Memory Simulations, Security and Optimization in a Verified Compiler

David Monniaux

Memory Simulations, Security and Optimization in a Verified Compiler

David Monniaux

Univ. Grenoble Alpes, CNRS, Grenoble INP*, VERIMAG, 38000 Grenoble, France.

December 11, 2023

Abstract

Current compilers implement security features and optimizations that require nontrivial semantic reasoning about pointers and memory allocation: the program after the insertion of the security feature, or after applying the optimization, must simulate the original program despite a different memory layout.

In this article, we illustrate such reasoning on pointer allocations through memory extensions and injections, as well as fine points on undefined values, by explaining how we implemented and proved correct two security features (stack canaries and pointer authentication) and one optimization (tail recursion elimination) in the CompCert formally verified compiler.

1 Introduction

Formally verified compilation aims at providing an end-to-end mathematical, machine-checked proof of preservation of the semantics of code through the compilation process. There are currently only two formally verified compilers for general-purpose languages: CompCert [20][1], for almost all of the C programming language,[2] and CakeML for a substantial subset of Standard ML.[3]

Our goal in this paper is to illustrate how certain proofs methods for reasoning on semantics and transformations of programs involving pointers can be used to implement optimization and security features not found in these compilers.

In this paper, we shall be concerned with CompCert. CompCert is used for safety-critical applications, including fly-by-wire controls in commercial aircraft

*Institute of Engineering Univ. Grenoble Alpes

[1]CompCert is distributed both as source code, free of charge for research purposes, to which we will refer when discussing CompCert modules, available at `https://github.com/AbsInt/CompCert` and as a commercial package by Absint `https://www.absint.com/`

[2]There also exist other front-ends for CompCert for languages other than C. Vélus is a front-end for a subset of the Lustre or Scade synchronous programming languages. A front-end for a subset of OCaml was also developed [10].

[3]In addition, some optimization passes for LLVM were formalized with the Vellvm project [44, 45, 28], but this was not an end-to-end project.

[3]. It however does not support many optimizations and security features found in current compilers such as gcc and LLVM (clang), which may limit its appeal for other contexts. One reason is that implementing and proving these features correct is considered challenging, particularly if *memory* is involved.

The internal representations (IR) of many compilers, including LLVM and CompCert, distinguish between values found in (pseudo-)registers and in memory. Registers are CPU registers and thus are in limited number, while infinitely many pseudo-registers are available. The *register allocation* phase, occurring after most optimizations have been run, maps pseudo-registers to CPU registers or to stack locations and adds suitable memory load and store instructions to *spill* values to them. Conversely, some early compilation phase (in LLVM, the mem2reg phase) tries to map local variables to pseudo-registers, if possible, otherwise to stack memory locations. The difference is that the address of pseudo-registers cannot be taken, they cannot be referred to through pointers, and there is no possibility of aliasing (two pseudo-registers are identical if and only if they have the same identifier, while memory locations may be identical or not according to complex pointer arithmetic relationships).

In CompCert, the correctness of code transformations and optimization phases is established by exhibiting a "match" relation between the program states before and after the transformation and proving that sequences of steps of the program pre-transformation are matched by sequences of steps post-transformation, a form of *simulation*. The overall correctness of the compiler is established by composition of these simulations.

In this paper, we will show how to use forms of memory simulations, known as memory extensions and injections [4], to prove the correctness of certain compiler transformations that are available in mainstream compilers yet missing from official releases of CompCert:

1. Official releases of CompCert do feature *tail call elimination*, which will in particular turn some recursive calls into tail recursive calls. However these tail recursive calls will still restore and save registers, destroy and reconstruct stack frames. Tail recursion elimination improves on tail call elimination by bypassing the function epilogue and prologue by directly branching to the top of the function body (after, of course, placing correct values in the parameters). The optimization itself is rather trivial to implement, but its correctness proof is nontrivial; it illustrates the use of *memory injections*.

2. *Stack canaries* are a basic but effective countermeasure to certain buffer overflow attacks and have long been supported by gcc and LLVM. Again, the transformation is trivial but the correctness proof involves *memory extensions*.

3. *Return address authentication* is another countermeasure to certain buffer overflows, supported on recent ARM processors. Here, suitably limited modification to memory invariants enable supporting this feature.

An important aspect, when dealing with formally verified software, is its *trusted computing base* (TCB). We extensively surveyed CompCert's TCB [26], both in the "official" releases and in our own "chamois" fork.[4] The trusted

[4] `https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert`

computing base can be extended in many ways, the most obvious of which is by adding axioms.

Axioms can be used to declare the existence of functions that are, through extraction directives, instantiated with OCaml code. In CompCert, this is in particular the case for many very short OCaml functions that look up the value of command-line flags. Just checking if an optimization or security feature should be active or not entails declaring an axiom that there exists such a Boolean flag, and then adding an extraction directive that explains how this flag may be obtained with a short bit of OCaml code. This usage does not increase the TCB.

In our work, the only axioms that we add, except for those dealing with command-line options, are the existence of a "canary value", which is only known when running the compiled program, as well as of processor-specific pointer encoding and decoding functions such that encoding then decoding a pointer yields that pointer. We will discuss this in more detail in §5.2.

We implemented our work on top of the "chamois" version (`c145bbc5`),[5] proving our passes sound with no admitted results. The two new passes and the modifications to the stacking pass that we describe in this paper should however be usable with very minimal changes with the "official" version. Options `-ftailrec`, `-fstack-protector`, `-fretaddr-pac`, respectively turn on tail recursion optimization (this also needs `-ftailcalls` so that tail recursive calls are exposed as tail calls), "canary" stack protectors and pointer authentication for return addresses. All these options are on by default and may be turned off by, e.g., `-fno-tailrec`. Finally, `-fstack-protector-all` in addition to `-fstack-protector` forces protection on all functions, not just "vulnerable" ones.

2 Semantic Preliminaries: Undefined Behavior and Refinement

We shall begin with considerations about how to model memory and undefined behavior.

The semantics of the intermediate and final representations in CompCert are defined as deterministic transition systems where the program moves from a state to another state.[6] The state, depending on which representation is used, may include certain elements such as the values of pseudo-registers or actual CPU registers, a more or less abstract vision of the call stacks, and a vision of memory. A transition may or may not emit externally observable events. Externally observable events includes reads and writes to volatile variables and calls to external functions. Various forms of simulations are used, and they always must preserve the *trace of externally observable events*.

In contrast, the values inside registers, memory, etc. are deemed internal issues of the program and they may or may not be preserved by transformations and optimizations: most operations generate the empty trace of events ϵ_0. This reflects the fact that the C standard mandates that the program behaves as

[5] A frozen version is archived on Zenodo (`10341149`).

[6] We shall see later in this section how some constrained form of nondeterminism can still be expressed in such semantics. See [8] for more general views on modeling nondeterminism in Coq formalizations of semantics.

though certain variables contain certain values, but does not mandate that these should be actually stored in memory at any specific location or even stored at all.

Programs transformations in CompCert are proved correct by exhibiting *forward simulation* relations [19]. Consider a transformation from a language L to a language L', with semantics respectively defined as transition systems over state spaces Σ and Σ'. A "match" relation $M \subseteq \Sigma \times \Sigma'$ relates states of the program before and after the transformation. Σ and Σ' are equipped with deterministic transition relations $\rightarrow$ and $\rightarrow'$: $a \rightarrow_t b$ means that state a steps to state b emitting an event trace t (in many cases, t is the empty trace ϵ_0). The simplest case of simulation is *lockstep*: if $\sigma_1, \sigma_2 \in \Sigma$ are such that $\sigma_1 \rightarrow_t \sigma_2$, and $\sigma'_1 \in \Sigma'$ is such that $M(\sigma_1, \sigma'_1)$ then there exists σ'_2 such that $M(\sigma_2, \sigma'_2)$ and $\sigma'_1 \rightarrow'_t \sigma'_2$. More complex cases of simulation replace several steps by several steps.[7]

One common way to model memory in the C programming language is as a set of disjoint memory blocks; a pointer consists in a block identifier and an offset within the block.[8] This is the basis of the formalization of memory in CompCert, which considers blocks made of bytes [21, 22].[9] This model was reused for the Vellvm formalization of the LLVM intermediate representation [44, Sec. 6.3].

