Formally Verified Hardening of C Programs against Hardware Fault Injection

Basile Pesin, Sylvain Boulmé, David Monniaux, Marie-Laure Potet

Formally Verified Hardening of C Programs against Hardware Fault Injection

Basile Pesin
Ecole Nationale de l'Aviation Civile
Toulouse, France
basile.pesin@enac.fr

Sylvain Boulmé
University Grenoble Alpes - CNRS - Grenoble INP - VERIMAG
Grenoble, France
Sylvain.Boulme@univ-grenoble-alpes.fr

David Monniaux
University Grenoble Alpes - CNRS - Grenoble INP - VERIMAG
Grenoble, France
David.Monniaux@univ-grenoble-alpes.fr

Marie-Laure Potet
University Grenoble Alpes - CNRS - Grenoble INP - VERIMAG
Grenoble, France
Marie-Laure.Potet@univ-grenoble-alpes.fr

Abstract

A fault attack is a malicious manipulation of the hardware (e.g., electromagnetic or laser pulse) that modifies the behavior of the software. Fault attacks typically target sensitive applications such as cryptography services, authentication, boot-loaders or firmware updaters. They can be defended against by adding countermeasures, that is, control flow checks and redundancies, either in the hardware, or in the software running on it. In particular, software countermeasures may be added automatically during compilation.

In this paper, we describe a formally verified implementation of this approach in the CompCert verified compiler for the C language. We implemented two existing countermeasures protecting the control flow of the program as program transformations over a middle-end intermediate representation of CompCert, RTL. We proved that these countermeasures are correct, that is, they do not change the observable behavior of the program during an execution without fault injection. We then modeled the effect of a fault on the behavior of the program as an extension of the semantic model of RTL. We used this new model to formally prove the efficacy of the countermeasure: all attacks are either caught, or produce no observable effects. In addition to this formal reasoning, we evaluated the protected program using Lazart, a tool for symbolic fault injection, and measured the effect of optimizations on security and performance.

Keywords: Formally Verified Compiler, Software Counter-Measure, Control Flow Integrity, Coq Proof Assistant.

This is the authors version of the peer-reviewed paper

Basile Pesin, Sylvain Boulmé, David Monniaux, and Marie-Laure Potet. 2025. Formally Verified Hardening of C Programs against Hardware Fault Injection. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'25), January 20–21, 2025, Denver, CO, USA*, https://doi.org/10.1145/3703595.3705880

1 Motivation

Fault injection techniques allow an attacker to alter the execution of a program, even in the absence of any vulnerability (e.g., undefined behavior) in software. These attacks usually target sensitive embedded systems, and proceed by physical means such as a precisely timed laser or electromagnetic pulse, or suitable perturbations to the power supply or clock signal [6]. Consequences vary with the attack method and the targeted processor: some instructions may be skipped, or may have unexpected effects, altering the control flow of the program. The values in registers or loaded from memory may also be modified.

Fault injections may allow an attacker to accomplish attack objectives such as leaking secrets [15] or bypassing authentification [50]. For instance, consider the function presented in listing 1, which checks a PIN code entered by the user. This code does not contain any vulnerability, but is written in a style that is weak against fault injection attacks: the `ok` value is set to `true` by default. This can be exploited by a single fault injected at the conditional branch instruction compiled from the loop condition. Indeed, inverting this condition at the first iteration would end the loop immediately and completely bypass the PIN check.

To protect against these powerful attacks, defensive checks commonly known as *countermeasures* may be implemented

```
extern char expected[PIN_LENGTH];
int verify_pin(char entered[PIN_LENGTH]) {
  int i, ok = 1;
  for(i = 0; i < PIN_LENGTH; i++) {
    if(entered[i] != expected[i]) ok = 0;
  }
  return ok;
}
```

Listing 1. The `verify_pin` function

either in hardware or in software. For instance, in listing 1, adding the test `if (i != PIN_LENGTH) exit(1)` just before `return ok` would detect the attack described above and halt the program gracefully. Adding a countermeasure to an existing program is commonly known as *hardening* it. While software-based countermeasures specific to an application may be written directly in the source code, more generic countermeasures may also be added a posteriori by an automatic hardening tool [11]. In particular, countermeasures may be inserted during the compilation process, and thus be implemented as transformations of the intermediate representations of the compiler [24]. Generally, two complementary properties are expected of a countermeasure:

- **correctness**: if no fault is injected, the countermeasure does not change the defined observable behavior of the program
- **robustness** (also known as *adequacy* or *efficacy*): the countermeasure actually detects injected faults, and corrects or aborts [45] the execution[1]

The second property depends on the *attacker model*, which specifies the maximum number and types of faults the attacker may inject. Depending on the level it is expressed on (source language, intermediate representation, assembly, binary), this model may be more or less abstracted with regard to the physical attack implementing it.

Significant work has been invested in validating and verifying the robustness property for select attacker models and countermeasures. Tools have been developed to simulate fault injection at the binary [23] and source [38] levels, and evaluate the behavior of the program under attack using symbolic execution. Other approaches rely on formal methods, such as model checking [27], or formal proofs [18, 32, 40] to verify the correctness and robustness of a countermeasure. These approaches focus on validating the robustness of a complete program with embedded countermeasures.

In this paper, we propose a novel approach to compositional verification of countermeasures introduced automatically by a compiler, using the interactive theorem prover Coq [19]. Our work is integrated in the end-to-end verified compiler CompCert, which we briefly introduce in section 2. We develop the following contributions:

- a general methodology for implementing and proving the compositional correctness and robustness of program hardening passes, presented in section 4
- implementations and proofs for two countermeasures from the literature [22, 41] based on this methodology, discussed in sections 3 and 5
- an experimental validation process based on symbolic evaluation, described in section 6
- an extension to CompCert's separate compilation model for function specialization passes, presented in section 5.3

Our whole Coq source code, including cross-referenced Coq definitions and proofs, is available from
https://certicompil.gricad-pages.univ-grenoble-alpes.fr/Chamois-Arsene/

2 The CompCert Verified Compiler

CompCert [30] is an optimizing compiler for the C language. It supports code generation for a variety of instruction sets (x86, ARM, RISC-V, ...). It is *formally verified*, in the sense that it is accompanied by a machine-checked proof of semantics preservation: any behavior of the generated code is an authorized behavior of the source program. This is formally stated as a *backward simulation* theorem:

$$\textbf{if } \texttt{compile}(P) \Downarrow B \textbf{ then } P \Downarrow B$$

where P is a source program and B a model of its behavior. CompCert is structured as a series of passes rewriting the program between intermediate representations, all with deterministic semantics except at the very beginning. Each of these passes comes with a *forward simulation* theorem:

$$\textbf{if } P \Downarrow B \textbf{ then } \texttt{compile}(P) \Downarrow B$$

For deterministic languages, forward and backward simulations are indeed equivalent, and forward simulation is usually easier to prove.

The RTL Language. In this paper, we focus on the middle-end representation of CompCert, RTL (Register Transfer Language)[2]. An RTL program is represented as a Control Flow Graph (CFG) of instructions. For instance, the RTL representation of the `verify_pin` function is presented in fig. 1. The program contains three types of RTL instructions. First, application of operations from the instruction set, (`x9 = x2 + x3`). Second, loading of data from memory (`x6 = int8u[x9]`), which corresponds to array dereferencing in the source code. Last, conditional branching, stemming from the loop (`if (x3 < 4)`) and the `if-then-else` (`if (x6 != x7)`). RTL programs manipulate pseudo-registers, here labeled `x1` through `x9`. This means a program transformation may always add new registers for storing intermediate values, without worrying about register allocation, which happens later in the compilation chain. This last property, along with the simplicity of its CFG representation,

[1] In some cases, it may also make the execution state unusable to the attacker. We will not consider this case in our formalization.

[2] The RTL representation of CompCert, presumably inspired by the one in GCC, should not be confused with the Register Transfer Level used for hardware description.

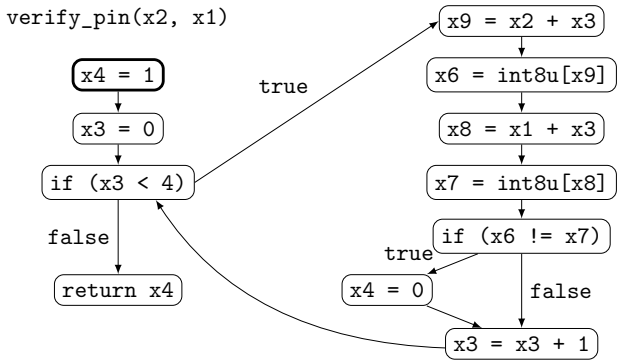


Figure 1. RTL representation of the `verify_pin` function

$$\frac{f.(pc) = \lfloor \mathbf{op}(op, \vec{r}, r_d, pc') \rfloor \qquad op^{\#}(R(\vec{r})) = \lfloor v \rfloor}{P \vdash \mathbf{S}(\Sigma, f, \sigma, pc, R, M) \xrightarrow{\epsilon} \mathbf{S}(\Sigma, f, \sigma, pc', R\{r_d \leftarrow v\}, M)}$$

$$\frac{f.(pc) = \lfloor \mathbf{cond}(cond, \vec{r}, pc_1, pc_2) \rfloor \qquad cond^{\#}(R(\vec{r})) = \lfloor b \rfloor}{P \vdash \mathbf{S}(\Sigma, f, \sigma, pc, R, M) \xrightarrow{\epsilon} \mathbf{S}(\Sigma, f, \sigma, b \ ? \ pc_1 : pc_2, R, M)}$$

Figure 2. RTL semantics of operation and conditional branch

makes RTL well-suited to implement a range of program transformations, including most of CompCert's optimizations. For the same reasons, we also implemented countermeasures as RTL program transformations.

RTL Semantics. The behavior of an RTL program is modeled as a sequence of small-step transitions. The judgment $P \vdash st_1 \xrightarrow{t} st_2$ asserts that "in program P, state st_1 transitions to state st_2 while producing observable events t". Observable events include accesses to volatile variables, calls to external functions, and special "builtin" functions affecting system state. This trace must be preserved from source to assembly code. Steps related by forward simulations must therefore emit exactly the same events.

