THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Laboratoire de conception et d'intégration des systèmes

Compilation pour la sécurité matérielle : au-delà de la sémantique

Compilation beyond semantics for hardware security

Présentée par :

Sébastien MICHELLAND

Direction de thèse :

Laure GONNORD PROFESSEURE DES UNIVERSITÉS, Grenoble INP – UGA	Directrice de thèse
Christophe DELEUZE MAÎTRE DE CONFÉRENCES, Grenoble INP – UGA	Co-encadrant de thèse

Rapporteurs :

Karine HEYDEMANN
SECURITY SENIOR EXPERT (HDR), Thalès CDI

Jens GUSTEDT
DIRECTEUR DE RECHERCHE, Centre Inria de l'Université de Lorraine

Thèse soutenue publiquement le **24 octobre 2025**, devant le jury composé de :

Isabelle PUAUT, PROFESSEURE DES UNIVERSITÉS, Université de Rennes	Présidente
Laure GONNORD, PROFESSEURE DES UNIVERSITÉS, Grenoble INP – UGA	Directrice de thèse
Karine HEYDEMANN, DOCTEURE EN SCIENCES HDR, Thalès DIS	Rapporteure
Jens GUSTEDT, DIRECTEUR DE RECHERCHE, Centre Inria de l'Université de Lorraine	Rapporteur
Marie-Laure POTET, PROFESSEURE DES UNIVERSITÉS EMÉRITE, Grenoble INP – UGA	Examinatrice
Albert COHEN, DIRECTEUR DE RECHERCHE, Google	Examinateur
Daniel GRUSS, UNIVERSITY PROFESSOR, Graz University of Technology	Examinateur

Invités :

Christophe DELEUZE
MAÎTRE DE CONFÉRENCES, Grenoble INP – UGA

Remerciements

Je tiens à remercier en premier lieu Laure, à qui je dois non seulement cette thèse mais aussi mon engouement pour la compilation et pour l'enseignement. La thèse est une expérience humaine autant que scientifique, et tu as été précieuse sur tous les plans.

Merci également à Christophe pour ton encadrement ancré dans la réalité qui m'a sauvé de quelques idées folles. Parfois le TikZ est superflu et l'ASCII art est bien suffisant.

Merci à Karine et Jens pour la relecture de ce document et votre enthousiasme extrêmement valorisant, ainsi qu'à Isabelle, Marie-Laure, Albert et Daniel pour votre participation à mon jury ; j'ai beaucoup apprécié vos questions minutieuses sur ces travaux dont je suis fier.

Les idées de cette thèse doivent beaucoup aux discussions informelles avec notamment François de Ferrière, Thomas Rubiano, Basile Pesin et Benjamin Gaudin ; et elles sont directement construites sur les travaux brillants et inspirants d'Ihab Alshaer et Son Tuan Vu.

Un grand merci à tous mes collègues : Marie et Arthur D. pour *nos* mèmes et les gâteaux, Clara pour ton enthousiasme, Valentin pour nos parties d'échecs, Julien pour cette JDD de tous les temps, Yann pour les aventures fonctionnelles, Daniel pour la musique… et bien d'autres.

Rien de tout cela n'aurait été possible sans les IATS du LCIS, notamment Carole, Caroline, Johanna, Patricia, Marie et Arthur ; toute notre recherche repose sur vos efforts quotidiens.

Je gage que ni Antonin Dudermel, ni Alban Reynaud, ni Antoine Domenech n'auront anticipé que les peluches qui animaient nos cours à l'ENS de Lyon animeraient aussi nos thèses. Merci à vous et aux supers collègues de promotion, Victor, Aymeric, Bastien, et tous les autres que je laisserai se reconnaître par manque de place. L'informatique est fun grâce à vous.

Des remerciements du fond du cœur aux étudiant·es de l'Esisar, entre autres les P2026 et P2027, pour les super moments passés en cours. Quand au TP de C débutant on voit du Vulkan et du JIT, on n'a pas une seconde pour s'embêter. Zoë, Julien, vous êtes géniales.

J'ai enfin eu l'honneur d'héberger sur mon tableau un projet d'ampleur de plus d'un an, par Arthur Baudet, Laure Gonnord, Stéphanie Chollet et Zoé Lagache, qui ont collectivement chamboulé le paysage de la programmation en établissant l'équation suivante :

$$\text{C} \overset{\text{A.B.}}{<} \text{Zig} \overset{\text{A.B.}}{<} \text{Perl} \overset{\text{L.G.}}{<} \text{Ocaml} \overset{\text{A.B.}}{<} \text{Vlang} \overset{\text{A.B.}}{<} \text{Javascript} \overset{\text{A.B.}}{<} \text{HTML} \overset{\text{S.C.}}{<} \text{CSS} \overset{\text{S.C.}}{<} \text{SQL} \overset{\text{Z.L.}}{<} \text{C}$$

Et c'est bien là, chère Lectrice, l'ultime succès de mes efforts. Au cas toutefois où ce thorème vous soit déjà familier, j'adjoins ci-après quelques autres résultats mineurs de mon travail.

Zen rides onwards to more science. There be compilers!

Résumé

Les systèmes informatiques sont construits par couches pour limiter leur complexité. Par exemple, le logiciel et le matériel sont globalement indépendants ; ils s'accordent sur une interface commune appelée « assembleur », et ensuite ignorent (*s'abstraient*) chacun des détails internes de l'autre. Cette construction décompose élégamment le système en éléments simples et est omniprésente en informatique ; il y a d'ailleurs une dizaine de ces couches d'abstraction entre le logiciel utilisé pour visualiser ce document et les lois fondamentales de la physique. Cependant, les interfaces successives entre couches n'isolent que les fonctionnalités, et pas par exemple les performances... ou la sécurité.

Les injections de faute sont des attaques matérielles qui induisent des comportements anormaux (*fautes*) en interférant volontairement avec des circuits (signaux, alimentation, champs électromagnétiques, etc.—tous les coups sont permis). Elles représentent une menace majeure pour les systèmes embarqués qui va au-delà de simples fautes aléatoires dues à des défauts de fabrication : ce sont des attaques ciblées qui peuvent exploiter même les faiblesses mineures dans un système. Et comme les abstractions ne se préoccupent pas de la sécurité, les contremesures ne peuvent pas être décomposées en couches ; elles doivent tenir compte de tout le système d'un seul coup, ce qui est sensiblement plus difficile.

De fait, il est déjà difficile de couvrir les couches logicielles, allant du code applicatif haut-niveau typiquement écrit en C au code assembleur. Il est connu que *compiler* des programmes peut détruire les parties logicielles des contremesures de sécurité. Par exemple, pour sécuriser on peut faire les calculs sensibles en double et vérifier qu'il n'y a pas eu d'erreur causée par une faute ; le compilateur, ignorant la menace de sécurité, supprimera les doublons (jugés inutiles) pour optimiser les performances du programme. Cette relation antagoniste fait de la compilation sécurisée un deuxième domaine de l'informatique où des générateurs de texte produisent des résultats fonctionnellement valides mais fondamentalement inadéquats.

Cette thèse analyse cette friction entre les contremesures de sécurité (principalement contre les injections de faute) et la compilation de code C. J'y démontre que les menaces envers le code de sécurité sont inhérentes à la descente en abstraction et ne peuvent pas être esquivées juste en désactivant les optimisations, avec une analyse détaillée dans le cas du compilateur LLVM. J'expose également de multiples subtilités sémantiques qui rendent des certifications formelles improbables pour les primitives de sécurité dans un compilateur de production, ce qui motive une approche expérimentale.

La production centrale de la thèse est Tracing LLVM, une surcouche libre légère pour LLVM qui enrichit l'interface entre le code et le compilateur pour faciliter l'implémentation de contremesures de sécurité. J'illustre ces extensions sur une palette d'exemples et montre qu'on peut les composer pour obtenir un contrôle fin sur le code généré tout en écartant la majorité des menaces provenant des optimisations et descentes en abstraction. Au centre de cet outil est un système de « tracé » permettant de connecter des éléments du programme source avec les programmes intermédiaires, ce qui aide les contremesures à naviguer la pile d'abstraction.

Ces contributions au développement de la compilation sécurisée ouvrent des portes pour faciliter la co-conception logicielle/matérielle des mécanismes de défense contre les attaques matérielles et consolider leur sécurité.

Abstract

Computer systems are built in layers to contain their complexity. For instance, software and hardware are mostly independent; they simply agree on a common interface called an “assembler” programming language, and then ignore (*abstract away*) each other’s internal details. This brilliantly breaks down the system in manageable pieces and is used extensively; in fact, the program visualizing this document is probably a dozen such abstraction layers away from the fundamental laws of physics. However, these successive layers only abstract away the system’s functionality, not other aspects like its performance... or its security.

Fault injections are hardware attacks that induce abnormal behaviors (*faults*) by interfering with circuits (through signals, power, electromagnetic fields, or otherwise—anything goes). They pose a major threat to embedded systems that’s not just random defect-induced faults but targeted, engineered attacks that can slip through even minor cracks in a system. And since abstractions don’t cover security, countermeasures can’t be broken down into layers and need to cover the entire system at once, which is significantly more challenging.

In fact, just covering the software layers, spanning from high-level application code typically in C to assembler code, is difficult. It is well understood that *compiling* programs can destroy the software components of security countermeasures. For instance, it makes sense to run sensitive computations twice to check for errors induced by faults, but a compiler will proudly delete the duplicates, which are redundant in its functional-only abstract semantics. This adversarial relationship makes compiling for security a second field of computer science that struggles over generating text that is functionally valid yet substantially inadequate.

This thesis analyzes this friction between security countermeasures against hardware attacks (mostly fault injections) and the compilation of C code. I show that the threats to security code are inherent to the descent in abstraction and can’t simply be dodged by disabling optimizations, with a detailed breakdown in the case of the widely-used production compiler LLVM. I also highlight many of the semantic subtleties that lie between security primitives and formally-provable security properties, motivating an experimental approach to the problem.

The central production of the thesis is Tracing LLVM, a lightweight open-source extension of LLVM which enriches the interface between program and compiler to facilitate the implementation of security countermeasures. I demonstrate these extensions on a variety of examples and show that they can compose to great effect, providing fine control over generated code while eliminating most threats from lowerings and optimizations. Key to this proposition is a new “tracing” system that assists in maintaining a connection between the source and intermediate programs, which helps countermeasures navigate the abstraction stack.

These contributions constitute a significant step towards security-aware compilation, which opens co-design opportunities and promising reliability improvements for defense mechanisms against these tricky hardware attacks.

Contents

Introduction 1

Benjamin Lee Whorf (1897–1941),
American linguist and chemical engineer

Image credit:
The Hartford Agent Magazine/Benjamin Lee Whorf Papers,
Manuscripts and Archives, Yale University Library (ID 10009314)

Although the core conclusions from this thesis all relate to programming languages, I have it on good authority that spoken languages have been around a bit longer than modern programming ones. Maybe I shouldn't have been surprised, then, to find elements of linguistic theory that share striking similarities with the rest of this document. Here's my modest attempt at establishing this relationship.

There is a concept in linguistics that language doesn't just describe our perception of reality, but also influences it; that our constant use of language for classifying and communicating the world warps our perceptions so they fall neatly into what our spoken language can describe. This principle is commonly associated with Benjamin Lee Whorf [Who97], popularized by quotes such as *"Language shapes the way we think"* (though he did not invent it).

This notion of has been the subject of intense debate. The idea is often split into two separate flavours [Cha94]; the strong *linguistic determinism* hypothesis that our thinking is determined by language, and the weak *linguistic relativity* hypothesis that language influences our perception and interpretation of the world. Nowadays, there appears to be a general agreement against the strong hypothesis, while the weak one remains debated.

One might wonder whether linguistic relativity applies to programming languages (after all, programming and linguistics often intersect, such as with the theories of grammar and parsing). Well there is, indeed, no shortage of testimony that varied programming paradigms lend themselves to equally varied formalizations of real-world problems, influencing developers' intuitive models and solutions. Jenna Zeigen addresses this question in a talk [Jen14] where she casts a mould of linguistic relativity around Paul Graham's "Blub Paradox" (from his talk-then-essay *"Beating the Averages"* [Gra01]).

One important distinction is that Whorf aimed to "calibrate" spoken languages such as English and Hopi with each other, allowing translations between their differing perceptions of the same reality. On the other hand, Zeigen's and Graham's discussion is underpinned by the claim that there is an order of power in programming languages, and that this order is abstraction; Graham writes, "if you have a choice of several languages, it is, all other things being equal, a mistake to program in anything but the most powerful one".

In this thesis I explore problems related to security, which, as it turns out, makes all other things very much not equal. Studying programming languages and their compilation under the lens of non-functional requirements forces us to capture the complexity of the entire language stack. So there is no single language to solve hardware security, let alone high-level.

In essence, my central claim is that low-level security concerns escape the modeling power of high-level programming languages, which only capture a chosen subset of functionality. Each lower abstraction level may attempt to cast the threat into a form that the next level can perceive accurately and counter. But on a basic level, the technical framework of countermeasures against hardware vulnerabilities is defined by the strong principle of linguistic determinism applied to computing abstractions, each being constrained in its perception of the threat by its limited ability to model the system.

1.1. Hardware security

Computer systems today are built in layers. We start with transistors on a chip and create logic gates and memories. From then on, we forget about the transistors, and directly use the gates and memories to create processors. We gradually go higher by building more abstract stages on the interfaces of lower stages. At some point, we start writing software in the processor's assembly language while ignoring the processor's implementation. Eventually, we forget the assembler too and write programs in "high-level" programming languages like C that we compile to assembler automatically. Each new layer is an *abstraction* of the previous, hiding tedious implementation details while providing more expressive tools for building complex systems.

This design has many advantages, chief among which is breaking down the complexity of computers over multiple fields of engineering. Hardware designers can change microarchitectures, and so long as they use the same assembly language, existing software will keep working, stabilizing the work of software engineers. This is why I can write the dot product function of Listing 1.1 in C and be confident it'll run on pretty much anything.

```c
long dot(int *x, int *y, size_t n) {
  long result = 0;
  for(size_t i = 0; i < n; i++)
    result += x[i] * y[i];
  return result;
} /* did you spot the bug? */
```

Listing 1.1: Integer dot product function in C

There are limits, of course; each stage is designed to abstract the *functionality* of lower stages, but can't guarantee other properties, called "non-functional", such as *performance* and *security*.

Looking at Listing 1.1 again, the C language specifies exactly what the multiplication operator computes, but not how long it takes to produce its result. The performance characteristics of the program depend greatly on the processor's implementation.

Thus, systems with non-functional requirements cannot rely on abstractions alone and must be aware of internal software and hardware details. This thesis is concerned with the particular case of hardware security vulnerabilities, which come in two flavours.

Fault injection attacks are physical attacks that deliberately cause *faults* in the system. A fault is any abnormal condition that leads to incorrect behavior. Traditionally the threat of faults was limited to manufacturing defects or extreme operating conditions (famously illustrated with cosmic rays flipping bits in avionics), but the development of *fault injection* techniques [She+21] made them an explicit attack vector.

On Listing 1.1, a typical fault attack might lead to an assembler instruction being skipped. Depending on the precise assembler code, this could skip a product, an accumulation step, use a wrong operand somewhere, end the loop early, extend it, use data or run code from other functions, and many other effects. This variety is a key reason why fault attacks constitute a significant threat. Occasionally a particularly impactful fault attack reaches the mainstream public, like Rowhammer [Kim+14] which had the rare ability to target servers remotely.

Side-channel attacks are attacks that exploit sensitive information extracted from a system without compromising its execution. A side-channel is any observable physical or functional property of the system that is correlated with its internal operations, like timing, power consumption, or all sorts of internal processor state that can be observed indirectly.

Again with Listing 1.1, the multiplication instruction on the targeted processor might be faster for small inputs, allowing attackers to learn information about the values of x and y by measuring execution times. If either of the arrays contains secret data, this would constitute a severe breach of confidentiality. Side-channel attacks also frequently go mainstream, like the haunting legacy of Spectre's [Koc+19] speculative execution leaks, or more recently when Apple processors' aggressive optimizations led to the GoFetch vulnerability [Che+24].

This combination of factors—extreme non-functionality and endless physical variations—makes hardware vulnerabilities particularly difficult to deal with. No one I've met entertains the notion of a complete defense against known attacks, let alone future ones. Obviously this expectation of incompleteness doesn't preclude the study and deployment of countermeasures, but it makes any *guarantees* about behaviors or coverage that much more valuable.

Figure 1.2 shows the process for designing a countermeasure against a fault attack, which will be our main use case. Just like how functional abstractions are spread over different fields of specialty, modeling vulnerabilities and designing countermeasures involves experts at many levels of the system.

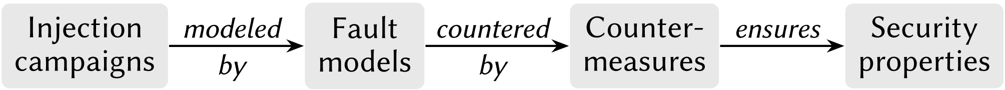


Figure 1.2: Usual process for countering a fault injection attack

1. First, hardware experts run *injection campaigns* with appropriate equipment and collect faulted execution traces.
2. Then they formulate a *fault model* that summarizes the most common effects and describes them at a higher level of abstraction.
3. From there, hardware or software engineers devise mechanisms for nullifying, detecting, or recovering from the attack, which can be in hardware, in software, or both.
4. Deploying the countermeasure then guarantees that a specific security property holds even when attacked, typically some sort of integrity or confidentiality requirement.

The main challenge at the moment remains providing countermeasures that are effective against attacks without large performance or resource overheads. The rapid improvement of attacks means that protections are often not yet secure enough [Yuc+16]—sometimes fundamentally insufficient [Ran23]—all while the industry of embedded systems puts a premium on cost-effectiveness. My work tackles two aspects of the reliability objective: countering accurate fault models, and eliminating friction around the compilation of countermeasures.

1.2. Modern C compilers, LLVM IR, and RISC-V

A central part of this thesis is understanding and improving the interactions between compilers and security countermeasures. A compiler is a tool that translates programs from a high-level programming language to another, lower-level programming language. In the context of embedded systems, engineers will most commonly write their software in C then compile it to assembler so it can be run on real hardware.

C is not a particularly high-level programming language; it may well be the lowest among those used to write large-scale applications. Developers can often predict what assembler code a compiler is likely to generate, leading to a sense that the compiler produces "the" assembler code for any given C program[1]. However, modern compilers only guarantee that they'll output *some* assembler program whose functionality matches the C source, with no guarantees on its specific implementation, performance or security.

This rewriting process takes the program through multiple intermediate languages which were introduced over the years either for optimization or software engineering purposes. Optimizing compilers are generally centered around one internal language called the *Intermediate Representation* or IR (sometimes multiple of them). This naturally leads into a separation of the compiler into three different stages, shown in Figure 1.3:

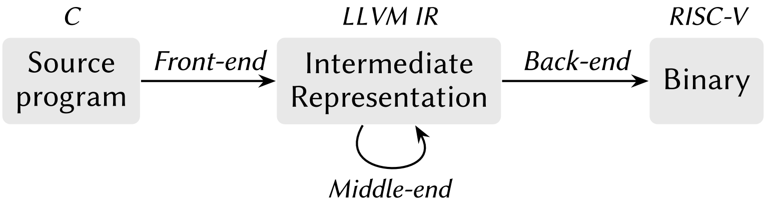


Figure 1.3: Bird's-eye view of compilation stages

[1] For a notable example, consider https://www.youtube.com/watch?v=MShbP3OpASA&t=21m58s.

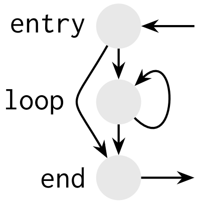
LLVM IR code (simplified)

```
define i32 @dot(ptr %x, ptr %y, i32 %n) {
entry:
  %empty = icmp eq i32 %n, 0
  br i1 %empty, label %end, label %loop

loop:
  %i = phi i32 [%new_i, %loop], [0, %entry]
  %result = phi i32 [%new_result, %loop],
                    [0, %entry]
  %xiptr = getelementptr i32, ptr %x, i32 %i
  %xi = load i32, ptr %xiptr, align 4
  %yiptr = getelementptr i32, ptr %y, i32 %i
  %yi = load i32, ptr %yiptr, align 4
  %xiyi = mul i32 %xi, %yi
  %new_result = add i32 %xiyi, %result
  %new_i = add i32 %i, 1
  %exit = icmp eq i32 %new_i, %n
  br i1 %exit, label %end, label %loop

end:
  %res = phi i32 [0, %entry],
                 [%new_result, %loop]
  ret i32 %res
}
```

Control-flow graph



RISC-V assembler code

```
dot:
  # a0 is x, a1 is y, a2 is n
  li      a3, 0
  beqz    a2, .end
.loop:
  lw      a4, 0(a0)   # a4 = *x
  lw      a5, 0(a1)   # a5 = *y
  mul     a4, a5, a4
  add     a3, a3, a4
  addi    a2, a2, -1  # n--
  addi    a1, a1, 4   # y++
  addi    a0, a0, 4   # x++
  bnez    a2, .loop
.end:
  mv      a0, a3
  ret     # returns a0
```

Figure 1.4: Dot product: LLVM IR code, control-flow graph, and RISC-V code

- the *front-end*, which rewrites the source into the intermediate representation;
- the *middle-end*, which optimizes the intermediate representation;
- and the *back-end*, which finally rewrites the program into target code.

The steps that rewrite the program into a lower-level language or representation are called *lowerings*. We'll encounter a few more when delving into LLVM's back-end (Figure 5.2).

In our case, the source code is C and the target code is RISC-V assembler, but the design is intended to allow any supported source language to be compiled to any supported target language, by exploiting the same shared middle-end logic, which is the core of the compiler. (Although, when it comes to security, all components will be about equally as relevant.)

LLVM's intermediate representation is called *LLVM IR*. Figure 1.4 (left) shows the (simplified) LLVM IR code for the dot product function after the front-end and middle-end have run. Unlike C functions, LLVM IR functions do not have nested control-flow structures; they consist of a flat list of *basic blocks* (here entry, loop and end). Each basic block is made of a sequence of straight-line instructions, ending either in a jump to another block, a conditional branch to two other blocks, or a function return. This induces a *Control-Flow Graph* (CFG) whose nodes are the basic blocks and edges are the possible jumps and branches between them, also shown in Figure 1.4 (top right).

A key feature of LLVM IR is that it is in *Static Single Assignment* (SSA) form [RT22]. This means that every variable in the function is assigned only once; intuitively speaking, a second assignment would be considered a new variable. If there are conditional assignments, multiple "versions" of the same variable can reach a given control point. For instance, on Figure 1.4 when entering the loop body on line 7, `i` might be `0` (if we came from the entry block) or from the incremented value `%new_i` generated by the previous iteration (if we came from the loop body). This kind of conflict is reconciled using a special `phi` operator which redefines the variable based on the control path at runtime. Essentially, the SSA form determines what versions of variables are visible to each statement and encodes it explicitly. We won't see much of `phi` in this thesis, but the splitting aspect in worth keeping in mind.

Other than its specific format using SSA in a control-flow graph, LLVM IR mostly resembles three-address code annotated with types (`i32` being 32-bit integers and `i1` being booleans). The LLVM IR code we'll see will focus on basic instructions such as memory loads (`load`) or arithmetic (`add`, `xor`, etc.). For more detail, please see the LLVM language reference[2].

Figure 1.4 also shows the RISC-V assembler code produced by the compiler's back-end (bottom right). RISC-V is an open-standard Instruction Set Architecture[3]; for our purposes, it mostly defines an extensible assembler language. It is a fairly standard RISC architecture; for the purpose of reading this thesis, the following aspects of 32-bit RISC-V are relevant:

- The main registers are `a0`–`a7` (*arguments*, caller-saved/scratch), `t0`–`t6` (*temporaries*, caller-saved/scratch) and `s0`–`s11` (*saved*, callee-saved/permanent).
- Special registers mostly include the return address `ra` and stack pointer `sp`.
- Output operands, if any, come first.
- The notation "`off(reg)`" refers to the memory address or operand at `reg+off`.
- Arguments are passed in `a0`, `a1`, etc.; the return value is `a0`.

The following instructions will make an appearance:

- Moving values: `mv` (MoVe), `li` (Load Immediate);
- Memory accesses: `lw` (Load Word), `sw` (Store Word), `lbu` (Load Byte Unsigned);
- Unconditional control-flow: `j[r]` (Jump [Register]), `call`, `ret`;
- Conditional control flow: `beq` (Branch if EQual), `beqz` (Branch if EQual to Zero), and variations with other two-letter comparison codes;
- The usual arithmetic: `add`, `mul`, `xor`, etc.

The assembler code for the dot product function should hopefully be straightforward. Note however how the logic differs from the IR code; `i` has been eliminated, instead the pointers `x` and `y` (`a0` and `a1`) are incremented at each iteration, and the counting is performed by decrementing `n` (`a2`) directly.

This example already illustrates some misconceptions about compilation. A user invoking LLVM would provide the C code as input and obtain the RISC-V assembler as output, and might assume that `result` was simply mapped to register `a3`, setting up the expectation that a single C variable would reasonably be assigned to a single register. However, the IR code

[2] https://llvm.org/docs/LangRef.html

[3] https://riscv.org/. RISC stands for "Reduced Instruction Set Computer".

hidden inside the compiler reveals that this is not the case, as both assignments to `result` are considered different variables. The fact that both end in `a3` is not random, but it's also not guaranteed (and we'll see it in action later).

This kind of discrepancy severely limits programmers' ability to incorporate security countermeasures in programs or ensure that they get compiled in a satisfying manner. It's emblematic of the tension between functionality-oriented compiler design, and the real but unmodeled threats that countermeasures attempt to defeat. Fortunately, there are a few things we can do about that.

1.3. Contribution and outline

This thesis addresses the general problems of designing countermeasure for low-level attack models, and improving the reliability of secure code generation with a compiler.

Chapter 2 surveys the methods for writing security countermeasures in software, with a focus on compiler integration. Its main point is that approximations in fault models combined with semantic subtleties in countermeasures make it hard to obtain strong guarantees, and that compiler "tricks" are insufficient.

Chapter 3 lays out my vision for improving the reliability of fault injection countermeasures, from design to implementation and validation (on the software side). This consists of two axes: (1) exploiting the lowest-level attack models possible, and (2) improving the reliability of secure compilation. It also contains examples demonstrating typical security violations that arise from compilation.

