. *PACMPL*, 2(POPL):38:1–38:23, 2018.
[17] 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.
[18] A. El-Korashy, R. Blanco, J. Thibault, A. Durier, D. Garg, and C. Hriţcu. SecurePtrs: Proving secure compilation with data-flow back-translation and turn-taking simulation. *CSF*, 2022.
[19] A. El-Korashy, S. Tsampas, M. Patrignani, D. Devriese, D. Garg, and F. Piessens. CapablePtrs: Securely compiling partial programs using the pointers-as-capabilities principle. *CSF*. 2021.
[20] C. Fournet, N. Swamy, J. Chen, P.-É. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. *POPL*. 2013.
[21] A. L. Georges, A. Guéneau, T. V. Strydonck, A. Timany, A. Trieu, S. Huyghebaert, D. Devriese, and L. Birkedal. Efficient and provable local capability revocation using uninitialized capabilities. *PACMPL*, 5(POPL):1–30, 2021.
[22] A. L. Georges, A. Trieu, and L. Birkedal. Le temps des cerises: efficient temporal stack safety on capability machines using directed capabilities. *PACMPL*, 6(OOPSLA):1–30, 2022.
[23] 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.
[24] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. *PLDI*, 2017.
[25] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. *CSF*. 2011.
[26] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown. WaVe: a verifiably secure WebAssembly sandboxing runtime. *IEEE S&P*. 2023.
[27] 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. *CSF*, 2016.
[28] D. Kilpatrick. Privman: A library for partitioning applications. *USENIX FREENIX*. 2003.

[29] M. Kolosick, S. Narayan, E. Johnson, C. Watt, M. LeMay, D. Garg, R. Jhala, and D. Stefan. Isolation without taxation: near-zero-cost transitions for WebAssembly and SFI. *PACMPL*, 6(POPL):1–30, 2022.
[30] D. Kästner, U. Wünsche, J. Barrho, M. Schlickling, B. Schommer, M. Schmidt, C. Ferdinand, X. Leroy, and S. Blazy. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. *ERTS*. 2018.
[31] X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.
[32] X. Leroy. A formally verified compiler back-end. *JAR*, 43(4):363–446, 2009.
[33] M. Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. BlueHat IL, 2019.
[34] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: Better, faster, stronger SFI for the x86. *PLDI*. 2012.
[35] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: From general purpose to a proof of information flow enforcement. *IEEE S&P*. 2013.
[36] M. S. New, W. J. Bowman, and A. Ahmed. Fully abstract compilation via universal embedding. *ICFP*, 2016.
[37] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *TOPLAS*, 2015.
[38] 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.
[39] M. Patrignani and D. Clarke. Fully abstract trace semantics for protected module architectures. *CL*, 42:22–45, 2015.
[40] M. Patrignani, D. Devriese, and F. Piessens. On modular and fully-abstract compilation. *CSF*. 2016.
[41] M. Patrignani and D. Garg. Secure compilation and hyperproperty preservation. *CSF*, 2017.
[42] M. Patrignani and D. Garg. Robustly safe compilation, an efficient form of secure compilation. *ACM Trans. Program. Lang. Syst.*, 43(1), 2021.
[43] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. *EuroSys*. 2009.
[44] G. Tan. Principles and implementation techniques of software-based fault isolation. *FTSEC*, 1(3):137–198, 2017.
[45] The Chromium Project. Memory safety. chromium.org.
[46] J. Thibault, A. Azevedo de Amorim, R. Blanco, A. L. Georges, C. Hriţcu, and A. Tolmach. SECOMP2CHERI: Securely compiling compartments from CompCert C to a capability machine. Presentation at PriSC, 2023.
[47] J. Thibault and C. Hriţcu. Nanopass back-translation of multiple traces for secure compilation proofs. PriSC, 2021.
[48] A. VanHattum, M. Pardeshi, C. Fallin, A. Sampson, and F. Brown. Lightweight, modular verification for WebAssembly-to-Native instruction selection. To appear at ASPLOS'24.
[49] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith. BreakApp: Automated, flexible application compartmentalization. *NDSS*. 2018.
[50] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SOSP*, 1993.
[51] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, R. Grisenthwaite, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, 2020.
[52] 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. *S&P*. 2015.
[53] C. Watt, X. Rao, J. Pichon-Pharabod, M. Bodin, and P. Gardner. Two mechanisations of webassembly 1.0. *FM*. 2021.
[54] 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.
[55] L. Zhao, G. Li, B. D. Sutter, and J. Regehr. ARMor: Fully verified software fault isolation. *EMSOFT*. 2011.