It is possible, through pointer arithmetic, to move from any location in a block to any location in the same block, but not from a block to another; and ordered comparisons between pointers is defined only if they reside within the same block[10] (equality tests, however, are always defined). Each global or local variable resides in its own block, as well as each block allocated through `malloc()`. A few examples:

```
int a[3], b[3];
int i = b > a;  //undefined behavior
int j = b == a; //defined, yields false
int *a1 = a+1;  //defined, points to a[1]
```

In other words, in that model, a memory address consists of a block identifier b and an offset o such that $0 \leq o < l$ where l is the length of the block. Pointer arithmetic modifies the offset while leaving the block identifier unchanged.

[7] See `Smallstep.v`.

[8] The C standard states that pointers point into *objects*, and that testing the ordering of pointers into two different objects has undefined behavior [18, §6.5.8-5]. It is however possible to test for (dis)equality pointers to different objects [18, §6.5.9-6]; this suggests a modeling of pointers as the pair of an object identifier and a location within the object. In the Frama-C's *typed memory model*, an address if a pair of base and offet, both integers [43, Sec. 3.7]. In Astrée, the memory model considers disjoint blocks of bytes, each representing a toplevel C object [25].

[9] Actually, the values stored in individual memory locations in CompCert cannot be exactly bytes. CompCert's semantics allow for an unbounded number of memory blocks to be created without failure within a program execution, thus it is impossible to map pointers to true 4-byte or 8-byte words, which allow finitely many different values. For the purpose of simplicity, we will call "bytes" values that for the purpose of address computations take up one byte, but that may belong to an infinite type.

[10] In C, it is allowed to compute a pointer just past the end of an allocated block, so that loops such as `for (int i=0; i<n; i++) *(t++)=42;` have defined behavior. Such a pointer may be compared to pointers within the block, but not be used for accessing data. For the sake of simplicity, we will leave out pointers just past the block from the discussion.

The C standard lists many cases of *undefined behavior*. According to the standard, if a program executes one instruction with undefined behavior (such as integer division by 0 or out-of-bound memory load from an array), then anything can happen (actually, the whole execution trace has undefined behavior).

Depending which of CompCert's formal semantics is concerned, undefined behavior is modeled either as an explicit "stuck" or "none" condition terminating the evaluation, or implicitly as the absence of a successor state in the transition relation. The provision in the C standard allowing execution to continue arbitrarily after reaching undefined behavior is reflected by the simulation relations, which states that if a sequence of steps is executed in the pre-transformation program, then a matching sequence of steps is to be taken in the transformed program, but does not put any constraints on the case where the pre-transformation program cannot take a step due to being stuck.

In CompCert, some forms of undefined behavior, instead of stopping program execution, are deemed to evaluate to a special Vundef value: all local variables contain Vundef at function entry, all bytes are "undefined" just after memory allocation, etc. Furthermore, arithmetic operations are strict with respect to Vundef: $\mathsf{Vundef} + 42 = \mathsf{Vundef}$, etc.

Vundef models some form of "lazy" undefined behavior, which simply propagates in the values instead of stopping program execution. Note, however, that Vundef as an operand will stop program execution in the semantics when certain instructions are used. This is in particular the case of conditions: branching on a condition evaluating to Vundef, instead of true or false, aborts execution. This is because one cannot define a unique next instruction in this case, which is required by the semantics.

It is always possible, during code transformation phases, to *refine* a case where execution is stuck into a case where a variable gets assigned Vundef (but not the converse). This is for instance the case of integer division: division by zero has undefined behavior at the C level and stops execution, but it is semantically possible at any point in the compilation chain to replace this division operator by another that, instead of not returning a value on division by zero, returns Vundef. This models the fact that, on many current processor architectures, division by zero does not trap. Such a transformation may, in turn, allow further optimizations.

Consider the following program, assuming `x` is dead at the end of the code and its final value does not matter. One cannot move the division instruction before the test if it has "stuck" semantics on division by zero unless one can prove that division by zero never occurs:

```
instructions modifying neither a nor b
if (condition) {
  x = a/b;
  y = x+3;
}
```

Yet, if we refine the division operator to another one that returns Vundef on division by zero, we can, for optimization purposes, turn this program into:

```
x = a/b;
instructions modifying neither a nor b
if (condition) {
```

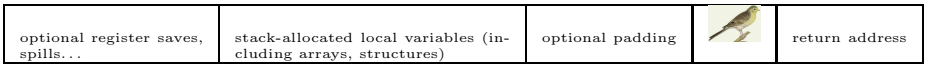

Figure 1: Stack frame with canary on x86 (see `Stacklayout.v` for the general layout; the canary is appended to the stack-allocated variable zone)

```
  y = x+3;
}
```

This may be advantageous for efficiency: a division instruction may take time and it may be preferable to anticipate it. An out-of-order processor will do so on its own, but an in-order processor needs the compiler to anticipate the instruction.

The C standard also mentions *unspecified* behavior, which does not terminate execution. Some operations may yield an unspecified value in some circumstances. The compiler is allowed to replace that "unspecified" value by an arbitrary actual value. Unspecified behaviors may also be modeled by Vundef. In other words, *Vundef can be used inside the deterministic semantics to model some form of nondeterminism.*

Program simulations are allowed to replace Vundef by any value at any point. This is formalized by a *definedness* partial ordering on values, where Vundef is the bottom element and all other values are deemed greater than it (and not comparable with each other). This ordering extends pointwise to orderings on lists of values, memories, etc. It is always possible, at any point, to *refine* Vundef into any other value. Semantics for all program constructs are expected to be monotone with respect to this ordering: if some construct returns a defined value v even if some of its operands are Vundef, then, *a fortiori*, it should return v if some of these Vundef are replaced by normal, defined, values.

3 Memory Extensions and Canaries

Stack canaries are countermeasures against buffer overflow attacks on variables, especially arrays, allocated on the stack. We shall first see how they operate, and then how they can be inserted inside a verified compilation chain.

3.1 Canaries

It is well-known that a major cause of security vulnerabilities in programs in the C language (or any language based on C, such as C++, unless pointer use is restricted by programming guidelines) is the fact that memory accesses are never checked for validity. In particular, the language does not check if accesses to an array occur within its bounds (an array is considered as a pointer to its first element, and the length information is not carried).

The usual way of compiling C-like languages is to have functions allocate a *stack frame* upon entry and deallocate it upon exit. The stack frame contains local variables (including, most interestingly, local arrays and structures handled through pointers), spilled register values (a register is *spilled* to memory when not enough registers are available to store all live values), *callee-saved* and *caller-saved* register values (those that must be saved across function calls,

depending on who is responsible for saving them), and, most importantly, the *return address*, that is, the address in the code to which the program will branch when returning from the function. The stack frame is generally arranged in descending addresses (the stack frame of a function has lower addresses than that of its caller).

The position of the return address in the stack frame depends on the architecture and, possibly, on the compiler. In stack-oriented CISC architectures, such as 680x0 and x86, the return address is pushed on the stack during procedure calls, and then the stack frame is created below it (Fig. 1). The return address thus resides at the end of the stack frame, after the local variables. It is then possible to overwrite this return address by writing past the end of a local array, a not uncommon programming mistake. In RISC processors, the return address is typically passed in a register (whether a special register, or a register defined by convention to contain the return address in the *application binary interface*). In the case of leaf functions, that is, functions making no function calls, one can simply keep the return address in this register (*leaf function optimization*). However, if function calls are made, the calls will overwrite this register, and thus the return address must be saved to the stack frame. Some compilers opt to save it at the beginning of the stack frame, which avoids making it vulnerable to writes past the end of the arrays. However, in this setting, the return address of the caller function, which resides in the stack frame of the caller function in memory (thus after the stack frame of the current function) may be overwritten by a write past the end of the array.

To summarize, if an array or structure is allocated on the stack and the function, for some reason, writes past the end of that array (or calls a function, such as a string manipulating function, that writes past the end of that array), it is possible to overwrite the return address of the function (or, depending on the architecture and compiler peculiarities, the return address of the caller function). In most cases, this will just result in the program crashing upon function return, because the address that will get loaded will contain garbage data that will not form a valid code pointer. If, however, the data written there is under the control of a malicious party, then it may point to executable code, which the program will branch to upon function exit. This is the infamous *stack smashing* attack [23].