Chapter 4 addresses the first axis by demonstrating the use of semantic modeling to capture low-level effects in a fault injection countermeasure. In this chapter, I develop, implement, and validate a software/hardware co-designed countermeasure against a tricky attack model called *fetch skips*. The countermeasure builds upon a semantic model and I formally prove its security in Appendix A.

Chapter 5 and onwards address the second axis by describing different aspects of Tracing LLVM, my LLVM mod that extends languages and compiler to facilitate the implementation of security countermeasures. Chapter 5 describes Tracing LLVM from the perspective of a countermeasure developer, illustrates how it solves the examples from Chapter 3, and showcases a combination of four countermeasures on a longer PIN verification example.

Chapter 6 takes a short dive into the research questions that arise from the implementation of Tracing LLVM, mainly relating to language design and maintainability. These choices are key to extracting guarantees out of LLVM extensions without requiring complete knowledge of its multi-million-line codebase.

Finally, Chapter 7 takes a critical look at the interface between the compiler and the security assumptions made by countermeasures to prepare for a proper formalization in future work. I discuss the goals that such an interface should achieve, then illustrate the delicate rift between experimentally-predictable behaviors and formal guarantees with a series of theoretical but legal security-breaking program transformations.

Chapter 8, as one would expect, wraps up by summarizing the contributions and lays out future work for this project.

What to read?

- If you want to understand Tracing LLVM's design and how to use it, read Chapters 3 and 5.
- For the research questions surrounding secure compilation, read Chapters 3 and 5 to 7.
- For countermeasures, read Chapter 4 (free-standing) or Chapters 3 and 5 (Tracing LLVM).

Chapter 2 provides valuable context but is not a hard dependency of other chapters; any relevant terms that it defines for later are listed in the index at the end of the document. The index collects definitions for the most common terms as well as thematic references for recurring concepts.

1.4. Dissemination

The proven countermeasure from Chapter 4 was the subject of a publication and presentation at the Compiler Construction conference of 2024 (CC'24) [MDG24]:

- Sébastien Michelland, Christophe Deleuze, and Laure Gonnord. "From low-level fault modeling (of a pipeline attack) to a proven hardening scheme". In: *Compiler Construction (CC'24)*. Edinburgh (Scotland), United Kingdom, Mar. 2024.
  DOI: `10.1145/3640537.3641570`. URL: `https://hal.science/hal-04438994`.

At the time of submitting the final version of this document, the Tracing LLVM bases described in Chapters 5–7 are cycling the submission pipeline; they have however been showcased at the international Workshop on the Principles of Secure Compilation of 2026 [MDG26].

- Sébastien Michelland, Christophe Deleuze, and Laure Gonnord. "Compiling Countermeasures Against Fault Attacks with Tracing LLVM". In: *Workshop on Principles of Secure Compilation (PriSC'26)*. Rennes, France. Jan. 2026.
  URL: `https://popl26.sigplan.org/details/prisc-2026-papers/8`.

This thesis is related to the PEPR project ARSENE[4] (« *Architectures Sécurisées pour le Numérique Embarqué* », or "Secure Architectures for Embedded Computing") which aims to design sovereign secure platforms for embedded systems with a RISC-V base. The ARSENE project is funded by the "France 2030" government investment plan managed by the French National Research Agency, under the reference `ANR-22-PECY-0004`.

The state-of-the-art from Chapter 2 is based on a deliverable from the ARSENE project (reusing only the sections that I personally wrote).

I gave a number of talks on the subject of secure compilation:

- At the 2024 Journée thématique sur les Attaques par Injection de Fautes (JAIF) workshop (Rennes, France);
- At the 2025 PHISIC workshop (Gardanne, France);
- At French national community meetings: Journées GLsec (Paris, Nov. 2022), Journées du GDR Sécurité (Caen, June 2025);

[4] `https://www.pepr-cyber-arsene.fr/`

- At a number of local seminars: CASH team (Lyon, Feb. 2024), ANSSI (March 2024), the Verimag lab (Grenoble, April 2024), CyberAlps seminar (Grenoble, May 2024), SemSécuÉlec seminar (Rennes, Sept. 2024), ÉPICURE team (Rennes, Sept. 2024), PACAP team (Rennes, April 2024) and finally Verimag again (Grenoble, June 2025).

In parallel to the work described in this document, I also completed a previous research project relating to the modeling of static analyzers in monadic semantics. This work was published and presented at the 2024 Internal Conference on Functional Programming (ICFP'24) [MZG24]:

- Sébastien Michelland, Yannick Zakowski, and Laure Gonnord. "Abstract Interpreters: A Monadic Approach to Modular Verification". In: *Proceedings of the ACM on Programming Languages* 8.ICFP (Aug. 2024), pp. 1–28.
  doi: `10.1145/3674646`. url: `https://hal.science/hal-04628727`.

This avenue of research is related to hardware security through its potential for formal verification, which other brilliant people have started to explore [Pes+25].

2 Incompleteness in the security stack

Software protections against hardware vulnerabilities face a variety of challenges. This chapter covers the state-of-the-art of these software components, with an emphasis on countermeasures against fault injection attacks. The particular angle I want to highlight is how the challenges in defeating attacks manifest in the incompleteness of protections. This highlights why any guarantees are really valuable and motivates the focus of this thesis in improving reliability.

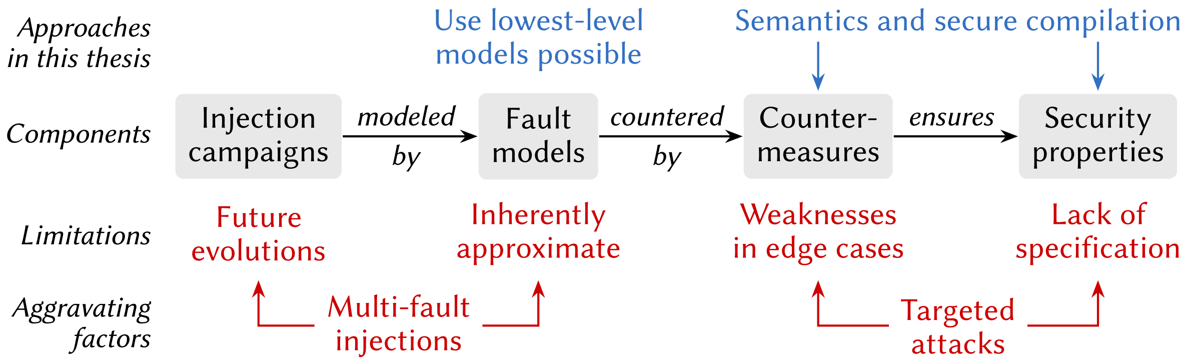


Figure 2.1: Limitations in the process for countering fault attacks

Figure 2.1 extends Figure 1.2 with the main hurdles in designing countermeasures against fault injection attacks.

- As before, injection campaigns are first run to sample execution traces. This analysis needs continuous updates as the set of faults that can be reliably injected by attackers keeps evolving.
- The fault model is then formulated; by construction it ignores rarer outcomes, and lifts the fault's description to a higher level of abstraction. Both effects lose real outcomes (and may introduce unrealistic ones), so the attack being protected against isn't *quite* the real attack.
- The step of building countermeasures doesn't have inherent weaknesses, but it's difficult to design protections that are complete even in edge cases. Broadly speaking due to community fragmentation countermeasures either counter accurate low-level models, or are formalized and proven, but rarely both.
- And a minor point, there is no unified framework for expressing security properties. They are often under-specified, with functional correctness as the implicit default.

These hardships are compounded by aggravating factors such as multi-fault injections (which add to the variety of attack methods and seriously complicate modeling) and targeted attacks (which may turn any single weakness into a serious vulnerability).

With this in mind, let's explore how literature deals with software contributions to this field. Starting with the requirements, Section 2.1 reviews usual attack models and Section 2.2 lists and attempts to categorize common targeted security properties. Section 2.3 covers the design of countermeasures, including their placement in the compiler and an overview of proof methods. Section 2.4 explores in more depth the problem of preserving security during compilation, which is tightly related to matters of *traceability* addressed in Section 2.5. I end with an overview of co-designed countermeasures in Section 2.6.

2.1. Quick overview of fault and side-channel models

Fault injection techniques are quite varied: some methods require contact on pins, such as clock/voltage glitches or body bias; some don't, like laser/X-ray pulses or electromagnetic interference. In this thesis, I only consider *transient* faults, which cause incorrect behaviors but do not actually damage the target (as opposed to *permanent* faults). The interested reader is referred to Shuvo et al. [Shu+23] for a more thorough listing.

Side-channels are similarly diverse; attackers can target programs' execution time [Koc96; HWH13; RG24], power consumption over time [Man03], or even electromagnetic emissions [QS01; SLS19]. All sorts of micro-architectural state affected by speculation is also a target, commonly categorized under the umbrella of Spectre [Koc+19] (and, to a lesser extent, Meltdown [Lip+20]). Leaks can further be analyzed over multiple executions by statistical methods for improved precision; such techniques include Correlation Power Analysis [BCO04] or Differential Power Analysis [KJJ99; Wan+17].

In both cases the complexity of the attacks calls for simplified models to smooth out the physical details and lift possible outcomes to higher abstraction levels. A single attack can be approximated by many models; generally speaking a higher-level model is easier to study and counter but a lower-level model better describes the possible outcomes of the attack. Note that approximations go both ways: they remove real behaviors (leading to worse protections) but also add fictional ones (leading to more expensive protections). Chapter 4 of this thesis will advocate for using the lowest-level models we can handle to avoid irrecoverable approximations.

2.1.1. Fault models

The following are common fault models, roughly sorted by decreasing abstraction level.

At source level:

- **"Statement skip"**: skip a C statement (also confusingly called "instruction");
- **Branch inversion**: a conditional statement's targets are swapped [Pot+14];
- **Corrupted control flow**: arbitrary jumps in a function [HLB19]; calls to invalid targets…

At assembler/ISA level:

- **Instruction skip/re-execution**: an instruction is either skipped or executed twice [VHM03];
- **Branch inversion**: targets of a conditional branch instruction are swapped;
- **Bus corruption**: data read or written corrupted en route on the CPU/memory bus [DSL17].
- **Wrong CFG edges**: a branch instruction jumps to the start of an arbitrary block in the function CFG, even if there was no CFG edge there [HLB19];
- **PC corruption**: bit flip or other types of corruption of the PC register;
- **Architectural SEU** (Single Event Upset): bit flip in an architectural component, like a register [Bar+14b], memory location, or instruction opcode;

At lower levels:

- **Forwarding error**: pipeline forwarding triggers when it shouldn't or doesn't when it should [Lau20];
- **Fetch skip** and **skip-and-repeat**: skip or repeat a CPU fetch's worth of code data (e.g. 4 bytes), leading to a single or multiple instructions decoding incorrectly [Als+21];
- **Low-level SEU** (Single Event Upset): bit flip in RTL latches [Tol+22];
- **Partial** or **delayed update**: an RTL register has only part of its bits updated, or it updates a cycle later than it should [Als+24];
- **Transistor failure**: a transistor assumes a semi-permanent state [Anc+17].

For each of these models, one can also consider *multi-fault* variations where more than one fault may occur. Generally speaking, multiple simultaneous faults greatly increase the burden of modeling while multiple faults spread over time mostly complicates countermeasures.

The term "soft error" was also frequently used in early works to describe any potential mis-execution of a program, but the interpretation of the term tends to vary from one publication to the next. It often ends up meaning a Single Event Upset (SEU, bit flip) but the abstraction level at which the SEU is considered still isn't always clear.

The chosen abstraction level for models is a clear accuracy-versus-simplicity trade-off; low-level descriptions are more true to practical attacks, but high-level approximations make it practical (in many cases *possible*) to reason about and protect against them. As it turns out, ISA-level models aren't always precise enough; faulted behaviors often depend on micro-architectural features and can only be described accurately by including hardware details [Lau+18]. Pipeline analysis by Yuce et al. [Yuc+16] further shows that targeted fault attacks can and do defeat many ISA-level countermeasures by exploiting unmodeled low-level effects. In other words, finer, more accurate countermeasures are still needed.

2.1.2. Side-channel models

While not a focus of this thesis, side-channel models serve the same purpose as fault models. Like faults, leaks of secret information can be difficult to analyze as they can be hidden by complex correlations. [Mar18] explains how countermeasures and other signal noises designed to defend against power analysis can be circumvented using different techniques including *multidimensional probabilistic representation* [CRR03; Arc+06].

Simple examples would be the *Hamming Weight* [Bel+13] and *Hamming Distance* models for power analysis. The Hamming weight model states that an attacker can observe the number of 1-valued bits in a piece of data. This is based on the fact that 1 bits in RAM are charged capacitors that need to be refreshed often; as a result, the power consumed to refresh the RAM is proportional to the number of 1 bits. Similarly, the Hamming distance model states that an attacker can determine the number of bits that flipped values between two observations of a single piece of data. This is relevant for storage elements that consume most of their energy not over time but when their stored value changes. In both cases, the model formalizes a *statistical correlation* between power consumption and the program's data by accounting for implementation details of the storage.

Models can also functionally capture non-functional properties like timing. For instance, RSA algorithms spend most of their time in modular exponentiation, and naive implementations perform varying numbers of modular multiplications depending on the value of the secret keys. This leads to a timing attack that observes the number of multiplications and thus part of the secret key [Koc96], which can be studied functionally.

However, at low levels of abstraction, non-functional elements tend to dominate. For example, timings can leak through instructions' interrupt latency [WMP21], data-dependent power policy (like Hertzbleed [Wan+22] monitoring Dynamic Voltage and Frequency Scaling), or just through speculative decisions not reflective of the program's code [Che+24].

So faults and side-channel models indicate how attackers may disturb or observe the system. The other piece of the puzzle for countermeasures is what needs protecting in the application.

2.2. Broad categories of security properties

Security properties formalize high-level security objectives such as *confidentiality*, *integrity*, or *availability* by specifying them within a given language or execution environment. There is no unified framework for this formalization, which leads to subtle variation between works; generally more so for properties related to fault tolerance compared to side-channels.

I want to draw attention to the fact that not all properties can be expressed at the same levels of abstraction: some are inherently low-level; some are high-level but may not translate well to low-level programs; a few are somewhat universal. This variety doesn't help with deriving a unified formalism.

Functional correctness. The most common security property is *functional correctness*[1]: the requirement that the program behaves as per its original semantics even when attacked. It's the natural target for fault tolerance (although hard to reach!), and is implicit in many early works like SWIFT [Rei+05]. Side-channels never break it as they don't modify executions.

Partial functional correctness. Most countermeasures detect faults *after* the program runs into an incorrect state. When no recovery is possible countermeasures allow erroneous outcomes, such as explicit termination, countermeasure-specific signals, or simply crashes. This idea of *partial correctness* generally comes with guarantees that the program stops "quickly" after an attack (with a suitable bound on the delay). This can satisfy security

[1] Compiler people call this "safety", but here *safety* is security for non-targeted/random attacks.

requirements when only chosen parts in a program are sensitive, such as the password check before the entry to a privileged section. For instance, NEMESIS [DSL17] detects attacks and stops the program when it's unable to recover.

Countermeasures against fault injections have a strong tendency to *default to functional correctness* as the only security property. I suspect this happens because of the field's background in *fault tolerance* (i.e. safety against faults), where there is no attacker and incorrect behavior is the only threat. Although not as obvious, fault injections can be used to threaten availability or confidentiality all the same. In any case, for fault-related works that do not explicitly provide a security property, some form of (partial) functional correctness should be assumed.

Control flow integrity (CFI). Control flow integrity is a family of properties about the integrity of execution paths taken by programs at runtime. They can appear at multiple levels:

- *Within a function's CFG.* Most commonly, control flow integrity refers to control following only valid edges of a function's control flow graph. Theißing et al. [The+13] list a number of countermeasures that achieve variations of this property with different types of checks.
- *Within a sequential block.* Instruction skips or arbitrary jumps might impact the execution of a sequential block (SSA basic block, C compound statement, or otherwise). Control flow integrity at this level is the property that all statements execute exactly in the source order.
- *For a whole program.* Control flow integrity can also extend to the inter-procedural level, requiring control to follow edges of the call graph much in the same way as a function's CFG edges [De 19].

Specialized forms of CFI. Weaker types of CFI may also be relevant:

- *Computation order*: the requirement for sensitive pure computation to occur in source order, such as refreshing the mask on a secret value. (See general literature on the analysis of masking, e.g. Prouff and Rivain [PR13]; the algorithms are very sensitive in general.)
- *Interleaving*: the CFI sub-scheme *Step Counter Incrementation* [HLB19] performs regular checks of a CFI witness value (the Step Counter) within sequential blocks. To be effective, the scheme requires the checks not be reordered relative to surrounding source code.

Data erasure. The property that a given piece of data is unavailable in program memory at certain control points. This is tricky to do because almost every abstraction level redefines data storage in a way that allows previous values to remain in internal state that's functionally invisible, which requires control of low-level systems such as CPU registers. This problem occurs independently in many systems, such as in redundant file-system storage [ORK18].

Non-interference. A common security property[2], *non-interference* formalizes data confidentiality as the absence of dependence from a sensitive input to an observable output. The sensitive input may be functional, for instance when studying the influence of secrets on the statistical distributions of observable *probes* in a program [Bar+16]. It can also be non-functional, like micro-architectural state, which can often be observed indirectly and may require system-level support to eliminate [Bar+14a].

[2] More specifically, a hyperproperty, since it is defined by quantifying over multiple executions.

The constant-time property. The constant-time property is a ubiquitous non-interference property for timing side-channel resistance. It is usually defined as the absence of secret-dependent branches (observable by timing) or memory accesses (observable via cache). This characterization is surprisingly very functional as it can be studied at all abstraction levels between C and assembler, but has limits in the context of modern Spectre-style attacks because they reveal many more micro-architectural details [Cau+20]. When discussed at low levels "constant-time" may also exclude some secret-dependent computations on processors with e.g. variable-delay multiplication or division [Gau+23].

2.3. Techniques and design of hardening compilation

After starting mostly as source transformations in the early days of fault tolerance, countermeasures now consistently involve automatic transformation in compilers. Manual source protections do exist, most notably constant-time cryptographic implementations, but this practice doesn't scale to entire systems, other attack models, or non-functional security properties. Involving toolchains allows automated protection tuned to each attack model or target device, and gives access to internal program representations for transformation.

2.3.1. Hardening at every level in the toolchain

Depending on the type of countermeasure (and thus attack model and targeted security property), any stage of the compilation chain might be involved in hardening, as shown by Figure 2.2. However, I'd like to underline that no single level provides all desirable properties for countermeasures; usually either fine control of assembly or source annotations is missing. Works discussed in this section are summarized in Table 2.3, sorted by publication year.

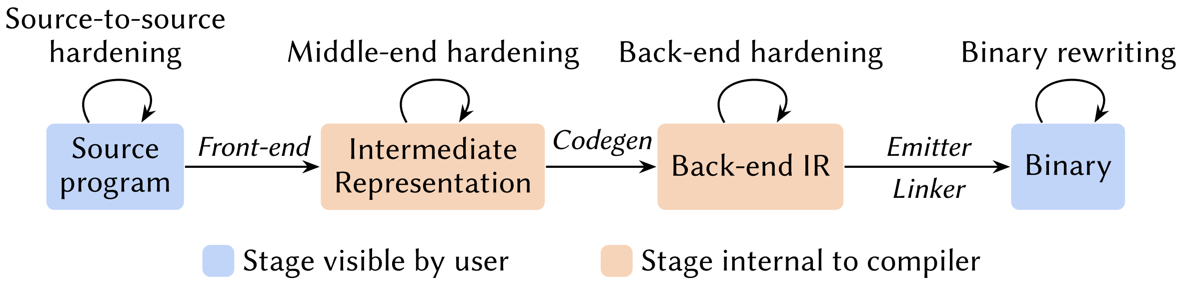


Figure 2.2: Common compilation stages for hardening

Source-to-source hardening. Countermeasures can be inserted by rewriting the source program (usually C) before compiling it.

Rebaudengo et al. [Reb+01] show an early example, targeted against transient memory errors, but claimed applicable to a wider range of transient faults. Their translation tool ThOR features standard transformations like duplicating variables, statements and function arguments; only returning from functions by address so return values can be duplicated; and control-flow integrity in the form of signature checks and duplicated conditions.

Cite	Scheme	Toolchain	Hardening stage	Attack model
[Reb+01]	ThOR	*N/A*	Source-to-source	"everything" (tests SEU†)
[VHM03]	ACFC	gcc	Preprocessor	Ins. skip/re-exec (multi-fault)
[Rei+05]	SWIFT	OpenIMPACT	Assembler	SEU†
[CRA06]	SWIFT-R	gcc 3.4.1 (PowerPC)	Early back-end	SEU†
[CRA06]	TRUMP	gcc 3.4.1 (PowerPC)	Early back-end	SEU†
[CRA06]	MASK	gcc 3.4.1 (PowerPC)	Early back-end	SEU†
[Lid+12]	ROSE::FTTransform	*N/A*	Source-to-source	"everything" (tests SEU†)
[Bay+13]	Sleuth	LLVM/Klee	Back-end	Leak info on ins. outputs
[BCR16]	*unnamed*	LLVM 3.6 (ARM)	Back-end	Instruction skip
[DS16]	nZDC	LLVM 3.7 (ARMv8-a)	Late back-end	SEU†, "soft errors"
[DSL17]	NEMESIS	LLVM 3.7 (ARMv8-a)	Late back-end	SEU†, wrong load/store
[Pro+17]	*unnamed*	LLVM 4.0 (ARM)	Back-end (SSA IR)	Ins. skip, reg. corruption
[Boh+18]	COAST	LLVM (MSP430)	Late middle-end	SRAM bit flip
[Van+18]	RACFED	*unknown*	Back-end CFG	Bit flip in PC
[De 19]	SecSwift	LLVM	Mostly middle-end	Multiple control/data attacks
[HLB19]	*unnamed*	*N/A*	Source-to-source	Wrong/random jumps
[AA19]	Smokestack	LLVM 3.9	Middle-end (+ libs)	Data-Oriented Programming
[Kia+21]	*unnamed*	LLVM (lifting)	Binary rewriting	SEU†, instruction skip
[WMP21]	*unnamed*	(Secure)LLVM 13.0	Back-end	Leak interrupt delay
[Gei+23]	CompaSeC	COMPAS (LLVM)	Back-end	Multi instruction skip
[Pes+25]	*unnamed*	Chamois CompCert	Middle-end (RTL)	Branch inversion transition

†SEU: Single Event Upset (single bit flip in registers, memory, micro-architectural components...)

Table 2.3: Sample of hardening schemes implemented in or around a compiler

Lidman et al. [Lid+12] describe a recovery scheme for large-scale computing systems where they expect fault tolerance to allow undervolting machines for efficiency. Their countermeasure ROSE::FTTransform (implemented in the source-to-source compiler framework ROSE [QL11]) runs computations up to N times and selects results based on configurable policies.

More recently, Heydemann, Lalande, and Berthomé [HLB19] provide a verified CFI countermeasure against intra-procedural jumps or wrong function calls in C code. They use signature-based CFI with step counter incrementation to track the progress of each block, which can detect any jumps of at least two statements regardless of block size. Signatures are chosen globally (each function having a unique signature interval) so the scheme can also detect inter-procedural control flow faults.

The main benefit of source-to-source hardening is of course portability, while the crucial drawback is that the compiler can drastically alter security measures while compiling. This approach provides next to no control over assembly code and often requires disabling optimizations, which is slow and increases attack surface due to the larger code size.

Compiler middle-end. Being the quintessential transformation pipeline in optimizing compilers, the middle-end is a natural choice for hardening.

Bohman et al. [Boh+18] evaluate an LLVM IR implementation of the Var3 scheme [Chi+12], called COAST, against neutron irradiation. Their countermeasure can duplicate or triplicate registers, computations and memory operations (but not control flow, although the implementation also independently supports CFCSS [OSM02]). Functional redundancy is typical for middle-end countermeasures, which don't have access to target-specific information. Despite the high cost, COAST causes a 7x mean-work-to-failure increase in the radiation setup.

SecSwift [De 19] implements slightly less usual transformations; it rewrites functions' prototypes to duplicate parameters and return values (which effectively modifies the ABI) and supports a dual intra-/inter-functional CFI scheme with signatures. The countermeasure is protected against back-end optimizations with intrinsics (I'll come back to that) and results in close to 100% coverage on random injection experiments.

Middle-end hardening can run after *most* optimizations while benefiting from typically superior documentation and language formalism than back-end representations. However, it can't easily relate to either source code or architectural details and still leaves all assembly code control to back-end algorithms.

Compiler back-end and beyond. Countermeasures against low-level attack models are commonly implemented near the back-end, close to hardware.

Winderix, Mühlberg, and Piessens [WMP21] close an interrupt-based side-channel that leaks the execution times of instructions on simple microcontrollers. The countermeasure equalizes the traces of all possible control flow paths in the program with no-ops and silent copies of functions. This pass couldn't be performed before the back-end because earlier representations don't fix specific instruction traces; it can also hardly be performed on binaries directly due to the difficulty of inserting large amounts of new code in linked programs.

NEMESIS [DSL17] is a back-end triplication countermeasure inspired by SWIFT-R [CRA06], which uses three instruction streams as reference, error detection, and recovery respectively. I believe it covers all bit flips in registers, memory address corruptions, and memory operand corruptions, although its claim of detecting all "soft errors" has some openings.[3] NEMESIS has very detailed widgets and notably protects silent writes (no-op writes where the value written is also the value stored) against address corruption, which can't be detected by the usual precaution of re-reading written values.

My fetch skips countermeasure detailed in Chapter 4 [MDG24] reached the linker through ELF relocations. The scheme involves computing a checksum of assembly code, whose final encoding is not decided until the linker assigns explicit addresses to symbols. This could be performed on executables directly but only if it doesn't interact with any other relocation. I'll discuss more co-designed back-end countermeasures in Section 2.6.

Back-end hardening is the go-to choice for any countermeasure that handles low-level details and is quite effective as long as no source input is needed. Owing to the complexity of modern back-ends, most intra-functional transformations can be performed there, although the compiler's pipeline may be limiting.[4]

[3] In particular, it mentions instruction corruption as a potential threat, but changing the target register of an instruction can corrupt two execution streams at once. This leads, for example, to an attack that skips a memory write if the program writes the same value to memory multiple times, like e.g. `memset()` would do.