We recall two of the rules defining this judgment in fig. 2: they specify how the state is updated when executing an operation and conditional branching, respectively. In both cases, the initial state is of the form $\mathbf{S}(\Sigma, f, \sigma, pc, R, M)$, where: $\mathbf{S}$ denotes a state ready to execute an instruction; f is the function being evaluated; and pc is the current Program Counter (PC) inside it. Together, they identify the instruction to be executed: in the rules presented, looking up pc in the CFG of f is a partial operation, which must succeed (written with $\lfloor . \rfloor$), and yield an operation (resp. conditional) instruction; these two rules are syntax-directed. R maps pseudo-registers to their content; this map is total, but may map to the special `Vundef` value, meaning uninitialized or undefined. Finally, parameter Σ specifies the current call stack, M the memory as a finite map of memory address to block, and σ the location of the current stack frame in M.

The first rule applies when the current instruction is an operation op with operand registers $\vec{r}$, writing in destination register r_d, and with successor pc'. The second premise indicates that applying op to the values associated with $\vec{r}$ must yield a value v. The specific rules for the evaluation of an operation are not detailed here. The conclusion indicates that the execution of the instruction updates the program counter to pc', and writes value v into r_d. The execution of an operation does not produce any observable event, and so yields the empty trace ϵ.

The second rules applies to a conditional branching instruction with condition $cond$, operands $\vec{r}$ and destinations pc_1 and pc_2. The evaluation of the condition must yield a boolean value b, which is used in the conclusion of the rule to determine the next program counter.

The full semantic model [30] contains one more semantic rule for each other instruction type, plus a few rules dedicated to function calling and returning. We do not detail them, as they are not directly relevant to our contributions.

3 Application to Control-Flow Integrity

Before delving into our generic framework, we present the first countermeasure we applied our methodology to, as a guiding example. Control-flow integrity (CFI) seeks to prevent intruders from diverting the program control-flow.

The SWIFT countermeasure was first introduced as a protection against accidental faults [41] (e.g., due to cosmic rays). It was later adapted to protect against fault injection attacks, and implemented in undistributed versions of the LLVM compiler [22]. This countermeasure protects the control flow from attacks that may invert or skip conditional branching instructions, such as the one described in the introduction. The countermeasure works as follows. A signature is statically assigned to each basic-block in the program. Throughout execution, the General Signature Register (GSR) stores the signature of the block currently executed. Before a jump, a composition (e.g., exclusive-or) of the current GSR value and the signature of the target block is stored as the Run-Time Signature (RTS). After the jump, it is decomposed by the reverse operation into GSR. A test then checks that GSR indeed contains the signature of the current block. If it does not, it means execution somehow jumped into the wrong block, which triggers halting the program.

We implemented this version of the countermeasure in CompCert, as a rewriting pass on RTL. An example of this transformation is presented in fig. 3, with the source conditional on the left, and the generated code on the right. We assign one signature for each basic block inferred from the RTL CFG: here `33` is assigned to the source conditional, `42` for the `true` branch and `78` for the `false` branch. The GSR corresponds to register `x1`, and initially contains the signature of the source condition, `33`. The RTS is stored in register `x2`. The first instruction is a preliminary test of the condition,

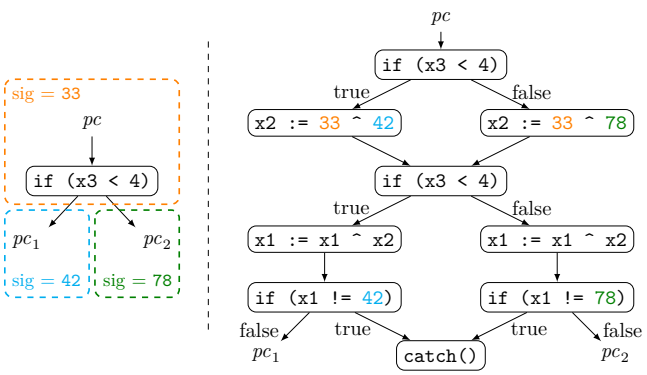


Figure 3. RTL conditional with our CFI countermeasure

here `x3 < 4`, and assigns the RTS depending on its value. To compute the RTS, we use the exclusive-or operation, written `^` in the code, as it is easily reversible. Then, the condition is tested again. In both branches, GSR and RTS are combined again by xor into the new value of GSR. Then, a final test checks that the new GSR is equal to the signature of the current block. If it is not, execution jumps to a special `catch()` instruction which stops the program immediately. If GSR contains the expected signature, the execution continues.

It seems easy to convince oneself that the source and target programs are equivalent: since the xor operator is involutive, GSR ^ GSR ^ *sig* = *sig*, and therefore the `catch()` instruction should never be reached. Similarly, it seems clear that testing the condition twice effectively protects from an attack inverting one conditional. These simple intuitions are however, not the whole story: in a complex compiler, such as CompCert, there are many moving parts and technical details to keep track of; the formal reasoning described in the next section ensure that none of them are forgotten.

4 Verifying Countermeasures

In this section, we present our methodology for implementing and verifying countermeasures such as the one discussed above. We first detail a framework for CFG transformations, designed to reduce technical details. We then show how this framework helps build the proof of semantic correctness. The latter part of this section describes our formalization of an attacker model and the corresponding robustness theorem, and the proof techniques used to establish this theorem.

4.1 Verifying CFG Transformations

In the previous section, we described our implementation of the SWIFT countermeasure as a transformation of the CFG where each conditional instruction is systematically rewritten into an equivalent sequence of instructions. A large class of fault injection countermeasures [27, 45] follow this form (data redundancy, conditional test duplication, instruction duplication, ...). We propose a Coq framework to simplify the implementation of such transformations. For the sake of clarity, we have simplified some of the definitions; the full implementation is available in our artifact.

4.1.1 Monadic CFG Transformations.

To understand the technical issues involved in implementing this type of transformation, we first recall the encoding of an RTL CFG. The nodes of the CFG are identified by their PC of type `node`, which is encoded in CompCert by positive integers. The `cfg` type itself is represented by a finite mapping of each node to the corresponding instruction, encoded by an efficient binary tree representation.

A CFG transformation can be specified as a sequence of two primitive transformations: rewriting an existing instruction, and adding a new instruction, associated with a *fresh* PC. At first, we mechanized this intuition using a state monad [52], where the state encodes both the next PC to be generated and the current shape of the CFG.

To implement a countermeasure, we could then write a monadic program P that sequentially updates the state to add new instructions to the CFG. In order to prove semantic properties of the transformation, we would then need to specify and prove a relation between the source *code* and the transformed program $P(code)$. However, this was not so easy: P being monadic prevents the easy decomposition of its effect on the different parts of the graph. Instead, we would need to reason using complex invariants relating the pre and post-states of the monadic program. These particularly tedious proofs would need to be repeated for each new countermeasure. To avoid this issue, we propose a generic framework which factors out technical difficulties into a once-and-for-all proof.

In addition to the countermeasures described in this paper, we also applied our framework to the "canary" countermeasure from [34], and we were able to reduce by one third the lengths of both the implementation and its proof by removing tedious reasoning on the insertion of code fragments.

4.1.2 Instruction-to-Sequence Rewriting.

Our framework is based on two key insights. First, the transformations we want to implement amount to locally rewriting individual instructions into sequences of instructions. Second, most of the complexity in proving a specification for these transformations comes from the lack of structure of the RTL CFG representation. Therefore, we propose a generic graph transformation framework, where the user specifies a countermeasure through functions producing structured sequences of instructions, which are then used by the framework to build a CFG.

In listing 2, we propose a type of structured sequences of instructions, `seq`. Each sequence has one entry point, and a set number of continuations, specified by the `nat` type parameter. Sequences may contain conditional branching and join points. The first three constructors mark the end of a sequence. Constructor `Ei` is used when the last expression has a successor. Its first parameter is not an RTL instruction, but

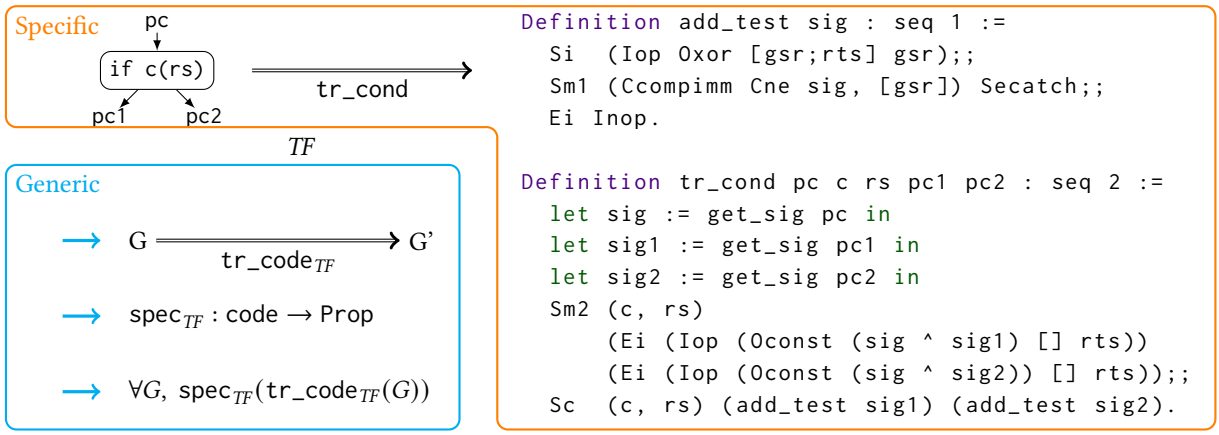


Figure 4. Using the CFG transformation framework to implement the SWIFT transformation

```
Inductive seq : nat → Type :=
(* End cases *)
| Ei: (node → instr) → seq 1
| Er: instr → seq 0
| Ecatch: seq 0
(* Sequencing cases *)
| Si: (node → instr) → seq e → seq e
| Sc: cond → seq 1 → seq 1 → seq 2
| Sm1: cond → seq 0 → seq e → seq e
| Sm2: cond → seq 1 → seq 1 → seq e → seq e.
```

Listing 2. RTL sequence AST

an instruction with its successor PC missing. Indeed, PCs do not appear explicitly in sequences; they are generated by the framework, automatically. Constructor `Er` is typically used with a `return` instruction which has no successor. Constructor `Ecatch` is used for introducing the special instruction `catch()`. The next four constructors indicate a composed sequence. Constructor `Si` adds an instruction at the head of a sequence. Constructor `Sc` represents a sequence that branches depending on a condition; therefore, it has two continuations. Constructors `Sm1` and `Sm2` represent a conditional branching that then merges back into a single sequence. The first specifies only one branch with no continuation (`return` or `catch()`), while the second merges its two branches.