Consider for instance the following program:[11]

```
#include <stdio.h>
#include <stdlib.h>

void quack(void) {
  puts("quack"); exit(1);
}

void* vulnerable(int i, void* stuff) {
  void *t[10]; int j[1];
  for(j[0]=0; j[0]<i; j[0]++)
    t[j[0]] = stuff;
```

[11] We store the loop index in an array to force `gcc` to store it in memory before `t[]`. If is is a scalar integer, `gcc` stores it after the array, and the buffer overflow replaces it by the 32 low order bits of the address of `quack()`, which is typically higher than `i` and terminates the loop.

```
  return t[9];
}

void attack(int x) {
  vulnerable(x, (void*) &quack);
}

int main(int argc, char **argv) {
  attack(atoi(argv[1]));
  puts("normal");
  return 0;
}
```

. If compiled with "official" CompCert `c7b27e3` on x86-64 and run as `./canary 9` (or any number ≤ 10), the program prints `normal`. However, if run as `./canary 12`, it prints `quack`. What happens is that `t[10]` is the location for the return address of the `vulnerable` function, which gets overwritten with the address of the `quack()` function. With the protection we implemented into CompCert, which we shall describe below, it instead prints the following message before aborting:

```
*** stack smashing detected ***: terminated
```

The same applies to `gcc`-9.4.0 whether one uses `-fstack-protector` or `-fno-stack-protector`.

The above simple example is unrealistic, because it assumes the attacker is within the same program. More realistic attacks involve exploiting bugs in the targeted application or system software. In the simplest cases, the malicious party provides some incorrectly formatted data, which triggers some unforeseen and undefined behavior of the function that causes it to overflow some data structure and overwrite the return address to a value pointing into malicious data stored on the stack (or in blocks allocated by `malloc()`), which is then executed as code. Of course, this assumes that the attacker is able to predict the addresses at which this data is loaded, but this is often the case assuming the attacker has access to the binary that is run on the target platform (which is the case if it is a standard program), unless ASLR is used.[12]

Such simple attacks, which rely on *code injection* (having data from the outside pass off as code) have long been thwarted by making the stack (as well as memory allocated by `malloc()`)) nonexecutable by default, but this requires a processor capable of differentiating executable memory from nonexecutable memory (all current application-grade processors with a memory management unit are capable of this, but not necessarily processors for simple devices).[13] *Return into libc* eschews code injection by returning to a suitable function inside the standard C library [30]. *Return-oriented programming* (ROP) [7] greatly extends this idea of using existing code inside the targeted application. It works

[12] *Address space layout randomization* (ASLR), implemented in e.g. Linux and MacOS, changes the base addresses of various structures at every execution, so as to make them not predictable by the attacker. As of 2023, current Linux distributions, by default, turn ASLR on and make the stack and other memory data non executable. ASLR can be turned off for debugging purposes.

[13] Memory can be made executable through system calls, for instance to implement just-in-time compilation.

by writing a succession of return addresses to code fragments, also known as *gadgets*, already present inside the program code. The thesis behind ROP is that in any sufficiently large code base, one can always find a succession of gadgets that allow executing sufficiently arbitrary code to gain control of the application. However, all such kinds of attack may often be stopped by using *canaries*.

A *canary* or, more formally, a *stack protector*, is a special value written at function entry inside the stack frame, past any structure that may be adressed through pointers but before the return address, and checked to be still present at function exit [9]. If this value is not longer present, then stack smashing is detected and the program terminates. The idea is that in most cases, buffer overruns occur in contiguous areas and such an overrun, if it can reach the return address, must overwrite the canary.[14] The value of the canary is typically chosen randomly at program start so that it may not be guessed by the malicious party.

A compiler may implement canaries by

- adding to function prologue the sequence: read the canary, write it to the stackframe (optionally, clear the register containing the canary)
- adding to function epilogue the sequence: read the canary, read the value stored in the canary location in the stack frame, branch to an error crash sequence if the two values do not match

The informal argument for this code transformation working is that if the program executed correctly prior to the transformation (recall that writing to the canary counts as undefined behavior thus incorrect execution), then this execution is preserved after the transformation.

One can choose to install canaries on all functions or, to avoid unnecessary slowdown, only on "vulnerable" functions, according to some heuristic for vulnerability: for instance, functions that have arrays inside the stack frame.

3.2 Formal Semantics and Verified Insertion

Let us now see how the above informal description of stack canaries can be made to work inside a verified compiler. Assume now that stack-allocated data is semantically modeled as a single block, as is the case in CompCert.[15]

The informal argument above translates into the statement that if the program executes correctly with stack frames of certain lengths, then it executes correctly with longer stack frames. Note that the program with canaries and thus longer stack frames has *more* legal executions than the original program. This is an effect of modeling stack frames, including the canary, as a single memory block. Imagine for instance that the original function has 32 bytes of

[14] This system is named after the use of canaries or other small birds inside coal mines as a warning system. The birds would become sick before the miners when breathing noxious gases, especially carbon monoxide, and thus would warn the miners. In our case, the canary dies from incorrect memory accesses as a warning to the program before these can be exploited by the malicious party.

[15] CompCert, early in the compilation phases, allocates into the stack frame, considered as a single memory block, all local variables whose address is taken, including structures and arrays; the other local variables are allocated to pseudo-registers. Arguably, one would have made the choice of keeping each variable separate, which would allow, for instance, inserting canaries between variables on the stack and not just at the end of that single block.

stack-allocated arrays, and some instruction accesses these at offset 32; this program evidently has undefined behavior because that access is outside of those arrays, and execution will terminate in the semantics of the intermediate representation. Suppose now we add a canary to that function, inside the same block, at offset 32. Then, reading from that location will just return the canary value. An execution with a write to offset 32 gets stuck at the time of the write in the original program, but will proceed in the transformed program, until the canary is checked; thus the execution will last longer.

In addition to executions going further, extending stack frames may also turn values from Vundef to defined. Consider an order pointer comparison $p < p+34$ between p, the start of the stack frame of length 32, and $p+34$. Because $p+34$ points outside of the block, this comparison returns Vundef. However, if we extend the stack frame with a canary of size 8, this comparison becomes defined and returns true, as expected.

CompCert's notion of *memory extension* [21, §5.2] captures the points we have expressed before. A memory m' extends a memory m if m' contains all the blocks of positive size in m, with lengths greater than or equal to their length in m (it may also contain other blocks). If a memory location exists in m, then it must exist in m' and its contents must be more defined than in m, that is, if the contents at a location in m are defined, then the same contents must appear at that location in m'. Calls to external functions are axiomatized in a way that guarantees that if a function call with arguments a and memory m_1 succeeds and returns v, with modified memory m_2, then if m_1' extends m_1 and arguments a' are more defined than a, the same function returns v' more defined than v in a memory state m_2' that extends m_2.

The proof of correctness of the canary installation phase relies on matching execution states through a memory extension and "more defined" relations. Furthermore, for every function on the call stack, if that function has a canary, then that canary must be in the right position. In addition, we state that in the original program, the stack frame block has the same size as the stackframe (no extra space) and that in the transformed program the extra stacksize space for padding and canary is actually allocated.

3.3 Implementation and Proof

The Coq implementation of the transformation in CompCert is 97 lines long and the proof is 1689 lines long. It operates on the `RTL` representation, which is a form of "3-address code" where instructions are arithmetic operations, loads, stores, conditional branches, builtin calls, procedure calls, tail calls, and branches through jump tables.

We formalized the "get canary" operation as an operator returning an uninterpreted integer constant (c_{32} or c_{64} depending on whether the canary is 32- or 64-bit on the target platform), which gets expanded in a platform-dependent way when producing assembly (the method for retrieving the canary depends on the architecture and the operating system; for instance on x86 Linux it resides at a special offset within thread-local storage, which is to be accessed using specific segment registers).