[4] For example, in LLVM adding new code becomes impossible sometime *before* reaching assembly.

Binary rewriting. A popular conservative approach is to rewrite output binaries; this bypasses the compiler and doesn't require access to source code.

Wenzl et al. [Wen+19] survey the mechanics of binary rewriting for different applications, including hardening against (mostly software) attacks. In each case, the binary is first analyzed using disassembly and structural recovery; it is then modified and reassembled using standard tools, sometimes even a production compiler. The survey highlights important variations in the analysis steps, as there doesn't appear to be popular widely-adopted tools in this area apart from reverse-engineering frameworks (IDA[5], radare2[6], etc.).

Binary rewriting isn't limited to simple transformations. Abromeit et al. [Abr+21] implement a masking countermeasure against side-channel attacks that observe secret-dependent data. This substitutes a number of elementary instructions with masking-compatible implementations known as *gadgets*, which requires moving some data to the stack and inserting calls to the gadgets. The paper performs these tasks almost entirely from an off-the-shelf binary, requiring only source-level annotations to identify secret variables.

One limitation of binary rewriting countermeasures is that program reconstruction is fundamentally incomplete, so despite working well in practice it's hard to prove their security. Kiaei et al. [Kia+21] argue that binary rewriting works best when lifting higher, and illustrate a lifting to LLVM IR to harden in the middle-end before essentially recompiling. Unsurprisingly, when compiler support is possible (and interference can be ruled out, see Section 2.4), it is more straightforward to harden as a compilation pass.

Custom compilers. Custom compilers may have specific provisions for hardening. For example, Jasmin [Alm+17] is a language and compiler for cryptographic implementations. It eliminates some conceptual overheads of C by giving the programmer finer control, such as in memory allocation: variables are explicitly allocated to registers, stack, or compile-time constants. This provides predictable assembly code generation and allows programmers to manually insert countermeasures at what would be an intermediate level of a C compiler. Jasmin is used in particular to write constant-time cryptography code.

2.3.2. High-level design considerations

A few design considerations apply regardless of implementation.

Countermeasure scoping and placement. Systems often have non-critical code that could be left unprotected, but is hardened anyway by default. Targeting critical code sometimes leads to gigantic differences in performance [Gei+23], making otherwise unaffordable whole-program protections viable. Most papers do not address the task of selecting the critical parts of a system for hardening, instead leaving it to domain experts typically *via* code annotations supported by static analysis. The task of passing down this information to hardening passes is also sneakily left unaddressed in most cases; I'll delve into this in Section 2.4.

A similar question for which manual specification isn't quite viable is the local placement of countermeasures. Not all fault attack paths in a given code unit (e.g. function) actually

[5] `https://hex-rays.com/IDA-pro/`

[6] `https://www.radare.org/n/`, `https://rizin.re/` (fork)

result in exploitable faults. Automatic countermeasures can thus be over-protective, which is a liability since any extra hardening code increases the program's attack surface. Boespflug et al. [Boe+23] optimize the placement of countermeasures against multi-fault attacks by enumerating actually-vulnerable paths with the help of a symbolic execution engine.

Multi-pass hardening schemes. Hardening passes are most often atomic; they take an arbitrary program as input, and produce a secure output with a security property directly related to the attack model, with one or more consecutive passes on a single program representation. Some of these passes may be analyses, like in Proy et al.'s loop hardening [Pro+17] that relies on LLVM's back-end loop analysis and supplements it with additional analysis.

Smokestack [AA19] randomizes the layout of functions' stack frames at each invocation to thwart Data-Oriented Programming attacks. It consists of 5 LLVM IR passes but has to modify libraries and the C runtime as well; the complex composition of binaries, which contain compiled sources, statically-linked code, dynamically-loaded dependencies, and a runtime, may not always accommodate high-level program transformations.

There are few schemes that work on multiple representations. SecSwift [De 19] is notable in this category. Despite being mostly located in LLVM's middle-end, it needs some of its duplication features in the back-end (notably duplicating return values of functions). Thus, the implementation arranges for duplicated IR instructions to be preserved until the back-end.

Countermeasure composition. In general, countermeasures only compose for free if they sit at different abstraction levels, like software replication on top of hardware error correction [Rei+05].

A few works do compose interacting countermeasures, like the evaluation of the CompaSeC transformation by Geier et al. [Gei+23]. The crux of CompaSeC is a fine-tuned combination of Dual Module Redundancy (DMR), a duplication of both code and instructions, and Runtime Signature Monitoring (RSM), a CFI scheme. This scheme is evaluated by comparing with pairings of independently-designed data-flow and control-flow countermeasures. This escalates issues with attack surface increases, and some combinations strikingly end up less secure than one of their components, with none reaching CompaSeC's level.

The PROSECCO [Bel+21] compiler features a number of countermeasures, including masking, code polymorphism, instruction replication and a CFI scheme. In one test scenario, a program is masked at the source level before an instruction skip countermeasure is applied by the compiler. The post-compilation verification tools in PROSECCO validate that the output program is indeed secure. This is a pairing we can expect to work well because these two countermeasures sit at different levels of abstraction and do not interact.

To my knowledge, the theoretical bases for analyzing composed countermeasures, especially at the back-end/architecture level, are completely unexplored. Since many of these countermeasures are not formalized, in some cases lack a clear fault model, and in many cases are not complete, the nature of the guarantees and limitations that could be derived in composed schemes is not obvious either. As such, countermeasure composition is a direction in which experimental schemes are ahead of formal analysis by a long mile.

Sharing of security abstractions. In their work about Security-Enhanced LLVM [Win18], Winderix make a compelling argument for sharing security abstractions between source languages like C and Rust, and targets like Intel SGX and Sancus [Noo+17] through compiler IRs. Their implementation focuses on module isolation but advocates for language-level formalizations of security mechanisms. Although my target is slightly different, this thesis strongly echoes this idea of factoring security tooling in the toolchain.

2.3.3. Overview of security modeling and proof

Compared to the extensive literature on the semantic correctness of program transformations and compilers, there's less work on proving security properties (with a clear focus on the constant time policy). This is likely related to the invasiveness of low-level and non-functional effects that are difficult to capture in formal models. Still, wherever a formal framework can be defined, usual reasoning methods used for correctness do transfer to security proofs.

Many countermeasures in security literature (for instance [CRA06; DSL17; Gei+23]) achieve sophisticated and carefully-balanced compiler transforms, but lack a detailed enough programming and security model for a formalization. They often rely instead on randomized or exhaustive attack campaigns for validation. This complicates the development of a formal security proof, or, in the more likely case that the countermeasure is incomplete, a detailed characterization of the threats that are not covered.

Multiple automatic methods can tackle cases where a proof is possible. Translation validation [BDG22] is a post-verification that the output program is secure independent of the implementation of the transformation. Static analysis [Chr+13] provides a conservative approximation of program semantics which can rule out undesired behaviors. Symbolic execution [Pot+14] performs a symbolic interpretation of program fragments which can the interface with SMT solvers.

As an example for validating a side-channel vulnerability, Bayrak et al. [Bay+13] implement an analysis to determine whether any value in a program depends on secrets without also depending on random inputs. The analysis operates in an LLVM back-end and identifies dependencies by querying a SAT solver for the possibility of a given input change affecting a given output value. It is configurable with user-supplied leakage models at the assembler level and finds potential leaks quickly thanks to SAT solvers' efficiency.

Formal verification against hardware vulnerabilities has also reached the certified C compiler CompCert [Ler09]. For example, Hutin proves a modification of CompCert that preserves the constant-time policy when applied to constant-time source code [Hut21]. Pesin et al. [Pes+25] add middle-end (RTL) countermeasures in Chamois CompCert [Mon+23] and prove their security at insertion time. Their framework covers generic expansions of instructions, which may include CFG transformations. To my knowledge, there is no established work (yet) on formally certifying security preservation across optimizations or lowerings in CompCert.

2.4. Preserving security in the compilation chain

Dealing with security properties across compilation stages raises inherent challenges. First, compilers preserve semantics but not arbitrary security properties, so countermeasures are liable to being broken by optimizations and lowerings. Second, the progressive lowering across multiple languages impacts the expression of security properties, if not the ability to express them altogether; so it is not even clear what compilers should preserve. These cross-layer issues have been raised previously [Bar+17], but do not have foundational answers yet.

I'll cover this linearly, starting with known breakages of security properties by compilers (Section 2.4.1) and common tricks used to counter them (Section 2.4.2). Then I'll focus on solutions, with a few certified results (Section 2.4.3) and Vu's systematic approach to property preservation, by which my work is heavily influenced (Section 2.4.4).

2.4.1. Compiler interference threatening security properties

Compiler optimizations are well-known for breaking security properties that extend beyond functional correctness [DPS15]. Looking at some properties discussed in Section 2.2:

- CFI checks are always redundant when standard language semantics are not violated, and are liable to being removed by constant folding with the help of static analysis.
- The constant-time property is difficult to maintain because compilers have no shortage of speed-improving peephole identities that re-identify branchless computations, and to a lower extent loop optimizations that can unmask constant-time array operations.
- Memory erasure is affected by Dead Store Elimination (DSE), even for arrays.
- Computation order can be blurred significantly by peephole rewriting and is explicitly overruled by back-end instruction scheduling.
- Control and data flow redundancy is explicitly eliminated by optimizations like basic block merging, Common Subexpression Elimination (CSE) and Global Value Numbering (GVN).

In his thesis about the preservation of the constant-time property in CompCert, Hutin [Hut21] provides such examples of clang introducing jumps that break constant-time. Figure 2.4 illustrates how unoptimized output from clang keeps the same structure as the original program, but rounds of optimizations from -O1 lead to LLVM recognizing a branching structure.

Many works avoid these issues by disabling optimizations [Reb+01; HLB19; Kia+21; WMP21]. However, this is a costly decision since the performance gain for optimizations can compensate the cost of major countermeasures like the replication in SecSwift [De 19]. The other popular approach is to insert countermeasures late enough to operate *after* all major optimizations, which is typically the case starting in the early back-end (e.g. for LLVM).

Note that even disabling optimizations is not a get-out-of-jail-free card; lowerings are a problem too. LLVM's front-end lowering from clang to LLVM IR performs a minimalistic constant propagation. The construction of SSA can also be accompanied by Global Value Numbering (GVN) [Lem23], which removes duplicate expressions and is an obvious risk for replication countermeasures. LLVM's Selection DAG data structure also merges identical

```
int f(int b, int x) {
  // !!b converts to boolean
  return (!!b) * x;
}
```

Source code

```
f:  movl    4(%esp), %eax
    testl   %eax, %eax
    je      .LBB0_2
    movl    8(%esp), %eax
.LBB0_2:
    retl
```

clang-16 -m32 -O1 (i386)

```
f:  pushl   %ebp
    movl    %esp, %ebp
    movl    12(%ebp), %eax   # (dead)
    movl    8(%ebp), %eax    # (dead)
    cmpl    $0, 8(%ebp)
    setne   %al
    xorb    $-1, %al
    xorb    $-1, %al
    andb    $1, %al
    movzbl  %al, %eax
    imull   12(%ebp), %eax
    popl    %ebp
    retl
```

clang-16 -m32 -O0 (i386)

Figure 2.4: Constant-time program insecurely compiled by clang's optimizations [Hut21]

computations in the same block. None of these steps can be disabled, making them the more fundamental issue; I'll expand upon this point in Section 3.4.3.

Simon, Chisnall, and Anderson [SCA18] demonstrate the full effect of this compiler interference by cataloging common security-related requirements by programs and various ways in which compilers can break them. They conclude that technical complexity in the compiler and subtleties in the language's specification (especially for C) mean that a continued engineering effort to make security a first-class citizen is the only option. However, such a design hasn't emerged yet and most literature still relies on various types of tricks.

2.4.2. Techniques for property preservation

Beyond disabling compiler optimizations, programs that require property preservation usually rely on known, informal compiler behaviors to generate suitable code. The listing below is roughly in order of wild hacks to solid guarantees.

Relying on difficult optimizations. One option is to hide secure code behind difficult optimizations that the compiler is not expected to perform. Lidman et al. [Lid+12] implement (an extension of) Dual Module Redundancy (DMR), which requires duplicating function inputs. As arrays cannot easily be duplicated, the scheme duplicates their accesses instead, as shown on Listing 2.5. The global array c is accessed a second time through a different pointer c2 which is dynamically equal to c but statically separate. The duplication won't be removed unless the compiler is capable of inter-procedural analysis or inlines `kernel2`.

Volatile and function pointers. Another popular trick is to abuse `volatile` objects or pointers, using the fact that compilers always allocate volatile objects to memory, and C and C++ make accesses to these objects side-effecting, thus not removable. This is typically used to protect redundant or otherwise dead code, which tends to be subtly incorrect because volatility only protects *accesses*, not uses.

```c
double c[SIZE];
/* Original function */
void kernel1() {
  for(int i = 1; i < SIZE-1; i = i+1)
    /* use c[i] */
}
/* Hardened function */
void kernel2(double *c2) {
  for(int i = 1; i < SIZE-1; i = i+1)
    /* use c[i] and duplicate with c2[i] */
}
// call site
kernel2(c);
```

Listing 2.5: Dual Module Redundancy for an array input to a function

Listing 2.6 shows a common technique for zeroing the memory of a dead array, by calling `memset` through a function pointer whose value is statically unknowable due to being volatile.

```c
/* Bypass DSE by preventing the static identification of memset */
void *(* volatile memset_ptr)(void *, int, size_t) = &memset;
memset_ptr(array, 0, sizeof array);
```

Listing 2.6: Dubious usage of `volatile`

While this works in practice, using `volatile` only guarantees that the program will *access* `memset_ptr`, not *use* it. In particular, the compiler could legally add a dynamic check for `memset_ptr == &memset` and skip the call if this turns out to be true. This may seem unlikely but it makes sense for a profile-guided optimization to inline a 100% pointer call on a hot path, especially as `memset` is a built-in with further optimization opportunities.

Correct usage of `volatile` does exist; Bonnal et al. [Bon+23] use it to force the compiler to read recently-modified variables in a CFI scheme, shown below.

```c
// Force read/write access to a variable in memory
#define ACCESS(VAR) *((volatile typeof(VAR) *)&(VAR))

int x = 2;
ACCESS(x) + ACCESS(x); // = 4, two memory accesses
```

Listing 2.7: Correct usage of `volatile`

Accessing a non-volatile object through an lvalue of volatile type used to not count as a volatile access [C11, 5.1.2.3§2], which was fixed[7] in C23 [C23, 5.1.2.4§2].

Modifying the optimization pipeline. One way to avoid interference from optimizations is to analyze the compiler pipeline and iteratively disable or modify optimizations that are found to break security. For example, Proy et al. [Pro+17] harden loops against control flow faults

[7] `https://www.open-std.org/jtc1/sc22/wg14/www/docs/summary.htm#dr_476`

early in the back-end, and find that LLVM's "control flow optimization" and "branch folding" passes interfere with the scheme. They disable the first and alter the second in their build of LLVM to ensure that the protected control flow is preserved. While flexible, this solution relies heavily on coverage from tests and requires continued expert maintenance with each compiler version.

Inline assembly. One official feature that's often useful for security is GCC's rich inline assembler mechanic [GCC25], which interfaces C and assembly code and is also supported by clang. While not standard (C has no `asm()` statement; C++ does but with no semantics), it's documented and provides reliable guarantees. GCC's inline assembly blocks are *opaque* input-to-output blocks of code and are subject only to limited optimizations that can be further reduced with the `volatile` qualifier. This can protect otherwise-optimizable operations like updating a statically-constant control-flow signature [VHM03], seen below.

```
// vulnerable to optimization like constant propagation
signature ^= 9265;
// cannot be removed or introspected
asm volatile("xorl %0, 9265": "+r"(signature));
```

Listing 2.8: Inline-assembly implementation of an x86 signature update

As another example, the automated masking implementation from Abromeit et al. [Abr+21] propagates information about secret variables all the way from the C source down to the assembler code before performing binary rewriting. This annotation takes the form of an inline assembly statement tied to the initialization of these secret variables. Here inline assembly is just a marker used as input in later analysis during binary rewriting.

Indirect semantic guarantees. SecSwift [De 19] uses a system similar to inline assembly for protection, but at the IR level. This uses new *intrinsic functions* which are opaque to LLVM and come with explicit lowering to other intrinsics all the way down to assembly. The opacity is enforced by LLVM's internal description of functions (including non-determinism, side-effects, etc.) which the compiler is guaranteed by API to honor. This method was validated experimentally under both `-O3` and link-time optimization (LTO).

Explicit library functions. Libraries being easier to evolve than languages, multiple revisions of the C standard added security-focused functions in the standard library. C11 introduced the bounds-checking interface known as "Annex K" [C23, K], with a unique feature that the bounds-checking `memset_s` function is protected against optimization [C23, K.3.7.5.1§4]. Annex K was unsuccessful in reaching wide adoption[8] but this particular feature was ported over to C23's `memset_explicit` function [C23, 7.26.6.2], standardizing third-party options like FreeBSD's `explicit_bzero` or Windows' `SecureZeroMemory`. The Linux manual page `bzero(3)` has stated since 2017[9] "[anticipating] that future compilers will recognize calls to `explicit_bzero` and take steps to ensure that all copies of the sensitive data are erased,

[8] For reasons explained in n1967: https://open-std.org/jtc1/sc22/wg14/www/docs/n1967.htm

[9] https://git.kernel.org/pub/scm/docs/man-pages/man-pages.git/commit/man3/bzero.3?id=55e04d23701e8aa8fb3c0164e95885740ea9ef44

including copies in registers or in "scratch" stack areas", an ambitious extension that to my knowledge has yet to be implemented.

Explicit guarantees from domain-specific languages. Custom languages can of course provide guarantees of their own, like Jasmin's preservation of the constant-time property. One example that doesn't use a fully custom toolchain is the FaCT DSL [Cau+19], which differs in that it uses Dudect [RBV17] for constant-time preservation and compiles to LLVM bitcode. FaCT still needs LLVM back-ends to correctly translate into targeted assembly without breaking security properties.

2.4.3. A few certified security properties

A few security properties have been studied in great detail and their preservation certified in production compilers, including a type of non-interference called *compartmentalization* [Thi+24] which is used in architectures likes CHERI [Woo+14].

Unsurprisingly, the constant-time property has received the most attention in this area. Because it relates to branches, array accesses and data-flow dependencies, which are native concepts of all languages in a C-to-assembler compilation chain, it presents relatively few cross-abstraction complications. Through their respective theses, Trieu and Hutin certified a constant-time-preserving compilation process in CompCert [Tri18; Hut21]. The constant-time preservation property for Jasmin is also proven in Coq.

2.4.4. Preserving generic properties

The issue of finding a general model for security properties during compilation goes hand-in-hand with whether compilers can preserve them *in general*. Fault attacks tend to branch out in many different subtle attack models that all have slightly different security properties; thus, the constant-time approach of preserving one specific property isn't as widely applicable (not to mention Spectre highlighting the limits of constant-time anyway [Cau+20]). This calls for a general framework for specifying and preserving security properties.

The foundations for this can be found in Vu's Ph.D thesis [Vu21]. Vu shows that a general class of *functional properties* extended with a concept of *observation* can be preserved by LLVM. An example of an observation (along with the general prototype) is shown on Listing 2.9.

```
int __builtin_observe(<VARIABLES...>, <PROPERTY(str)>);
__builtin_oberve(T, i, x, "T[i] == x");
```

Listing 2.9: Prototype of observations (simplified) and an example [Vu21]

The semantics of an observation, defined in every language of the LLVM compilation chain, is:

1. there exists a control point where all the `VARIABLES` are defined;
2. each variable's value is as if the program was evaluated strictly according to the rules of C's abstract machine up to this observation point;
3. hence, evaluating the `PROPERTY` at that point in any intermediate program yields its value as per the abstract machine.

This can be rephrased at a fundamental level in terms of valid program transformations. C and C++ make surprisingly few guarantees about programs' behavior; any program transformation is allowed as long as I/O operations and accesses to volatile objects are not reordered[10]. In other words, I/O operations and accesses to volatile objects (plus program termination) are the only "observable effects" in C and C++ (although compilers usually define more side-effects). Vu's contribution is key in that it adds *expressions* to the set, extending the base vocabulary available to express security properties at all abstraction levels with a single definition.

Another way to view this is that observations force LLVM IR's and assembler's respective loose notions of "variables" to follow part of the semantics of C variables, by appropriately inhibiting lowerings and optimizations. This closes the abstraction gap that prevented functional properties at the C level from being expressed at assembler level. Crucially, the implementation of observations is based on existing observable items of LLVM's intermediate languages (namely side-effecting intrinsics, like SecSwift [De 19]), meaning LLVM is guaranteed to preserve them. This is an important building block in the preservation of security properties.

Vu's work still leaves the question of what security properties can be captured by observations open. However, this vision of the property preservation problem at least shows that compiler integration can be decomposed into (1) generic preservation mechanisms in the compiler, and (2) implementation of security properties using these mechanisms. The main challenge addressed in this thesis is designing the preservation mechanisms.

2.5. Traceability in compilation

Security isn't the only client for passing down extra-functional information down compiler stages; so is *traceability*, the ability for users to determine how a compiler got to generating particular aspects of its output (which gives its name to my main contribution "Tracing LLVM"). This intersects domains such as Worst-Case Execution Time estimation [LPR14], but mostly runs into the validation of debug information [Li+20]—which is famously unreliable at high optimization settings for many of the same reasons as security countermeasures—and other runtime information like the precise memory layout of program data [KHM20].

Some properties can be formally studied in this way, such as information flow being partitioned into independent domains in CompCert [CSG16] or a generic representation of security property based on automata, intended to be checked by translation validation [NT20] after running a stock compiler. Traceability can be used more generally to evaluate implementations with regard to their specification [CGZ+12].

Similar objectives arise with *split compilation* [CR10; Les+07; Tag11], a technique that transfers analysis and optimization information from early stages to late stages of a compiler. It is specifically intended to connect the offline and online steps of a bytecode compiler, which offers a practical solution to the lowering problem: annotating the bytecode [KC01], which is a stable and well-documented interface between both sides.

[10] `https://en.cppreference.com/w/{c,cpp}/language/as_if`. Direct links: for C, for C++

2.6. Software/hardware co-design

Co-designed software/hardware schemes provide an interesting middle-ground. Pure hardware solutions often suffer from limitations in flexibility or performance; meanwhile, pure software solutions struggle to provide full security within reasonable time and space overhead due in part to a lack of foundational guarantees. The extra effort in implementing support on both sides pays off with a significantly larger design space, as shifting around responsibilities between both sides allows countermeasures to avoid each side's weaknesses while exploiting their strengths. Co-designed countermeasures are thus quite varied in nature and balance, with some recurring benefits.

2.6.1. Improving performance

Manssour et al. [Man+22] use hardware support to improve the performance of a countermeasure against data corruption that runs critical instructions multiple times, comparing the results to detect anomalies. Doing this in software is very costly due not only to repeated execution, but also code size, register pressure, and many extra branches. They propose a new instruction `rpl w n` which repeats the next `w` instructions `n` times, and compares results in hardware with an exception in case of a mismatch. In addition to major performance benefits, this reduces compiler work to only marking sensitive instructions. This does however increase the effort of hardware qualification/certification.

2.6.2. Addressing micro-architectural fault models

Gaudin et al. [Gau+23] augment the RISCV ISA with two instructions "lock" and "unlock" which allow a program to lock and unlock a cache line, ensuring that memory accesses at locked addresses are constant-time. Such approaches have been tested before, but eliminating information leaks is difficult because locking lines and accessing them still affects the publicly-visible state of the cache. This particular paper improves the security of the scheme by ensuring (among other things) that locked lines cannot be evicted implicitly and accessing them doesn't update LRU information. This leads to traces for an appropriate leakage model being independent of secret inputs in a simulator evaluation of the protection.

2.6.3. Enriching the interface layer between software and hardware

Guarnieri et al. [Gua+21] introduce the concept of *software/hardware contracts* that enrich the ISA with a specification of processors' speculative behavior. Each contract would formalize a hardware security guarantee that could be exploited by software to avoid side-channel attacks. Compilers could then target contract sets like they target ISAs, producing optimally secure code for each platform [GP20]. To my knowledge, this idea is not implemented yet, with most work focusing on finding appropriate formalisms for contracts.

Other approaches shift the balance to mostly hardware logic with minimal software support. For example, the CHERI/Morello initiative [Woo+14] adds enhanced pointers with security features directly in the ISA, allowing old programs in memory-unsafe languages like C to run securely. Other hardware primitives for security [Hu+20], such as True Random Number

Generators (TRNG), make it possible to implement secure cryptographic functions, Physically Unclonable Functions (PUF), and authentify devices securely.

This idea is not limited to security; RISC-V reserves many opcodes for `HINT` instructions whose purpose is to provide performance information, such as “memory-system spatial and temporal locality hints, branch prediction hints, thread-scheduling hints, security tags, and instrumentation flags for simulation/emulation” [RV1, 2.9].

2.7. Chapter conclusion

Countermeasures against fault injections frequently have to deal with compilation. Some protections are entirely in hardware, some can be performed directly on assembler code, but a significant number involves either transformations of compiler intermediate programs or a need to connect the source code with later stages. The extra-functional nature of these security endeavors means that compilers provide no explicit help, and with their current complexity, they’re not particularly predictable either. This leads to an adversarial relationship commonly fought with semi-reliable tricks, an approach that’s empirically tenable but hampers progress towards verifiable security guarantees and formal validation. Previous work has outlined a path towards security-aware compilation based on compiler extensions that wall off security code from interference, which I now wish to improve upon.

3 The vision: threading security through abstractions

This chapter describes the general direction of this thesis. The essential point made is that compilers are well-positioned to address the main reliability problems with software countermeasures (in large part because they cause them). I'll advocate for first-class compiler support for security as a separate research target (see Figure 3.1) and a secure compilation flow that doesn't require low-level information to establish security at intermediate stages (see Figure 3.2). Note that some of the components described in this chapter are still future work—the vision captures more than the contribution.