Our generic framework for instruction-to-sequence rewriting implements a kind of *visitor pattern*: to implement a transformation, the user only provides one rewriting function for each type of instruction. For instance, Figure 4 shows, in the top frame, the implementation of the SWIFT transformation, where a conditional (on the left) is rewritten by the function `tr_cond` (on the right). Since the source instruction is a conditional branching, the target sequence has two continuations. The `;;` notation clarifies the sequencing of constructors, and corresponds to function application.

4.1.3 CFG Rewriting Framework.

Rewriting functions, forming the *visitor* specific to the countermeasure, are passed

```
Definition tr_code_TF code : mon unit :=
  mfold (fun pc i ⇒
      let (seq, cont) := tr_instr_TF pc i in
      do cs ← cseq pc seq cont;
      add_seq abort cs) code.
```

Listing 3. Transforming each instruction in the CFG

as a parameter *TF* to the generic framework schematized on the left of fig. 4, which provides (i) a function `tr_code`$_{TF}$ transforming the source CFG *G* into a new CFG G' (ii) a specification `spec`$_{TF}$ of the content (nodes) and shape (edges) of G' (iii) proofs that G' respects this specification (theorems 4.1 and 4.2). We now describe these components.

Listing 3 presents the function `tr_code`, which iterates through the source CFG using the previously described state monad, of type transformer `mon`. Each instruction is passed to the `tr_instr` function, which dispatches it to the functions in *TF* to return the corresponding sequence `seq`, as well as `cont`, which carries the continuation PCs of the original sequence. Then, the monadic function `cseq` *concretizes* the sequence by annotating it with fresh PCs. Finally, `add_seq` recursively adds the instructions in `cs` to the CFG.

Two "forward" and "backward" theorems specify the relation between source and target CFGs. The first, theorem 4.1, specifies how an instruction *i* appearing in the source CFG is reflected as a sequence in the target CFG. The conclusion asserts that `seq` is related to a concretized sequence `cs` by the inductive relation `concretized`, which abstracts the `cseq` monadic function, and that the target CFG respects the specification generated by function `spec` for sequence `cs`.

Theorem 4.1. *Forward correctness of CFG rewriting*

```
tr_code_TF code st = ((), st') →
code!pc = Some i →
tr_instr_TF pc i = (seq, cont) →
∃ cs, concretized pc seq cont cs
      ∧ spec NoPred st'.(code) cs
```

```
Fixpoint spec mode code cs : Prop :=
  match cs with
  | Cseq pc (CSi i (Cseq pc1 cs1)) ⇒
    code!pc = Some i
    ∧ (mode = Pred → uni_pred code pc1 pc)
    ∧ spec mode code (Cseq pc1 cs1)
  | ... end.
```

Listing 4. Specification generator

An excerpt of `spec` is presented in listing 4; it recursively generates a conjunction of assertions about nodes and edges in the CFG. The case presented here is that of the `CSi` concretized sequence constructor, which corresponds to sequence constructor `Si`, and denotes a sequence starting at `pc` with first instruction `i` and successor sequence `cs1` starting at `pc1`. The specification indicates that the instruction at `pc` in the CFG is `i`. Then, `spec` generates the specification for the successor sequence starting at `pc1`. The `spec` function has an optional `Pred` "mode", which triggers the generation of assertions about the uniqueness of backward edges (`uni_pred`) in the CFG.

Theorem 4.2. *Backward correctness of CFG rewriting*

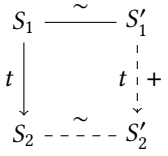
```
tr_code_TF code st = ((), st') →
st'.(code)!pc = Some i →
∃ pcf i' seq cont cs,
  code!pcf = Some i'
  ∧ tr_instr_TF pc i' = (seq, cont)
  ∧ concretized pc seq cont cs
  ∧ cs!pc = Some i
  ∧ spec Pred st'.(code) cs
```

This mode is only used in the theorem above, which, assuming the existence of an instruction `i` in the target CFG, asserts the existence of an instruction `i'` in the source CFG which was translated into the sequence surrounding `i`.

The proofs of these two generic theorems are established once and for all. The proof effort is reasonable (around 2300 lines of proof in total). They are then specialized for each countermeasure, to specify the content of the generated CFG. The proof of their specialization exploits the general theorem, and usually proceeds by case analysis on the source instruction and simplification of the `tr_instr` and `spec` functions.

4.2 Correctness Proof

As stated earlier, CompCert is a formally verified compiler, in the sense that it comes with a proof that the semantics of the source program are preserved in the generated assembly. As the compiler is essentially a sequence of passes, this proof is built by composing the correctness proofs of each pass. When adding a new pass, such as the one implementing the CFI transformation described above, we must provide a corresponding correctness proof of forward simulation.



if $P \vdash S_1 \xrightarrow{t} S_2$ **and** $S_1 \sim S'_1$
then $\exists S'_2$, $\texttt{compile}(P) \vdash S'_1 \xrightarrow{t}{}^{+} S'_2$
and $S_2 \sim S'_2$

Figure 5. Forward simulation diagram and obligation

As a first approximation, this means proving that any step $P \vdash S_1 \xrightarrow{t} S_2$ in the source is reflected by an identical step $\texttt{compile}(P) \vdash S_1 \xrightarrow{t} S_2$ in the target. The actual proof is often more complex, for two reasons. First, the source step may actually be reflected by more than one steps in the target; this is particularly true when one instruction is replaced by a sequence of instructions. Second, the states manipulated in the source and target programs may not be identical, but merely *similar*: for instance, the content of registers or the memory layout may change without affecting the overall correctness. To account for this, forward simulation is proven with regard to a "matching" simulation relation between source and target states, written $S_1 \sim S'_1$. To prove a pass, we must first exhibit this relation, and prove the theorem presented in fig. 5. It states that, if a state S_1 may step to S_2, and is simulated by state S'_1, then S'_1 may do one or more steps and reach S'_2 which simulates S_2.

In practice, the most difficult part of any correctness proof is often to find and define a suitable simulation relation; proving the simulation theorem is then mostly mechanical. As an example, we present in fig. 6 an excerpt of the simulation relation used for the proof of correctness of the pass implementing the CFI transformation. The first rule defines simulation between standard states (ie from constructor **S**). It indicates that state S' simulates S if the current registers of S' simulates that of S in the sense specified by the second rule: all the registers used in the source function must contain the same values, and the new `gsr` register must contain the signature of the instruction to be executed. The second premise of the first rule imposes this relation recursively to each call frame in the call stacks. Because the countermeasure involves neither the call stack nor the memory, the first rule imposes them to be identical within S and S'. Finally, S and S' must also have the same program counter: each source instruction and its translation start at the same *pc*.

This relation is fairly straightforward and similar to that used for several RTL optimization passes. It is not too difficult to prove that it is indeed a simulation relation. The proof exploits the forward rewriting specification (theorem 4.1) to identify the sequence of instructions involved in the target multi-step derivation $\texttt{compile}(P) \vdash S'_1 \xrightarrow{t}{}^{+} S'_2$.

4.3 Attacker and Robustness Models

The methodology described up until now allows us to implement and verify the semantic correctness of a countermeasure. This is only part of our overall objective: we also want

$$\frac{R \overset{regs}{\sim}_{f,pc} R' \qquad \Sigma \overset{stk}{\sim} \Sigma'}{\mathbf{S}(\Sigma, f, \sigma, pc, R, M) \sim \mathbf{S}(\Sigma', \mathtt{compile}(f), \sigma, pc, R', M)}$$

$$\frac{\forall r \in \mathtt{regs}(f),\ R'(r) = R(r) \qquad R'(\mathtt{gsr}(f)) = \mathsf{Vint}(\mathtt{get\_sig}_f(pc))}{R \overset{regs}{\sim}_{f,pc} R'}$$

$$\frac{}{\epsilon \overset{stk}{\sim} \epsilon} \qquad \frac{R \overset{regs}{\sim}_{f,pc} R' \qquad \Sigma \overset{stk}{\sim} \Sigma'}{F(r, f, \sigma, pc, R).\Sigma \overset{stk}{\sim} F(r, \mathtt{compile}(f), \sigma, pc, R').\Sigma'}$$

Figure 6. Simulation relation for CFI

to prove the *robustness* property, which states that, once protected by the countermeasure, the program effectively resists an attack. To state and prove this property, we must first define and encode the *attacker model*, which specifies the classes of disruptions the attacker may inflict to the program.

Since we implement countermeasures as RTL transformations, the attacker model must also be specified at the RTL level. This necessarily makes the attacker model an abstraction of hardware attacks. This level of abstraction is acceptable, as long as the high-level attacker model represents a sound over-approximation of the behaviors that may be triggered by physical attack. In the following, we discuss, for each attacker model introduced, the corresponding concrete attacks and the soundness of the abstraction.