The transformation pass identifies functions that must be fitted with canaries, computes the offset at which the canary is inserted in the stack frame (some padding may be needed to satisfy alignment constraints) and the size

of the extended stack frame, inserts a "load canary, save it to the appropriate location" sequence at function entry, adds a call to a special function `stack_chk_fail`, and then, for each return or tail call (see 4.1), precedes these call with a "load canary, load saved value, compare them, and branch to the special function call if they differ" sequence.

We do not add axioms to those already present in CompCert, except those necessary to interface with new command-line options, and one that states that there exists a canary value: thus the impact to the TCB is null.

For each platform that we support (currently, only Linux on x86, x86-64, RISC-V, AArch64 and ARM), we provide specific code for retrieving the canary.[16]

The proof of correctness is, as usual in CompCert, proof that the transformed program simulates the original program. This proof relies on a matching invariant $\sim$ between the states of the original and the transformed program.

At `RTL` level, a regular state σ is a tuple $(stack, f, sp, pc, r, m)$ where *stack* is the abstract call stack, f is the function currently executed, *sp* is the address of the current concrete stack frame in memory, *pc* is the number of the current instruction under execution within f, *rs* is the state of the bank of pseudo-registers, and m is the memory; there are also *call states* for when a function is just about to start to be executed, and *return states* for when a function is just about to return. The abstract call stack is a possibly empty list of abstract stack frames, with the immediate caller of the function currently under execution at the start of the list. Each abstract stack frame is a tuple $(ret, f, stack, f, sp, pc, r)$ where *ret* is the number of the pseudo register in the caller function where to store the return value, and f, *sp*, *pc* and r have the same meaning as for the function currently executed, except that they pertain to the function whose execution was stopped when it called another; thus *pc* points to the instruction to which to return.

We use a "plus" simulation, meaning that if there are states σ_o and σ'_o in the original program such that it may take a step $\sigma_o \to \sigma'_o$, and $\sigma_o \sim \sigma_t$, then there must exist σ'_t and a nonempty sequence of steps $\sigma_t \to \ldots \to \sigma'_t$. The reason we need a nonempty sequence of steps in the transformed program, and not just a single one, is that the prologue, epilogue, and tail call instructions, which take one single step in the original program, need several steps to retrieve and store or check the canary. The other operations are handled in lock-step.

The matching invariant $\sigma_o \sim \sigma_t$ essentially states that in the current function, as well as all stack frames in the abstract stack frame:

- the memory m_t of the transformed program *extends* that of the original program, m_o
- the canary value is stored at the appropriate position in the concrete stack frame block in m_t, with appropriate permissions
- the canary value is inaccessible in m_o, meaning that it cannot be read or written in this memory.

[16] We have not found authoritative documentation on how to retrieve the canary and thus follow what gcc and clang do, though not necessarily with the same instruction sequences. In particular, gcc will erase the register in which the canary is used, but CompCert's register allocation will optimize out erasure instructions. Fixing this is left for future work.

The formal proof essentially argues that if a store operation or a call to an external function succeeded in the original memory m_o, where the canary value is inaccessible, then it cannot attempt touching the canary value (otherwise execution would have stopped, because the canary location is inaccessible in that memory), thus these operations never touch the canary value in m_t. In addition to that, it argues that the prologue sequence actually loads the canary at the correct location in the concrete stack frame block (necessary for the matching invariant), and that the function return and tail call sequences never branch to the "stack smashing detected" procedure. A tedious part of the proof shows that the prologue, tail call and return sequences are correctly added to the function without clobbering other operations.

Note that we do not prove that canaries constitute an *adequate* security measure, that is, that the program will go to the "stack smashing detected" procedure if stack smashing happens in the original program. It is in fact possible to prove that in the transformed program, the call to "stack smashing detected" is unreachable using normal execution semantics. The definition of adequation is delicate. Ideally, we would like to ensure that the function would return to its caller and not elsewhere; but a canary does not ensure that, because it is possible for the attack to "jump" over it. The design of a weaker adequation property and its proof are left to future work.

4 Memory Injections and Tail Recursion Elimination

In functional programming languages, it is commonplace to program iterations through *tail recursion*, that is, calling the function itself as its final action before exiting. It is in fact expected, in such languages, that the compiler will turn tail recursive calls into loops. We shall see here how such an optimization can be formally proved.

4.1 Tail Recursion Optimization

Consider the following function:

```
int f(int x) {
  instructions
  return g(x+5);
}
```

The normal way of executing `f` would be to save the return address of `f` (the address to which `f` must return) and create a stack frame for `f`, save callee-saved registers, compute $x + 5$, then call `g`, restore callee-saved registers, destroy the stack frame for `f`, restore the return address of `f`, and finally return from `f` (there are some architectural variations whether the return address is saved before or after the stack frame is created, and restored before or after the stack frame is destroyed).

Remark that the value returned by `g` is directly used as the return value for `f` with the same memory state. One could instead destruct the stack frame for `f`, restore the return address, and jump to `g`. Then `g` gets executed and returns

to whoever called `f`. This avoids accumulating useless frames on the stack. This optimization is known as *tail call optimization*.

This optimization is correct only if no pointers into the local variables of `f` escape to `g`. A sufficient condition for this is that `f` has no stack-allocated local variables.[17] This is how CompCert's existing tail call elimination pass `Tailcall` operates; we do not relax this restriction.[18]

A common case for tail calls is *tail recursion*: the tail call is made to the same function. If the above tail call elimination is used, a tail call from a function f to itself amounts to restoring callee-save registers, destroying the stack frame, jumping to the head of the function, creating a stack frame of the same exact size and saving the callee-save registers (in the same positions). This seems like a waste of time; we would expect *tail recursion elimination*, which turns tail recursion into a loop that keeps the stack frame as is.

For instance, this function:

```
int descend(tree *t, char *u, int n) {
  if (n <= 0) return t;
  if (*u)
    return descend(t->right, u+1, n-1);
  else
    return descend(t->left,  u+1, n-1);
}
```

can be turned into:[19]

```
int descend(tree *t, char *u, int n) {
  loop: if (n <= 0) return t;
        if (*u) { t=t->right; u++; n--;
                  goto loop; }
        else    { t=t->left;  u++; n--;
                  goto loop; }
}
```

Tail recursion elimination improves on tail call elimination by witnessing that destroying an "old" stack frame just to create a "new" one of the same length is useless and one should instead reuse the same stack frame.

4.2 Formal Semantics and Verified Optimization

Again, we must find a formal counterpart to the informal argument that one can freely reuse the existing stack frame. The proof of this optimization relies on the notion of memory injection, a generalization of extensions.

A *memory injection* [21, §5.4] consists in mapping all currently allocated memory blocks in the original program to sub-segments of memory blocks in the

[17] Because this constraint is appreciated prior to register allocation, the stack-allocated local variables it takes into account are arrays, structures, and scalar variables whose address is taken. Variables that are spilled to stack locations because of register allocation do not count.

[18] The `Tailcall` pass operates over the `RTL` representation. Tail calls are however available in previous intermediate representations starting from `Cminor`, but the C frontend does not generate them. They may however be generated by alternate frontends [10, §7.3.4].

[19] This could be more elegantly be written using a `while` loop, but the way optimizations work over a control-flow graph is to insert `goto`'s.

transformed program. Injections are notably used for proving the correctness of the *inlining* transformation, where a call from a function `f` to a function `g` is replaced by the code for `g` (with suitable parameter passing and register renaming): the stack frame of the combined function contains the stack frames for `f` and `g`.

A memory injection μ maps a block b either to nothing (see below for one use for this), or to a block b' and offset o'. A byte at offset o in block b is mapped to offset $o + o'$ in block b'. Memory injections thus constrain the semantics of program operations: they must commute with injections. Adding an integer i to a pointer (b, o) commutes with injections: $(b, o) + i = (b, o + i)$; $\mu(b, o + i) = (b', o' + o + i) = \mu(b, o) + i$. In contrast, checking that specific bits in the offset have certain values is an operation that does not commute with injections. The formal requirement that program semantics commute with memory injections corresponds, partially, to the informal idea that "program behavior must not depend on the addresses to which variables are allocated".[20] The same applies to the semantics of external calls.

For the proof of our tail recursion elimination phase, the injection maps the identifier of the "new" stack frame in the original program to the identifier in the new program to which the "old" stack frame was mapped, while the "old" stack frame maps to nothing.

