SECOMP2CHERI: Securely Compiling Compartments from CompCert C to a Capability Machine

(Extended Abstract, Last Updated on January 10, 2023)

Jérémy Thibault[1] Arthur Azevedo de Amorim[2] Roberto Blanco[1]
Aïna Linn Georges[3] Cătălin Hrițcu[1] Andrew Tolmach[4]
[1] MPI-SP, Bochum, Germany [2] Boston University, USA [3] Aarhus University, Denmark [4] Portland State University, USA

Undefined behavior is endemic in the C programming language: buffer overflows, use after frees, double frees, 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 memory safety issues alone [6, 14, 17].

We study how *compartmentalization* can mitigate this problem by restricting the scope of undefined behavior both *(a)* spatially to just the compartments that encounter undefined behavior [12], and *(b)* temporally by still providing protection to each compartment up to the point in time when it encounters undefined behavior [1]. While our past work has focused on formally secure compilation of compartmentalized code for toy languages with buffers and procedures [1, 12], in this talk we report on our ongoing work on scaling up these ideas to a realistic C compiler, based on CompCert [13]. While in prior work [1] we used software-fault isolation (SFI) or a tagged architecture [4] to enforce compartmentalization at the lowest level, in this talk we will focus on a new secure compilation backend for CompCert targeting a variant of the CHERI capability machine [20].

When completed, our work will show that compartmentalized code in a mainstream programming language can be compiled by a realistic compiler with machine-checked security guarantees. This will be a milestone for secure compilation. Proving secure compilation even for toy compilers can be a daunting task, with careful paper proofs often spanning hundreds of pages [8]. We believe that scaling such proofs to realistic compilers has to rely on proof assistants like Coq for ensuring that the proofs are correct, even if building such machine-checked proofs requires serious proof-engineering work. The good news is that proof assistants do not only check proofs, but also allow proofs to be built interactively, refactored, simplified, maintained, and evolved together with the compilation chain.

Machine-Checked Proofs in Coq for the Secure Compilation of Compartmentalized C Code. We have extended the CompCert C compiler [13] and its correctness proof in Coq with secure compartments that can only interact via procedure calls, as specified by cross-compartment interfaces. We disallow cross-compartment inlining and tail-call optimizations. Moreover, for the moment, compartments cannot pass each other pointers and are prevented from accessing each other's memory, except for call arguments spilled on the caller's stack frame. We applied this extension has to all the levels of CompCert, from CompCert C all the way down to CompCert's formalization of RISC-V assembly.

The changes we made to add secure compartments to the semantics of RISC-V assembly are particularly interesting. Even at this low level the security of compartments is enforced "magically"[1] by the semantics (before we go even lower and implement this enforcement, for instance using capabilities, as explained in the next section). At this level, most information about control flow is gone, and calls and returns are done through ordinary jumps (including jump-and-link). To identify calls and returns, and to enforce the cross-compartment interfaces, we made two changes: (1) we use a shadow stack that tracks cross-compartment calls and returns; and (2) we allow certain jump instructions to be tagged as calls or as returns, so that only such appropriately tagged instructions can attempt to cross compartment boundaries. When we encounter a call-tagged jump, we check that the call is allowed by the interface, and push the expected return address and stack pointer to the shadow stack. When we encounter a return-tagged jump, we use the shadow stack to check that the return address and the stack pointer have been correctly restored by the callee, ensuring that an attacker cannot return to an arbitrary location. Together, these checks ensure the well-bracketedness of cross-compartment control flow [3], which can be efficiently enforced at an even lower level using capabilities (see below) or micro-policies [1, 4].

We have also extended CompCert's trace model with new events that record cross-compartment calls and returns, and proved our extension correct w.r.t. these events. Adapting

[1]Where by "magic" we mean a high-level, abstract implementation of compartments in an otherwise low-level machine.

CompCert's compiler correctness Coq proof to account for all these changes was a substantial amount of work. We wanted to change the proof as little as possible, but since CompCert is a realistic multi-pass compiler with 20 passes across 10 different languages, it was not always obvious from the beginning how best to do this. Several times, we made design decisions that seemed adequate, but that turned out to actually be inadequate much later (e.g., choosing at which precise step to insert a given check), when we discovered that they interacted poorly with some particular compilation pass (e.g., inlining or tail-call optimization) or language (e.g., RISC-V assembly). These issues often did not affect the correctness of the compiler, but made the proofs much more difficult, so we had to backtrack and find alternative ways to structure the changes so as to simplify the proofs.

We have finished adapting CompCert's compiler correctness Coq proof to account for all the changes above. Our development is available online.[2] In the near future, we plan to use this compiler correctness proof as a key ingredient for proving two secure compilation criteria called Robust Safety Preservation (RSP) and Robustly Safe Compartmentalizing Compilation (RSCC), by applying the proof technique from our prior work [1]. Adapting the trace-based back-translation proof step of this technique should hopefully be fairly straightforward, since traces have the same overall structure, and the source language of CompCert is expressive enough for us to generate similar code. The other important proof step is recomposition, which relies on traces being expressive enough (e.g., by recording cross-compartment calls and returns) to synchronize two executions of different programs when crossing compartment boundaries.

Before we can start the secure compilation proof along these lines though, at least two more compiler changes will be needed. First, in order to achieve secure compilation we need to make all registers be caller saved before cross-compartment calls, since in our setting we cannot trust the callee compartment to save and restore the caller compartment's registers. Second, we still need to make the semantics invalidate all non-argument registers on cross-compartment calls and all non-return registers on cross-compartment returns (by making them undefined values), since the recomposition proof step requires all information passed between compartments to be captured by the trace.