An attacker model specifies the possible behaviors of a program under an attack. Therefore, it is natural to formalize it as an extension to the standard RTL semantics, which specifies the nominal behavior of the program. Specifically, we encode fault attacks as new, "faulty" transitions in the step-or-fault relation, written $P \vDash_{\mathsf{F}} S_1 \xrightarrow{t} S_2$. The basic rules defining this relation are presented in fig. 7. Any step transition allowed in the standard model is also allowed in this extended model. There also is a transition modelling the behavior of the `catch()` instruction: entering the special state **Caught**; this abstracts halting the program with an error.

The last rule models a fault injection; specifically, an inversion of conditional branching, the fault against which the countermeasure described in the previous section protects. The inversion is naturally encoded by inverting the branch destinations pc_1 and pc_2 in the conclusion of the rule. This model abstracts attacks that would alter conditional flags (zero flag, sign flag) in the program status register before a conditional branch [12]. It also covers an instruction skip fault occurring during the execution of the conditional branch [5]. We consider that a fault injection is an observable event, and record it in the trace as $TI(f)$, which specifies that a test inversion occurred during the execution of function f.

Having faults explicitly appear in the trace allows us to state hypotheses on the attacker in our robustness theorems: on the presence of a fault during an execution, and in the case

$$\frac{P \vdash S_1 \xrightarrow{t} S_2}{P \vDash_{\mathsf{F}} S_1 \xrightarrow{t} S_2} \qquad \frac{f.(pc) = \lfloor \mathbf{catch} \rfloor}{P \vDash_{\mathsf{F}} \mathbf{S}(\Sigma, f, \sigma, pc, R, M) \xrightarrow{\epsilon} \mathbf{Caught}}$$

$$\frac{f.(pc) = \lfloor \mathbf{cond}(cond, \vec{r}, pc_1, pc_2) \rfloor \qquad cond^{\#}(R(\vec{r})) = \lfloor b \rfloor}{P \vDash_{\mathsf{F}} \mathbf{S}(\Sigma, f, \sigma, pc, R, M) \xrightarrow{TI(f)} \mathbf{S}(\Sigma, f, \sigma, b\ ?\ pc_2 : pc_1, R, M)}$$

Figure 7. Transitions in faulted model, with test inversion

of attack $TI(f)$, that function f is actually protected against this attack. Let us now introduce our formalization, sketched in definition 4.3, of the robustness property for single-fault attacks. We say that a program P is robust against a single-fault attack belonging to model F if, for any execution stepping from an initial state S_0 to state S, with a fault injection occurring at the last position in the trace, then continuing the execution may only lead to two possible outcomes: (i) either the fault gets caught, that is, execution reaches a **Caught** state, or (ii) the fault did not actually have any observable side effects—this is modeled by the faulted execution "merging" back with the standard execution at a future state S'.

Both scenarios respect a very important property: *the trace of the faulted execution does not contain any observable events that do not appear in the nominal execution, other than the fault event itself.* This ensures that the attacker cannot use fault injection to leak secrets through new observable events. The property also guarantees that the steps following the fault are well-defined. This ensures that the fault can not lead to undefined behavior, which the attacker could exploit.

Definition 4.3. Robustness to a single-fault attack F

$$\begin{aligned}
&\mathsf{resists}^F(P) \overset{def}{=} \\
&\quad \textbf{if}\ \ \mathsf{initial_state}\ P\ S_0\ \textbf{and}\ P \vDash_{\mathsf{F}} S_0 \xrightarrow{t_1+F}{}^{\star} S\ \textbf{and}\ \mathsf{nofault}\ t_1 \\
&\quad \textbf{then}\ \ P \vDash_{\mathsf{F}} S \xrightarrow{\epsilon}{}^{\star} \mathbf{Caught} \\
&\quad \textbf{or}\ \ \exists\ S'\ t_2,\ \mathsf{nofault}\ t_2 \\
&\qquad\qquad \textbf{and}\ P \vDash_{\mathsf{F}} S \xrightarrow{t_2}{}^{\star} S'\ \textbf{and}\ P \vdash S_0 \xrightarrow{t_1+t_2}{}^{\star} S'
\end{aligned}$$

This characterization of robustness encodes a fundamental assumption about the attacker model: only functions in the current compilation unit may be attacked. Indeed, in CompCert, a call to an external function may emit an event, constrained by a few hypotheses. We added a new hypothesis stating that this event may never be a fault. This is a necessary assumption, regardless of the compiler, as we cannot guarantee the security of any function not compiled with security features, and would therefore not be able to prove any robustness result about a program calling vulnerable external functions. In practice, this hypothesis is reasonable, as the security-critical components of an application, that an attacker would target, are generally well-identified by the developer who would therefore make sure that they are compiled by CompCert and protected.

```
1  induction P ⊧ S0 --t1+F-->★ S as [|IH|STEP].
2  - (* F in empty trace is absurd *) ...
3  - (* F in previous steps *) apply IH...
4  - (* F in current step *)
5    inversion (STEP: P ⊧ S1 --F--> S).
6    apply theorem 4.2 in f.(pc) = ⌊cond(c, a⃗, pc1, pc2)⌋.
7    (* f.(pca) = ⌊assert(= sig, gsr, pcc1)⌋
8       f.(pcc1) = ⌊cond(c, a⃗, pcxt, pcxf)⌋
9       f.(pcxf) = ⌊op(sig^sig1, [], rts, pcc2)⌋ ...
10      f.(pcc2) = ⌊cond(c, a⃗, pct, pcf)⌋ ...
11      EQ: pc = pcc1 ∨ pc = pcc2 ∨ pc = pct ∨ pc = pcf *)
12   destruct EQ.
13   + (* fault in pre-test *) ...
14   + (* fault in test *)
15     repeat inv_previous_step.
16     (* GSR: R(gsr) = sig, RTS: R(rts) = sig^sig1 *)
17     (* goal: P ⊧ S --ϵ-->★ Caught *)
18     repeat econstructor...
19   + (* fault in true branch check *) ...
20   + (* fault in false branch check *) ...
```

Listing 5. Robustness proof sketch

4.4 Robustness Proof

Proving that the CFI countermeasure is robust against the Test-Inversion attack model amounts to proving theorem 4.4, where the conclusion is a specialization of the above robustness definition. The theorem has only two hypotheses: that the source program is well-formed statically, and that the function being attacked was actually marked to be protected. While establishing the proof of this theorem, we have identified a general proof structure that should apply for any countermeasure and attack model fitting our framework; it is outlined in listing 5.

Theorem 4.4. *Robustness of the CFI countermeasure*

if $\mathsf{well\_typed}(P)$
and $\mathit{CFI} \in \mathsf{attributes}(f)$
then $\mathsf{resists}^{TI(f)}(\mathsf{compile}^{\mathit{CFI}}(P))$

The proof proceeds by induction over the multiple-step semantic derivation (line 1), until it finds the faulted step (line 4). This part of the proof is generic, and is provided as a parameterized lemma. Then, the faulted step is analyzed by inversion (line 5), which exposes the type of instruction being attacked, since semantic rules are syntax directed. In this example, we know that the instruction being attacked must be a conditional branching instruction. The backward specification (theorem 4.2) is then applied (line 6) to retrieve the shape of the CFG around the attacked instruction (lines 7-11). In particular, this exposes the immediate predecessors to the attacked instruction, which were necessarily executed

$$\frac{f.(pc) = \lfloor \mathbf{assert}(cond, \vec{arg}, pc_1) \rfloor \quad \forall i, \mathrm{eval\_arg}(arg_i, R, M, \sigma) = \lfloor vs_i \rfloor \quad cond^{\#}(\vec{vs}) = \lfloor \mathbf{true} \rfloor}{P \vdash \mathbf{S}(\Sigma, f, \sigma, pc, R, M) \xrightarrow{\epsilon} \mathbf{S}(\Sigma, f, \sigma, pc_1, R, M)}$$

Figure 8. Semantic rule for **assert**

before the fault. Inverting the corresponding steps recovers information about the processor state during the fault, with in particular the content of registers and memory (lines 16-17). We automated this complex inversion process with reusable lemmas and tactics (line 15).

Some necessary state information may not be recoverable from these immediate predecessors, because it depends on updates occurring earlier in the program. For instance, in the CFI transformation, we must show that GSR always contains the signature of the block being executed. This is encoded in the second rule of the correction invariant presented in fig. 6. To propagate this knowledge to the robustness proof, we introduced a new instruction `assert` to the RTL language, which semantics are given in fig. 8. An `assert` does not update the state other than the current PC. However, it imposes that its condition evaluates to **true** on the value of its arguments $\vec{arg}$, which may be either in registers or memory. We introduce this instruction in the hardened code; for instance, for the CFI transformation described in fig. 3, we add the instruction `assert (x1 = 33)` at the start of the sequence. In the correctness proof, it is easy prove that this instruction executes properly thanks to the correction invariant. In the robustness proof, inverting the local semantic derivation for the `assert` instruction recovers the global invariant.

Finally, knowing enough about the current state, we can reason on the successors of the attacked instruction to build the continued semantic derivation. To satisfy the robustness theorem, we prove that these steps lead either to the `catch` instruction or back to the nominal state (line 18).

4.5 Informal Extra Robustness Against ROP

The CFI countermeasure described in section 3 is likely to be useful against attacks other than faults. For instance, *Return-Oriented Programming* (ROP) [16] consists in overflowing the call stack and writing into it a sequence of return addresses that the processor will use to jump into code fragments (often called "gadgets") each ending in a function return instruction, so that the effectively executed sequence is the composition of these gadgets. With our countermeasure configured (through a dedicated command-line option) to insert an additional GSR check before each return instructions, ROP may not jump in the middle of protected functions, as the GSR would then not be properly initialized, which would trigger the check. However, since it seems difficult to identify an attacker model (what is and what is not an interesting composition of gadgets), formally proving robustness against such attacks seems elusive within our framework.

5 Application to Inter-procedural CFI

Even if the CFI countermeasure does protect from ROP, it still has one limitation: it does not prevent an attacker from jumping to the head of a function and executing it. Indeed, the countermeasure initializes GSR at function start and checks that execution has followed permissible control flow only since function start. SWIFT solves this issue with a complementary inter-procedural variant [22].