In practice, for CompCert, for each function we identify tail calls to the same function, and for each of these calls we replace them by

- a sequence of copies of the effective arguments to the tail call to temporary variables
- a sequence of copies from these temporary variables to the positional arguments
- an unconditional branch to the start of the function.

Note that this creates a loop that is amenable to any loop optimization in further passes.

Our copy sequences perform, in effect, a parallel assignment of effective to positional arguments. There are algorithms for generating minimal sequences of moves for such assignments. We however opted for simplicity. Further optimisations of assignments and register allocation anyway suppress most such moves in practice.

Again, what is done is a refinement, not an exact simulation. On normal function start, all pseudo-registers contain `Vundef`. However, when a tail-recursion call is replaced by the above sequence, the pseudo-registers that are not positional arguments to the function retain their last value. This is not a problem, because the pseudo-registers at function entry in that case are "more defined" that the pseudo-registers at normal function entry, which satisfies the "match" relation.

4.3 Implementation and Proof

The Coq implementation of the transformation is 69 lines long (including some sanity check about instruction numbers that could be removed). Again, it op-

[20] In C, comparing for order pointers to distinct variables, living in distinct memory blocks, has undefined behavior.

erates on RTL. The proof of correctness is 2641 lines long, half of which are rather tedious proofs about inserting instructions at correct locations and preserving them during further insertions, and the other half the inductive invariants discussed above. The invariants and the proofs of memory injections were nontrivial.

Consider for instance this tail recursive implementation of factorial:

```
static int fac_rec(int x, int acc) {
  if (x >= 1) return fac_rec(x-1, x*acc);
  else return acc;
}
int fac(int x) {
  return fac_rec(x, 1);
}
```

Our modified version of CompCert will turn the recursive call in `fac_rec()` into a loop. Since `fac_rec()` then becomes a non-recursive function callable only from one location, in `fac()`, it gets inlined into `fac()`. Since there are no more possible calls to `fac_rec()` (recall that a `static` function cannot be called from outside of the module), it is deemed useless and removed from the module. The `fac()` function is thus compiled to assembly code equivalent to (showing a `while` loop instead of `goto` for readability):

```
int fac(int x) {
  int acc = 1;
  while(x >= 1) {
    acc = acc * x;
    x = x-1;
  }
  return acc;
}
```

Similarly, if compiling

```
void twisted_quicksort(data x,
    data *A, int len) {
  if (len < 2) return;
  data pivot = A[len / 2];
  int i, j;
  for (i = 0, j = len - 1; ; i++, j--) {
    while (x*A[i] < x*pivot) i++;
    while (x*A[j] > x*pivot) j--;
    if (i >= j) break;
    data temp = A[i];
    A[i]      = A[j];
    A[j]      = temp;
  }
  twisted_quicksort(x, A, i);
  twisted_quicksort(x, A + i, len - i);
}
```

the second recursive call to `quicksort()`, the tail one, gets compiled into a subtraction, a move and a jump to the head of the function.[21]

We do not add axioms to those already present in CompCert, except those necessary to interface with new command-line options, and thus the impact to the TCB is null.

5 Pointer Authentication for Return Addresses

We shall now investigate how to implement in a formally verified manner another mechanism for thwarting stack smashing: pointer authentication (PAC), as offered by the ARM v8.3 architecture [32, 2, 1, 31].

5.1 Semantics

The operating system, if it supports this functionality, sets up secret keys in the processor. The processor has two implementation-specific functions $E(p, m, k)$ (encoding) and $D(c, m, k)$ (decoding), where p is a pointer, m is a modifier, k is a secret key, such that $D(E(p, m, k), m, k) = p$ for any p, m, k. The modifier is an arbitrary value, the only requirement on it is that it should be the same for encoding and decoding; suitable examples include zero and the stack pointer (if one is sure that the value is decoded in the same function it was encoded in). The idea is that it would be in practice impossible for a malicious party to guess e so that $D(e, m, k)$ is a valid pointer.

Before storing a pointer p to a location where it could be overwritten by vulnerable code, one applies $e = E(p, m, k)$, and apply $p' = D(e, m, k)$ to recover $p' = p$ before using it. Then, even if the malicious party can overwrite this location with arbitrary data, it cannot freely redirect the pointer stored there. If there were only p and k, the malicious party could perhaps, by triggering some memory copy operation, have an otherwise valid pointer copied and reused in another context in which it was not intended to be used; the modifier is intended to make this harder.

In particular, there exist two instructions (`paciasp` and `autiasp`) that respectively apply $r := E(r, s, A)$ and $r := D(r, s, A)$ where r is the return address register and s is the stack pointer. The idea is that one applies `paciasp` before saving the return address to the stack, and `autiasp` before restoring it. On earlier processors not supporting these two instructions, they are executed as "no operation", which amounts to saying that E and D are identity functions ($E(p, m, k) = p$ and $D(e, m, k) = e$).

We modified CompCert's matching invariants that state that the return address is stored at a certain offset in the stack frame so that instead of storing it always in the clear it may be stored as its image by E. As E and D are implementation-specific, we just axiomatize them as uninterpreted functions mapping values to value such that $D(E(p, m, k), m, k) = p$ if p is a pointer.[22]

[21] This function, where `data` can be instantiated to be `uint32_t` or `uint64_t`, sorts elements according to $a \prec_x b \iff (x.a) \bmod 2^n < (x.b) \bmod 2^n$; for odd x, $a \mapsto x.a$ is a permutation and thus this is a total ordering. We use it for evaluation purposes because it is possible to obtain different orderings on the same data by just changing x.

[22] Such a symbolic view of encoding and decoding, when applied to encryption / decryption primitives, is known as the *Dolev-Yao model* [11].

Now, D and E certainly do not commute with memory injections. They are indeed intended so that knowing that $e = E(p, m, k)$ does not help the malicious party forge e' such that $e' = E(p', m, k)$ and p' is a pointer of interest to the malicious party. However, since the handling of return addresses appears late in CompCert's code transformation phases, this lack of commutation does not hinder proofs, since no further transformation phase relies on injections for its proof of correctness.

If we run the attack described in §3.1 without canaries but with pointer authentication, we obtain:

```
normal
Segmentation fault (core dumped)
```

The attack modified the return address of `main()`, but did not overwrite it with an authenticated pointer, thus the `autiasp` instruction yielded an incorrect pointer, which led to a segmentation fault when executing the return instruction.

Finally, the ARM 8.3 instruction set introduced the `retaa` instruction, which combines `autiasp` and the function return instruction. This instruction does not work on previous generations of processors; we thus generate it only if a certain option is passed. For this, we extended the existing postpass optimizer introduced by Gourdin [14]:[23] if the option is active, this pair of instructions is replaced by a single `retaa`.

We had to deal with some slight semantic issue in order to do so, since this instruction is not exactly equivalent to the pair of instruction it replaces: `autiasp` "decodes" the return address into the return address register, which is then used by the return instruction; `retaa` does not change the return address register. The peephole optimizer verifies instruction replacement through symbolic execution and the two sequence of instructions must have identical final result, which is not the case here. We worked around this difficulty by stating that both `retaa` and the normal return instruction leave an undefined value (Vundef) in the return address register—this is acceptable since their actual behavior, which is to leave the value of that register unchanged, is a refinement of this. Then, the two-sequence instruction `autiasp` plus return, and the single `retaa` instruction have the same semantics and the transformation is validated.

5.2 Implementation and Proof

The implementation consists in a 54-line `PAC` module and 11 changes to invariants and proofs in the `Mach` and `Stackingproof` modules of CompCert, those that implement the storing or restoring of return addresses. We also added the `autiasp`, `paciasp` and `retaa` instructions at the assembly level.

To each function is associated a pointer authentication kind. For each architecture, we provide two functions: one (`choose_pac_kind`) chooses, given a program function, the kind of pointer authentication used in this function; the

[23] This optimizer replaces combinations of machine instructions by a single instruction with identical semantics. It operates post all other optimizationpasses. So far, it was used to replace load and store instructions of single values in consecutive memory locations by load and store instructions capable of moving multiple values. An untrusted oracle proposes a new instruction sequence; its result is validated by performing symbolic execution on both the original and modified sequences and checking the final symbolic results are identical, which implies they have identical semantics.

other, (`pointer_auth_encode`), encodes the pointer according to the authentication kind, the pointer, and the modifier. One must also provide a property that if both the pointer and the modifier are of the pointer type (that is, a 32-bit or 64-bit integer depending on the architecture), then the encrypted pointer belongs to that same type.