Sections 3.1 and 3.2 describe the broad objectives that this thesis targets in terms of improving reliability, and why treating secure compilation as its own subject is necessary. Section 3.3 goes into more detail as to the benefits that a security-aware compiler would bring to the design and validation of countermeasures against hardware vulnerabilities. Section 3.4 introduces small examples to showcase typical friction between compilation and security, with extra discussion in Section 3.5 and solutions (using Tracing LLVM) in Chapter 5.

Research questions

→ What are the lowest-hanging targets to improve the reliability of software protections?
→ How could a high-level compiler contribute to low-level security?

3.1. Targets of interest in the security process

There are a couple of directions I think are high-value targets to help address the effectiveness and cost of existing countermeasures. For brevity, I will refer to the process of analyzing attacks, designing, implementing and testing countermeasures as the *security process*.

3.1.1. Delimit uncertainties to facilitate unit validation

Managing uncertainty is key when dealing with hardware vulnerabilities. Some incompleteness is structural, like when devising fault models; some is "just" a hard engineering problem, like compilers interfering with code hardening. Obviously any uncertainty we can eliminate reduces the risk that countermeasures could be defeated in the real world. Essentially it would be great to have fully formalized attack models and security properties with theorems to prove that the countermeasures are effective, or get as reasonably close as possible.

A second objective, less obvious but in my opinion just as important, is to characterize the boundary around uncertain behaviors that we can't eliminate. It's more informative to refine a limitation that "compiler might optimize away instructions" into "compiler saturates its known set of peephole rewrites" because it helps characterize what compiling steps might defeat protections and provides a basis for evaluating the countermeasure.

Such specific evaluation is needed for the feedback loop of the security process. Hardware attack campaigns are the ultimate evaluation, but they don't readily explain causes of failure. Checking hardened circuits against simulated attacks, and hardened programs against emulated attacks, and compiler stages against a specification of hardened code, provides insight into individual steps and translates much more easily into design improvements.

3.1.2. Account for high-level security requirements

Countermeasures against fault injections in literature place more emphasis on dealing with the low-level effects of attacks than building up security properties. For instance, it's quite common to run into countermeasures that protect against multiple fault models but rare to find one that guarantees multiple security properties. In fact, as I discussed in Section 2.2, many just fix (partial) functional correctness by default.

However, applications have high-level security requirements that are both stronger and weaker:

- **Stronger requirements** include any other non-functional properties, such as availability (denial of service by repeatedly triggering a countermeasure may not be acceptable), confidentiality (faults can leak information both functionally and through side-channels), or performance (costly replay countermeasures for instance may not suit a real-time system).
- **Weaker requirements** because most systems are not worried about all failures; for instance a bootloader might not care if it's faulted *as long as* it doesn't boot an untrusted system. Most programs also have lots of non-critical code that doesn't need protecting.

Accounting for these requirements to strengthen the countermeasures while weakening their scope would make them more attractive to widely-used embedded systems. Currently, the performance cost mostly limits them to the most critical systems such as smart cards. (This is not helped by a tendency to use unoptimized builds as a baseline in performance evaluations, even though unprotected builds would be optimized.)

Hardware countermeasures are the most obvious examples where security properties are fixed for the entire program (being the furthest away from source-level requirements). Ultimately, a hardware-only countermeasure is a stronger claim than might first seem: it's a hardware behavior complete and secure enough to fulfill the security property software needs, in a cost-effective way, without information or intervention. Since hardware is traditionally only given software's *functional* requirements (through code), any consideration of other requirements will likely require some software contribution.

Of course, to provide requirements to countermeasures we need to connect the higher levels of abstraction (where the requirements are initially specified) with the lower levels (where the attack and countermeasure operate). This will be one of the main purposes of Tracing LLVM, echoing the need for compiler *traceability* with a distinct tint of security.

3.1.3. Facilitate software/hardware co-design

Finally, adding the fact that accurate attack models often involve hardware components, the dual contributions of software and hardware become clear:

- **Software** is a versatile basis for countermeasure logic, has fast development and deployment cycles, and hosts application-specified requirements and protection scope.
- **Hardware** can protect against the initial effects of attacks, can self-assess better than software (which may not run when hardware is compromised), and can improve performance for common security code patterns.

My contributions are geared towards co-designed software/countermeasures, which are rarely explored compared to pure-software or pure-hardware protections, hence my emphasis on providing users with fine control of the compilation process until the lowest levels.

Of course, in most institutions it's not particularly straightforward to bring software and hardware designers to collaborate on such a project. The distribution of skills into different projects, teams, and research communities mirrors the functional layers that computers are built on and can be a high hurdle to clear. Still, improved software/hardware interfaces can emerge from thoughtfully-constructed countermeasures; software/hardware contracts [Gua+21] attempt this for side-channels. Hopefully, we will see more work in this direction for fault injection countermeasures too.

Exploring the design space of co-designed countermeasures is not the focus of this thesis, sadly; I stop at Chapter 4's demonstration that a co-designed approach can protect against an accurate model with good coverage and reasonable costs. Many details of the software/hardware interface deserve more attention and will hopefully be analyzed in future works.

3.2. Distribution of responsibilities

My key argument is that the interests above invite (if not downright require) splitting off secure compilation as a new responsibility. Figure 3.1 shows how this fits in the entire process.

Left is the end-user, typically an industry actor working on a secure application. Middle is the countermeasures themselves (top), with potential hardware contributions (bottom), which will be featured in Chapter 4 with an example of co-design. Right is the "new" secure compilation support block, which is the subject of Chapters 5 through 7.

Everyone already assumes that countermeasures could be designed once and used by multiple applications, and thus either publish their details, provide research artifacts, or otherwise document them. Compilation techniques are different; papers and artifacts mostly skim over their handling of the compiler and stick to the tricks needed for the countermeasures at hand without generalizing. A long-term solution to these recurring issues requires treating secure compilation as a first-class target [Win18; Vu21].

An important side-effect of splitting off secure compiler development is that there should be an interface linking it back to countermeasures. The follow-up question of *what tools programs*

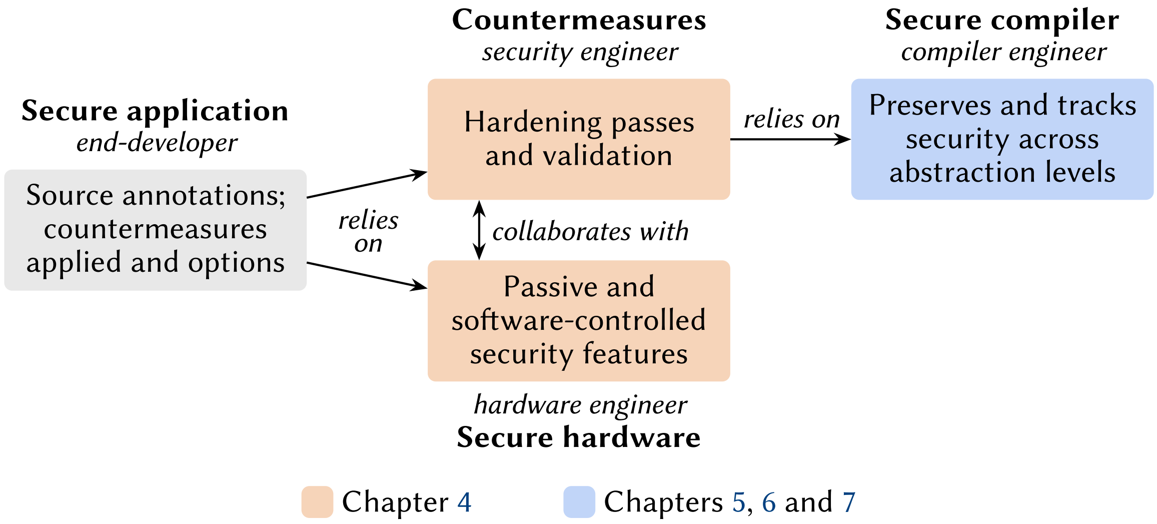

Countermeasures
security engineer
Secure compiler
compiler engineer
Secure application
end-developer
Hardening passes and validation
relies on
Preserves and tracks security across abstraction levels
Source annotations; countermeasures applied and options
relies on
collaborates with
Passive and software-controlled security features
hardware engineer
Secure hardware
Chapter 4
Chapters 5, 6 and 7

Figure 3.1: Distribution of responsibilities for countermeasure development

need from compilers for security is quite deep and will not be answered in this document, but hopefully Tracing LLVM can be a playground for exploring it.

3.3. The compiler as the cornerstone of hardening

A secure compiler for countermeasures isn't just about avoiding breakage—it could factor out the many uses cases explored in Section 2.3.1 and enable end-to-end, source-to-binary countermeasures. As the main driver of program analysis and code generation for the target system, the compiler is best positioned to perform hardening and assist by providing information that connects intermediate programs to the source code. Getting the compiler to cooperate this way is certainly not easy, but it's deliberate design, as opposed to the fundamental challenge of erecting solid protections while using an adversarial black box for code generation.

An off-the-shelf C compiler produces an assembly program whose behavior is constrained by a specification given as C source code. Compared to this, a secure C compiler should:

- Further constrain the assembly based on requirements given by countermeasures (security);
- Characterize these security requirements for each intermediate language so that every compiler stage can be studied and debugged individually (validation);
- Assist the implementation of countermeasures by connecting intermediate programs to the source specification (support).

Maybe the most obvious hurdle is that the compiler doesn't have access to the lower-level details of attacks, and so it needs a different model to track security during compilation.

3.3.1. Model of a secure build

There are two common visions for how compilers may generate secure programs. We speak of:

- **Property preservation** when the input program is already protected and the compiler is tasked with *preserving* this desirable property in its assembler output;

- **Hardening** when the input program is not protected and the compiler is tasked with transforming it into a secure form.

These approaches may look orthogonal because they place the responsibility of hardening either on the programmer or the compiler. But a closer analysis that accounts for compilers' intermediate languages reveals that they overlap while missing important concerns. For one, any hardening not performed on the very last program representation must be preserved during later compilation stages, so we should almost always be using both. Property preservation also implies that source properties automatically make sense on the target, which is definitely not the case in general. Instead of a semantic idea of "resisting attacks", we often have low-level attack models (that can't be captured at high levels) and high-level annotations (that direct hardening passes and need to be preserved until then).

Figure 3.2 illustrates a secure compilation model that accounts for this internal complexity. For clarity, the model is instantiated on a countermeasure against an assembler-level attack model, which takes source annotations and hardens in two steps at the LLVM IR and assembler levels. The left side describes the hardening steps while the right side lists the security property associated with each intermediate program.

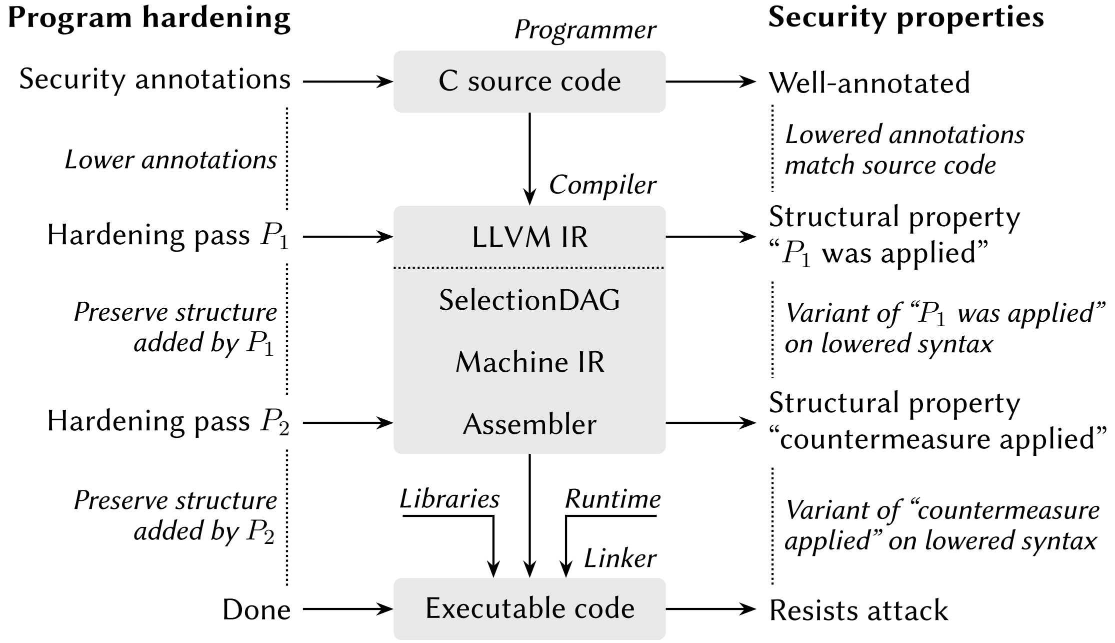

Program hardening
Programmer
Security properties
Security annotations
C source code
Well-annotated
Lower annotations
Lowered annotations match source code
Compiler
Hardening pass P_1
LLVM IR
Structural property "P_1 was applied"
SelectionDAG
Preserve structure added by P_1
Variant of "P_1 was applied" on lowered syntax
Machine IR
Hardening pass P_2
Assembler
Structural property "countermeasure applied"
Libraries
Runtime
Preserve structure added by P_2
Variant of "countermeasure applied" on lowered syntax
Linker
Done
Executable code
Resists attack

Figure 3.2: End-to-end security modeling for a two-pass countermeasure

In this approach, the compiler mixes hardening and preservation/lowering of security annotations or code. Notice how only the last security property is "resists the attack" as the attack model is formulated on assembly code. The initial security property is for the source to be properly annotated with respect to high-level security requirements. This doesn't reflect the source's functional semantics but instead the fact that we can compile it to a secure assembly program. Intermediate properties encode the *progress* of the hardening instead of attempting to describe the intermediate programs' response to an attack that they can't model. Since security code usually has non-functional aspects, these intermediate properties are encoded with a combination of:

- **Structural (syntactic) properties**, such as the presence of annotations, declarations for sensitive data using a dedicated type, or control-flow integrity checks being correctly interleaved with computations. These are straightforward to check in the source code and fit well in syntax-directed optimizations and lowerings.
- **Semantic properties**, such as data sensitivity being fully annotated on data-flow paths, or non-removable security checks being side-effects. These properties can be difficult to check (requiring, e.g., data-flow analysis) and their preservation relies on the compiler's correctness.

It may feel unnatural to depart from semantics and use structure to encode a notion of security. However, this makes sense for multiple reasons:

- First, the entire premise of non-functional requirements is that they don't fit within semantics. If the attack model is micro-architectural, even the assembly output won't have a semantic "resists attack" property to express.
- Syntax is already used for other non-functional targets like performance. For instance, LLVM IR has conventions on normalizing certain expressions to facilitate matching by InstCombine (the peephole rewriter), later leaving back-ends to undo it if the normalized form is not the fastest for the targeted architecture. Pattern-matching optimizations also rely on specific structure (e.g. detecting polyhedral loop nests).
- Finally, structure composes better. Implementing hardening passes in LLVM often requires technical adjustments, like splitting a pass in two; be it to work around a problematic optimization, to wait for a lowering, or simply to factor out a reusable component. In such a case, partially-hardened programs in-between passes are unlikely to have good semantic properties of their own, and are easier to characterize based on syntax. This generally makes it easier to assess whether nearby optimizations would interfere as well, as most transformations are syntax-directed.

This view suggests that we are compiling security "source code" (annotations) to security "target code" (countermeasure logic) where security compilation steps (hardening passes) are interleaved with functional compilation steps (analyses, optimizations and lowerings). For these interleaved processes to not interfere with each other, we need hardening passes to not affect functionality (which is usually fine) and optimizations and lowerings to not affect security, which is a universal concern for optimizations, but rarely-brought-up complication for lowerings. I will illustrate the importance of lowerings in examples in Section 3.4.

3.3.2. Potential for integration with the rest of the toolchain

A clear formalism for specifying and controlling secure code would benefit much more than just the compiler.

Any clean specification for a useful set of security annotations could be picked up by linters. It would allow tools like Coccinelle [Pad+06; Pal+11] to flag and update more suspicious patterns. And of course static analyzers like Frama-C [Cuo+12] could use it them to check top-level "well-annotated" security properties at a large scale.

The story is similar near the end of the compilation chain. On this side compilers interface with linkers, whose relevance will be illustrated in Chapter 4 with a checksum relocation. Debuggers are also heavy users of all traceability features and can contribute to security testing [Vu21, 4.4.1]. And emulators and simulators that check runtime traces for violations of expected security behaviors would benefit from cleanly lowering security specifications.

3.3.3. Lack of language support

So far, I've only advocated for security features against hardware attacks to be supported by compilers based on language extensions (like annotations, types, built-ins, etc.). A natural question is whether such features could be made official or standardized in the languages themselves. My perception is that this is a tricky proposition and unlikely to happen, on multiple counts.

First, there is no clear formalism for security mechanisms yet. In my approach the security properties for high-level and intermediate programs depend on the countermeasures being applied (through the intermediate structural properties), which is unlikely to generalize well to language-level features unless recurring patterns arise long-term.

Second, hardware attacks and countermeasures are in an arms-race dynamic that programming languages couldn't keep up with. Consider the huge can of worms Spectre [Koc+19] opened, with new micro-architectural attack vectors being found year after year. There is no single complete counter, so protections are best-effort and require continuous updates by a specialized community of both academics and industry experts. This process is comically out of tune with the way modern general-purpose or system programming languages evolve. Even Rust, which is lauded for its rapid adoption, has still only chipped the dominance of C in embedded development, despite addressing memory management questions that are better understood than hardware security—the time scale just doesn't match.

Finally, updating programming languages always leaves large amounts of legacy code behind, making their adoption into existing projects difficult in practice. Compiler extensions also suffer from that shortcoming but at least invite automated annotations and analyses that reduce the hardening effort.

3.4. Examples and use cases

Let's now go through a couple of examples illustrating the ways in which increased control of the compiler can serve security applications. In this chapter, I'll just introduce the examples and show the shortcomings of a stock compiler. Section 5.3 will present ways in which Tracing LLVM can accomplish the stated objectives. Table 3.3 lists the examples.

I will loosely group them in three categories based on the objective:

- **Take the source literally** refers to examples in which the source program accurately captures the desired property (via the abstract C machine's golden execution), and there is an "obvious" way to lower the code, but the compiler may not do it due to inner complexity.
- **Control lowerings** refers to examples in which the target code can be obtained by lowering abstractions in a specific manner, such as by using particular instructions or registers.

Take the source literally	Intro	Solution	Contribution
Strict variable accesses	3.4.1	5.3.1	New
Sequencing at variable writes	3.4.2	5.3.2	Solved by Vu
Control lowerings	**Intro**	**Solution**	**Contribution**
Avoid optimizations during lowerings	3.4.3	5.3.3	Made easier
Map source variables to registers	3.4.4	5.3.4	New
Trace source to target code	**Intro**	**Solution**	**Contribution**
Cleanup sensitive registers	3.4.5	5.3.5	New
Split register allocation	3.4.6	5.3.6	New

Table 3.3: Summary of the examples discussed in Chapters 3 and 5

- **Trace source to target code** refers to examples in which the desired property in the target code is expressed by linking it back to the source code, such as hardening code relating to a given source variable or expression.

Some examples rely on a macro `IO` which produces a side-effect of its first argument and returns it. Writing `IO(n)` forces the compiler to compute `n` and blocks optimizations such as constant propagation. The macro itself expands to an assembly comment "I/O modifies ⟨register⟩". Here is its definition (see Section 6.2.2 for an explanation of the inline assembly):

```
// IO: int -> int
#define IO(_var) ({ \
    __asm__ volatile ("# <I/O modifies %0>": "+r"(_var)); \
    _var; \
})
```

This macro is only used to keep relevant code alive (for the sake of writing short example code) or as pre-protections in examples in which the focus is elsewhere. While its behavior is nearly identical to a Tracing LLVM feature (specifically `__builtin_tracing_opaqueio`), `IO` should be interpreted as scaffolding for the examples and never as a part of the solution.

3.4.1. Strict variable accesses

- *Category:* Take the source literally
- *Requirement:* Keep accesses to a variable as in the source code, without forcing it to memory
- *Motivation:* Preserve logic surrounding redundant variables like CFI signatures
- *Failures:* Optimizations break requirement; `volatile` uses memory

One way source semantics are modified by compilers is by simplifying away the redundancies introduced by countermeasures. Programs protect against this by creating artificial semantic variations around redundant code. For Control-Flow Integrity (CFI) schemes, which focus on maintaining and checking a control-flow signature, a natural option is to obfuscate accesses to the signature variable, as all CFI logic is based on reading or writing it.

Figure 3.4 shows a dispatcher for a "get key" operation parametrized by key size and protected with the chain-of-trust CFI scheme by Bonnal et al. [Bon+23] (it's simplified from an example

```
/* Signature values for each section. C01 is the transition C0 -> C1 */
enum { C0 = 0xb7, C1 = 0x2a, C01 = C0 ^ C1 };

void get_key_1(int key_size) {
  int volatile CoT = C0;

  CoT ^= key_size;
  switch(key_size) {
  case 128: get_key128(); CoT ^= (C01 ^ 128); break;
  case 256: get_key256(); CoT ^= (C01 ^ 256); break;
  default: abort();
  }
  if(CoT != C1) abort();
}
```

Figure 3.4: Strict variable accesses: source code using chain-of-trust

```
get_key_1:
  addi    sp, sp, -16
  sw      ra, 12(sp)
  li      a1, 0xb7   # C0
  sw      a1, 8(sp) # 8(sp) is CoT

  # CoT ^= key_size
  lw      a1, 8(sp)
  xor     a1, a1, a0
  li      a2, 256
  sw      a1, 8(sp)

  # Dispatch key_size
  beq     a0, a2, .case256
  li      a1, 128
  bne     a0, a1, .abort

.case128:
  call    get_key128
  lw      a0, 8(sp)
  xori    a0, a0, 29  # C0^C1^128
  j       .end

.case256:
  call    get_key256
  lw      a0, 8(sp)
  xori    a0, a0, 413 # C0^C1^256

.end:
  sw      a0, 8(sp)
  # Check that CoT == C1
  lw      a0, 8(sp)
  li      a1, 0x2a    # C1
  bne     a0, a1, .abort
  lw      ra, 12(sp)
  addi    sp, sp, 16
  ret

.abort:
  call    abort
```

Figure 3.5: Strict variable accesses: slow assembly with `volatile` (-O3)

in that paper). Here, the objective is to hide the values read from or written to the signature variable `CoT` to avoid optimizations. (Chain-of-trust incorporates program data like key_size in the signature but peephole optimizations can still see through this.)

The original solution is to make the `CoT` variable volatile[1], which makes all accesses side-effects and eliminates the compiler's assumption that each value read is the last value written. This successfully gets the compiler to keep all the checks; Figure 3.5 shows the resulting assembly.

[1] More accurately, to access it through a volatile pointer, which leads to the same result.

Unfortunately, even when optimizing aggressively at -O3, volatility forces the variable on the stack, which is detrimental for performance and a potential security issue for other attacks (as buses and memory are generally more vulnerable to observation and interference than CPU registers).

3.4.2. Sequencing at variable writes

- *Category:* Take the source literally
- *Requirement:* Partially enforcing a sequence on normally pure operations
- *Motivation:* Preserving the order of control-flow checks with respect to surrounding code
- *Failures:* Optimizations break requirement; -O0 produces slow code with no guarantees

This example is motivated by Step Counter Incrementation [HLB19], a CFI scheme in which, among other things, progress along straight sections of code is tracked by counters. Figure 3.6 shows the code for a sequential PIN verification function protected by SCI to detect if a digit check is skipped.

```
#define SCI_CHECK(_n) { SCI = IO(SCI) + 1; ASSERT(SCI == _n); }

int verify_pin_1(uint8_t *userPIN, uint8_t *secretPIN) {
  int valid = true, SCI = 0;
  SCI_CHECK(1); if(userPIN[0] != secretPIN[0]) valid = false;
  SCI_CHECK(2); if(userPIN[1] != secretPIN[1]) valid = false;
  SCI_CHECK(3); if(userPIN[2] != secretPIN[2]) valid = false;
  SCI_CHECK(4); if(userPIN[3] != secretPIN[3]) valid = false;
  return valid;
}
```

Figure 3.6: Sequencing at variable writes: source code with Step Counter Incrementation

For this demonstration, the step counter is opacified at each step through an identity I/O so the compiler doesn't remove the entire scheme. However, no attempt is made yet to ensure the proper interleaving of SCI checks and PIN digit checks, which is the focus of this example.

Figure 3.7 shows the assembly code produced by LLVM at -O1 (which is identical to -O3 here). The back-end decides to rewrite the element-wise array comparison into a branchless test. The bitwise difference is computed for each index with a xor instruction, then folded over the entire array with three or instructions. If the result has any non-zero bits, then the array differs for some index.

In the process of rewriting the comparison, the compiler lumps together the SCI_CHECK() calls at the beginning of the function, rendering the scheme effectively useless. In fact, after the rewrite a placement of the checks in-between the logical operations that would be equivalent to the source sequencing doesn't even exist. Producing a correct output here will require the compiler to keep each index test separated.

As a side remark, the compiler also optimizes the SCI variable in subtle ways, leading to no actual incrementations. Seeing that the assertions are all "IO(SCI)+1 == _n" where _n is a constant, LLVM subtracts 1 from the right-hand-side and delays the incrementation of SCI

```
verify_pin_1:
  addi    sp, sp, -16
  sw      ra, 12(sp)
  li      a2, 0
  # <I/O modifies a2>
  bnez    a2, .abort
  li      a2, 1
  li      a3, 1
  # <I/O modifies a3>
  bne     a3, a2, .abort
  li      a2, 2
  li      a3, 2
  # <I/O modifies a3>
  bne     a3, a2, .abort
  li      a2, 3
  li      a3, 3
  # <I/O modifies a3>
  bne     a3, a2, .abort
  lbu     a6, 0(a0)
  lbu     a7, 0(a1)
  lbu     a4, 1(a0)
  lbu     a5, 2(a0)
  lbu     a2, 2(a1)
  lbu     a3, 1(a1)
  lbu     a0, 3(a0)
  lbu     a1, 3(a1)
  xor     a2, a2, a5
  xor     a5, a6, a7
  xor     a3, a3, a4
  xor     a0, a0, a1
  or      a0, a0, a2
  or      a3, a3, a5
  or      a0, a0, a3
  seqz    a0, a0
  lw      ra, 12(sp)
  addi    sp, sp, 16
  ret
.abort:
  call    abort
```

Figure 3.7: Sequencing at variable writes: reordered assembly (-O1)

until after the test. However, code beyond the assertion is unreachable unless the condition holds, which allows LLVM to infer that IO(SCI) did indeed return _n-1. Thus, there is no need to store the updated value of SCI at all, and no need to perform the incrementation either; LLVM just materializes the propagated constant at the next point of use with the following logic (but at LLVM IR level).

```
SCI = 0;
// ... check #0 ...
SCI = IO(SCI);
ASSERT(SCI == 0);       /* simplified from _+1 == 1 */
                        /* since we didn't abort, we know SCI==0 */
// SCI = SCI + 1;       /* constant-propagated to SCI==1  */
// ... check #1 ...
SCI = IO(1);            /* simplified from IO(SCI) */
ASSERT(SCI == 1);       /* simplified from _+1 == 2 */
```

I won't address this effect directly; it comes up again in the complex PIN verification program in Section 5.4, but the optimization won't apply there (as the panic handler isn't non-returning).