5.1 Inter-Procedural SWIFT

This inter-procedural countermeasure works by adding GSR as an extra in-out parameter—passed through a pointer—to functions. Each internal function `g` is therefore associated to a *protected* version, called `ipcfc.g`, having the extra parameters. In the transformed code, while the `g` function remains with the original signature, its body is a simple wrapper of the protected version, initializing the hardened execution.[3]

Let us explain this idea on the example in fig. 9. It gives the code generated by this transformation, for source code containing a function `g(x1)` returning `f(x1*3)` and `f(x1)` itself returning `x1+1`. The countermeasure computations are shown in gray whereas the original computations are represented in black. The execution starts in function `g` at top-left, which first allocates and initializes some space on the stack for GSR. Then, `g` calls its specialized, protected version, `ipcfc.g`, by passing GSR (in pointer `x5`), the interprocedural RTS (in integer `x7`), which contains the exclusive-or of GSR and the signature of `g`, and its source argument. At the start of `ipcfc.g`, RTS is checked against the entry signature, `190`. Then, `ipcfc.g` directly calls `ipcfc.f`, the protected version of `f`, making the same updates to RTS and GSR. At the end of their execution, the protected functions set the value pointed by GSR to be the exclusive-or of the caller and callee exit signatures; this is checked after returning to the caller. As with the intra-procedural SWIFT, this approach catches both immediate attacks, and the delayed effects of a past attack, since any corruption of GSR gets propagated through the exclusive-ors to further checks. This transformation is implemented using the generic framework presented in section 4.1.

5.2 Correctness Proof

The simulation relation used for the correctness proof of this pass is more complex than that of the intra-procedural transformation, for two major reasons. First, at any point in the execution, the stack Σ of the transformed program may contain more stack frames than that of the source program, due to the execution traversing initialization functions. The shape of the stack is fairly complex, as calls may occur between initialization, protected, and unprotected functions.

[3] This approach does not work with variable-length argument lists: the wrapper would not know how many arguments are to be copied. We therefore disable this countermeasure for variadic functions. The main use of such functions is `printf()` and variants, which anyway are likely not to be used in a security-critical system with reduced code memory footprint.

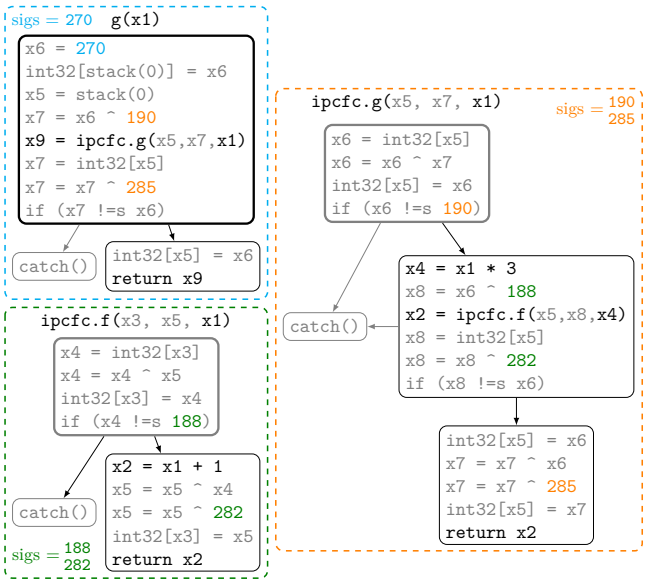


Figure 9. Example of the inter-procedural CFI

Second, the memory layout M of the target program may be different from that of the source. Indeed, the pass may change this layout in two respects: (i) introduce new functions, which essentially changes the text segment of the program, and therefore its initial memory layout (ii) allocate more memory in initialization stackframes to store the GSR. To account for this change, we reason modulo a memory injection, described in [31, §5.4], which specifies where each source memory block is located in the target layout.

5.3 Function Specialization

As seen in the example, the transformation produces, for each source function, two functions: one with the original name, and one with two supplementary parameters and a derived name. The "initialization" function bearing the original name needs to be retained in two cases: (i) if its address is taken and stored in a pointer, because GSR cannot be adjusted and checked across indirect calls, as the signatures must be determined statically (ii) if it has external linkage, because it may be called from other modules; adding an extra parameter would break the Application Binary Interface (ABI).

Such a transformation is closely related to those of *function specialization*, when a compiler emits different versions of the same function specialized according to the value of certain parameters, to allow further optimizations in the specialized versions. This class of optimizations is not present in CompCert at the moment. We describe below the extensions to CompCert's linking mechanisms that we introduced for our inter-procedural countermeasure. They should be also useful for other kinds of function specialization.

Hacking CompCert's Model of Linking. Introducing new functions in the program is normally not supported by the *linking* invariants that CompCert maintains to ensure

the soundness of separate compilation [28, approach A]. Indeed, consider a program consisting of two modules `A` and `B`, respectively containing function `f`, and functions `f1` and `g`. Imagine the compiler decides to specialize `f` into `f1`; then there would be a name clash between the compiled modules whereas none was present in the source program.

In order to avoid this issue, we extended the naming system of CompCert to support structured, *qualified names* to specify which names may or may not appear before a compilation pass. For instance, all the functions introduced by the interprocedural SWIFT pass have a name prefixed by `ipcfc`. Before running, the pass defensively checks that no function name in the source has the same prefix. From this check, we can deduce that no name collision can be introduced by the pass, and complete the proofs of linking preservation.

Pruning Unused and Unprotected Functions. A further optimization pass in CompCert discards unused `static` (module-local) functions. We have also set up CompCert's code generation to allow the linker to remove non-`static` unused functions from the final executable.[4] However, since we keep the normal calling conventions for functions outside the current module (because we do not know if they are compiled with CompCert or another compiler, and, if they are compiled with CompCert, if they use the interprocedural protection), procedure calls between modules will still be unprotected and thus retain the unprotected version. CompCert views external calls as events, and its correctness argument is that it preserves the sequence of events; thus it is not possible, with current definitions, to replace one external call by another. Replacing external calls by other "equivalent" ones would likely entail some notion of refinement of events and perhaps some more refined notions about separate compilation, which fall outside the scope of this paper. In the meantime, a workaround for small embedded code is to put the whole program inside a single C module.

5.4 Robustness Proof

The formalized attacker model for this countermeasure includes two possible attacks, presented in fig. 10. The first, $CS(f, id)$, corresponds to skipping a call to the function named id in function f. Although instruction skips are not inherently related to function calls, skipping a call may be particularly interesting to an attacker, as it may allow bypassing a lot of the program in a single fault; this protection is therefore useful. The second, $CW(f, f')$, models calling the wrong function f' from function f. This model only encodes jumping to the entrypoint of the function. Protection against an attacker jumping to the middle of the function is ensured by the intra-procedural CFI countermeasure, thanks to the propagation of a corrupted intra-procedural `GSR`.

We have proven that the inter-procedural SWIFT countermeasure effectively protects the program from these two attack models, in the sense of definition 4.3. As expected, this is true only if all functions involved (caller, expected callee and fault callee) are protected by the countermeasure. The proofs for each attack model proceed as described in section 4.4. In particular, they exploit an assertion about the value pointed to by `gsr` to recover global information from the local derivation.

6 Evaluation and Practical Concerns

The previous sections introduced a methodology to prove the resistance of countermeasures against a given attacker model. However, these proofs are limited in scope: they are only established on the code right after the transformation implementing the countermeasure. Semantic-preserving compilation passes occurring later in the compilation chain may still undo these protections. The main concern is optimization passes, which may detect that defensive checks are functionally redundant, and eliminate them.

Positioning our Countermeasures within CompCert. Because our countermeasures require extra temporary registers, they must be applied before register allocation: we thus implement them within RTL. Moreover, our inter-procedural CFI cannot be applied to the "tail call" control transfer instruction, which destroys the current stack frame and calls the target function. It thus must be applied before tail call elimination, which replaces normal function calls in tail position by this instruction, because this optimization may still be useful on unprotected functions. And, since this pass is the first RTL optimization, we introduced our countermeasures before all RTL optimizations (including constant propagation, common subexpression elimination, etc).

Experimental Evaluation of Robustness. To check if RTL optimizations destroy countermeasures, we propose a methodology to evaluate the robustness of the intermediate, optimized RTL program. We build our approach on top of Lazart [14, 38], an established tool for experimenting with fault attacks in software.[5] Lazart operates on the LLVM Intermediate Representation (IR) through concolic execution of a modified program, using the Klee concolic execution engine. CompCert outputs assembly code, not LLVM intermediate code. We thus added a CompCert backend for LLVM. This backend is not formally verified. It branches off Chamois CompCert at the level of the BTL representation, a variant of RTL structured with basic blocks [26]. Since the LLVM IR uses Single Static Assignment (SSA) form, as opposed to RTL/BTL, pseudo-registers have to be renamed to prevent multiple assignment.

[4] `-ffunction-sections -Wl,--gc-sections` on Linux `-Wl,-dead_strip` on MacOS.