On all architectures except AArch64, the type of authentication kinds is a singleton (the only authentication kind is "none"), the encoding function is the identity, and the lemma about the pointer type is trivial.

On AArch64, we implement two authentication modes: "none" and "key A" and thus the kind type has two constructors.[24] We add 6 axioms to CompCert. One is extracted into an OCaml function that deals with the command-line option and selects the kind type accordingly. The kind also formally depends on the function under compilation, but we currently do not take it into account; we could for instance implement a command-line option disabling pointer authentication on leaf functions that do not restore the return address from memory. This function does not add to the TCB.

The 5 remaining axioms are the only genuine axioms added in this work. One states that there exists an encoding function for key A; another that there exists a decryption function for key A; another that encoding then decoding is the identity function; another that if the result of encoding with key A is Vundef, then either the pointer or the modifier (or both) was Vundef, and finally that the result of encoding a value with pointer type with key A also has pointer type.

Could these axioms introduce logical inconsistency? We show that this cannot be the case by showing that these axioms can be satisfied by instantiating the encoding and decoding functions for key A by the identity and that, in this case, the remaining axioms can actually be proved.

Could we do without adding axioms? Since ARM does not specify exactly the actual computations that the pointer encoding and decoding functions perform, and in fact reserves the right to implement them differently in different processors, it is inevitable that, whatever the formalization we choose, there is a point when we need to state as axioms that there exist some functions that the processor performs and that satisfy certain properties.

As for canaries, the correctness proof relies on a simulation between states in the original and transformed programs. In this case, however, we need a "star" simulation, where a succession of steps in the original program is simulated by a succession of steps in the transformed program. In almost all cases, our proof is in fact lock-step (one step in the original matches one step in the transformed program), but, when tail recursion elimination is applied, we replace a sequence of two steps (entering the call state of the tail-called procedure and stepping from that call-state to the first normal state in the called procedure) by a sequence that copies parameter values to temporary pseudo-registers, then copies them into the parameter variables, and finally branches to the top of the function.

The matching invariant relies on a memory injection. In most cases, the construction of the injection follows both executions: if the original program creates a memory block b_o and the transformed program creates a memory block b_t, then the injection maps b_o to b_t at offset 0. When it comes to the tail recursive call, the injection, at the end of the call, maps the new stack frame

[24] AArch64 also has a B key. We currently support only key A, since this is what gcc uses.

block b'_o of the current function (the one that has been re-created after the old stack frame block b_o was destroyed) to the current stack frame block b_t, while b_o, the old, destroyed stack frame block in the original programs, is removed from the map.

In order to show all the required properties for the injection to hold, we need an auxiliary invariant that, for the stack frame of the current function as well as all functions on the stack, that stack frame is "weakly allocated", that is, if one has permission to access a memory byte in the concrete stack frame block, then the offset o of the byte in the block is such that $0 \leq o < s$ where s is the size of the stack frame.[25]

There is also a significant amount of tedious proofs showing that the tail recursive call sequences are correctly placed, that their placement did not clobber other instructions, and that the parameter copies that they perform to and from temporaries truly place the parameter values into the correct pseudo-registers.

6 Experimental Evaluation

CompCert's correctness proof states that if it succeeds in compiling a program, the formal semantics of the assembly code that it produces simulates the formal semantics of the C source code. It does not state that CompCert will necessarily succeed in compiling within reasonable time.

Furthermore, at the end of CompCert's backend there are a 2-3 phases (depending on the target architecture) that are implemented in OCaml and are trusted to be correct, including the one that prints assembly code in text form. Bugs have been found in those passes through testing [26], as well as in the formal semantics given for (pseudo-)assembly instructions. Since, for pointer authentication, we added new instructions along with their semantics, and slightly modify the semantics of one existing instruction, testing is again needed.

To detect such bugs, we use the same test suite we used previously [27] in our continuous integration system and with which we identified code generation bugs in both our fork and the "official" releases of CompCert. This test suite consists of the small test suite shipping with CompCert "official" releases, to which we added hundreds of randomly generated programs produced by 3 different generators (CSmith, YarpGen, CCG), and most of `gcc`'s "torture test" suite. In some cases, the resulting object code is run and must match a known result, in other cases one only checks that the compiler ran successfully. We ran this test suite without encountering errors.

We also provide a larger test suite for benchmarking the speed of executable code, suitable for measuring the impact of optimizations and security features.[26] Because pointer authentication is only available on AArch64, we ran the test

[25] One would expect that the converse (a byte at address (b, o) such that $0 \leq o < s$ where s is the size of a stack frame and b is the number of the block for that stack frame is always accessible) would also be true. However such an invariant cannot be proved within CompCert, because the semantics of external calls is too lax. Specifically, the axiomatization of external calls allow the external function to free arbitrary memory locations, including the stack frames of caller functions.

Of course this does not make sense with respect to final assembly-level semantics: a function may only free its own stack frame. Even when considering exception handling or `longjmp()`, one cannot transfer control to a function whose stackframe has been destroyed.

[26] `https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/chamois-benchs`

suite on a Raspberry Pi 3 equipped with an ARM Cortex A53 processor, which interprets pointer authentication as "no operation". Return address authentication and pointer authentication result in a $< 1\%$ slowdown. The same applies to the Apple M1 processor, which does implement pointer authentication: the slowdown is too about 1%. (The stack protector is not yet available on MacOS, and thus we did not time its impact.)

Stack canaries inserted only on "vulnerable" functions (functions with stack-allocated variables, register spills excluded) results in a $< 1\%$ slowdown, and a 5% slowdown if applied to all functions. This vindicates the strategy, already implemented in `gcc` and LLVM, to only apply them to vulnerable functions. Future work should include a better strategy for designating which functions are at risk (pointer arithmetic / array accesses, pointers to local variables passed to other functions?).

Our purpose in this work was to implement standard optimizations and security features and to show how they can be proved to be semantically correct, not to show that these standard optimizations bring large speed improvements. Improvements to tail recursion affect only a narrow category of functions: rather small functions with tail calls. Let us see two examples of these.

The use case for tail recursion elimination for functions traversing data structures can be illustrated by:

```
int last(struct list *p) {
  if (p->next == NULL) return p->val;
  else return last(p->next);
}
```

On a RISC-V "Rocket" core implemented in a FPGA, this procedure ran 100 times on a 100000-element list takes 6.6 s using CompCert's tail call elimination, and only 5.7 s (the same timing as `gcc -O2`) if using tail recursion elimination, a 14% reduction. (Disabling tail call optimization results in 11.5 s, about the same time as `gcc -O`, which does not perform tail call optimization either.) We chose this core for benchmarking because it is very simple and performance or lack thereof is easy to explain.

On an x86-64 machine[27], this procedure ran 10000 times on a 100000-element list takes 2.16 s using CompCert's tail call elimination, and only 1.76 s (the same timing as `gcc -O2`) if using tail recursion elimination, a 19% reduction. This means that function prologue and epilogue can have nontrivial cost on small functions even on a highly out-of-order, speculative core. (Disabling tail call optimization results in 8.54 s.)

Consider now our "twisted quicksort" procedure with the ordering $\prec_x$. We run it $m = 10000$ times, with $i = 0 \ldots m - 1$, on an array of length 1000, initially containing the integers from 0 to 999999, for $x = 1000000001 - 2i$, on the "Rocket". Without tail call elimination, execution time is 37 s; it falls very slightly with tail call elimination, and drops to 34.4 s with tail recursion elimination.

We ran it with $m = 100000$ times on the x86-64. Without tail recursion elimination, the timing is the same as with `gcc -O`, but it increases with tail recursion elimination. CompCert's code generation is geared towards RISC processors with a reasonable number of registers (it does not exploit the memory-

[27] Intel® Core™ i7-10610U CPU @ 1.80 GHz

to-register operations of CISC processors) and it is possible that adding a loop path results in suboptimal register allocation. Moreover, the x86-64 is very complex and it is known that conditions such as the memory alignment of a loop body may have notable consequences on speed.