3.4.3. Avoid optimizations during lowerings

- *Category:* Control lowerings
- *Requirement:* Disable or bypass fixed optimizations performed during lowerings
- *Motivation:* In LLVM, back-end duplication and literals obfuscation
- *Failures:* Systematic failure, even at -O0

The next example illustrates the specific threat of optimizations mixed in with lowerings, of which there are mostly two in LLVM:

1. Immediate constants propagation when lowering the clang AST to LLVM IR;
2. Common Subexpression Elimination in blocks in the DAG-based instruction selector.

For the first lowering, let's turn to Figure 3.8, which is an attempt at obfuscating a constant by splitting it into two independent sets of bits[2].

```
#define CONSTANT 0x00c0ffee
#define CONSTANT_a (CONSTANT &  0xa5a5a5a5)
#define CONSTANT_b (CONSTANT & ~0xa5a5a5a5)

uint32_t get_constant_1(void) {
  return CONSTANT_a + CONSTANT_b;
}
```

Figure 3.8: Avoid optimizations during lowerings: source code with obfuscated constant

In resulting IR in Figure 3.9 the entire expression gets constant-folded, and this is key, *by the lowering to LLVM IR*, making it one of the rare instances where even -O0 will optimize, despite producing a function with the optnone attribute[3] which explicitly turns off optimizations. (This happens because clang uses an LLVM IR builder class whose instruction constructors greedily fold into constants whenever possible, regardless of optimization settings.) The desired behavior is to preserve the addition while propagating the masks.

```
define i32 @get_constant_1() #0 {
  ret i32 0xc0ffee
}
attributes #0 = { noinline optnone } ; ... and many more
```

Figure 3.9: Avoid optimizations during lowerings: LLVM IR with folded constant (-O0)

For the second lowering, consider Figure 3.10, which is the LLVM IR code for a function with a duplicated addition and the same attribute for disabling optimizations, optnone.

Here, the lowering to an instruction selection DAG coalesces the two additions because LLVM's implementation of the selection DAG data structure automatically merges identical nodes to enforce a uniqueness invariant. Figure 3.11 reveals this by showing the entry block's selection DAG just after construction[4].

The DAG represents a single basic block; its internal nodes are computations, its leafs are operands. The edges represent data dependencies (black lines) or control/ordering dependencies (dashed blue lines). Nodes have their inputs on the top and outputs at the bottom; looking at the highlighted add node (in red), the upper 0 and 1 cells represent its inputs, and

[2] Literals encoding is not so common in fault countermeasures. The idea comes from protections against reverse-engineering; see for instance https://tigress.wtf/encodeLiterals.html.

[3] https://llvm.org/docs/LangRef.html#function-attributes

[4] Actually, just after type legalization. The initial combining pass cleans up some dead nodes.

```
define i32 @f(i32 %x, i32 %y) #0 {
  %z1 = add i32 %x, %y
  %z2 = add i32 %x, %y
  %eq = icmp eq i32 %z1, %z2
  br i1 %eq, label %bb.ok, label %bb.err

bb.ok:
  ret i32 %z1
bb.err:
  call void () @abort()
  unreachable
}
attributes #0 = { noinline optnone }
```

Figure 3.10: Avoid optimizations during lowerings: LLVM IR with duplicate add

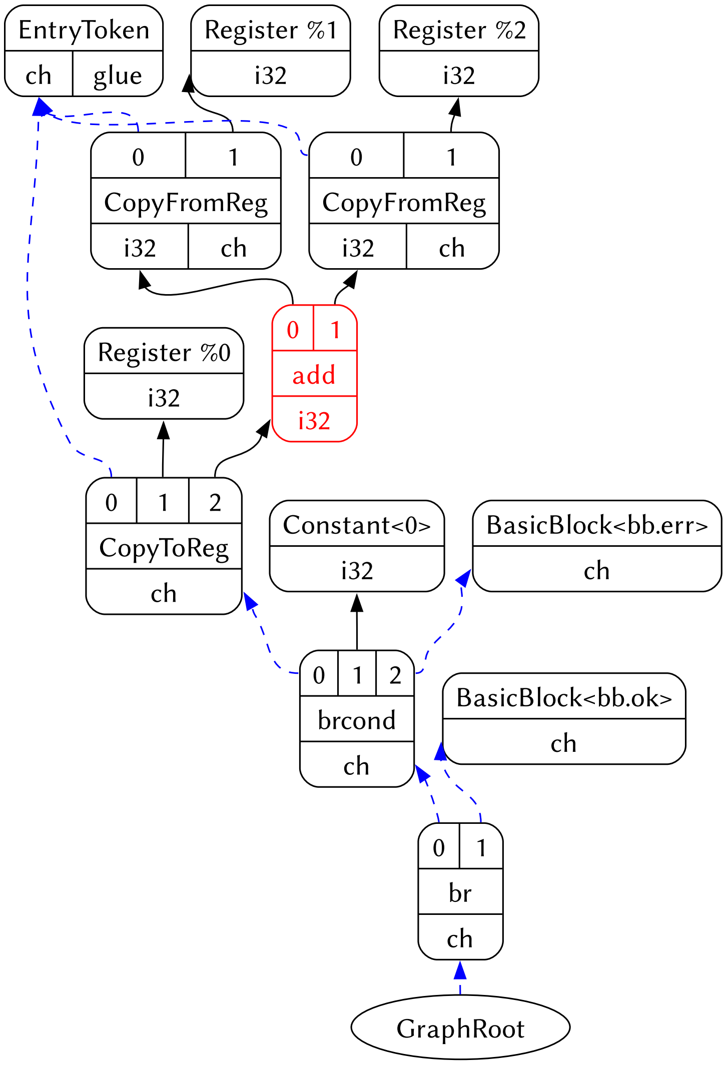


Figure 3.11: Avoid optimizations during lowerings: Selection DAG with merged add

the lower i32 cell is the type of its output. The type ch represents the *chain* token which is used for ordering (and models side-effects, among other things). Here, ignoring the registers and copies thereto/from, we are left with a single addition (add), and one comparison to zero (brcond) before we end the block on a jump (br) to the success block, instead of having both additions and a fault-detecting comparison as intended.

These lowering-related transformations are uniquely tricky because unlike dedicated optimization passes, lowerings can't be disabled, so the optimizations can only be blocked by changes to the compiler.

3.4.4. Map source variables to registers

- *Category:* Control lowerings
- *Requirement:* Assign all of a variable's live ranges to a single, consistent register
- *Motivation:* Runtime introspection, interfacing with hardware
- *Failures:* Variable's identity is lost in translation at SSA

Looking deeper now at the transformations inherent to abstraction lowerings, let's consider storage mechanisms. Storing small pieces of data with function lifetime is done with local variables in C and, pressure notwithstanding, in CPU registers in RISC-V assembler.

For this example, I'll repurpose the chain-of-trust function from Figure 3.4. The CFI signature CoT is a native-sized integer and remains alive through the entire function. It may be relevant to store it in a single register for the entire function:

- to provide runtime inspection without a full debugger and debug information;
- to simplify interfaces with hardware, where a fixed register could serve as an implicit argument in jumps and other accelerated updates or checks;
- or for inter-procedural CFI, to chain it across calls as an extension of the calling convention without materializing it explicitly as a formal argument.

However, LLVM doesn't construct a one-to-one correspondence between the variable and a target register. Consider the intermediate code shown in Figure 3.12, which, for the sake of demonstrating the issue without removing the protection, is compiled with -O0 followed by just the SSA register promotion pass (mem2reg).

```
define void @get_key_1(i32 %key_size) {
entry:
  %xor = xor i32 183, %key_size
  switch i32 %key_size, label %abort [
    i32 128, label %case128
    i32 256, label %case256]

case128:
  call void @get_key128()
  %xor1 = xor i32 %xor, 29
  br label %end
case256:
  call void @get_key256()
  %xor3 = xor i32 %xor, 413
  br label %end

end:
  %CoT = phi i32 [ %xor3, %case256 ],
                 [ %xor1, %case128 ]
  %cmp = icmp ne i32 %CoT, 42
  ; ...
```

Figure 3.12: Map source variables to registers: IR with variable identity lost (-O0 -mem2reg)

The construction of the SSA form of the intermediate program (which is at the heart of the compiler's design) makes each live range of CoT a different SSA value, leading to four different "variables" %xor, %xor1, %xor3 and %CoT. The relationship between these values (that they are all assigned to CoT) is lost at the very first step.

Similar to the "Strict variable accesses" example, disabling optimizations or using volatile lead to CoT being stored on the stack. In fact, the optimization that promotes CoT to a register is also the one that breaks its live ranges apart.

3.4.5. Cleanup sensitive registers

- *Category:* Trace source to target code
- *Requirement:* At end of function, zero all registers that held sensitive data
- *Motivation:* Modular reasoning for side-channel resistance
- *Failures:* By the time registers have been allocated, sensitivity info is lost

As with any countermeasure, reasoning about data leaks for side-channels is easier if leaks can be contained to a comfortable unit of code such as a function. This can be tricky for micro-architectural or physical side-channels due to hidden state, but in any case one has to start at the architectural level, with the base requirement that a function manipulating sensitive data erases it before returning so it can't be observed by someone else.

This issue of secure erasure is usually presented on arrays or other buffers stored in memory, as memory behaves predictably and the leak can be modeled at a high-level. Naturally, sensitive values can also leak through registers, but erasing them is a much trickier problem. Figure 3.13 demonstrates the limitations of attempting this at a high level of abstraction.

```c
/* T1 contains sensitive data */
int compare_arrays_1(int *T1, int *T2, unsigned N) {
  int equal = true, T1i;

  for(unsigned i = 0; i < N; i++) {
    T1i = T1[i] + 1;
    if(T1i != T2[i])
      equal = false;
  }

  T1i = 0;
  return equal;
}
```

Figure 3.13: Cleanup sensitive registers: C source code with sensitive array T1

The (imaginary) function compare_arrays checks whether an "obfuscated" array T1 containing sensitive data is equal to an array T2 containing non-sensitive data. To keep the example short, the obfuscation is limited to adding 1. While executing this function, the values of individual array elements T1[i] are read and decoded, and thus the storage for these intermediate values, T1i, is zeroed upon leaving the function.

The assembly code generated with `-O1` is shown in Figure 3.14 (`-O3` happens to be identical). Unsurprisingly, the assignment `T1i = 0` is eliminated due to being dead, as C doesn't have a notion of shared local state between functions.

```
compare_arrays_1:
  # a0 is T1, a1 is T2, a2 is n
  li      a3, 1
  beqz    a2, .end
.loop:
  # Load secret into a4
  lw      a4, 0(a0)
  lw      a5, 0(a1)
  # Branchless condition using a4
  addi    a4, a4, 1
  xor     a4, a4, a5
  snez    a4, a4
  addi    a4, a4, -1
  and     a3, a3, a4
  addi    a2, a2, -1
  addi    a1, a1, 4
  addi    a0, a0, 4
  bnez    a2, .loop
.end:
  # No zeroing!
  # Returns while a4 is sensitive.
  mv      a0, a3
  ret
```

Figure 3.14: Cleanup sensitive registers: Leaky assembly (-O1)

The inner equality check is rewritten to a branchless form, leaving only a minimal leak of register a4, which at the time of returning only carries one bit of information. However, with no safeguards these leaks can quickly spiral out of control, which I'll showcase in Section 5.4 (with a PIN verification function which basically leaves the entire secret PIN in registers for its callers to see).

The expected behavior is for any registers touched by the load or the "de-obfuscation" and not alive at the return statement to be reset to zero. Naturally, this should be done at a later compilation stage, at or around register allocation. However, it's not easy—sometimes not possible—to identify sensitive data on these late representations because the source information is lost.

3.4.6. Split register allocation

- *Category:* Trace source to target code
- *Requirement:* Allocate sensitive and non-sensitive values to different registers
- *Motivation:* Special hardware handling of sensitive data, facilitate validation
- *Failures:* By the time registers have been allocated, sensitivity info is lost

This final example is a variation on the array comparison program from "Cleanup sensitive registers". This time, we want sensitive and non-sensitive data to be allocated to disjoint sets of registers (ignoring fixed registers for function arguments and return values). The compiler won't do it on its own, and unsurprisingly this doesn't happen in Figure 3.14.

Nonetheless, this property would be of interest, mainly for interfacing with hardware. Indeed, if hardware knows which registers hold sensitive data then it can more easily reject leaky behaviors, select constant-time implementations of operations, or other countermeasures.

3.5. Discussion of the examples

At this stage, we can already highlight the primary angles I've been describing.

3.5.1. End-to-end management of security

In most of these cases we need to concern ourselves with the entirety of the abstraction stack, from C to IR to the instruction selection DAG and various details of Machine IR.

Three examples revolve around variables (3.4.1, 3.4.2, 3.4.4) and suffer from the notion being lost as soon as we lower to SSA, despite the fact there is a somewhat obvious direct lowering from C to assembly that would satisfy their needs.

Another two examples deal with source data that needs to be protected at a lower level (3.4.5, 3.4.6), which doesn't necessarily require a one-to-one mapping between source and target but requires *some* sort of mapping to implement the passes (similar to debug information). Such a mapping needs to be maintained for the whole compilation process.

And of course, the lowering optimizations example (3.4.3) demonstrates that it may only take one lowering to defeat a countermeasure. These problems naturally get worse the more representations there are.

3.5.2. Threats from lowerings rather than optimizations

In these examples, lowerings present three types of threats.

- **Unskippable optimizations**: I showed two basic ones, but the problem is more general. For instance, one can engineer the lowering from clang to LLVM IR (the construction of the SSA form) to also perform Global Value Numbering [Lem23], an optimization that would eliminate complex redundant code on the spot.
- **Inadequate abstractions**: the attempt to erase sensitive data in Section 3.4.5 fails because it goes through *C variables*, which is not an adequate abstraction. Using C variables fails to account for temporary expressions such as `T1[i]` which are eventually materialized as *SSA values* or *assembly registers* and need protection too. Ultimately, sensitivity here is a data-flow property and assigning variables can't capture data-flow in C.
- **Improper lowering of adequate abstractions**: even when the source code expresses the right property, such as the desired ordering of SCI checks and surrounding code in Section 3.4.2, the compiler may not preserve it. In this case, the back-end legally trades it off for performance.

Tracing LLVM was designed around this focus on lowerings and abstractions. Optimizations are still a threat, but they can be dealt with mostly predictably by means of opacification; this is because opacification is set up at a semantic level so an optimization breaking it would be functionally invalid. I'll come back to this in Section 5.2.

3.6. Chapter conclusion

This chapter has hopefully motivated my thesis that the compiler occupies a central role in software contributions to security countermeasures against hardware vulnerabilities. Even when there's no compiler interference to avoid, the support that a security-aware compiler can provide by generating predictable code, maintaining connections between source and assembly, or interfacing with the toolchain would be very valuable. The immediate target of this thesis is to block interference from optimizations and lowerings, and provide basic tools for tracing relevant program elements through abstractions. These goals will be demonstrated in Chapter 5 through my main contribution of Tracing LLVM.

Now for a direct answer to this chapter's research questions.

→ What are the lowest-hanging targets to improve the reliability of software protections?

In my opinion, these are the hurdles caused by the relative obscurity of the compiler on matters of security: its interface with high-level code (e.g. annotations specification), its internal behavior on optimizations and lowerings, and its interface with low-level code (including on fine details useful for co-designed countermeasures).

→ How could a high-level compiler contribute to low-level security?

Even if the attacks are low-level, co-designed countermeasures can greatly benefit from carefully-constructed assembler code, whether it is through eliminating indirect jumps, separating sensitive and non-sensitive data, or pre-computing validation data (as will be showcased in Chapter 4). This would constitute, in my opinion, a credible lightweight alternative to the full-featured hardware designs.

4 Countermeasures at the lowest levels of software

In Section 3.1.3, I proposed software/hardware co-design as a flexible solution against low-level, typically micro-architectural attack models, with software handling the logic and hardware providing base defense mechanisms and acceleration. This chapter demonstrates this idea by describing, implementing and proving a co-designed countermeasure against *fetch skips*, a micro-architectural refinement of instruction skips.

Key to this contribution is formalizing and proving the countermeasure. Indeed, the natural interface between software and hardware, the ISA, doesn't capture the intricate details of micro-architectural attacks (or they'd be assembly-level). Thus, a proper formalization and proof requires including hardware components in the formal model. Of course, verifying hardware is not a novelty by any means; the non-trivial part here is mixing software and hardware in a single semantic model that's guided by assembly syntax.

As we'll see, the co-designed approach reaches a compelling sweet spot: we counter an accurate model, with limited complexity on both software and hardware sides, good time performance, proven full coverage against single-fault attacks, and significant coverage against multi-fault attacks whose limitations are clearly outlined. The main costs are the code size and the need to involve hardware, the latter of which at least is already expected in the industry.

This chapter is largely reworded content from my CC'24 paper *"From low-level fault modeling (of a pipeline attack) to a proven hardening scheme"* [MDG24]. The implementation doesn't use any fancy tracing features as the work on Tracing LLVM had not started at the time, but it presents enough compilation complications to motivate them.

The chapter is organized as follows. Section 4.1 formalizes the fault in an extended assembler language. Section 4.2 proposes a countermeasure based on a hardware extension and a compile/link-time program transformation. From the semantics for the fault, Section 4.3 formalizes a security theorem proven in Appendix A. Section 4.4 describes my implementation in LLVM and GNU `ld`, its security evaluation with emulated fault campaigns, and its performance evaluation with processor simulations.

Research questions

→ Can software still contribute significantly to defeating a micro-architectural fault model?
→ How to prove an assembly program secure against a lower attack model?

4.1. Formalizing fetch skips

The fault model I want to counter, "fetch skips", was developed by Alshaer et al. [Als+22] after analyzing injection campaigns for clock and voltage glitches on ARM and later RISC-V microcontrollers. The target system is a 32-bit little-endian RISC-V processor with the "C" extension for compressed instructions (the base case being RV32IC [RV1]), which means it has a mix of 16-bit and 32-bit instructions. In this chapter, all 16-bit instruction encodings are prefixed with "`c.`" to distinguish them from 32-bit encodings. For a quick refresher on RISC-V syntax and semantics, please see Section 1.2.

Broadly speaking, a "fetch skip" will be a fault in which the processor skips (or replaces) bytes that should have been fetched from the instruction stream in a given memory access. If these bytes constitute a single whole instruction, that'll lead to an instruction skip; if not, then corrupted or brand-new instructions can make their way through the pipeline. Modeling this attack and the associated countermeasure will require accounting for memory alignment and the processor's internal fetch-decode-execute process.

To ease the presentation, I'll first describe the program and fault model informally (Sections 4.1.1 and 4.1.2) and then move to proper operational semantics (Section 4.1.3).

Terminology and notations in this chapter.

- aligned, unaligned: a multiple of 4 bytes; a multiple of 4 bytes plus 2
- line: 4 bytes of data at an aligned address (mimics table representation)
- $\mathtt{u}_\mathtt{N}$: type of `N`-bit unsigned integers
- $\mathtt{LSH}, \mathtt{MSH} : \mathtt{u}_{32} \to \mathtt{u}_{16}$: least and most significant halves of a 32-bit value
- Structure-like notations $\{\langle\text{field}\rangle : \langle\text{type}\rangle\ , \ldots\}$ and $\langle\text{struct}\rangle.\langle\text{field}\rangle$ are used for bit fields and named collections of values

4.1.1. Informal description of RISC-V programs

For this model, RISC-V programs are collections of *blocks* made of non- or conditionally-branching instructions, terminated by an unconditional jump[1]. Figure 4.1 shows the code for a function `g(x) = f(x) + 1` consisting of two blocks, one that calls `f` (implicitly forwarding the argument) and one that increments the return value `a0` then returns.

The execution of a program will be a sequence of *execution steps* in which the CPU obtains the next instruction (with a combination of a buffer read and/or a memory fetch), decodes it, and executes it. For instance, g executes in 8 steps (ignoring the call to `f`), with each step running one instruction and consuming either 2 or 4 bytes of code.

The central behavior of interest here is that *there is not always one fetch for one step*: certain instructions are fetched in advance, and others are read by the CPU over two consecutive fetches. This is because most CPUs *always fetch 4 aligned bytes in memory*, that is, a *line* in the memory layout table. For example, the first step in running g fetches the 4 bytes at address

[1] Traditionally conditional branches would end a block, but here only unconditional jumps do. This difference is of minor importance and serves to simplify the security proof.

C source code

```
int g(int x) { return f(x) + 1; }
```

Disassembled RISC-V code (instruction bytes in gray)

```
g:
  24: 41 11        c.addi    sp, sp, -16
  26: 06 c6        c.sw      ra, 12(sp)
  # call f (linker inserts address later)
  # ra is both target and return address
  28: 97 00 00 00  auipc     ra, 0
  2c: e7 80 00 00  jalr      ra, 0(ra)

  # add 1 to return value a0 of f
  30: 05 05        c.addi    a0, a0, 1
  32: b2 40        c.lw      ra, 12(sp)
  34: 41 01        c.addi    sp, sp, 16
  36: 82 80        c.ret
```

Memory layout (4 bytes per row)

24:	c.addi 41 11	c.sw 06 c6
28:	auipc (1/2) 97 00	auipc (2/2) 00 00
2c:	jalr (1/2) e7 80	jalr (2/2) 00 00

30:	c.addi 05 05	c.lw b2 40
34:	c.addi 41 01	c.ret 82 80

Figure 4.1: C code, object code, and memory layout of a function g(x) = f(x)+1

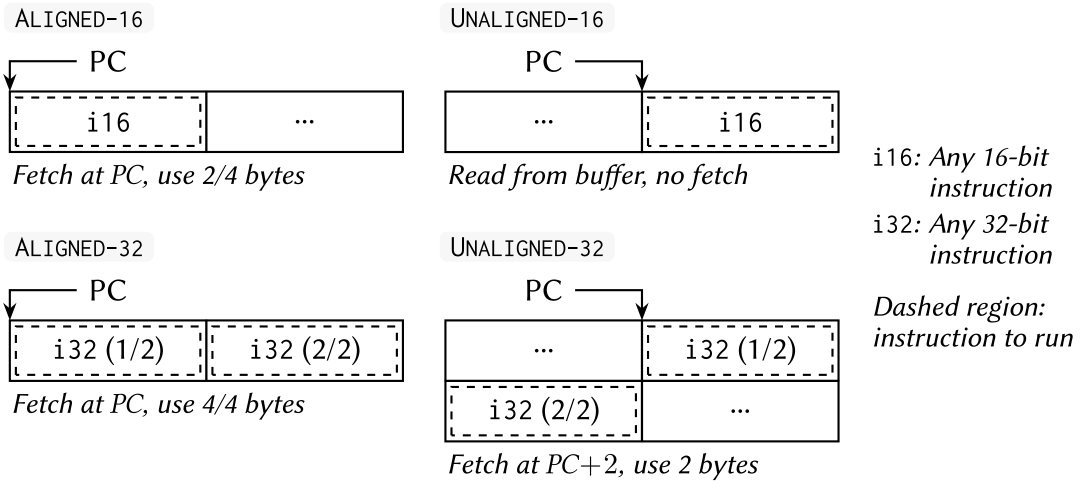

ALIGNED-16
PC
i16
...
Fetch at PC, use 2/4 bytes
UNALIGNED-16
PC
...
i16
Read from buffer, no fetch
ALIGNED-32
PC
i32 (1/2)
i32 (2/2)
Fetch at PC, use 4/4 bytes
UNALIGNED-32
PC
...
i32 (1/2)
i32 (2/2)
...
Fetch at PC+2, use 2 bytes
i16: Any 16-bit instruction
i32: Any 32-bit instruction
Dashed region: instruction to run

Figure 4.2: Visual representation of execution step rules

24 and puts them in a decoding buffer, but only consumes the first 2 to execute c.addi. Then, the second step executes c.sw from the buffer without needing another fetch.

Figure 4.2 shows all the ways a step may use buffered data and/or fetch from memory when executing one instruction. When PC is aligned (ALIGNED-*), the next instruction is on a new line, so a fetch is performed. When PC is unaligned, 2 bytes of instruction data are left in the decoding buffer. If they form a 16-bit instruction (UNALIGNED-16), the CPU runs it immediately without a fetch. If they form the first half of a 32-bit instruction (UNALIGNED-32), a fetch is performed at PC + 2 to obtain the second half and piece together the opcode. Each step then increments PC by 2 or 4, setting up the next cycle. This description accounts for the first stages of the pipeline while collapsing all the execution and write-back stages, which are not relevant in this particular fault model.