[5] There exist special circuit boards that allow experimenting with actual power or clock glitches, specialized equipment for laser pulses etc., but such hardware experiments would be way beyond the scope of this paper.

$$\frac{f.(pc) = \lfloor \mathbf{call}(id, \vec{r}, r_d, pc') \rfloor}{P \vdash_{\mathrm{F}} \mathbf{S}(\Sigma, f, \sigma, pc, R, M) \xrightarrow{CS(f,id)} \mathbf{S}(\Sigma, f, \sigma, pc', R, M)} \qquad \frac{f.(pc) = \lfloor \mathbf{call}(id, \vec{r}, r_d, pc') \rfloor}{P \vdash_{\mathrm{F}} \mathbf{S}(\Sigma, f, \sigma, pc, R, M) \xrightarrow{CW(f,f')} \mathbf{C}(F(r_d, f, \sigma, pc', R).\Sigma, f', R(\vec{r}), M)}$$

Figure 10. Attacker model for inter-procedural SWIFT

	No CM, 00			CM, 00			CM, 01			CM+cpy, 01		
Prog.	#IP	1F	2F	#IP	1F	2F	#IP	1F	2F	#IP	1F	2F
vp	4	3	3	16	0	5	15	1	4	16	0	5
ark	1	1	0	4	0	2	3	1	0	4	0	2
aes	2	2	3	8	0	4	8	0	4	8	0	4
fu	11	4	13	41	0	5	23	3	1	34	0	3

Table 1. Results of Lazart simulation (01 is with optimization; there are no higher optimization levels in CompCert)

Lazart proposes some benchmark programs, each coupled with an attack objective, modeled by a boolean predicate on the variable and memory at the end of the execution, and designed to only be satisfiable during a faulted execution. We compiled four of these programs using our instrumented version of CompCert: `vp` is a more complete version of the pin verification program described early in this paper; `aes` is an implementation of the Advanced Encryption Standard algorithm; `ark` is the function implementing key scheduling; `fu` is a skeleton of a firmware updater program. We ran Lazart on the LLVM code compiled from these programs, using the Test Inversion (TI) attack model[6]. The results of this analysis are presented in the first 3 columns in table 1. Each column corresponds to a different configurations of CompCert. Let us first focus on the first three: with neither countermeasure nor optimization, with countermeasure but without optimization, and with both countermeasure and optimizations. Each column indicates the number of possible fault injection points in the program (#IP). For the TI attack model, there is one injection point for each conditional branching instruction. Then comes the analysis results, measured as the number of traces reaching the attack objective with either one (1F) or two (2F) faults.

There are two major takeaways. The first is that, when countermeasures are activated and optimizations are deactivated, all one-fault (1F) traces are neutralized. As expected, the countermeasures are robust relative to the single-fault model implemented in Lazart, but not against two-fault attacks. The second is that, when optimizations are also activated, some one-fault traces subsist for all examples except `aes`. This means that, as expected, optimizations do remove at least part of the countermeasures.

[6]Lazart does not natively support inter-procedural attack models, and it is therefore not possible to evaluate our inter-procedural countermeasure.

Protecting Countermeasures Against Optimizations. We have identified that the Constant Propagation and Common Subexpression Elimination optimizations are responsible for removing the countermeasures. There is an obvious way to fix this issue: selectively disable these optimizations for protected functions. However, this would mean sub-optimal performances for the generated code. Instead, we opted for a more elegant solution, based on "opacification" of redundant code, as proposed in [51]. The goal is to prevent optimizations from identifying the redundancy in the countermeasure logic, and therefore protects countermeasures from optimizations. In practice, our opacification mechanism proceeds in two steps.

First, each countermeasures transformation adds `observe` instructions: intuitively, `observe`$(x_1, ..., x_n)$ should be understood as a side-effect that both depends on and modifies the value of registers $x_1, ..., x_n$; therefore, instructions that write or read these registers should not be moved across the `observe`. For instance, in the example of fig. 3, three observers are added: First `observe(x3)` before the second conditional: it ensures that the two conditions cannot be merged by Common Subexpression Elimination. Second, one `observe(gsr)` before each condition checking `gsr` (i.e. `x1` on fig. 3): they protect `gsr` from Constant Propagation. In the formal semantic model, `observe` does not actually have any effect: this makes its introduction easy and facilitates the proofs of the countermeasure pass.

Second, the actual opacification is introduced in a dedicated pass which runs after each transformation introducing `observes`. It expands each "`observe`$(x_1, \ldots, x_n)$" instruction into a sequence of opaque copies "x_1`=copy(`x_1`,`pc_1`);` $\ldots$; x_n`=copy(`x_n`,`pc_n`)`". Semantically, `copy(`x`,`pc`)` returns x, but optimization passes ignore that fact. Therefore, because optimizations consider `copy` like an uninterpreted function symbol, they have to remain correct for any interpretation of the symbol. The second argument of `copy` is a uniquely generated number (from identifiers of the CFG) that forces optimizations to distinguish two calls of `copy` on the same first argument. Instructions "x`=copy(`x`,`pc`);`" are simply eliminated at the end of the backend, when generating the final assembly code.

From our experiments, the technique seems successful, as shown by the last column of table 1, where opaque copies are inserted and all optimizations are active: there is no successful one-fault attack left. However, although countermeasure opacifications is common in the field [51], we have currently no formal proof that this approach is sound, in the sense that

	max	Q3	median	Q1	min	avg
CM+O0	347.15	134.05	87.99	52.26	-1.2	98.68
CM+cpy	419.7	79.75	42.68	16.35	-1.43	59.49

Table 2. Running Times Loss of CM+O0 and CM+cpy wrt O1.

it preserves robustness through the compilation process. In order to establish one, we would need to adapt both the deterministic semantics of CompCert backend and its forward simulation framework (as discussed in Conclusion section).

Other Testing. We also successfully applied the testing methodology of Chamois CompCert [35] : both nonregression tests and benchmark evaluation. The purpose of benchmarking was to measure the loss of performance due to countermeasures, and how the optimizations still improve performance after opacification of countermeasure. Our benchmarking compares 3 configurations of our CompCert version: • O1: with optimizations but without countermeasures nor opacification • CM+O0: with countermeasures but without opacification nor optimizations • CM+cpy: with opacified countermeasures and optimizations. For CM+O0 and CM+cpy, all our countermeasures (including the one against ROP of section 4.5) are systematically applied to all functions.

We ran their generated code on a RISC-V 64 bits architecture, on a SiFive in-order dual-issue U740 core (HiFive Unmatched), over a set of 173 procedures, each compiled with 3 sets of initial state, using Chamois benchmarking framework and methodology [35, §6]. For each of these running times, we compute the *running time loss* wrt the fastest config, namely O1. The loss for a configuration C wrt O1 is computed using the formula: $\text{loss}(C) = ((C - \texttt{O1})/\texttt{O1}) \times 100$. The summary of the losses are given in table 2 (lower is better; Q1 and Q3 are respectively the first and third quartile). The loss variability is pictured in fig. 11 on a representative subset of these programs. The loss variability being quite high, a too small benchmark or with too fast programs would give insignificant results. Therefore, Lazart benchmark did not suit performance benchmarking.

These results show that, except on a few pathological cases, CM+cpy is much faster than CM+O0: optimizations are still useful despite opacification of countermeasures. The median loss is two times lower with CM+cpy than with CM+O0. We also see that applying systematically our limited countermeasures is already costly: on average, CM+cpy generates a code 1.6 times slower than O1. This illustrates the importance of carefully selecting the code parts to harden.

7 Related Works and Discussion

Countermeasures Against Fault Attacks. Reasoning on compiler-assisted countermeasures against fault injections [7, 24, 39] requires a *fault model*. Fault models range from source level (e.g., branch inversion [38], which inspired our work) to hardware description level (e.g., bit flips in the Register Transfer Level [47, 48]). Precise fault models require a fine analysis of microarchitectural behaviors and thus may need to expose hardware details [29]. For a compiler-assisted countermeasure with hardware support against "*fetch skip*" attacks, Michelland et al. [33] propose a formal pen-and-paper model based on such a microarchitectural analysis. They prove that faulty executions (within the attack model) are well-detected by the hardened program. We recognize here a well-known dichotomy: abstract models give more tractable security proofs whereas precise models give stronger guarantees. Experience with cryptographic protocols indicates that both kinds of models are useful [13].

Given-Wilson and Legay [25] formalize a general notion of *fault injection* and *countermeasures* within the framework of parallel Turing machines. For the property expected of countermeasures, they do not consider our conjunction of *correctness* and *robustness*, but a stronger property called *effectiveness*, which they define as the *functional equivalence* of *the composition of the fault and the hardened program* with the *source program*. They prove interesting results about this notion (such as the impossibility to have an effective countermeasure against all fault models). On the one hand, we think it is a too strong property, because it does not allow the countermeasure to abort execution in case of attack. On the other hand, aborting in case of attack, which our notion of robustness allows, permits *denial-of-service* attacks.

Formally Verified Countermeasures. Control-flow security is a well-established topic since the seminal work of Abadi et al. [1]. Many software countermeasures [17, 43] aim to protect control-flow against software vulnerability attacks (exploiting memory corruption errors). The mathematical models of these software attacks, such as initially introduced in [2] or more recently in [20], generally assumes that attackers cannot modify the behavior of instructions (e.g., attackers can modify data but not code). This does not reflect how hardware attacks can corrupt control flow.

In recent years, the emerging field of *secure compilation* achieved *formally verify compartmentalization* [4, 46]. Compilers—possibly with hardware support—provide protections against low-level software exploits of undefined behaviors. The formal guarantee, called *robustly safe compilation* [3, 37], ensures that compilation preserves security properties stated at the source level of safe components *even when linked to vulnerable ones*. This work has been integrated within a branch of the CompCert formally verified compiler. Compartmentalization however does not protect safe components against hardware attacks.