7 Conclusion, Related Work, and Perspectives

Proving the correctness of program transformations involving a block-based memory model is often difficult. For several transformations available in mainstream compilers, but previously not available in formally verified compilers, we identified the key simulation arguments and successfully implemented these transformations into CompCert. We have successfully run hundreds of test cases using these transformations.

Some other improvements to CompCert [15, 37, 14, 36, 16, 35, 40, 39, 41], starting with the register allocator [34], instead of proving the total correctness of transformation passes, use *translation validation*: a formally verified checker is run after the pass and checks that the source and target programs match. This checker may be helped by suitable annotations.

Translation validation has also been successfully applied, on select architectures to verify that the object code implementing the seL4 kernel matches the source code. Similarly, Alive2 [24] checks that the result of a LLVM transformation matches the source. In both cases, the checking is done by a SMT solver. In our context, that would entail either trusting the SMT solver, or getting from it some kind of certificate that could be used in a verified checker such as the one inside SMTCoq [12], assuming we find one that supports a rich enough theory. This however seems a heavyweight solution, both because the solver needs to be run during every compilation run, and because of the interfacing code needed.

The formal semantics of LLVM were formalized as part of Vellvm, and illustrated with the formal verification of a pass for hardening programs against memory violations [45], a formally verified version of the SoftBound protection [28]. Their system attaches to each pointer the bounds of the zone to which it can legally point to. Because affixing this metadata to the pointer itself would change the number of bytes needed to hold a pointer, leading to compatibility issues, they store the metadata in lookup tables in separate memory blocks. One difference with our approach for canaries is that we change the layout of memory blocks used by the program, by extending stackframe blocks, whereas they keep the original memory blocks. Memory metadata is kept in a disjoint memory space, the access to which is axiomatized.

One difficulty with pointer authentication is that, in the way that we axiomatized it, it does not respect memory injections. This is not an issue in our case, because we add pointer authentication late in the compilation process: the phase that deals with return addresses is not followed by phases whose correctness relies on injections. It would be an issue if we wanted to allow arbitrary code and data pointers to be encoded through special qualifiers for pointers, as allowed by LLVM under MacOS [31]. In order to have pointer authentication respect memory injections, we could add a new kind of value to CompCert's `val` type, for encoded authenticated pointers. This would however likely entail pervasively modifying CompCert, since this type is used pretty much everywhere; and thus we might end up with a fork of CompCert that would be hard

to synchronize with the official releases. We may however study this approach in the future.

Torrini and Boulmé [38] proposed a more ambitious view of a CompCert backend with a semantics incorporating symbolic encryption, for compilation targeting a special processor with built-in code encryption.

The abstract vision of pointers as pairs (b, o) of block identifier and offset has advantages, but also limitations. For once, the semantics of programs doing bit manipulations on pointers (such as storing data in low-order bits of pointers known to be multiples of 2, 4 or 8) is undefined. It also unnaturally views the stack as a list of independent blocks, whereas in reality they are contiguous in memory, which has some adverse consequences for efficiency (CompCert stores a pointer to the next stack frame in the current stack frame). Some alternative models viewing pointers as integers were developed for CompCert [5, 6].

CHERI [42] proposes an execution model where pointers can carry *capabilities*, including buffer bounds, that are enforced in hardware. With such a system, one can render buffer overflows impossible. An extra bit, unforgeable in software, distingues capabilities from normal data. CHERI has been successfully implemented as the "Morello" variant of the AArch64 architecture [17]. At minimum, a compiler for a CHERI-enhanced platform must cope with 32-bit pointers being replaced by 65-bit pointers, 64-bit pointers by 129-bit pointers, and provide memory copy operations that copy the capability bits. For CHERI to be useful against buffer overflows, pointers to memory objects must be created with capabilities limited to these objects. Finer-grained security could also involve further restricting pointer capabilities, especially block sizes, automatically or through explicit program specification (for instance, one may want to restrict a function being called to operate only on part of an object, such as an array slice). In any case, supporting CHERI in CompCert would entail substantial and pervasive changes.

Some high-security or high-safety systems duplicate information and/or computations: mismatches indicate hardware faults, memory corruption, or malicious attack. Duplicating computations can be done in hardware (run two copies of the same core in lockstep and check if their output pins match, trap if they don't) or in software, either by modifying the source code or through a compilation pass [29, 33] (as in STMicroelectronics's modified LLVM, *SecSwift* [13]). We envision such a duplicating pass could be added to CompCert. Duplicating information in memory at compilation time is however more difficult, at least if compiling from C or a language similar to it, because a pointer designates one single block of memory, not one block and its copy.

Our proofs of correctness of phases that insert security countermeasures just show that they preserve legal executions. We do not prove that these countermeasures have any effect on any attack. Doing so would probably involve incorporating a threat model inside the semantics of the program and show properties such as "if a program executes a branch in the attack semantics that is not executed in the original semantics, then its execution must stop before issuing an observable event". While pen-and-paper proofs of such *adequation* properties on toy languages are feasible, formal proofs for them in the context of semantics of compiler intermediate representations are a challenging task.

In the case of pointer authentication, one further difficulty is that this proof of security would have to be relative to the security of the encoding and decoding primitives, in much the same way that the security of a cryptographic protocol

is generally not proved in absolute terms, but rather relative to the security of the primitives involved, often through some *game reductions* (if the attacker can win against the protocol, then, up to some margin in probabilities, it can win against the primitives).

One goal of our work was to minimally change CompCert, so that our additions could be presented as independent passes that could eventually be submitted for integration into the official releases. This was an important constraint. Many of the extensions or alternate solutions described in this section entail modifying global definitions (such as the value type) with possibly far-reaching consequences.

Acknowledgments

This work is partially supported by the ARSENE project funded by the "France 2030" government investment plan managed by the French National Research Agency (ANR), under the reference ANR-22-PECY-0004.

References

[1] Apple. *ARMv8.3 Pointer Authentication in xnu*. 2021. URL: https://opensource.apple.com/source/

[2] Brandon Azad. *Examining Pointer Authentication on the iPhone XS*. Google Project Zero. 2019. URL: https://googleprojectzero.blogspot.com/2019/02/examining-po

[3] Ricardo Bedin França et al. "Towards Optimizing Certified Compilation in Flight Control Software". In: *Workshop on Predictability and Performance in Embedded Systems (PPES 2011)*. Vol. 18. OpenAccess Series in Informatics. Grenoble,France: Dagstuhl Publishing, 2011, pp. 59–68. URL: http://hal.archives-ouvertes.fr/inria-00551370/.

[4] Lennart Beringer et al. "Verified Compilation for Shared-Memory C". In: *Programming Languages and Systems (ESOP)*. Ed. by Zhong Shao. Vol. 8410. Lecture Notes in Computer Science. Grenoble, France: Springer, 2014, pp. 107–127. DOI: 10.1007/978-3-642-54833-8_7.

[5] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "A Concrete Memory Model for CompCert". In: *Interactive Theorem Proving (ITP)*. Ed. by Christian Urban and Xingyuan Zhang. Vol. 9236. Lecture Notes in Computer Science. Nanjing, China: Springer, 2015, pp. 67–83. DOI: 10.1007/978-3-319-22102-1_5.

[6] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. "CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics". In: *J. Autom. Reason.* 63.2 (2019), pp. 369–392. DOI: 10.1007/s10817-018-9496-y.

[7] Erik Buchanan et al. "Return-oriented Programming: Exploitation without Code Injection". In: *Black Hat*. Las Vegas, Nevada, United States, 2008. URL: https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Re

[8] Nicolas Chappe et al. "Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq". In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 1770–1800. DOI: 10.1145/3571254.

[9] C. Cowan et al. “Buffer overflows: attacks and defenses for the vulnerability of the decade”. In: *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*. Vol. 2. Hilton Head, South Carolina, United States: IEEE Computer Society, 2000, pp. 119–129. DOI: 10.1109/DISCEX.2000.821514.

[10] Zaynah Dargaye. “Vérification formelle d'un compilateur optimisant pour langages fonctionnels. (Formal verification of an optimizing compiler for functional languages)”. PhD thesis. Paris Diderot University, France, 2009. URL: https://tel.archives-ouvertes.fr/tel-00452440.