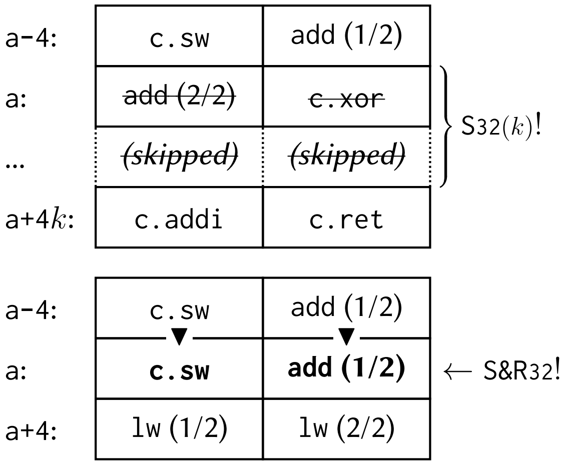


Figure 4.3: Effect of fetch skip attacks

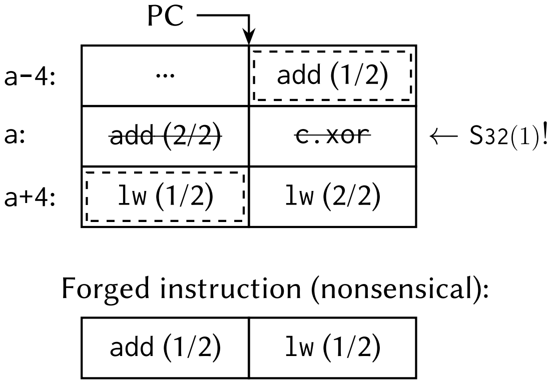


Figure 4.4: Forging a 32-bit instruction by attacking an UNALIGNED-32 step

4.1.2. Informal description of fetch skips

Fetch skips refine the traditional *instruction skip* (briefly discussed in Section 2.1.1 and addressed by multiple countermeasures from Table 2.3) by accounting for this mismatch between instructions and fetches. In the authors' original work, about 80% of faults injected by clock and voltage glitches were fetch skips. Because the model is more accurate with respect to physical effects, a system protected against fetch skips will be more secure in practice than a system protected against plain instruction skip.

The different types of fetch skips are illustrated in Figure 4.3. In this model, attempts by the CPU to fetch at an address a may result in three outcomes:

- **Normal behavior**: The memory contents at address a are returned.
- **Skip 32 bits**, k times (S32(k)): The memory contents at address a $+ 4k$ are returned, and PC is incremented by $4k$.
- **Skip and repeat 32 bits** (S&R32): The last fetched line (usually at a $- 4$) is returned, and the contents at address a later become the last fetched line.

A brief intuition as to why this happens is as follows. A typical fault leading to a fetch skip would be a *clock glitch*, where a short clock cycle is incorrectly introduced in the clock signal. This cycle is not long enough for CPU logic to propagate entirely. The S32(k) fault (observed for $k = 1, 2$) might happen if the decoder is unable to decode quickly and doesn't issue an instruction for the execution stage[2]. The S&R32 fault might happen if a fetch is started during the short cycle but doesn't complete in time, leading the decoder at the next cycle to capture the previous contents of the fetch stage's output register.

For the countermeasure, the threat model is a "level N attacker" that can independently attack every fetch with either one S32(k) attack ($1 \leq k \leq N$) or one S&R32 attack.[3]

The connection between fetch skips and the memory layout of instructions creates a new

[2] [Als+22] observes that the CPU sometimes runs a nop during a S32(k) attack, sometimes not. This modeling difference is accounted for in the countermeasure.

[3] I landed on these patterns because the behavior of more complex combinations cannot be inferred from [Als+22] without more hardware details.

effect of **instruction forging**, illustrated in Figure 4.4, that cannot be captured by the usual "instruction skip" model. Here, PC $=$ a-2 and the decoding buffer contains the first half of a 32-bit unaligned add instruction. The CPU fetches at a to obtain the second half (Figure 4.2, case `Unaligned-32`). Attacking this fetch with `S32`(1) results in an unrelated 16-bit value (the start of an `lw`) being used to complete the add, causing the execution of an opcode not originally in the program. This opcode can be "anything" (including illegal). Alshaer et al. [Als+22] demonstrate how this enables new vulnerabilities (e.g. by forging control flow instructions).

Once PC is out of sync, forging can continue without repeated fault injection. Continuing with Figure 4.4, after running the forged instruction we get PC $=$ a $+ 6$. Now the second half of `lw` is interpreted as its own 16-bit or (first half of) 32-bit instruction. Thus, the attack carries over to the next line; in the worst case, execution might not resynchronize with the intended sequence of instructions until a jump (which resets PC the start of an intended instruction).

4.1.3. Operational semantics of RISC-V programs with fetch skips

To design and prove a countermeasure against such a low-level attack, it is helpful to reflect the faults' effects into language features, and study the updated language with semantic tools. Let's now put formal definitions on this CPU and attack models with operational semantics. To avoid modifying this later I'll give the full model with countermeasure included right away, and explain the components related to hardening in Section 4.2.

Definition 1.
An instruction `i` *is a 16- or 32-bit integer (i.e. a value of type* u16 *or* u32*) corresponding to a RISC-V opcode.*[4] *The size of the instruction in bytes is written* $\|$`i`$\|$ *(equal to either 2 or 4).*

A block bb *is a nonempty sequence of instructions loaded in memory at a 2-aligned address.*

A program P is a collection of non-intersecting blocks, supposed well-formed in that every jump points to the beginning of a block.

Fetches are memory reads that update the fetch output buffer called ρ, and sometimes PC (as a side-effect of `S32`(k) attacks).

Definition 2.
A fetch is a statement $(PC, \rho)\ a \Rightarrow d\ (PC', \rho')$ *representing a load of 32 bits at 4-aligned address* a : u32 *yielding the "data" value* d : u32*, updating PC and the fetch buffer* ρ *in the process.*

Figure 4.5 shows the semantic rules for deriving fetches.

- `NoFault` is the normal behavior. Loading address a yields $[a]$, the contents of memory at a; this value is stored in the fetch buffer ρ, and PC is not affected.
- `S32`(k) is the generalized "skip 32 bits" attack: the program counter advances $4k$ bytes before the fetch is performed.
- `S&R32` is the "skip-and-repeat 32 bits" attack: the fetch is performed but the loaded data only reaches ρ at the next fetch, and the previous value of ρ is used for the current line.

[4] There is no risk of type confusion because 16- and 32-bit instruction encodings differ on their lowest 2 bits.

$$\textsc{NoFault}\ \frac{}{(\mathrm{PC},\rho)\ a \Rightarrow [a]\ (\mathrm{PC},[a])} \qquad \mathsf{S32}(k)\ \frac{1 < k \leq N}{(\mathrm{PC},\rho)\ a \Rightarrow [a{+}4k]\ (\mathrm{PC}{+}4k,[a{+}4k])} \qquad \mathsf{S\&R32}\ \frac{\rho \neq [a]}{(\mathrm{PC},\rho)\ a \Rightarrow \rho\ (\mathrm{PC},[a])}$$

Figure 4.5: Semantic rules for fetches

Before formalizing steps, I need one final element which is a description of the processor state. Since the attack and countermeasure are only concerned with fetch logic, this only needs to include ρ and standard architectural state; no other micro-architectural elements are needed.

Definition 3.
A program state is a quintuplet $\langle PC, \rho, \delta, \sigma, \alpha\rangle$, where
- *PC : u_{32} is the program counter;*
- *ρ : u_{32} is the last row fetched from code memory (fetch output buffer);*
- *δ : u_{32} is the current 32-bit value being decoded (decoder working buffer);*
- *σ describes registers associated with the countermeasure, detailed in Section 4.2;*
- *α : $\mathsf{u}_{32}[32]$ is the architectural state (registers `x0`...`x31`) as an array of 32 values.*

For simplicity, memory is not included; it could be handled in α like registers. Note that ρ and δ are 32-bit values and may not necessarily be legal instructions. From there, an execution is simply a series of state-updating steps ending in program termination.

Definition 4.
A step is a statement describing the execution of a single instruction, written $\langle PC, \rho, \delta, \sigma, \alpha\rangle \to r$ where r is either a program state or one of two termination reasons:

- *$\bot$, denoting an exception or crash;*
- *$\mathsf{end}(\alpha)$, denoting successful completion with final state α.*

An execution is a sequence of program states ending with a termination reason, such that all pairs of consecutive elements are related by a step:

$$\langle PC, \rho, \delta, \sigma, \alpha\rangle \to \ldots \to \langle PC_n, \rho_n, \delta_n, \sigma_n, \alpha_n\rangle \to \begin{cases} \bot \\ \mathsf{end}(\alpha') \end{cases}$$

All the alignment and decoding logic lies in how steps are constructed; the rules are given in Figure 4.6 and correspond directly with the visual illustration from Figure 4.2.

- `ALIGNED-*` are the cases where PC is aligned. They require a fetch to load the next 32 bits of code. Either `ALIGNED-16` or `ALIGNED-32` will then apply depending on whether the first 16 bits of the loaded word constitute a 16-bit instruction or the first half of a 32-bit instruction.
- `UNALIGNED-16` is the case where PC is unaligned, and the two unused bytes remaining in δ constitute a 16-bit instruction. This is the one case in which no fetch is needed to decode the next instruction; thus steps using the `UNALIGNED-16` rule can't be faulted by a fetch skip.
- `UNALIGNED-32` is the last case, where PC is unaligned and the remainder of δ forms the start of a 32-bit instruction. Here a fetch is needed to load the next line and get the other half of

the instruction. This is the only case in which the instruction to run is computed based on two different fetches, which presents the most opportunities for attacks.
- CHECKSUM-DELAY-SLOT (which is triggered by $\sigma.\mathtt{CCSDS} \neq 0$) is an extension from the countermeasure and will be discussed in Section 4.2.

The $[\![\cdot]\!]$ and $[\![\cdot]\!]_{ccs}$ functions represent the semantics of individual instructions; they return either an updated triple $(\text{PC}, \sigma, \alpha)$ or a termination reason. Both are oblivious to ρ and δ (which are micro-architectural implementation details) so the $\bullet$ function recombines their architectural output with the new ρ and δ. Notice how ρ is exclusively passed around the fetch statements (as it represents internal state of the fetch unit) while δ holds the fetched values seen by the decoder.

The formal definition of instructions' semantics is provided in Appendix A. For now, they only capture the usual RISC-V behavior (i.e. update registers in α, increment PC by 2 or 4).

$$\textsf{ALIGNED-32}\quad \frac{\begin{array}{cc} \text{PC aligned} & (\text{PC}, \rho)\ \text{PC} \Rightarrow \delta\ (\text{PC}', \rho') \\ \sigma.\mathtt{CCSDS} = 0 & \mathtt{LSH}(\delta) \text{ is a 32-bit leader}^{5} \end{array}}{\langle \text{PC}, \rho, \_, \sigma, \alpha \rangle \to [\![\delta]\!](\text{PC}', \sigma, \alpha) \bullet (\rho', \delta)}$$

$$\textsf{ALIGNED-16}\quad \frac{\begin{array}{cc} \text{PC aligned} & (\text{PC}, \rho)\ \text{PC} \Rightarrow \delta\ (\text{PC}', \rho') \\ \sigma.\mathtt{CCSDS} = 0 & \mathtt{LSH}(\delta) \text{ is a 16-bit ins.}^{5} \end{array}}{\langle \text{PC}, \rho, \_, \sigma, \alpha \rangle \to [\![\mathtt{LSH}(\delta)]\!](\text{PC}', \sigma, \alpha) \bullet (\rho', \delta)}$$

$$\textsf{UNALIGNED-32}\quad \frac{\begin{array}{cc} \text{PC unaligned} & \mathtt{MSH}(\delta) \text{ is a 32-bit leader}^{5} \\ \sigma.\mathtt{CCSDS} = 0 & (\text{PC}, \rho)\ \text{PC} + 2 \Rightarrow \delta'\ (\text{PC}', \rho') \end{array}}{\langle \text{PC}, \rho, \delta, \sigma, \alpha \rangle \to [\![\mathtt{MSH}(\delta) + 2^{16}\mathtt{LSH}(\delta')]\!](\text{PC}', \sigma, \alpha) \bullet (\rho', \delta')}$$

$$\textsf{UNALIGNED-16}\quad \frac{\begin{array}{c} \text{PC unaligned} \qquad \sigma.\mathtt{CCSDS} = 0 \\ \mathtt{MSH}(\delta) \text{ is a 16-bit ins.}^{5} \end{array}}{\langle \text{PC}, \rho, \delta, \sigma, \alpha \rangle \to [\![\mathtt{MSH}(\delta)]\!](\text{PC}, \sigma, \alpha) \bullet (\rho, \delta)}$$

$$\text{where}\quad \begin{array}{rcl} (\text{PC}, \sigma, \alpha) \bullet (\rho, \delta) & = & \langle \text{PC}, \rho, \delta, \sigma, \alpha \rangle \\ \bot \bullet (\rho, \delta) & = & \bot \\ \mathsf{end}(\alpha) \bullet (\rho, \delta) & = & \mathsf{end}(\alpha) \end{array}$$

$$\textsf{CHECKSUM-DELAY-SLOT}\quad \frac{\text{PC aligned} \qquad \sigma.\mathtt{CCSDS} \neq 0 \qquad (\text{PC}, \rho)\ \text{PC} \Rightarrow \delta\ (\text{PC}', \rho')}{\langle \text{PC}, \rho, \_, \sigma, \alpha \rangle \to [\![\sigma.\mathtt{CCSDS}]\!]_{ccs}(\text{PC}', \sigma, \alpha, \delta) \bullet (\rho', \delta)}$$

Figure 4.6: Semantic rules for execution steps

The step rules are exclusive, so execution is deterministic for any given attack sequence. For example, attacking the fetch at `0x30` when running g from Figure 4.1 produces this execution:

[5] As per the RISC-V ISA [RV1], a 32-bit instruction leader is a value $i : \mathsf{u}_{16}$ such that $i \equiv 3\ [4]$ and a 16-bit instruction is any other 16-bit value.

Secure RISC-V assembly (numbers on the right refer to lines of Algorithm 1 that added the instructions)

```
g: # PC >= 0x40000 is a protected region
   400e4: 41 11         c.addi    sp, sp, -1
   400e6: 06 c6         c.sw      ra, 12(sp)
   400e8: 97 00 00 00   auipc     ra, 0         # auipc was relocated
   400ec: 0b 64 00 00   ccscallb  8             # checksum was set, LSB flipped      (22,10)
   400f0: e2 75 06 c6   .word     0xc60675e2                                         (22,11)
   400f4: e7 80 00 fb   jalr      ra, -80(ra)   # jalr was relocated
   400f8: 02 90         c.ebreak                # repeats 8 times                      (28)

   40108: 05 05         c.addi    a0, a0, 1
   4010a: b2 40         c.lw      ra, 12(sp)
   4010c: 41 01         c.addi    sp, sp, 16
   4010e: 01 00         c.nop                                                          (19)
   40110: 0b 10 00 00   ccs                     # checksum set, no flip needed       (24,13)
   40114: 51 16 b3 40   .word     0x40b31651                                         (24,14)
   40118: 82 80         c.ret
   4011a: 02 90         c.ebreak                # repeats 8 times                      (28)
```

Figure 4.7: Linked object code for g after hardening ($N = 2$)

$$\begin{tikzcd}
\langle \texttt{0x30}, \rho_0, \delta_0, \sigma_0, \alpha_0 \rangle \arrow[d, "\texttt{S32}(1)\ \ \texttt{ALIGNED-16}"', "(\texttt{0x30}, \rho_0)\ \texttt{0x30} \Rightarrow [\texttt{0x34}]\ (\texttt{0x34}, [\texttt{0x34}])\ \ \text{Runs: } \texttt{c.addi sp, sp, 16}"] \\
\langle \texttt{0x36}, \rho_1 = [\texttt{0x34}], \delta_1 = [\texttt{0x34}], \sigma_1, \alpha_1 \rangle \arrow[d, "\texttt{UNALIGNED-16}"', "\text{Runs: } \texttt{c.ret}"] \\
\langle \alpha_1.\texttt{ra} = \texttt{0x30}, \rho_2 = \rho_1, \delta_2 = \delta_1, \sigma_2, \alpha_2 \rangle
\end{tikzcd}$$

c.ret jumps to the address stored in register ra, which is still 0x30 because the load from the stack was skipped, later leading to a stack corruption crash.

4.2. A co-designed countermeasure

This section describes a software/hardware countermeasure, based on code instrumentation with hardware support. This countermeasure draws inspiration from previous work on instruction skips [YS18] and control-flow integrity protections [Zgh+22; MCG22]. However, complications associated with multi-fault attacks incentivized me to aim for a design that guarantees security *locally* (at every block) to keep formal reasoning simple.

4.2.1. Overview

The key ideas of the countermeasure are as follows. Machine code is augmented ("hardened") during compilation with checksum protections that react to a fault attack by forcing execution to trap before the end of the current block. This limits exploits by ensuring a sufficiently tight window between attack and detection (like most countermeasures it doesn't prevent side-effects immediately resulting from the fault, which is inherently difficult due to timing). Figure 4.7 shows the hardened code for the g function from Figure 4.1.

Hardware is modified to automatically maintain a running checksum (in fact, a simple *sum*) of every line of instruction data fetched during the execution of a block, independent of instruction alignment. Blocks are compiled so that every exit is guarded by a ccs instruction (from our ISA extension), which traps if the running checksum is not equal to a reference value computed at compile-time. Blocks thus act as autonomous "jails", in that faulty executions causing the checksum to deviate from its expected value cannot leave their current block.

The co-design lies in the cross-checking of information between hardware, whose monitoring produces a trace (checksum) sensitive to fault attacks, and software, which provides reference checksum values to interpret that trace.

Section 4.2.2 describes the hardware extension used by the countermeasure. Section 4.2.3 details the hardening algorithm. Section 4.2.4 highlights the subtleties of implementing the hardening algorithm in LLVM. Finally, Section 4.2.5 discusses the design choices in a more general context.

As always, one can attempt to attack the countermeasure itself. The security theorem formulated in Section 4.3 and proved in Appendix A shows that no attempt at skipping, repeating or forging ccs instructions or jumps can succeed, leading to a strong security guarantee.

4.2.2. ISA and hardware extensions

The countermeasure is based on a custom ISA extension named Xccs for *Code CheckSum* (the "X" denotes an unofficial RISC-V extension). All examples in this section refer to Figure 4.7. Xccs introduces four new Control and Status Registers (CSR), which together form the σ field of the program state:

- `CCS` : u_{32} is the running checksum for the current block. For example, the region of g from address `40108` up to (but excluding) `40114` adds up in little-endian to

  `0x40b20505 + 0x00010141 + 0x0000100b = 0x40b31651`

  so the value of `CCS` at `40114` will be `0x40b31651` if no attack is performed.
- `CCSPROT` : u_{32} indicates the PC value at which a protected instruction (jump) is expected to execute. It is zero most of the time and non-zero for a single step after a checksum is validated. In Figure 4.7, it is set at `400f4` and `40118`.
- `CCSD` : $\{\texttt{E} : u_1, \texttt{JO} : u_5\}$ holds control information; the *enable* bit (`E`) indicates whether checksum protection is active for the current block (which currently is whenever PC $\geq$ `0x40000`), and the *jump offset* field (`JO`) is set by `ccscall` as described below.
- `CCSDS` : u_{32} ("delay slot") is used to trigger the Checksum-Delay-Slot rule. This is needed because Xccs instructions are encoded on two consecutive 32-bit words, so the second half shouldn't be decoded as a normal RISC-V instruction.

The meat of Xccs is the addition of four guard instructions whose encoding is shown in

31 25	24 20	19 15	14 12	11 7	6 0	
`funct7`	`rs2`	`rs1`	`funct3`	`rd`	`opcode`	
`0`	`0`	`0`	`001`	`0`	`0001011`	`ccs`
`0`	`0`	`0`	`010`	`N`	`0001011`	`ccscall N`
`0`	`0`	`0`	`101`	`0`	`0001011`	`ccsb`
`0`	`0`	`0`	`110`	`N`	`0001011`	`ccscallb N`

Figure 4.8: Encoding of Xccs instructions

Figure 4.8, all of which are followed by a 32-bit ⟨**`checksum`**⟩ argument:

- **`ccs`** ⟨**`checksum`**⟩ compares the `CCS` register with the provided argument and traps if they differ. Otherwise, it sets `CCSPROT` $= \mathrm{PC} + 8$ so a jump or function call can execute at the next step. For instance, at `40114` in Figure 4.7, the dynamic value of the `CCS` register is compared to `0x40b31651`. When no faults are injected, these are equal, so execution proceeds to the `c.ret` instruction.
- **`ccscall`** N ⟨**`checksum`**⟩ is similar; it is used before function calls. It sets `CCSD.JO` $= N$, which causes the next function call's return address to increase by $2N$ bytes. For instance, the `ccscallb 8` at `400ec` changes the return address after the `jalr` (call) instruction from `400f8` to `40108`, skipping over the `c.ebreak` block.
- **`ccsb`** and **`ccscallb`** are variations that flip the least significant bit (LSB) of the checksum before comparing it. This is leveraged to ensure that no ⟨**`checksum`**⟩ argument decodes as a jump or Xccs instruction, as detailed in Section 4.2.3. Again with Figure 4.7, the sum of the first block up to `400f0` is `0xc60675e3`, but that decodes as a `bltu`, creating a vulnerability. The LSB is flipped to avoid this.

All Xccs instructions further trap when run at unaligned PC, which prevents most attempts at forging them. Finally, existing CPU behavior is modified as follows when `CCSD.E` $= 1$:

1. All branch instructions trap if PC $\neq$ `CCSPROT`. Taking a branch resets `CCS` and `CCSPROT` to `0`.
2. Every other instruction traps if `CCSPROT` $\neq 0$.
3. Every value retrieved from a fetch is added to `CCS`.
4. Call instructions add $2 \times$ `CCSD.JO` to the return address and clear `CCSD.JO` before jumping.

Hardware support in this countermeasure serves an important dual purpose. First, it limits the reach and exploitability of potential attacks. Second, it provides a checksum update method that's not vulnerable to attacks, so that comparisons in Xccs instructions are reliable even if the reference ⟨**`checksum`**⟩ argument is itself vulnerable.

4.2.3. Hardening algorithm

Algorithm 1 shows the hardening process for a single block in pseudocode, with my canonical example in Figure 4.7. As all blocks are independent, this algorithm is executed by the compiler for every block in the program.

The main **for** loop (line 15) iterates over the instructions of the original block. All instructions are copied to the hardened block (line 26). Jump instructions are preceded by a guard, which

Algorithm 1 Algorithm: HARDEN

```
Input: A (source) block $[\mathtt{i}_1, ..., \mathtt{i}_n]$
Input: Upper bound $N$ for S32($k$) rule ($k \leq N$)
Output: A hardened block hb.
 1: hb $\leftarrow$ [ ]
 2: sum : u32 $\leftarrow 0$
 3: offset $\leftarrow 0$                                          ▷ Blocks are 4-aligned.
 4: procedure addToBlock(i)
 5:     hb.append(i)
 6:     sum $\leftarrow$ sum + realign(offset, u32(i))
 7:     offset $\leftarrow$ offset + $\|i\|$
 8: procedure addChecksum(i, ib)
 9:     if sum + i is not a valid checksum literal then
10:         addToBlock(ib)
11:         addToBlock(sum $\oplus$ 1)
12:     else
13:         addToBlock(i)
14:         addToBlock(sum)
15: for i in $[\mathtt{i}_1, ..., \mathtt{i}_n]$ do
16:     if i is a jump or branch instruction then
17:         if offset $= 0$ then                                   ▷ No empty sections.
18:             addToBlock(encode(nop))
19:         else if offset $\equiv 2\ [4]$ then                     ▷ Force alignment.
20:             addToBlock(encode(c.nop))
21:         if i is a function call then
22:             addChecksum(encode(ccscall ($2N+4$)),
                            encode(ccscallb ($2N+4$)))
23:         else                                                   ▷ Jumps/branches.
24:             addChecksum(encode(ccs), encode(ccsb))
25:         offset $\leftarrow 0$
26:     addToBlock(i)
27: for $j = 1$ to $2N+4$ do                                       ▷ Add trap barrier.
28:     addToBlock(encode(c.ebreak))
29: return hb
```

is ccscall/ccscallb (line 22) for function calls and ccs/ccsb for other jumps (line 24). nops are used to ensure that guards are aligned and jumps are always separated by at least one non-jump instruction, both of which prevent subtle attacks against the countermeasure.

The second **for** loop (line 27) adds a barrier of c.ebreak instructions, which raise a distinctive exception when executed. Their role is to prevent control from leaving the block by skipping over the terminator. Up to $2N+4$ are needed to address the worst case where control reaches the middle of a checksum whose second half is a 32-bit opcode, in which an attack of a S&R32 followed by a S32(N) would reach $4N+6$ bytes past the checksum.

Procedure addToBlock appends instructions to the hardened block while computing the reference checksum value sum by summing instruction's opcodes. This accounts for instructions'

```
g:  (...) # push ra to stack
    PseudoCALL @f
    $a0 = nsw ADDI $a0, 1
    (...) # pop ra from stack
    PseudoRET $a0
```

(a) Machine IR before back-end pass.

```
g:  (...) # push ra
    PseudoCALL @f, .LBB1_0
    PseudoCCSTRAP 2
.LCCS_Region_Start0:
    $a0 = nsw ADDI $a0, 1
    (...) # pop ra
    CCS .LCCS_Region_Start0
    PseudoRET $a0
    PseudoCCSTRAP 2
```

(b) Machine IR after back-end pass.

```
g: e4: 41 11 06 c6    (...)     # push ra
   e8: 97 00 00 00    auipc     ra, 0
   ec: 0b 24 00 00    ccscall   8
   # ▼ R_RISCV_CHECKSUM: g
   f0: 00 00 00 00    .word     0x00000000
   f4: e7 80 00 00    jalr      ra, 0(ra)
   f8: 02 90          c.ebreak # 8 times (2N+4)
.LCCS_Region_Start0:
  108: 05 05          c.addi    a0, a0, 1
  10a: b2 40 41 01    (...)     # pop ra
  10e: 01 00          c.nop     # ccs alignment
  110: 0b 10 00 00    ccs
  # ▼ R_RISCV_CHECKSUM: .LCCS_Region_Start0
  114: 00 00 00 00    .word     0x00000000
  118: 82 80          c.ret
  11a: 02 90          c.ebreak # 8 times
```

(c) Object code before linking.

Figure 4.9: Stages of hardening g in our LLVM implementation ($N = 2$)

alignment with the function

$$\texttt{realign}(\texttt{offset}, \texttt{i}) = \begin{cases} \texttt{i} \text{ if } \texttt{offset} \equiv 0 \ [4] \\ 2^{16}\mathsf{LSH}(\texttt{i}) + \mathsf{MSH}(\texttt{i}) \text{ otherwise} \end{cases}$$

This process is equivalent to summing the lines of the final layout table, a fact I will prove in Lemma 4. Finally, addChecksum selects whether to use ccs/ccscall or their -b variants. The -b variants are selected when the checksum value is an "invalid checksum literal", i.e. it decodes as a jump, Xccs instruction or c.ebreak. Such values could be misused as instructions if an attacker were to skip the guard that precedes it. Flipping the LSB ensures that these sensitive values do not appear in code. Figure 4.7 is obtained by executing this algorithm on both blocks of g's original code from Figure 4.1 and linking it.