Official releases of CompCert do not implement any security feature. The *Chamois* branch of CompCert implements three features available in gcc and clang: *branch target identification*, *stack canaries* and *return address authentication*, all three meant to prevent software attacks that divert the

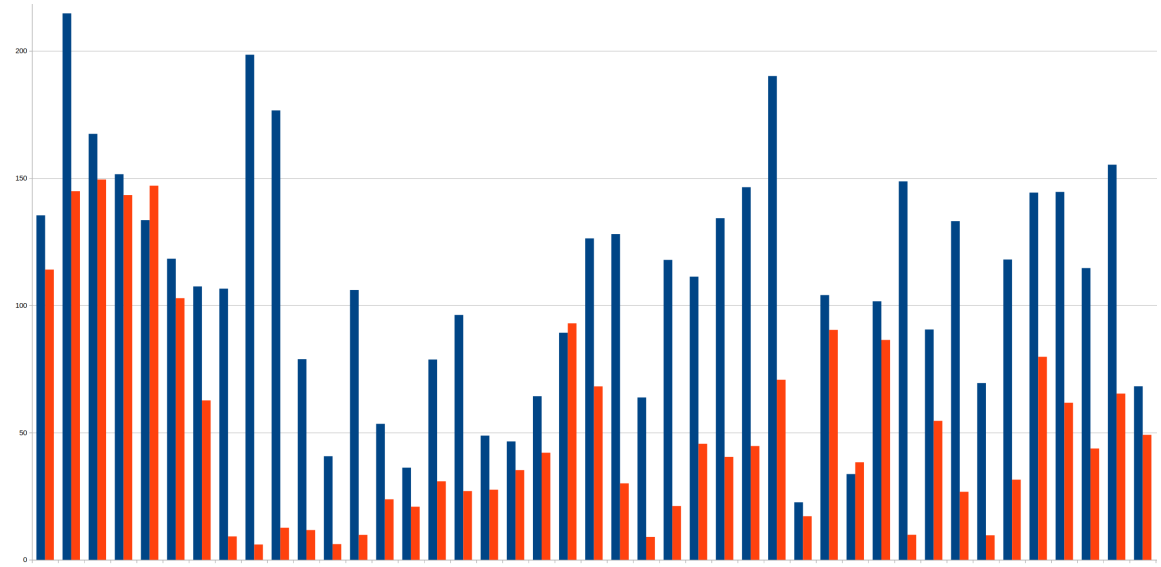


Figure 11. Representative Running Times Loss (in %) of `CM+O0` on **left bars** and `CM+cpy` on **right bars** wrt `O1` (zero axis).

control flow by altering code pointers, generally using buffer overflows. Soundness was proved, but not robustness [34].

Stack canaries are memory locations that must be overwritten by a contiguous buffer overflow in the stack before it touches return addresses; an incorrect canary value at function return aborts execution. We think that our approach could be used to prove them robust with respect to a semantics where the program does unauthorized memory accesses, though the attacker model seems awkward due to contiguity.

Previously, another fork of CompCert [49] inserted control flow protections based on the hardware support of a hardened RISC-V core [42]. The countermeasure is expected to protect against some hardware attacks, such as instruction skips, but it does not protect against branch inversion. Moreover, authors only formally verified the correctness of their protection: their attacker model remains unclear.

Vellvm [54] is a Coq formalization of the semantics of the LLVM IR, which was illustrated with a formally verified version of SoftBound, a code instrumentation pass hardening program against memory violations [36]. During formalization, they discovered semantic corner cases where the original SoftBound would fail to detect a memory violation.

Some hardware countermeasures against software vulnerabilities have also been formally verified in Coq [10]. Their countermeasure, a shadow stack implemented in a bounded array, can abort execution due to lack of memory even if there is no issue in the program. Thus, the formulation of their correctness theorem is weaker than ours, because they allow the countermeasure to stop legal executions of the program (with an error state different from those indicating an attack). They also prove a form of robustness theorem: any modification of a return address in the real stack leads to an error state. They do not consider hardware attacks.

Side-channel attacks (leakage of information through timing, electromagnetic emissions, etc.) are a concern especially for cryptographic applications. Naive implementations of cryptographic primitives (e.g., RSA where a multiplication is performed or not whether a secret key bit is 1 or 0, AES where S-boxes are implemented by table lookup…) must be eschewed in favor of secure implementations operating in *cryptographic constant time*, a notion defined by behaviors observable directly or indirectly by the attacker. Secure compilation must preserve cryptographic constant time in addition to correctness [8, 9]. The semantic approach is different from ours: we consider a nondeterministic semantics that includes faults from an active attacker, while they have no attacker but must preserve a more precise semantics.

Inter-Module Program Transformations. In [20], the authors describe a countermeasure for enforcing control-flow integrity at both the intra and inter-procedural level. Their implementation, as a plugin of the LLVM compiler, is able to protect calls between functions present in different source files linked into a unique binary, and even between functions from different binaries.

Our approach cannot, at present, protect such calls, because CompCert considers that external calls (calls to functions defined in other modules) are observable events, and no code transformation is allowed to alter the sequence of observable events. All external calls thus go through the normal calling convention, not the one with the extra parameter implementing the countermeasure. Going beyond this would involve a more general theory of modules, external calls, calling conventions, and linking, perhaps that of [53].

Verified CFG Transformations. One major difficulty in implementing CFG transformations is the lack of structure of

	Implementation	Correctness	Robustness
Common	736	842	1067
Intra CFI	163	680	427
Inter CFI	309	3262	492

Table 3. Size of the Coq development

RTL. We addressed this issue with our monadic framework of section 4.1. Such an issue was previously tackled in the Chamois fork of CompCert by BTL, a variant of RTL with a general representation of loop-free structured blocks [26]. The expressivity of BTL blocks seems, at first sight, slightly more general to that of our structured sequences, allowing for nested conditional branching, merging and exiting. We did not use the BTL language to implement our transformations because of some of its limitations. First, it imposes observable events to only occur at a final instruction of a block: this is not compatible with our attacker models in robustness proofs. Second, as function calls also need to be at such a final instruction, it would not be convenient for our interprocedural countermeasure, which inserts some defensive code after each function call.

Our approach is currently limited to transformations that replace one instruction with several ones; we do not know yet how to generalize it to sequences-to-sequences transformations, which would allow for defining a more general class of countermeasures. One solution would be using a structured representation like that of BTL; we would therefore need to remove the BTL limitations discussed above.

Another way would be to implement pattern-matching on RTL CFGs, as in [44] as part of a partially verified CFG rewriting engine. It would then still be difficult to prove forward simulation, which normally matches one source step to several target steps. Not following this scheme usually requires more complex simulation invariants.

8 Conclusion and Perspectives

We have successfully implemented and verified countermeasures in the CompCert end-to-end formally verified compiler. On the way, we have developed tools likely to be useful for other kinds of passes, including optimizations: a monadic framework to program verified CFG transformations and an extension to CompCert's separate compilation model to support function specialization. Our development and proof effort is summarized in table 3, in term of lines of code for the generic framework and both countermeasures. We think our approach may be valid for any class of attacks expressible by adding nondeterministic attack transitions, and any countermeasure that checks that certain invariants are preserved.

Promising applications of our framework could include duplicating computations or memory loads, or counting instructions. In this respect, we envision that the main difficulties will not be in the general design of invariants, but in "details" such as the value semantics of CompCert (for instance, due to undefined values, the equality test is not reflexive). Other countermeasures such as shadow stacks may prove more difficult to integrate, as they may introduce new traps in the hardened program (in case of stack overflows), which does not fit with the current notion of semantic correctness. Formalizing countermeasures against transient attacks (à la Spectre and Meltdown) involves semantic modeling of speculation [21], which is not only nondeterministic but also has a notion of cancellation of wrongly speculated behaviors, as well as some modeling of side channels.

One limitation of our work is that our robustness theorem and proofs only capture single-fault attacks. Considering attack scenarios with multiple fault being injected would require extending the theorem, with a definition allowing for interleaved faulted and non-faulted steps. We also expect that robustness proofs would become more complex, because of the combinatorial increase in the number of possible paths, which we do not know yet how to manage efficiently.

Another limitation is the absence of formal theorem that our opacification mechanism is sufficient to preserve the robustness property, proved on each hardening pass, by subsequent optimizations. As sketched in section 6, it seems difficult to establish such a theorem within CompCert, without a big redesign of its whole framework. In particular, it would be highly desirable to have a single notion of program simulation, possibly parametrized by the presence or absence of attacks, ensuring both correctness and robustness. For example, we would consider that an opacification operator such as "`observe`$(x_1, ..., x_n)$" enables "replaying" attacks within the semantics used in optimization proofs: this instruction mimics a kind of potential fault injection within registers $x_1, ..., x_n$. The overall goal would be to ensure that subsequent optimizations preserve the robustness w.r.t the attacker model by construction. We do not yet clearly envision what would be the right simulation relation. It might need expressive notions of nondeterminism that the forward simulation framework of CompCert cannot provide.

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 (https://www.pepr-cybersecurite.fr/projet/arsene/).

We are deeply indebted to Etienne Boespflug for his support on the Lazart tool, and to Olivier Lebeltel for his benchmarking of our Chamois version. We also thank anonymous CPP referees and our colleagues of PEPR Arsene for their useful feedback on this work.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (Alexandria, VA, USA) *(CCS '05)*. New York, NY, USA. https://doi.org/10.1145/1102120.1102165

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. A theory of secure control flow. In *Proceedings of the 7th International Conference on Formal Methods and Software Engineering* (Manchester, UK) *(ICFEM'05)*. Berlin, Heidelberg. https://doi.org/10.1007/11576280_9

[3] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2021. An Extended Account of Trace-relating Compiler Correctness and Secure Compilation. *ACM Trans. Program. Lang. Syst.* 43, 4, Article 14 (nov 2021). https://doi.org/10.1145/3460860

[4] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. 2018. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. https://doi.org/10.1145/3243734.3243745

[5] Ihab Alshaer, Gijs Burghoorn, Brice Colombier, Christophe Deleuze, Vincent Beroulle, and Paolo Maistri. 2024. Cross-layer analysis of clock glitch fault injection while fetching variable-length instructions. *Journal of Cryptographic Engineering* 14 (04 2024). https://doi.org/10.1007/s13389-024-00352-6

[6] H. Bar-El, Hamid Choukri, D. Naccache, Michael Tunstall, and C. Whelan. 2006. The Sorcerer's Apprentice Guide to Fault Attacks. *Proc. IEEE* 94, 2. https://doi.org/10.1109/JPROC.2005.862424

[7] Thierno Barry, Damien Couroussé, and Bruno Robisson. 2016. Compilation of a Countermeasure Against Instruction-Skip Fault Attacks. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems* (Prague, Czech Republic) *(CS2 '16)*. New York, NY, USA. https://doi.org/10.1145/2858930.2858931

[8] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.* 4, POPL (2020). https://doi.org/10.1145/3371075

[9] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. https://doi.org/10.1109/CSF.2018.00031

[10] Matthieu Baty, Pierre Wilke, Guillaume Hiet, Arnaud Fontaine, and Alix Trieu. 2023. A generic framework to develop and verify security mechanisms at the microarchitectural level: application to control-flow integrity. In *CSF 2023 - 36th IEEE Computer Security Foundations Symposium*. Dubrovnik, France. https://inria.hal.science/hal-04118645

[11] Nicolas Belleville, Karine Heydemann, Damien Couroussé, Thierno Barry, Bruno Robisson, Abderrahmane Seriai, and Henri-Pierre Charles. 2018. *Automatic Application of Software Countermeasures Against Physical Attacks*. Cham. https://doi.org/10.1007/978-3-319-98935-8_7

[12] Pascal Berthomé, Karine Heydemann, Xavier Kauffmann-Tourkestansky, and J. Lalande. 2012. High Level Model of Control Flow Attacks for Smart Card Functional Security. In *ARES '12*. https://doi.org/10.1109/ARES.2012.79

[13] Bruno Blanchet. 2012. Security Protocol Verification: Symbolic and Computational Models. In *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings (Lecture Notes in Computer Science, Vol. 7215)*. https://doi.org/10.1007/978-3-642-28641-4_2

[14] Etienne Boespflug, Abderrahmane Bouguern, Laurent Mounier, and Marie-Laure Potet. 2023. A tool assisted methodology to harden programs against multi-faults injections. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC) 23*.

