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

(Extended Abstract)

Jérémy Thibault\textsuperscript{1} Arthur Azevedo de Amorim\textsuperscript{2} Roberto Blanco\textsuperscript{1} Aïna Linn Georges\textsuperscript{3} Cătălin Hrițcă\textsuperscript{1} Andrew Tolmach\textsuperscript{4}

\textsuperscript{1} MPI-SP, Bochum, Germany \hspace{1cm} \textsuperscript{2} Boston University, USA \hspace{1cm} \textsuperscript{3} Aarhus University, Denmark \hspace{1cm} \textsuperscript{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 spatially to just the compartments that encounter undefined behavior [12], and temporally by still providing protection to each compartment up to the point in time when it encounters undefined behavior [1]. Our past work has focused on formally secure compilation of compartmentalized code for toy languages with buffers and procedures [1, 12], while in this talk we report on our ongoing work on scaling up these ideas to a realistic C compiler, based on CompCert [13]. While SFI and tagged architectures can also be used to enforce compartmentalization at the lowest level [1], in this talk we will focus on a new secure compilation backend targeting a variant of the CHERI capability machine [19].

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. This extension has been applied at 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” 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, 5].

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 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 the compiler correctness Coq proof of CompCert to account for all the changes above. Our development is available online [4]. In the near future, we plan to use this compiler correctness proof as a key ingredient for proving two secure compilation criterion called Robust Safety Preservation (RSP) and Robustly Safe Compartmentalizing Compilation (RSCC), by applying the
proof technique from our prior work [1]. Adapting the 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 expres-
sive enough to generate similar code. The other important 
proof step is recomposition, which relies on traces being ex-
pressive enough 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 for cross-compartment 
calls, since in our setting the caller compartment cannot trust 
the callee compartment to save and restore the caller’s regis-
ters. Second, we still need to make the semantics invalidate 
all non-argument registers after cross-compartment calls 
(by making them undefined values), since the recomposition 
step requires all information passed between compartments 
to be captured by the trace.

When completed, our work will show that compartmen-
talized 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 comp-
ilation. Proving secure compilation even for toy compilers 
can be a daunting task, with careful paper proofs often span-
ing 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. The good 
news is that proof assistants also allow such proofs to be 
built interactively, refactored, simplified, maintained, and 
evolved together with the compilation chain.

Secure Compilation to a Variant of CHERI RISC-V.
To show that the “magic” enforcement we added to the se-
manitics 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, 
18], 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 au-
thority 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].

We base this backend on a variant of CHERI RISC-V [19], 
which already supports not only normal capabilities, but also 
local [15], entry, and sealed capabilities [19], 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 com-
promised. 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. [20], 
in which compartmentalization is enforced using only the 
existing features of CHERI. In the long run, formally veri-
fying such backends in Coq is an interesting open research 
challenge, as also mentioned below.

Future work. A limitation of this work is the lack of 
memory sharing. Compartments can only communicate via 
scalar arguments, but many C design patterns require some 
amount of pointer passing and memory sharing. In recent 
work [7], we have shown that in a much simpler setting it 
is indeed possible to verify in Coq a secure compiler that 
allows memory sharing by passing secure pointers (e.g., ca-
pabilities) between compartments. With such fine-grained, 
dynamic memory sharing, however, proofs become much 
more challenging, and the proof technique of El-Korashy et 
al. [7] still has limitations that one would need to remove for 
it to work for CompCert (for instance, CompCert’s memory 
injections are more complex than the simple memory renam-
ing of El-Korashy et al. [7]). In the shorter-term, we could 
try to allow more limited forms of memory sharing, such as 
adding privileged compartments with unrestricted access 
to memory for (parts of) the C standard library; or 
allowing statically shared buffers, but no pointer passing.

Other interesting future work includes extending our se-
cure variant of CompCert to stronger criteria beyond robust 
preservation of safety properties [2]; building more secure 
compilation backends, for instance based on micro-policy 
machines or software-fault isolation [1] (maybe going via 
Wasm [11]); proving some of these backends secure; and 
extending this work to dynamic compartment creation.
References