4.2.4. LLVM implementation

Algorithm 1 cannot be implemented as-is in a single pass in a standard compiler, because reference checksum values depend on the exact bit-level encoding of each instruction, which is not decided until the linker relocates references to globals and functions. See for instance how the call in Figure 4.9c (before linking) has placeholder zero-offsets but the one in Figure 4.7 (after linking) has a proper target offset.

This is our first sign that the toolchain's lowering, which is designed around *functional* invariants, does not always serve *security* invariants well. Encodings are decided late because they don't matter to the compiler, which is only interested in the functional specification of instructions. The addition of a security countermeasure to the compiler breaks this assumption. This creates a new challenge of threading the security transform in-between functional steps that may not come in a suitable order. In this case, I was able to implement the algorithm in two steps: a late Machine IR pass followed by an extension to the linker relocation process.

- **Machine IR pass**: the program's Machine IR representation is first transformed late in the back-end (from Figure 4.9a to Figure 4.9b). This pass handles all tasks that *add* code into the program, including:

  - Aligning functions and blocks to 4-byte boundaries;
  - Adding aligned Xccs instructions before all Machine IR instructions that expand into RISC-V jumps, such as `PseudoRET`. A label indicates the start of the region that the checksum must cover;
  - Adding the trap barrier with the `PseudoCCSTRAP` N pseudo-instruction, which later expands into a series of $2N + 4$ `c.ebreak` instructions.

  At this stage some jumps are still hidden in pseudo-instructions, like the function call in `PseudoCALL`. This is because far jumps in RISC-V are implemented with a pair of instructions, `auipc` and a jump, to overcome the limited distance that can be encoded into single jump instructions. In LLVM, this is expanded later in the code emitter due to limitations in the back-end structure; I count this towards the Machine IR pass for simplicity of exposition.

  The Machine IR pass is followed by static branch relaxation (which unfolds far jumps into instruction sequences and compacts near jumps into single branches), which is the main reason why the hardening can't be delayed more; inserting extra code after relaxing would break short jumps.[6] Section 6.2.3 will explain why this is kind of a hack on LLVM's part.

  During object file generation, the 8-byte `CCS` Machine IR instruction is replaced with a 4-byte Xccs opcode and a placeholder zero-checksum. A custom relocation of type `R_RISCV_CHECKSUM` (marked by a comment in Figure 4.9c) is added to mark the checksum region for the linker.[7]

- **Linking**: the linker follows relocation entries to compute checksums and insert them in the provided spaces. The linker script is also updated so that hardened objects are linked to a different virtual address (`0x40000`) than non-hardened libraries and runtime files (`0x10000`), which tells hardware when to enable or disable Xccs protection.

4.2.5. Discussion

Fault's effect on CCS updates. The countermeasure relies on `CCS` updates not being vulnerable to attacks. This is a reasonable inference based on the fault model: recall that fetch skips are induced by clock glitches, which are known to cause problems locally along critical signal propagation paths. Alshaer et al. [Als+22] identify that such paths are mostly in the fetch stage of the pipeline, but `CCS` can be updated in the execution stage using the `realign` sum technique (described in Section 4.2.3), and is thus unlikely to be affected.

[6] I disabled linker relaxation for simplicity to maintain jumps' alignment, but it could be enabled after adding appropriate alignment relocations.

[7] The `R_RISCV_CALL` relocation for auipc/jump pairs is also replaced with a custom type to notify the linker of the newly-added ccs in the pair.

Interrupts. Common interrupts and signal handlers (that are invisible from the main thread) would not interfere with Xccs protections (with the only OS support needed being to save Xccs registers, which can be viewed as an extension of PC, to the CPU context structure). However, a non-returning interrupt (such as a signal exiting) would leave the current block without a check. I assume such a no-returning action implies abandoning the critical section where the interrupt occurred; otherwise, there might be a vulnerability.

Effect of faults on complex architectures. The fault model from Alshaer et al. [Als+22] does not describe hardware responses to fault attacks during speculative or out-of-order execution. The study and design of fault models at the micro-architectural level is already state-of-the-art, and applying it to these complex features is a completely open problem. While Xccs is amenable to speculative execution (mispredictions would not lead to false checksum exceptions because the checksum resets at the beginning of every block) and out-of-order execution (the checksum update is associative-commutative, allowing for reordering within each block) it remains unlikely that clock glitches would affect such complex designs in the same way as the simple processors from which fetch skips are derived.

Possibility of a hardware-only solution. Hardware-only fault countermeasures have their own challenges [CV17]. *Fault detectors* [Gom+14] create a performance trade-off between sensitivity and the rate of false alarms, and are limited to critical systems that accept the performance loss. Detecting corruption in the instruction stream requires extra hardware logic that risks being itself faulted. By contrast, a software/hardware proposition like Xccs minimizes exposure to the fault because the detection only relies on CCS updates (which occur in the execution stage, away from disrupted fetch logic) and a checksum check made after the fault's transient effect has subsided. (We carefully discussed the safety of these operations with authors of [Als+22].) It also incurs costs only in critical sections and is lightweight enough to be implemented in hardware late or during minor revisions.

Specificity of the attack model. Many (mostly early) works in fault literature attempt to protect against *all* program misbehaviors, described as "soft errors". By contrast, this targets a single vulnerability, which might appear overly specific. However, the fetch skips model results from extensive physical injection campaigns, where it described the impact of 80-90% of clock and voltage glitches on Cortex-M boards [Als23], making this countermeasure useful against common attack vectors on real boards. In addition, I argue that the lack of a precise definition for soft errors leads to tricky vulnerabilities[8] preventing any proof-based security standard from being met. Hence the focus fetch skips, for which I can prove security.

4.3. Security theorem

I'll now state the security theorems that provide the single-fault and multi-fault security properties for the countermeasure. The proofs are in Appendix A.

[8] For instance, triplication countermeasures such as SWIFT-R [CRA06] and NEMESIS [DSL17] tend to assume that a single "soft error" only affects one of the three execution streams, but this is not true of e.g. the kind of decoding errors mentioned in [DSL17]. I briefly outlined an attack in Section 2.3.1.

These theorems rely on definitions of legitimate *entry*, *executions* and *jumps*. Intuitively, a legitimate entry is a state that points to the beginning of a block while $\sigma.\mathsf{CCS} = 0$ (the correct value at the start of a block); a legitimate execution of a block is a series of steps that all take place within the block and end in a taken jump or exit; and a legitimate jump out of a block is a step that executes one of the block's branch instructions.

4.3.1. Security guarantee against multi-fault executions

The main theorem states that no matter what fetch skips are injected, the execution only proceeds if every intended checksum is passed before taking any control flow edge. The proof of this theorem is the appendix's Theorem 1.

Theorem (Security guarantee for multi-fault executions).
Let P a fully hardened program and $e = [s_0,\ ...,\ s_{|e|}]$ an execution such that

- *s_0 is a legitimate entry into a block of P;*
- *$s_{|e|}$ ends successfully, returning some $\mathsf{end}(\alpha)$.*

Then there exists a sequence $[\mathsf{hb}_1,\ ...,\ \mathsf{hb}_m]$ of blocks of P such that

1. *e can be partitioned into subsequences $(s_{t_i} \dots s_{b_i})_{1 \leq i \leq m}$ each a legitimate execution of hb_i ("t" means "top" and "b" means "bottom");*
2. *Each s_{b_i} $(i \neq m)$ is a legitimate jump of hb_i and $\sigma_{b_i}.\mathsf{CCS}$ is the correct checksum for that jump.*

A crucial aspect of this theorem is the *local security* at the block level; the security property of passing checksums is checked at every block, and deviations from that property are detected before the block ends. This facilitates both formal analysis (by removing difficulties associated with control flow) and testing (by providing an easy way to detect successful attacks).

This all means that the only way to defeat the countermeasure is to provoke a *checksum collision*, i.e. a modified execution path in a block whose checksum happens to be the same as the intended checksum.

Such collisions are rare; none was found while evaluating benchmarks in Section 4.4. Since blocks have on average less than 10 instructions it's statistically unlikely that collisions would appear on exploitable paths. In general, collisions can be ruled out entirely by checking all the combinations of attacks on individual blocks after linking, which is tractable unless there are huge blocks and the attacker is given unlimited targeted attacks. That said, if collisions do appear, they would be hard to solve consistently because the linker is unable to add or remove instructions, as the first could break short jumps and both would change jump offsets thus more checksums; this highlights further challenges with the toolchain's abstraction stack.

4.3.2. Security guarantee against single-fault executions

The problem of checksum collisions doesn't apply to single-fault executions when $N = 1$; an attacker that has access to only one S32(1) attack or one S&R32 attack will always invalidate the checksum. Intuitively, this is because the fetch skipped by S32(1) cannot have value 0 as no valid opcode can produce this pattern, and S&R32 cannot replace a fetch with the same value (that wouldn't constitute an attack). This is proven in the appendix's Theorem 2.

Theorem (Security guarantee for single-fault executions).
Let P a fully hardened program and $e = [s_0, \ ..., \ s_{|e|}]$ an execution such that
- *s_0 is a legitimate entry into a block of P;*
- *$s_{|e|}$ ends successfully, returning some $\mathsf{end}(\alpha)$.*

Let $[\mathsf{hb}_1, \ ..., \ \mathsf{hb}_m]$, $(s_{t_i} \ ... \ s_{b_i})_{1 \le i \le m}$ the partition of e into block executions given by Theorem 1. If each segment $s_{t_i} \ ... \ s_{b_i}$ uses at most one faulted fetch rule and the last segment $s_{t_m} \ ... \ s_{b_m}$ contains no fault, then e contains no faults.

I exclude the last block because the libc `exit()` function isn't protected in my setup, but this choice makes no significant difference on the proof.

The broad progression of the argument for both theorems is as follows:
1. Control cannot reach the end of a protected block due to the `c.ebreak`-based trap barrier;
2. Control can only jump out of a block after passing the associated checksum;
3. Using exactly one faulted fetch in a block always invalidates the checksum.

4.4. Implementation setting and evaluation

Next is the implementation and the experimental settings used to evaluate the countermeasure's functional correctness, security guarantees, and cost.

4.4.1. Implementation and experimental setting

I implemented the hardening scheme in LLVM [LA04] 12.0, linking programs with an off-the-shelf GNU C library using a modified GNU `ld` 2.40 linker. I prototyped the hardware extension Xccs in the system emulator QEMU 8.0 [Bel05] and the processor simulator Gem5 22.1 [Low+20]. The implementation is about 1150 lines in LLVM/`ld` for the countermeasure, and 850 lines in QEMU/Gem5 for evaluation.[9] The scheme is open-source and available online as a Git fork of LLVM:

→ URL: `https://gricad-gitlab.univ-grenoble-alpes.fr/michelse/fetch-skips-hardening`

I evaluate the countermeasure on programs from the MiBench benchmark suite [Gut+01], which is designed to be representative of embedded applications. I don't specifically use a cryptography- or security-oriented benchmark as the code is protected at a low enough level that the nature of the program is not significant. Programs were hardened with $N = 2$, i.e. assuming a strong attacker that can inject double-skips every fetch. There are three types of experiments, on a standard x86_64 GNU/Linux machine:

- Exhaustive single-fault injection campaigns in QEMU;
- Random multi-fault injection campaigns in QEMU;
- No-fault performance simulations in Gem5.

I instrumented QEMU to support fault injections by skipping or replacing data during the transpilation step. This takes advantage of the locally-correct design of the countermeasure and raises an explicit "countermeasure bypassed" exception at every successful exit of a block

[9] Gem5 only supported RV64 encodings at the time; I modified it to support RV32 for this evaluation.

where a fault was triggered. This ensures that successful fault attacks are reported and cannot be masked by program logic. I similarly extended Gem5 with the decoding and execution of Xccs instructions, but left out exceptional conditions since it was only used to evaluate performance without fault injections.

4.4.2. Functional correctness

The first assertion is that hardening preserves the functional behavior of programs when no fault is injected. This is checked by running hardened MiBench programs and comparing their output to a non-cross-compiled x86 build and a non-hardened RISC-V build. All programs pass this test.[10]

4.4.3. Security guarantee

I then subjected each program to two injection settings, both emulated in QEMU:

1. *Single-fault exhaustive injection*: attacks every PC in the protected text segment of the program with single-fault S32(1), S32(2), and S&R32.
2. *Multi-fault random injection*: attacks 2000 code intervals of 64 bytes with randomly-selected sequences of 2 to 6 faults in close succession.

This showcases the guarantees proven in Section 4.3. Outcomes of injections for which the targeted PC was reached are classified in four categories in Figure 4.10:

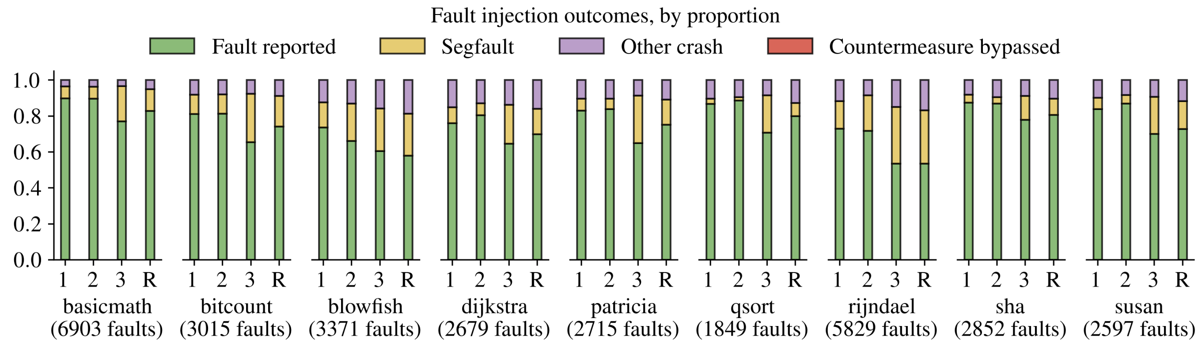
- **Fault reported**: An invalid checksum, illegal jump, or other Xccs-imposed constraint was violated; or `c.ebreak` was executed.
- **Segfault**: The program segfaulted from an incorrect memory access as a result of the injected fault.
- **Other crash**: Illegal instructions being decoded by the CPU (`SIGILL`) or other rare crashes such as `SIGBUS`.
- **Countermeasure bypassed**: The program successfully exited a block after attacking it. This did not happen in any of the tests.

Figure 4.10 shows that no attack was able to bypass the protection from the countermeasure, which in the multi-fault case means that no checksum collision occurred.

A significant majority of faults that are reached result in Xccs violations, meaning that failing checksums and invalid attempts to jump out of blocks are adequately reported. Because all exits of attacked blocks were guarded with “Countermeasure bypassed” exceptions, which were not reached, every crash we encountered also occurred within the attacked block. This shows that the countermeasure fulfills its goal of containing faults within blocks.

Interestingly, crashes alone do not provide sufficient security: unprotected programs (not depicted) experience ∼90% crashes but only ∼30% in the block where the fault is injected, meaning the window of exploitation is larger. So like with instruction skip, a system facing

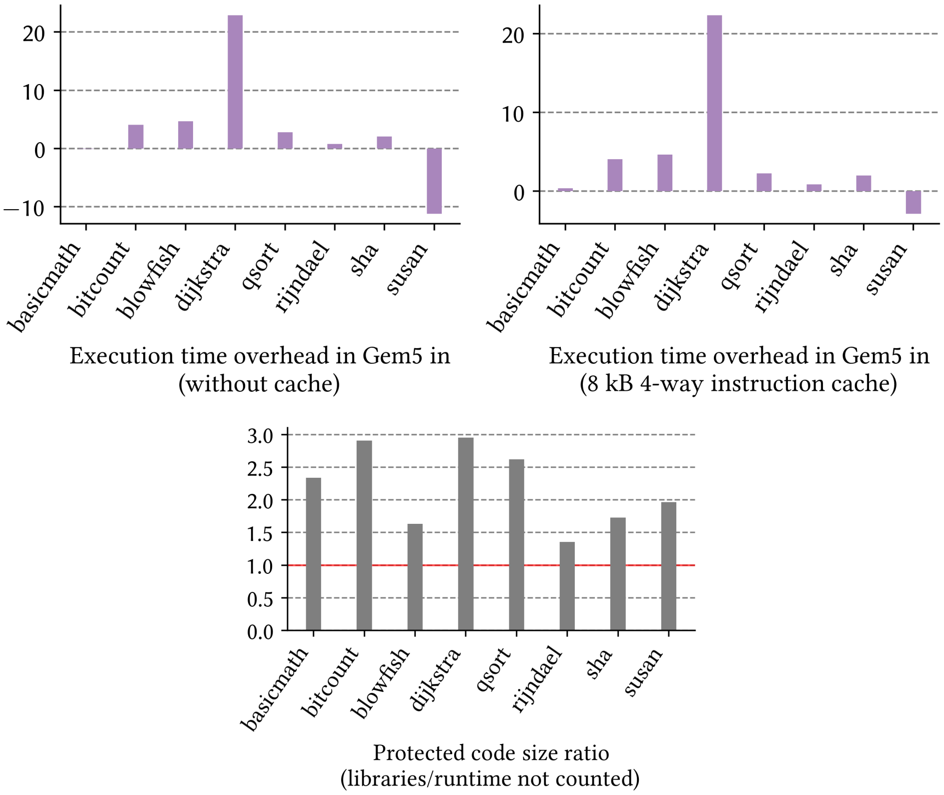
[10] I skipped the comparison with a native build for the few programs that were affected by mismatched precision in the RISC-V and x86 math libraries.



(1,2): Exhaustive injections of S32(1) *and* S32(2) *(3): Exhaustive injections of* S&R32

(R): Random multi-fault injections *Fault count: total for all categories*

Figure 4.10: Outcomes of exhaustive fetch skip attacks against countermeasure



Execution time overhead in Gem5 in (without cache)

Execution time overhead in Gem5 in (8 kB 4-way instruction cache)

Protected code size ratio (libraries/runtime not counted)

Figure 4.11: Evaluation of countermeasure code size and performance overhead

random failures could leave the code unprotected, but the threat of targeted attacks requires a countermeasure like Xccs.

4.4.4. Performance

On average, hardening with $N = 2$ increased the size of protected functions (i.e., excluding libraries and runtime files) by a factor of 2.46. This is a large increase, but typical for security applications especially against skips, due to duplication. Individual differences are given in Figure 4.11; unsurprisingly, programs with longer straight-line sections see less of an increase, while short, loop-intensive programs like `dijkstra` and `bitcount` get the largest overhead.

I ran Gem5 simulations for each program[11] to estimate the overhead in execution time, also reported in Figure 4.11. The simulated system mainly consists of a 1-GHz RISC-V core with a 1600 MHz DDR3 controller simulated by Gem5's *timing* model. This type of simulation doesn't have detailed timings for the CPU pipeline and execution but it accounts for bus, cache, and memory speed.

The main cost is the extra fetch-decode-execute cycles needed to run Xccs instructions. As expected, programs whose hot sections are control-heavy (like `dijkstra` and `bitcount`) have the largest overhead, while those with computation-heavy nests (cryptographic schemes) have the lowest. I expected the code size increase to reduce cache efficiency, and ran simulations with an 8-kB 4-way instruction cache (the median size of hardened sections; much smaller than the whole text segment), but the ratio of hardened to non-hardened speed remains consistent (with much faster overall executions). The mean performance hit is 10.2%, consistent with MiBench's 6–7 average instructions per block (recalling that blocks in this chapter's model are longer than traditional basic blocks).

Existing countermeasures without hardware support have a comparatively much larger overhead. Geier et al. [Gei+23] compare multiple combinations of countermeasures against single- to quadruple-instruction skips on a secure boot program. Only 3 combinations (nZDC + RACFED, NEMESIS + RACFED, and CompaSec) close more than 75% of vulnerabilities, all with space overhead above x4.85 and time overhead above x5.01.

4.5. Chapter conclusion

In this chapter I've presented a new countermeasure to a micro-architectural fault model targeting the instruction fetch unit in RISC-V processors. The co-designed approach combines an ISA extension with a compile- and link-time hardening step, and reaches an interesting compromise with good time performance (about 10.2% overhead) while completely countering the attack (guaranteed on single faults, and empirically on multi-faults).

Most importantly, I showed that it was possible to incorporate low-level details in a semantics of assembler to link program code to its direct effects on hardware behavior. This enables reasoning on the security of assembly code even if the attack model is lower, and in the case of this countermeasure, prove the security property. I believe this was the first case of a formally proven (partly-)software countermeasure against a low-level attack model.

I implemented this particular countermeasure in the late back-end. This is is pretty standard for low-level attacks, and keeps the abstraction gap between the attack and countermeasure fairly small. However, trouble with linker relocations, LLVM's branch relaxation pass, and the slight distance between Machine IR and assembly due to pseudo-instructions made this integration not trivial. It is clear from this example that the standard toolchain pipeline won't always accommodate security transformations.

[11] Except `patricia`, due to a 32- vs. 64-bit compatibility issue related to reused opcodes in the RISC-V ISA.

→ Can software still contribute significantly to defeating a micro-architectural fault model?

Absolutely. Even if the attack originates in hardware and its model is studied at hardware level, the end goal is to guarantee a security property for the entire system, which may very well be focused on software logic (e.g. hardware can break as long as the attacker doesn't gain unauthorized entry). Even if software can't ensure the security property alone, co-design can reduce the cost on both sides of the ISA, as was the case here.

→ How to prove an assembly program secure against a lower attack model?

The answer in this chapter would be "model hardware details in the program's semantics". The obvious caveat is that this only works if the relevant hardware behaviors are primarily a function of software. It wouldn't be possible to analyze hardware logic not directed by code (like predictor states) just by locally matching and deconstructing code.

5 Writing countermeasures with Tracing LLVM

Hardening against fetch skips demonstrated primarily low-level formalization, and the implementation ended up emblematic of the limitations of software countermeasures. Like many other low-level countermeasures, it takes place entirely in the back-end, and takes no input from source code; thus, it is unable to differentiate critical or non-critical code, which would be helpful for saving space. It also struggles a bit with instruction encodings, having to lower checksum regions from assembly to the binary level so the linker can compute the checksum with a relocation.

In Chapter 3, I argued that first-class compiler support for security countermeasures would enable such end-to-end countermeasures. My proposition revolves around the concept of *tracing* elements of the source program down the abstraction stack, while constraining compilation to ensure that the required correspondence between low-level and high-level program elements does exist.

I've implemented some of these ideas in an open-source LLVM extension called *Tracing LLVM*. The remaining chapters in this thesis describe different aspects of Tracing LLVM. This chapter showcases its features from the point of view of a countermeasure developer (from Figure 3.1), to illustrate the concept of tracing. Chapter 6 focuses on its implementation and internal details, while Chapter 7 addresses a few elements about validation.

Section 5.1 introduces the project and gives an overview of its features, which are then detailed in Section 5.2. Solutions to the examples from Chapter 3 are then provided in Section 5.3. Finally, I build up to a complex PIN verification examples with multiple layered countermeasures in Section 5.4.

5.1. Tracing LLVM

5.1.1. Project description

I implemented many (though not all, sadly!) of the ideas in this thesis in an extension of LLVM 17 called *Tracing LLVM*, which is open-sourced under the same Apache 2.0 license as LLVM. The distribution includes the compiler, documentation, and a basic test system.

→ URL: `https://gricad-gitlab.univ-grenoble-alpes.fr/tracing-llvm`
→ DOC: `https://gricad-gitlab.univ-grenoble-alpes.fr/tracing-llvm/doc`

The project's goal is to realize the distribution of responsibilities illustrated in Figure 3.1; that is, to provide security tooling for LLVM that can be reused across multiple countermeasures and may inform research on the design of secure software for vulnerable hardware systems. Maintainability is a concern and guided many choices for the implementation; I will discuss this along with implementation techniques in Chapter 6.

Tracing LLVM contains features imported from Son Tuan Vu's *property-preserving LLVM*; specifically, a side-effecting interpretation of its observation primitives [Vu21, 5] (ported from LLVM 12 to LLVM 17). The original thesis proposes pure primitives for observation and opacification, but these are vulnerable to some theoretical abuses that side-effects help limit; this is discussed in much greater detail in Section 7.3.

I received authorization to distribute property-preserving LLVM under the Apache License 2.0 used by upstream LLVM. The code is publicly available at the repository below, with the original code on the "vu" branch. Property-preserving LLVM doesn't have an official documentation, but I summarized its implementation in the Tracing LLVM docs by analyzing a diff with the latest upstream merge.

→ URL: `https://gricad-gitlab.univ-grenoble-alpes.fr/michelse/llvm-property-preserving`
→ DOC: Tracing LLVM documentation at `misc/llvmpp.md`

5.1.2. Overview of the main features

Tracing LLVM's features broadly fulfill three purposes:

- **Opacification**: The most significant semantically, opacification features hide details of the program such as data or instructions from either language or compiler. They are leveraged to extend preservation guarantees during optimizations.
- **Tracing**: The features transport information from high-level representations to lower levels during lowerings, and inhibit optimizations that would prevent this connection from existing.
- **Lowerings control**: These features inject custom lowering logic in the compiler, such as in instruction selection or register allocation.

Table 5.1 lists the features I implemented in each of these categories[1], which are explained below in Section 5.2.

Category	Feature name	Section	Level	Brief description
Opacification	Vu's `opacify` [Vu21]	5.2.1	C → Machine IR	Hides values
	Wrapper instructions	5.2.2	LLVM IR, Machine IR	Hides instructions
	Traced types	5.2.3	C, LLVM IR	Hides type representations
Tracing	Tracing variables	5.2.4	C → Machine IR	Tracks reads and/or writes
	Tracing dataflow	5.2.5	C → Machine IR	Tracks downstream dataflow
Lowerings	Wrapper lowering	5.2.6	Machine IR	Custom instruction selection
	Reg. alloc. for tracing	5.2.7	Machine IR	Multi-pass register allocation

Table 5.1: Summary of Tracing LLVM's features

[1] Vu's base code for generating runtime-checkable debug information has also been ported, but I still lack a debugger plugin and some validation for it.