Secure Compilation to a Variant of CHERI RISC-V. To show that the "magic" enforcement we added to the semantics of RISC-V assembly is efficiently implementable, we have recently designed a capability backend for our secure compiler. While various secure calling conventions targeting capabilities have been proposed in recent years [9, 15, 16, 19], our backend is based on the most recent proposal of Georges *et al.* [10]. This calling convention is based on two new kinds of capabilities: uninitialized [9] and directed [10]. In short, uninitialized capabilities "represent read/write authority to a block of memory without exposing the memory's initial contents" [9], preventing reading old values from the stack without excessive clearing, and directed capabilities allow one to efficiently implement stack safety [10].

[2]https://github.com/secure-compilation/CompCert

We base this backend on a variant of CHERI RISC-V [20], which already supports not only normal capabilities, but also local [15], entry, and sealed capabilities [20], so we extended CompCert's RISC-V language with these capabilities. On top of this, we add the aforementioned uninitialized and directed capabilities, and we use them to design a calling convention inspired by Georges *et al.* [10].

We adapt the calling convention of Georges *et al.* [10] to our setting in two ways: first, because we only enforce compartment isolation, not memory safety, we represent pointers as offsets into a large stack capability or into per-compartment heap capabilities. By not using directed capabilities for stack pointers, we overcome a potential limitation of Georges *et al.*'s [10] calling convention and can store cyclic data structures on the stack. Second, compared to Georges *et al.* [10] we consider a stronger attacker model, in which both the caller and the callee compartments of a call can be compromised. In our model we thus need to always maintain the distinction between the caller and callee compartments and enforce that no capabilities are exchanged between the two. We achieve this by adding privileged wrappers for calls and returns, which ensure that the passed arguments are not capabilities and which clear all remaining registers.

This backend is for the most part also implemented in Coq, but not yet fully integrated with CompCert and not verified. In the short run, we plan to finish implementing this backend and use property-based testing to get some confidence that it is secure. We are also investigating a second capability backend inspired by the original work of Watson *et al.* [21], in which compartmentalization is enforced using only the existing features of CHERI. In the long run, formally verifying such backends in Coq is an interesting open research challenge, as also mentioned below.

Future work. As for the mainstream compartmentalization mechanisms (e.g., SFI or 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 or on the micro-policies tagged architecture [4], which would allow a more efficient interaction model that is natural for C programmers, the main challenge we still have to overcome is *proving* secure compilation at scale in the presence of such fine-grained, dynamic memory sharing.

Recent work in a much simpler setting [7] shows that it is indeed possible to verify in Coq a secure compiler that allows passing secure pointers (e.g., capabilities) between compartments. However, with such fine-grained memory sharing proofs become much more challenging, and the proof technique of El-Korashy *et al.* [7][3] still has limitations that one would need to overcome for it to work for CompCert (for instance, CompCert's memory injections are more complex than the simple memory renaming of El-Korashy *et al.* [7]). For the moment we do allow pointer passing between compartments and trusted external libraries (e.g., component-aware variants of the C standard library), and in the near future we could try to also allow more limited forms of memory sharing between compartments, for instance of statically allocated buffers.

Other interesting future directions includes extending our secure variant of CompCert to stronger criteria beyond robust preservation of safety properties [2, 18]; building more secure compilation backends for CompCert (e.g., taking inspiration in our previous work in simpler setting [1] to target micro-policy machines or software-fault isolation; but for the latter maybe going via Wasm [5, 11]); proving some of these backends secure; and extending this work to dynamic compartment creation.

[3]This technique for structuring mechanized secure compilation proofs when secure pointers can be passed between compartments is in part inspired by a previous paper proof of full abstraction by El-Korashy *et al.* [8], who as a secondary contribution, implement an unverified compartmentalizing compiler from C to CHERI, using the `libcheri` library to construct sandboxes. Their attacker model is quite different from ours though, since we consider mutually distrustful compartments that can be dynamically compromised by C undefined behavior [1].

References

[1] C. Abate, A. Azevedo de Amorim, R. Blanco, A. N. Evans, G. Fachini, C. Hriţcu, T. Laurent, B. C. Pierce, M. Stronati, J. Thibault, and A. Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. *CCS*. 2018. Extended version on arXiv:1802.00588v5.

[2] 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.

[3] S. N. Anderson, L. Lampropoulos, R. Blanco, B. C. Pierce, and A. Tolmach. Security properties for stack safety. *CoRR*, abs/2105.00417, 2021.

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

[5] J. Bosamiya, W. S. Lim, and B. Parno. Provably-safe multilingual software sandboxing using WebAssembly. In K. R. B. Butler and K. Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. 2022.

[6] C. Cimpanu. Chrome: 70% of all security bugs are memory safety issues. ZDNet, 2020.

[7] 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.

[8] 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. pages 1–16, 2021.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

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

[15] L. Skorstengaard, D. Devriese, and L. Birkedal. Reasoning about a machine with local capabilities: Provably safe stack and return pointer management. *TOPLAS*, 42(1):5:1–5:53, 2020.

[16] L. Skorstengaard, D. Devriese, and L. Birkedal. StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *JFP*, 31:e9, 2021.

[17] The Chromium Project. Memory safety. chromium.org.

[18] J. Thibault and C. Hriţcu. Nanopass back-translation of multiple traces for secure compilation proofs. PriSC, 2021.

[19] S. Tsampas, D. Devriese, and F. Piessens. Temporal safety for stack allocated memory on capability machines. *2019*. 2019.

[20] 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.

[21] 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.