# A CompCert Backend with Symbolic Encryption

PAOLO TORRINI, INRIA, Grenoble, France
SYLVAIN BOULMÉ, Verimag, Grenoble, France

Binary encryption can be used to strengthen a verified compilation toolchain, in order to protect executable code from malware attacks. We present IntrinSec, a C backend that extends RISC-V with binary encryption and our work on its formalization and verification as a CompCert backend.

Attacks against computer systems can take advantage of software vulnerabilities such as buffer overflows in order to inject malicious code or divert control-flow. Protection against them typically involves ensuring integrity of executable code and stack data. Integrity can then be used to ensure higher-level properties of system behaviour, such as those represented by the control-flow directed graph (CFG) and non-interference. Various techniques have been introduced to ensure integrity, including mitigation tools, type-safe languages, the enforcement of CFG-based control-flow integrity [1] and encryption.

Binary code encryption can be used to safe-guard code and data integrity. Unlike other approaches, it generally requires the deployment of specialized hardware to execute the encrypted code. In [2, 4], integrity of C program execution by authenticated encryption of instructions is ensured by compilers that have been co-developed with RISC processors (the latter used as prototype for proprietary CEA hardware developed within the NanoTrust project). The encryption of binary programs is carried out at compile-time by the trusted compiler. Their decryption is achieved on the fly, at runtime, by the processor itself. The processor supports single-instruction decryption (it executes cyphertext by decrypting each instruction just before executing it).

## 1 INTRINSEC

The IntrinSec assembly, following the design of [4], extends RISC-V 32bits with additional registers and instructions, for control-flow monitoring (CFM), which appear in **blue** on Fig. 1. The compiler translates source code to encrypted binary code (EBC) after linking, producing cyphertext which is executable relying on single-instruction decryption. Encryption is based on stream cyphers (finite ones), each associated with a code block (cryptographic block) that has a single entry point. Instructions in the block are sequentially associated with CFM tokens (which are masks) [2]. The assembly code is instrumented at compile-time in order to update the CFM tokens accordingly to control-flow branch. A higher level of protection can be achieved by adding encryption of the whole program, associated with a stronger secret key, and by introducing data encryption, though these aspects will not be further discussed here.

Verified compilation ensures source-level behaviour preservation, i.e. that the target assembly code always behaves compatibly with the source code semantics. The CompCert C compiler [3] formally developed and verified in Coq is based on a chain of verified compilation passes between intermediate languages down to Asm. Each

Authors' addresses: Paolo Torrini, INRIA, Grenoble, France, Paolo.Torrini@inria.fr; Sylvain Boulmé, Verimag, Grenoble, France, Sylvain.Boulme@univ-grenoble-alpes.fr.

```
int fact(int n){
  if (n <= 1) return 1;
  return n*fact(n-1);
}
```

```
  ecr.enter
fact:
  mv       x30, sp
  addi     sp, sp, -16
  sw       x30, 0(sp)
  sw       ra, 4(sp)
  ecr.sw   emr, 8(sp)
  sw       x8, 12(sp)
  mv       x8, ra0
  ecr.lui  emb,%hi(.L100)
  ecr.addi emb, emb,%lo(.L100)
  addi     x31, x0, 1
  blt      x31, x8, .L100
  addi     ra0, x0, 1
  ecr.lui  emb,%hi(.L101)
  ecr.addi emb, emb,%lo(.L101)
  j        .L101
.L100:
  addi     ra0, x8, -1
  ecr.lui  emb,%hi(fact)
  ecr.addi emb, emb,%lo(fact)
  call     fact
  mul      ra0, x8, ra0
.L101:
  lw       x8, 12(sp)
  lw       ra, 4(sp)
  ecr.lw   emb, 8(sp)
  addi     sp, sp, 16
  jr       ra
```

Fig. 1. IntrinSec assembly produced by our version of CompCert

language is characterized in terms of executable semantics, and they all share a memory model which ensures separation between mutable data and code associated with functions. At the assembly level, a function is associated with a block and each instruction in the block with an offset. CompCert can target different assembly backends including RISC-V [5].

Here we discuss the formalization and verification of the IntrinSec CompCert 3.8 backend[1]. In order to deal with cryptographic tokens, we extend the RISC-V formal model with the specific instructions and registers, and we extend the memory state with a correspondingly modified notion of stack frame. This instrumented version of CompCert RISC-V constitutes the executable model of the IntrinSec backend. As CompCert verified compilation stops at the assembly code, the formal verification of our compiler can only consider an abstract model. We extend the IntrinSec executable model with an axiomatic model of link-time encryption and run-time decryption, basically corresponding to a symbolic encryption model of stream cyphers in Coq. Encryption is represented as a function that depends on a cryptographic block and an offset, returning the CFM mask of the corresponding instruction.

Our basic encryption model assumes that each function block, hence each function, is associated with a stream cypher. The first instruction in the block is associated with the initial mask of the stream (entry mask generated by the "`ecr.enter`" directive in the concrete assembly). Further instructions in the code are then sequentially associated with further masks in the stream. In order to decrypt an instruction, the processor needs to be given the correct mask: an incorrect mask is considered as the result of a control-flow

[1] https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-intrinsec.

attack, and the execution aborts. The processor fetches this mask from dedicated mask registers: MSK_CNT (for the mask associated with the next instruction address, given by the program counter PC), **emr**—called MSK_RTN in our Coq model—(for the mask associated with the return address, given by RA), and **emb**—called MSK_BRN in Coq—(for the mask associated to the next branching address). On each non-branching instruction, MSK_CNT is implicitly updated (with the symbolic encryption). For jumps, the correct mask (stored in the memory at conventional locations for indirect jumps), must be loaded by the program to **emb** by means of the special "**ecr.**" instructions (see Fig. 1).