[15] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. 1997. On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology — EUROCRYPT '97*. Berlin, Heidelberg.

[16] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*. ACM, 27–38. https://doi.org/10.1145/1455770.1455776

[17] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.* 50, 1 (2017). https://doi.org/10.1145/3054924

[18] M. Christofi, B. Chetali, L. Goubin, and D. Vigilant. 2013. Formal verification of a CRT-RSA implementation against fault attacks. *Journal of Cryptographic Engineering* 3, 3 (2013).

[19] Coq Development Team. 2020. *The Coq proof assistant reference manual*. Inria. https://coq.inria.fr/distrib/current/refman/

[20] Thomas Coudray, Arnaud Fontaine, and Pierre Chifflier. 2015. Picon: Control Flow Integrity on LLVM IR. In *SSTIC*. https://www.sstic.org/2015/presentation/control_flow_integrity_on_llvm_ir/

[21] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. 2023. ProSpeCT: Provably Secure Speculation for the Constant-Time Policy. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. https://www.usenix.org/conference/usenixsecurity23/presentation/daniel

[22] François de Ferrière. 2019. A compiler approach to Cyber-Security. 2019 European LLVM developers' meeting. https://llvm.org/devmtg/2019-04

[23] Louis Dureuil, Marie-Laure Potet, Philippe de Choudens, Cécile Dumas, and Jessy Clédière. 2015. From Code Review to Fault Injection Attacks: Filling the Gap Using Fault Model Inference. In *Smart Card Research and Advanced Applications - 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 9514)*. https://doi.org/10.1007/978-3-319-31271-2_7

[24] Johannes Geier, Lukas Auer, Daniel Mueller-Gritschneder, Uzair Sharif, and Ulf Schlichtmann. 2023. CompaSeC: A Compiler-Assisted Security Countermeasure to Address Instruction Skip Fault Attacks on RISC-V. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference* (, Tokyo, Japan,) *(ASPDAC '23)*. New York, NY, USA. https://doi.org/10.1145/3566097.3567925

[25] Thomas Given-Wilson and Axel Legay. 2020. Formalising fault injection and countermeasures. In *Proceedings of the 15th International Conference on Availability, Reliability and Security* (Virtual Event, Ireland) *(ARES '20)*. New York, NY, USA, Article 22. https://doi.org/10.1145/3407023.3407049

[26] Léo Gourdin, Benjamin Bonneau, Sylvain Boulmé, David Monniaux, and Alexandre Bérard. 2023. Formally Verifying Optimizations with Block Simulations. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (Oct. 2023). https://doi.org/10.1145/3622799

[27] Karine Heydemann, Jean-François Lalande, and Pascal Berthomé. 2019. Formally verified software countermeasures for control-flow integrity of smart card C code. *Computers & Security* 85 (2019).

[28] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight verification of separate compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. https://doi.org/10.1145/2837614.2837642

[29] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. 2018. On the Importance of Analysing Microarchitecture for Accurate Software Fault Models. In *2018 21st Euromicro Conference on Digital System Design (DSD)*. Prague, France. https://doi.org/10.1109/DSD.2018.00097

[30] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009). HAL:inria-00415861

[31] Xavier Leroy and Sandrine Blazy. 2008. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *Journal of Automated Reasoning* (2008). https://doi.org/10.1007/s10817-008-9099-0

[32] Thibault Martin, Nikolai Kosmatov, and Virgile Prevosto. 2022. Verifying Redundant-Check Based Countermeasures: A Case Study. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*.

[33] Sébastien Michelland, Christophe Deleuze, and Laure Gonnord. 2024. From Low-Level Fault Modeling (of a Pipeline Attack) to a Proven Hardening Scheme. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction, CC 2024, Edinburgh, United Kingdom, March 2-3, 2024*. https://doi.org/10.1145/3640537.3641570

[34] David Monniaux. 2024. Memory Simulations, Security and Optimization in a Verified Compiler. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2024)*. New York, NY, USA. https://doi.org/10.1145/3636501.3636952

[35] David Monniaux, Léo Gourdin, Sylvain Boulmé, and Olivier Lebeltel. 2023. Testing a Formally Verified Compiler. In *Tests and Proofs (TAP 2023) (Lecture Notes in Computer Science, Vol. 14066)*, Virgile Prevosto and Cristina Seceleanu (Eds.). Springer Nature Switzerland, Leicester, United Kingdom, 40–48. https://doi.org/10.1007/978-3-031-38828-6_3

[36] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. 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*. https://doi.org/10.1145/1542476.1542504

[37] Marco Patrignani and Deepak Garg. 2021. Robustly Safe Compilation, an Efficient Form of Secure Compilation. *ACM Trans. Program. Lang. Syst.* 43, 1, Article 1 (feb 2021). https://doi.org/10.1145/3436809

[38] Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. 2014. Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014*.

[39] Julien Proy, Karine Heydemann, Alexandre Berzati, and Albert Cohen. 2017. Compiler-Assisted Loop Hardening Against Fault Attacks. *ACM Trans. Archit. Code Optim.* 14, 4, Article 36 (dec 2017). https://doi.org/10.1145/3141234

[40] Pablo Rauzy and Sylvain Guilley. 2014. A Formal Proof of Countermeasures Against Fault Injection Attacks on CRT-RSA. *Journal of Cryptographic Engineering* 4, 3 (2014). https://doi.org/10.1007/s13389-013-0065-3

[41] G.A. Reis, J. Chang, Neil Vachharajani, Ram Rangan, and David August. 2005. SWIFT: software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*. https://doi.org/10.1109/CGO.2005.34

[42] Olivier Savry, Mustapha El-Majihi, and Thomas Hiscock. 2020. Confidaent: Control Flow protection with Instruction and Data Authenticated Encryption. In *23rd Euromicro Conference on Digital System Design, DSD 2020, Kranj, Slovenia, August 26-28, 2020*. https://doi.org/10.1109/DSD51259.2020.00048

[43] Sarwar Sayeed, Hector Marco-Gisbert, Ismael Ripoll, and Miriam Birch. 2019. Control-flow integrity: attacks and protections. *Applied Sciences* 9, 20 (2019).

[44] Zachary Tatlock and Sorin Lerner. 2010. Bringing Extensibility to Verified Compilers. *ACM SIGPLAN Notices* 45. https://doi.org/10.1145/1806596.1806611

[45] Nikolaus Theißing, Dominik Merli, Michael Smola, Frederic Stumpf, and Georg Sigl. 2013. Comprehensive analysis of software countermeasures against fault attacks. In *Proceedings of the Conference on Design, Automation and Test in Europe* (Grenoble, France) *(DATE '13)*. San Jose, CA, USA.

[46] Jérémy Thibault, Roberto Blanco, Dongjae Lee, Sven Argo, Arthur Azevedo de Amorim, Aïna Linn Georges, Catalin Hritcu, and Andrew Tolmach. 2024. SECOMP: Formally Secure Compilation of Compartmentalized C Programs. https://arxiv.org/abs/2401.16277

[47] Simon Tollec. 2024. *Formal Verification of Processor Microarchitecture to Analyze System Security against Fault Attacks*. Ph. D. Dissertation. Université Paris-Saclay.

[48] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. 2022. Exploration of Fault Effects on Formal RISC-V Microarchitecture Models. In *Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2022, Virtual Event / Italy, September 16, 2022*. https://doi.org/10.1109/FDTC57191.2022.00017

[49] Paolo Torrini and Sylvain Boulmé. 2022. A CompCert Backend with Symbolic Encryption. In *Sixth workshop on Principles of Secure Compilation (PriSC'22), part of the 49th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2022)*. Philadelphia, Pennsylvania, United States. https://hal.science/hal-03555551

[50] Aurelien Vasselle, Hugues Thiebeauld, Quentin Maouhoub, Adèle Morisset, and Sebastien Ermeneux. 2018. Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot. *IEEE Trans. Comput.* PP (07 2018). https://doi.org/10.1109/TC.2018.2860010

[51] Son Tuan Vu, Karine Heydemann, Arnaud de Grandmaison, and Albert Cohen. 2020. Secure delivery of program properties through optimizing compilation. In *CC '20: 29th International Conference on Compiler Construction, San Diego, CA, USA, February 22-23, 2020*. https://doi.org/10.1145/3377555.3377897

[52] Philip Wadler. 1992. Monads for functional programming. In *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992 (NATO ASI Series, Vol. 118)*. https://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf

[53] Ling Zhang, Yuting Wang, Jinhua Wu, Jérémie Koenig, and Zhong Shao. 2024. Fully Composable and Adequate Verified Compilation with Direct Refinements between Open Modules. *Proc. ACM Program. Lang.* 8, POPL (2024). https://doi.org/10.1145/3632914

[54] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. 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*. https://doi.org/10.1145/2103656.2103709