[11] Danny Dolev and Andrew Chi-Chih Yao. “On the security of public key protocols”. In: *IEEE Trans. Inf. Theory* 29.2 (1983), pp. 198–207. DOI: 10.1109/TIT.1983.1056650.

[12] Burak Ekici et al. “SMTCoq: A Plug-In for Integrating SMT Solvers into Coq”. In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 126–133. DOI: 10.1007/978-3-319-63390-9_7.

[13] François de Ferrière. *A compiler approach to Cyber-Security*. Slides presented at EuroLLVM. https://llvm.org/devmtg/2019-04/slides/TechTalk-Ferriere-A_compiler
Apr. 2019.

[14] Léo Gourdin. “formally verified postpass scheduling with peephole optimization for AArch64”. In: *AFADL*. 2021. URL: https://www.lirmm.fr/afadl2021/papers/afadl202

[15] Léo Gourdin. “Lazy Code Transformations in a Formally Verified Compiler”. In: *Proceedings of the 18th ACM International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems, ICOOOLPS 2023, Seattle, WA, USA, 17 July 2023*. Ed. by Eric Jul and Dimi Racordon. ACM, 2023, pp. 3–14. DOI: 10.1145/3605158.3605848.

[16] Leo Gourdin et al. “Formally Verifying Optimizations with Block Simulations”. In: *OOPSLA*. To appear. Oct. 2023.

[17] Richard Grisenthwaite et al. “The ARM Morello Evaluation Platform - Validating CHERI-Based Security in a High-Performance System”. In: *IEEE Micro* 43.3 (2023), pp. 50–57. DOI: 10.1109/MM.2023.3264676. URL: https://doi.org/10.1109/MM.2023.3264676.

[18] *ISO/IEC 9899:2018 — Information technology — Programming languages — C*. ISO/IEC. June 2018.

[19] Xavier Leroy. “A Formally Verified Compiler Back-end”. In: *J. Autom. Reason.* 43.4 (2009), pp. 363–446. DOI: 10.1007/s10817-009-9155-4.

[20] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Commun. ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814.

[21] Xavier Leroy and Sandrine Blazy. “Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations”. In: *J. Autom. Reason.* 41.1 (2008), pp. 1–31. DOI: 10.1007/s10817-008-9099-0.

[22] Xavier Leroy et al. *The CompCert Memory Model, Version 2*. Research Report RR-7987. INRIA, June 2012, p. 26.

[23] Elias Levy. “Smashing the stack for fun and profit”. In: *Phrack* 49 (Nov. 1996). The author is also known as “Aleph One”. URL: http://www.phrack.org/issues/49/14.html#a

[24] Nuno P. Lopes et al. “Alive2: bounded translation validation for LLVM”. In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 65–79. DOI: 10.1145/3453483.3454030. URL: https://doi.org/10.1145/3453483.3454030.

[25] Antoine Miné. “Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics”. In: ACM Press, June 2006, pp. 54–63. eprint: cs/0703074.

[26] David Monniaux and Sylvain Boulmé. “The Trusted Computing Base of the CompCert Verified Compiler”. In: *European Symposium on Programming Languages and Systems (ESOP '22)*. Vol. 13240. Lecture Notes in Computer Science. Springer, 2022, pp. 204–233. DOI: 10.1007/978-3-030-99336-8_8. HAL: hal-03541595v1.

[27] David Monniaux et al. “Testing a Formally Verified Compiler”. In: *Tests and Proofs - 17th International Conference, TAP 2023, Leicester, UK, July 18-19, 2023, Proceedings*. Ed. by Virgile Prevosto and Cristina Seceleanu. Vol. 14066. Lecture Notes in Computer Science. Springer, 2023, pp. 40–48. DOI: 10.1007/978-3-031-38828-6_3.

[28] Santosh Nagarakatte et al. “SoftBound: highly compatible and complete spatial memory safety for c”. In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. Ed. by Michael Hind and Amer Diwan. ACM, 2009, pp. 245–258. DOI: 10.1145/1542476.1542504. URL: https://doi.org/10.1145/1542476.1542504.

[29] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. “Error detection by duplicated instructions in super-scalar processors”. In: *IEEE Trans. Reliab.* 51.1 (2002), pp. 63–75. DOI: 10.1109/24.994913. URL: https://doi.org/10.1109/24.9949

[30] Alexander Peslyak. *Getting around non-executable stack (and fix)*. Bugtraq mailing list. The author is also known as “Solar Designer”. Aug. 1997. URL: https://seclists.org/bugtraq/1997/Aug/63.

[31] *Pointer Authentication*. Documentation for Apple's fork of LLVM. Apple. 2019. URL: https://github.com/apple/llvm-project/blob/apple/main/clang/docs/PointerAuth

[32] *Pointer authentication on ARMv8.3. Design and Analysis of the New Software Security Instructions*. Qualcomm Technologies, Inc. 2017. URL: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v

[33] George A. Reis et al. “SWIFT: Software Implemented Fault Tolerance”. In: *3nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005), 20-23 March 2005, San Jose, CA, USA*. IEEE Computer Society, 2005, pp. 243–254. DOI: 10.1109/CGO.2005.34. URL: https://doi.org/10.1109/CGO.

[34] Silvain Rideau and Xavier Leroy. “Validating Register Allocation and Spilling”. In: *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by Rajiv Gupta. Vol. 6011. Lecture Notes in Computer Science. Springer, 2010, pp. 224–243. DOI: 10.1007/978-3-642-11970-5_13.

[35] Cyril Six. “Compilation optimisante et formellement prouvée pour un processeur VLIW”. PhD thesis. Grenoble Alpes University, France, 2021. URL: https://tel.archives-ouvertes.fr/tel-03326923.

[36] Cyril Six, Sylvain Boulmé, and David Monniaux. “Certified and efficient instruction scheduling: application to interlocked VLIW processors”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 129:1–129:29. DOI: 10.1145/3428197. HAL: hal-02185883.

[37] Cyril Six et al. “Formally verified superblock scheduling”. In: *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*. Ed. by Andrei Popescu and Steve Zdancewic. ACM, 2022, pp. 40–54. DOI: 10.1145/3497775.3503679. URL: https://doi.org/10.1145/3497775.3503679.

[38] Paolo Torrini and Sylvain Boulmé. “A CompCert Backend with Symbolic Encryption”. In: *Workshop on Principles of Secure Compilation (PriSC)*. Jan. 2022. URL: https://hal.science/hal-03555551.

[39] Jean-Baptiste Tristan. “Formal verification of translation validators”. PhD thesis. Paris Diderot University, France, 2009. URL: https://tel.archives-ouvertes.fr/tel-004375

[40] Jean-Baptiste Tristan and Xavier Leroy. “A simple, verified validator for software pipelining”. In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, 2010, pp. 83–92. DOI: 10.1145/1706299.1706311. URL: https://doi.org/10.1145/1706299.1706311.

[41] Jean-Baptiste Tristan and Xavier Leroy. “Formal verification of translation validators: a case study on instruction scheduling optimizations”. In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. Ed. by George C. Necula and Philip Wadler. ACM, 2008, pp. 17–27. DOI: 10.1145/1328438.1328444. URL: https://doi.org/10.1145/1328438.13284

[42] Jonathan Woodruff et al. “The CHERI capability model: Revisiting RISC in an age of risk”. In: *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 2014, pp. 457–468. DOI: 10.1109/ISCA.2014.6853201. URL: https://doi.org/10.1109/ISCA.2014.6853201.

[43] *WP Plug-in Manual for Frama-C 28.0 (Nickel)*. CEA-List. 2023.

[44] Jianzhou Zhao. “Formalizing an SSA-based compiler for verified advanced program transformations”. PhD thesis. University of Pennsylvania, 2013. URL: https://www.cis.upenn.edu/~stevez/vellvm/Zhao13.pdf.

[45] Jianzhou Zhao et al. “Formalizing the LLVM intermediate representation for verified program transformations”. In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. Ed. by John Field and Michael Hicks. ACM, 2012, pp. 427–440. DOI: 10.1145/2103656.2103709.