Asm programs in CompCert are obtained from a chain of intermediate languages inclusive of Mach and Linear [3]. This makes it possible to take advantage of the Mach semantics in the verification of control-flow properties, in particular with respect to function entry points. A minor revision of the semantics preservation proof between Mach and Linear is needed in connection with stackframe operations. A more significant revision is needed for the semantics preservation proof from Mach to Asm. The verification invariant needed for IntrinSec strengthens RISC-V source-level behaviour preservation with a symbolic encryption invariant, ensuring that an instruction can be executed only if the mask which becomes available for its decryption matches the one that has been used to encrypt it (expressed as a relation between PC and MSK_CNT). The proof consists in a refactoring of the analogous one for Mach and RISC-V, and it involves a significant revision of the notion of straightline code, which is essentially meant to capture code without jumps.

From the point of view of symbolic encryption, under the assumption that masks cannot be guessed and encryption cannot be broken, our model can guarantee code integrity, i.e. the fact that it is not possible to make the processor execute assembly code that has been altered or introduced by the attacker. This protection extends at least partially to control flow, especially with respect to function calls (i.e. the forward edges in the CFG). In the case of direct jumps, the destination address is protected by mask encryption. In the case of indirect ones, the mask to decrypt the address is associated to the function entry point. This is safe, under the assumption that the conventional location at which the mask is stored cannot be guessed. Moreover, under the assumption that the mask increment function cannot be guessed, this suffices to guarantee that jumping into the middle of a function is not possible. Concerning the return addresses (i.e. the backward edges in the CFG), the problem is more delicate as both the return address and the associated return mask are stored as data on the stack. Therefore, in order to protect the backward edges, data encryption is needed.

## 2 RESETTING CRYPTO-BLOCKS

The basic version of IntrinSec assumes that cryptographic blocks coincide with function blocks. This means that each stream cypher has at least the same size of the block it is associated with. However, this is a rather artificial constraint and may be undesirable when function blocks are very large. It seems then appropriate to introduce an independent notion of crytographic block. Operationally, this can be done by using a special label to mark the start of a new block, and a reset instruction to initialize the stream cypher. Although the semantics of reset seems easy, this instruction complicates the refactoring of the proofs, as the encryption function comes to depend on the whole function code (labels may trigger a reset), and this makes the definition of the encryption invariant more complex.

The CompCert inductive definition of "straightline code" is semantic (i.e. bounding the PC shift between two sequential instructions). This definition does not depend on any instruction set, and therefore can be used for different backends. In general, this notion is slightly different from a syntactical one which could be obtained by checking for jump instructions in the function code (a jump which advances the counter by one does not break the semantic definition). In IntrinSec, however, this notion has to be modified, as we need to keep into account the MSK_CNT update. With the reset instruction, this complicates refactoring. Thus, we switched to a syntactic notion of straightline code, excluding jump and reset.

## 3 PSEUDO-ASM

In order to mitigate the refactoring problem which may arise when we enrich the Asm back-end (as the reset example shows), we want to split two main aspects that are dealt with in the translation from Mach to Asm. One is the shift from the structural character of the stack representation in Mach to the memory-embedded one of Asm. The other one is the shift from the Mach instruction set to the Asm one. We then introduce an intermediate language, which we call PseudoAsm, that has the same instruction set as Mach but has PC and RA registers and a memory-embedded stack similar to the Asm one. Not only we can factor the translation from Mach to Asm into two distinct ones, one from Mach to PseudoAsm and the other one from PseudoAsm to Asm. We can also define an inverse translation from PseudoAsm to Mach, under a state match relation that restricts Mach states to those with a stack that can be faithfully embedded in memory. An analogous restriction can be introduced in the translations from Linear to Mach, and from Mach to PseudoAsm. The advantage of this approach (currently work in progress) is not only to achieve better modularity, but also to make it possible to express security properties which can be preserved down to PseudoAsm, thus localizing their possible break-down to the shift from the Mach instruction set to the Asm one.

## ACKNOWLEDGMENTS

This work has been partially supported by the IRT Nanoelec (ANR-10-AIRT-05), funded by the French national program "Investissement d'Avenir".

We also wish to thank Olivier Savry, Thomas Hiscock, Marie-Laure Potet and David Monniaux for their helpful collaboration.

## REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. 2005. A Theory of Secure Control Flow. In *ICFEM (LNCS)*, Vol. 3785. Springer, 111–124.
[2] T. Hiscock, O. Savry, and L. Goubin. 2019. Lightweight instruction-level encryption for embedded processors using stream ciphers. *Microprocessors and Microsystems* 64 (2019), 43–52.
[3] X. Leroy, S. Blazy, Z. Dargaye, Jourdan J. H., M. Schmidt, B. Schommer, and J. B. Tristan. 2020. The CompCert C Compiler, Version 3.8. http://compcert.inria.fr/compcert-C.html
[4] O. Savry, M. El-Majihi, and T. Hiscock. 2020. Confidaent: Control FLow protection with Instruction and Data Authenticated Encryption. In *DSD 2020*. IEEE, 246–253.
[5] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi. 2014. *The RISC-V instruction set manual (TR EECS-2014-54).* Univ. of Calif. Vol. 26.