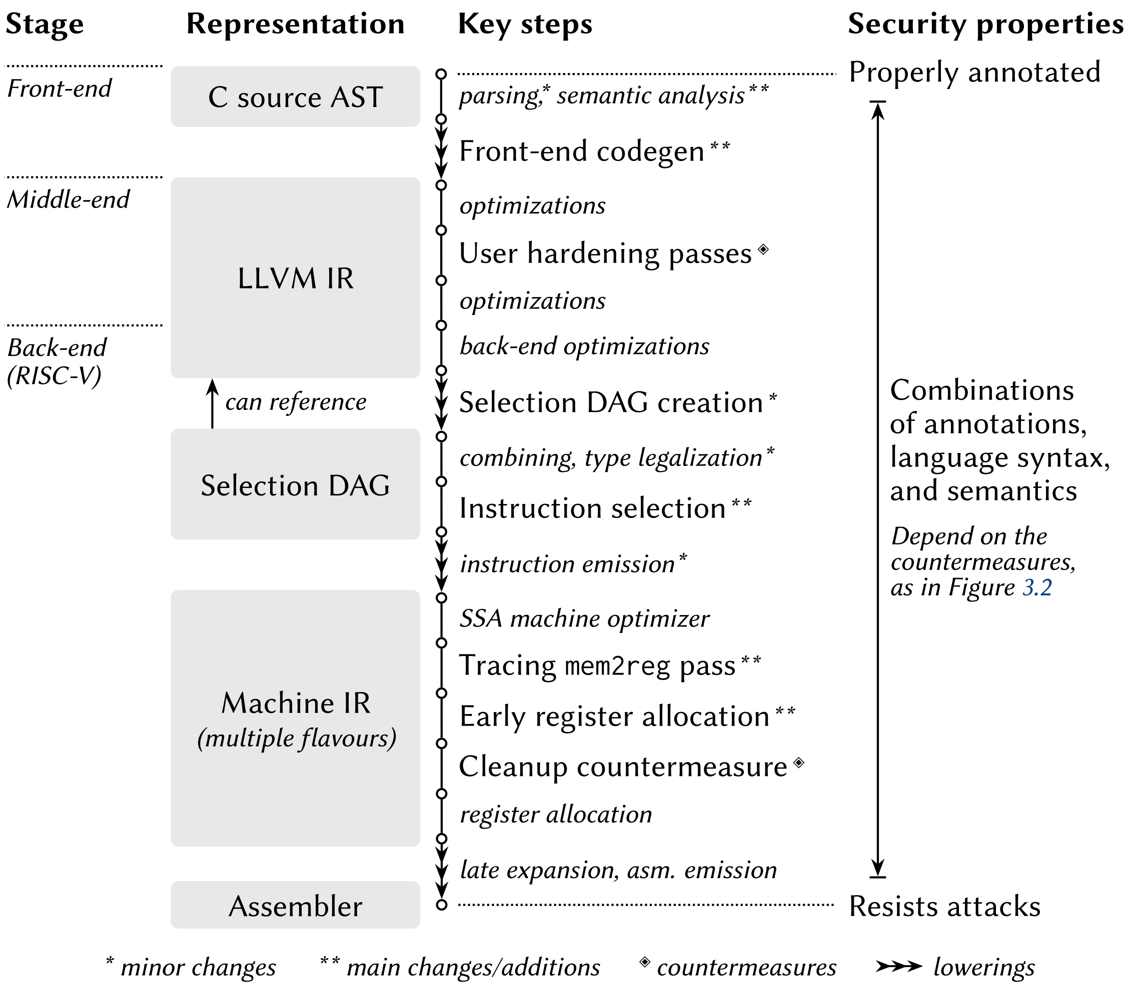


Figure 5.2: Overview of LLVM's build process and the main additions in Tracing LLVM

Figure 5.2 visualizes the key steps of compiling a C program to RISC-V assembler with LLVM, highlighting the changes and additions made by Tracing LLVM. It illustrates the abstraction level targeted by each of the features in Table 5.1 and the order in which they are involved.

As a first approximation, the build goes through a series of program representations, each time undergoing an abstraction lowering. The back-end is more detailed here than on Figure 1.3, revealing that back-end passes access the LLVM IR code in addition to two intermediate representations, the instruction selection DAG and the varying flavours of Machine IR[2]. The security properties on the right side are the same as in Figure 3.2, using a mix of structural and semantic properties that reflect the progressive hardening of the program rather than the attack model itself.

5.2. Feature descriptions

In this section, I want to briefly describe the features listed in Table 5.1, show their syntax and explain their semantics, to serve as a reference. Their use for solving security problems will be

[2] Modern LLVM back-end use the so-called "Global Instruction Selection" (GlobalISel) method, skipping the selection DAG in favor of selecting directly within Machine IR.

shown in the next section, whereas the research questions behind their implementation will be addressed by Chapter 6.

5.2.1. Opacification function

The opacification function is a built-in function available at all levels of abstraction, first introduced by Vu [Vu21]. It takes an argument of a given type T and returns another value of that same type T; it may also take more arguments variadically.

```
/* In C */
T __builtin_tracing_opaqueio(T arg, ...); /* for each type T */
int y = __builtin_tracing_opaqueio(x);
```

```
; In LLVM IR
declare T @llvm.tracing_opaqueio.T(T, ...) ; for each type T
%y = call i32 (i32, ...) @llvm.tracing_opaqueio.i32(i32 %x), !observation !8
```

```
# In Machine IR
%2:gpr = TRACING_OPAQUEIO !8, %1:gpr(tied-def 0), !DIExpression()
```

```
# In assembler (no code, just a comment)
# $a0 = TRACING_OPAQUEIO(observation: "x", $a0(tied-def 0), !DIExpression())
```

The opacification function is *implementation-defined*. In C, this means it has *"unspecified behavior where each implementation documents how the choice is made"* [C23, 3.5.1§1], using here the maximalist interpretation where the *implementation* consists of all the lower stages in the compiler, including the countermeasures and their parameters. The compiler's semantic preservation target implies that it must preserve the program's behavior *for all possible implementations* of any implementation-defined feature. This notion comes up frequently in C due to its wide range of target systems[3]. In practice, this is specified "by default" to the compiler by simply not providing any determined semantics.

In the case of the opacification function, we always substitute it in the back-end with the function that returns the first argument x (i.e. we only use one implementation). Essentially, we're telling the compiler that we're computing a mystery function whose results it semantically can't predict, and then at the last second reveal the ruse and substitute opacify(x) with x at no cost in machine code, leaving behind just a comment (and debug information).

The opacification function hides *values* by replacing an existing program value with an opaque one. However, because it's the identity function, it can't be used to compute anything.

My implementation of the opacification function carries a side-effect, unlike Vu's original which is pure. This design choice is conservative; having a side-effect forces stronger preservation guarantees from the compiler but also inhibits more optimizations. I'll discuss the fine implications of this difference in Section 7.3.

[3] Including extravagant assumptions by today's standards such as bytes not always having 8 bits [C23, J.3.5§1].

5.2.2. Wrapper instructions

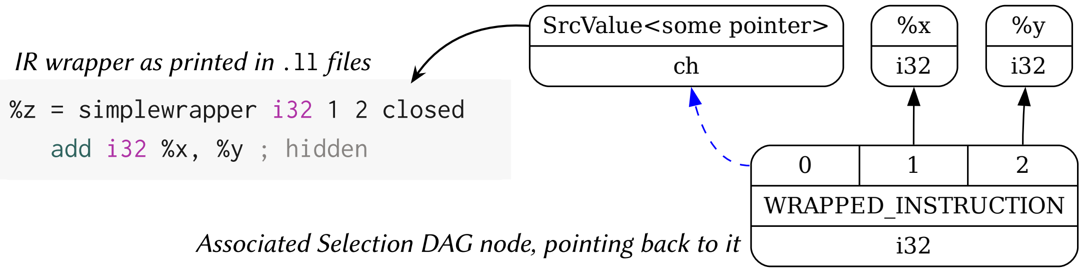
Wrappers are another kind of opacification that I implemented in LLVM IR and the instruction selector. These are opaque instructions (again, implementation-defined) whose semantics are secretly defined by a hidden (non-branching) instruction called the *linked* instruction. Essentially, a wrapper wraps the linked instruction and hides its behavior.

Wrappers follow the idea of hiding instructions by redefining them. For instance, one could opacify an add by replacing it with a new instruction opaque_add that takes the same operands but doesn't have any semantics other than its lowering. Wrappers achieve this but without the hassle of duplicating definitions, flags, type support, etc. for each instruction of interest. For example, the following `simplewrapper` is an opaque 32-bit integer addition:

```
%z = simplewrapper i32 1 2 closed   ; 1 hidden instruction, 2 arguments
   add i32 %x, %y                   ; hidden to LLVM
```

These two lines are the LLVM IR notation for a single `simplewrapper` instruction with one linked instruction (a hidden add) and two arguments (`%x`, `%y`). The notation for the wrapper puts the arguments after add for readability, but internally the arguments are on the `simplewrapper` to ensure that the add instruction is not reachable through SSA's use-def chains for `%x` and `%y` (which link values to all of their uses). More on the representation (including the role of the "`closed`" keyword) in Section 6.3.2.

In general, wrappers are used to wrap a single instruction. The system allows for hiding an entire block of sequential code, but because wrappers only expose the output of the final instruction, no storage would be allocated for any intermediate values. This shifts a lot more work on the countermeasure developer in terms of ensuring proper storage is available to lower all the instructions. I do use this mechanism to wrap no-op traced type casts, which I'll describe in Section 5.2.3.


IR wrapper as printed in .ll files
SrcValue<some pointer>
ch
%x
i32
%y
i32
0
1
2
WRAPPED_INSTRUCTION
i32
Associated Selection DAG node, pointing back to it

```
%z = simplewrapper i32 1 2 closed
   add i32 %x, %y ; hidden
```

Figure 5.3: Wrapper notation and their Selection DAG lowering.

Wrappers also exist in the Selection DAG, represented by a `WRAPPED_INSTRUCTION` node with the non-lowered IR source as an operand (Figure 5.3). This relies on an LLVM feature where the Selection DAG can refer to fragments of IR[4]. Using it for instructions avoids complex changes to the DAG structure for lowering wrappers, at the cost of having to select them manually. The "ch" operand is a token that chains side-effecting nodes in the right order.

[4] As far as I understand this is intended for complex values like structure initializers.

Currently, wrappers disappear during instruction selection, so they are mostly used to opacify against the main LLVM IR optimization pipeline in the middle-end.

Unlike the opacification function, which opacifies values, wrappers opacify *instructions*, which makes them useful for forcing computations to occur. For instance, value opacification cannot force an addition instruction to execute on all program traces. This is because the compiler can always add dynamic checks for special values regardless of how the operands were provided:

```
z = x + y;
// can always be rewritten into
z = (x == 0) ? y : x + y;
```

A wrapper, on the other hand, preserves the instruction itself and can force the computation to execute exactly as written.

5.2.3. Traced types

The last representation feature in Tracing LLVM is a data-flow abstraction called *traced types*. Traced types hide the domain and representation of existing types[5] and may capture data-flow by propagating within expressions. Like the opacification function, they are always implemented as the identity constructor, so the runtime type doesn't actually change.

Traced types appear in different forms until late Machine IR:

- In C, a type constructor written "!" with controllable attributes. In most cases that I'll explore in this thesis, traced type attributes behave like qualifiers (e.g. const or volatile).

  ```
  int !x;                         /* no attributes */
  int ! __attribute__((...)) x;   /* with attributes */
  ```

- In LLVM IR, also a type constructor written "traced()";[6]
- In back-end representations, where there are no type constructors, I provide traced counterparts tri1, tri8, etc. to integral *machine value types* i1, i8, etc. equipped with the same promotion logic.

Traced types feature most prominently in the front-end, where they influence typing and code generation based on what attributes are given. See the tracing features below (variables and dataflow) for some examples.

Using types provides strong guarantees in C and LLVM IR, as they have a lot of control over semantics and are only rarely bypassed[7]. Types don't work as well in the back-end because *machine value types* are a flat enumeration without constructors (requiring some duplication) and transformations are less type-directed. Once we get to register allocation, a "traced general-purpose" register class would be more effective (but hasn't been implemented yet).

[5] Except their storage size, which is too instrumental to be hidden completely.

[6] Attributes on IR traced types are not implemented yet but are planned in the design.

[7] The main case that comes to mind is untyped LLVM IR pointers and bitcasts.

Working with traced types in LLVM IR requires casts, which are provided through straightforward new trace and untrace instructions[8] (also no-op at runtime).

```
%t = trace i32 42       ; %t has type traced(i32)
%u = untrace i32 %t     ; %u has type i32
```

Naturally, hiding the domain and representation of types is not nearly as useful if the raw values are exposed in code because of untrace instructions. The intention is to expose raw values as intermediates in a wrapper, which keeps them hidden as well. For instance, the wrapper in Listing 5.4 adds two traced(i32):

```
; %x, %y: traced(i32)
%z = typewrapper traced(i32) 4 2 closed     ; 4 hidden instructions, 2 arguments
  %z_w0 = untrace i32 %x
  %z_w1 = untrace i32 %y
  %z_w2 = add i32 %z_w0, %z_w1
  %z_w3 = trace i32 %z_w2                   ; wrapper's return value (implicitly)
```

Listing 5.4: Multi-wrapper with hidden traced type casts

The mechanism is identical to a normal wrapper, with two differences. First, the value produced by the wrapper is the output of the last instruction, with all other intermediate values hidden. Second, the wrapper instruction is identified as a typewrapper (not a plain simplewrapper), which is a hint to the instruction selector about the structure of the linked instructions (some untrace, a computation, and a trace) to automate part of the selection process.

5.2.4. Tracing variables

Let's now look at the *tracing* features, which create and maintain a connection between the source code and intermediate programs. The first half of this system (which is implemented) consists in opacifying targeted elements of the source program to prevent LLVM from rewriting the program into a form that doesn't "match" the source (for various relations of interest). The second half of this system (which is not implemented at time of writing) would consist in attaching metadata to protected program elements so that hardening passes can *query* for traced elements, like "iterate on all traced variables in the function".

The first kind of tracing is *tracing variables*, which preserves the action of accessing variables, specifically writing. This feature is enabled by defining the targeted variable as a traced type with the trace(writes) attribute:

```
int ! __attribute__((trace(writes))) n;
n = 42; /* this store is traced */
```

The modified type triggers a special code generation rule for the assignment, which creates a store instruction in a wrapper. This ensures that the act of writing to n is recorded at a specific control point and the value stored must be materialized at that time. This also prevents n

[8] Intrinsics could do in principle, but putting them in wrappers would move the parameter designating the intrinsic function to the wrapper, and these are only allowed on calls.

from being promoted to a register, which we delay until the register allocation step for traced values (see the next sections).

```
%n = alloca i32, align 4
simplewrapper void 1 2 closed
  store i32 42, ptr %n, align 4
```

Tracing variables only makes a difference when the variable is used as an lvalue; in all other contexts `n` will behave as a normal integer, as the `trace(writes)` attribute decays during lvalue-to-rvalue conversion.

```
int p = n + 31; /* fine, p = 73 */
```

The opacified assignments are kept as side-effects until register allocation, at which point traced variables are promoted to registers and compilation continues as normal. In the current state, this leaves the writes exposed to a few back-end passes (in LLVM's RISC-V back-end, the only threat I identified is the copy propagation pass, which could alter the order originally set by the store side-effects).

Note that writes being side-effecting implies that any writes that were conditional in the source code *must* remain conditional in the target code. This can be used to indirectly preserve control flow (with limited guarantees).

5.2.5. Tracing dataflow

The other kind of tracing currently available in Tracing LLVM is *tracing data-flow*, which is similar to propagating and preserving a taint in *taint analyses* [Xia+25]. This feature is enabled by manipulating any expression whose type is a traced type with the `trace(dataflow)` attribute.

```
int ! __attribute__((trace(dataflow))) n;
n + 2; /* this addition and the resulting expression are traced */
```

`trace(dataflow)` is very different from `trace(writes)`. It behaves like a qualifier and propagates through operations; for instance, if either `n` or `p` has a dataflow-traced type, then `n+p` also has that dataflow-traced type and it lowers to an add in a wrapper.

```
%n = alloca i32, align 4
simplewrapper void 1 2 closed
  add i32 %n, i32 2
```

Dataflow tracing is useful to track sensitive/secret data and control how this data is manipulated. This is illustrated in Sections 5.3.5 and 5.3.6 for cleaning up registers having held sensitive data and separating sensitive and non-sensitive registers in a function.

Casting through traced dataflow types is not allowed (as that could be error-prone); instead, explicit built-ins are provided:

- `__builtin_tracing_trace()` adds `trace(dataflow)` on a pure expression, thus tracing its result and further uses;
- `__builtin_tracing_untrace()` removes `trace(dataflow)`; this can result in incomplete coverage and should be used cautiously.

Dataflow tracing is currently based entirely on traced types, and thus it stops after instruction selection. The mechanism could be extended until register allocation by tracking with register classes.

5.2.6. Wrapper lowering

Wrappers provide an opportunity to customize the instruction selector. This could be used in schemes where custom non-trivial implementations of operations are required, but the replacement cannot be performed on assembly directly due to a need for extra storage; for instance, masking.

This possibility isn't exploited much currently; rather the opposite. Section 5.2.2 mentions that linked instructions in wrappers are not lowered to the selection DAG but instead kept in their original IR form. This shuts down any option of letting LLVM lower the instruction automatically before putting it back in a (hypothetical) Machine IR wrapper. Instead, all linked instructions must currently be lowered manually, which is inconvenient. This friction could be solved by implementing a Selection DAG wrapper that doesn't keep the original instruction in IR form.

5.2.7. Register allocation for tracing

Tracing LLVM extends the register allocation process by preceding it with:

1. A late register-promotion pass (similar to the LLVM IR `mem2reg` pass, but much later) which promotes traced stack locations into registers. These are allocations that `mem2reg` can't promote because wrappers (purposefully) obscure the accesses.
2. A pre-allocation pass for allocating registers to values of traced types, such as sensitive values marked by `trace(dataflow)`.

The early register allocation can be used for a few different things. Its basic purpose is to provide special treatment to traced values, which can be done either by running an actual allocation pass or by setting up hints for the normal allocator.

Typically, one might change the allocation order to invite the allocator to place secret data in other registers than non-secret data. This can be enforced with a dual-allocator setup by reserving the registers used to store secret data before doing the second allocation.

Although fiddly, the pre-allocator can match which registers to allocate based on type instead of a register class, which I believe makes it easier to carry around metadata (similar to the traced type attributes in C).

Another option, not currently implemented, is to run the early register allocation *ahead* of the Machine IR SSA optimization pipeline (which normally takes place between instruction selection and register allocation) to protect sensitive values against late optimizations.

5.3. Examples and use cases

We can now come back to the use cases of Section 3.4, all of which can be accomplished to some degree with Tracing LLVM.

Note that the interface to Tracing LLVM is not stable, so user-facing options with clear names are not yet available for controlling every feature individually. As a result, some of the examples below are compiled with hardcoded passes, and the same top-level attributes may be used in multiple examples for different purposes. The focus of this section is to demonstrate the mechanisms rather than their interface (which I intend to refine in the future).

5.3.1. Strict variable accesses

In this example from Section 3.4.1, we want to obfuscate accesses to a CFI signature variable CoT to prevent the compiler from optimizing away CFI checks. Making the variable volatile accomplishes this objective but causes it to be stored in memory, which is inefficient.

Figure 5.5 shows a different approach by tracing the writes to CoT. This grants similar protection to volatility but Tracing LLVM still attempts to promote the variable to a register late in the back-end (just before register allocation). The code is identical to Figure 3.4 except for the type of CoT, which adds the annotation for tracing writes.

```
/* Signature values for each section. C01 is the transition C0 -> C1 */
enum { C0 = 0xb7, C1 = 0x2a, C01 = C0 ^ C1 };

void get_key_2(int key_size) {
  int ! __attribute__((trace(writes))) CoT;
  CoT = C0;

  CoT = CoT ^ key_size;
  switch(key_size) {
  case 128: get_key128(); CoT = CoT ^ (C01 ^ 128); break;
  case 256: get_key256(); CoT = CoT ^ (C01 ^ 256); break;
  default: abort();
  }
  if(CoT != C1) abort();
}
```

Figure 5.5: Strict variable accesses: hardened code by tracing writes

The protection takes a different form at each abstraction level:

- In clang, the type annotation triggers a particular code generation path;
- In LLVM IR, the write is protected by a wrapper;
- The wrapper is dropped during instruction selection, leaving a plain write;
- Tracing LLVM's late register promotion pass promotes the variable to a register just before register allocation, after all threatening optimizations (even at -O3).

Figure 5.6 shows the resulting assembler code, which holds CoT in the register s0. This saves 5 stack access instructions compared to the initial attempt from Figure 3.5.

```
get_key_2:
  addi    sp, sp, -16
  sw      ra, 12(sp)
  sw      s0, 8(sp)
  # CoT is never stored to memory
  li      s0, 0xb7
  # CoT ^= key_size
  xor     s0, s0, a0
  # Dispatch key size
  li      a1, 256
  beq     a0, a1, .case256
  li      a1, 128
  bne     a0, a1, .abort

.case128:
  call    get_key128
  xori    s0, s0, 29  # C0^C1^128
  j       .end

.case256:
  call    get_key256
  xori    s0, s0, 413 # C0^C1^256

.end:
  li      a0, 0x2a
  bne     s0, a0, .abort
  lw      ra, 12(sp)
  lw      s0, 8(sp)
  addi    sp, sp, 16
  ret
.abort:
  call    abort
```

Figure 5.6: Strict variable accesses: hardened assembly after tracing writes

Note that both solutions here make the simplifying assumption that preserving the accesses will lead to the compiler keeping the checks the way they are. This works in practice but there are theoretical subtleties, some of which are discussed in Section 7.3.

5.3.2. Sequencing at variable writes

This example from Section 3.4.2 requires control-flow checks to be properly interleaved with surrounding pure code. Vu already provides two solutions for this use case:

1. The "old" solution [Vu21, 4.4.3.3] places *observations* to synchronize the values of relevant variables before SCI checks. The observation is a side-effect and it reads all active variables, which forces them to be up-to-date when the observation is reached.

   ```
   if(userPIN[0] != secretPin[0]) valid = false;
   obs_pt: __attribute__((annotate("observe(SCI, valid)")));
   if(++SCI != 2) abort();
   ```

   Register promotion is still possible because the observation takes these variables as inputs and produces new so-called "artificial definitions" without taking their addresses.

2. The "new" solution [Vu21, 5.4.1.2] uses a data dependency chain with only pure functions. In the excerpt below, the opacification function always returns its first parameter, and the purpose of the second (unused) parameter is to add an extra data dependency.

   ```
   if(userPIN[0] != secretPin[0]) valid = __builtin_opacify(false, SCI);
   SCI = __builtin_opacify(SCI, valid) + 1;
   if(SCI != 2) abort();
   ```

   This strongly invites the compiler to evaluate in the desired order.

A third, comparable approach available in Tracing LLVM is to make the writes to relevant variables side-effects themselves by tracing them, as shown in Figure 5.7. This guarantees that the writes happen in the desired order, leading to correctly-interleaved assembler code shown in Figure 5.8.

```c
#define SCI_CHECK(_n) { SCI = SCI + 1; ASSERT(SCI == _n); }

int verify_pin_2(uint8_t *userPIN, uint8_t *secretPIN) {
  int ! __attribute__((trace(writes))) valid;
  int ! __attribute__((trace(writes))) SCI;
  valid = true;
  SCI = 0;

  SCI_CHECK(1); if(userPIN[0] != secretPIN[0]) valid = false;
  SCI_CHECK(2); if(userPIN[1] != secretPIN[1]) valid = false;
  SCI_CHECK(3); if(userPIN[2] != secretPIN[2]) valid = false;
  SCI_CHECK(4); if(userPIN[3] != secretPIN[3]) valid = false;
  return valid;
}
```

Figure 5.7: Sequencing at variable writes: hardened code by tracing writes

```asm
verify_pin_2:
  addi    sp, sp, -16
  sw      ra, 12(sp)
  li      a4, 1
  li      a3, 0
  li      a3, 1

.check0:
  mv      a5, a3
  bne     a3, a4, .abort
  mv      a7, a4
  lbu     a4, 0(a0)
  lbu     a2, 0(a1)
  li      a6, 2
  li      a5, 2
  beq     a4, a2, .check1
  li      a7, 0
  addi    a5, a3, 1
.check1: # (...)
# same digit check block 3 more times
# with other load offsets/SCI values
.end:
  mv      a0, a7
  lw      ra, 12(sp)
  addi    sp, sp, 16
  ret
.abort:
  call    abort
```

Figure 5.8: Sequencing at variable writes: hardened assembly with proper ordering

This is similar to Vu's "old" solution[9]. I'm not lingering on the subtle differences because ultimately none of the options truly *force* the expected ordering; they all leave the actual computations as pure, and pure computations cannot be forced to evaluate in a specific order. The opacifying version (which uses a pure opacification function) has the most theoretical angles of attack, which I'll explore in some detail in Section 7.3.

[9] Vu's solution is slightly more flexible because it doesn't enforce an update order between the variables involved within each step, whereas mine requires the source order to be followed.

5.3.3. Avoid optimizations during lowerings

Next up, we had two examples in Section 3.4.3 demonstrating optimizations hidden within lowerings that couldn't be disabled: propagation of constant literals when lowering the clang AST to LLVM IR, and merging of common subexpressions in the instruction selection DAG.

To bypass constant propagation (both the lowering one and the actual data-flow pass that comes later), one can simply opacify both halves of the constant, as shown in Figure 5.9 (compare to original: Figure 3.8).

```
#define CONSTANT 0x00c0ffee
#define MASK 0xa5a5a5a5

uint32_t get_constant_2(void) {
  return __builtin_tracing_opaqueio(CONSTANT & MASK) +
         __builtin_tracing_opaqueio(CONSTANT & ~MASK);
}
```

Figure 5.9: Avoid optimizations during lowerings: hardened code by opacifying constants

In this case, using a side-effecting opacification function is suboptimal. If the constant appears multiple times in a single function (either by inlining `get_constant_2` or defining the macro `CONSTANT` as the whole expression), merging them would be desirable.

To work around the merging of identical nodes in the Selection DAG, one needs to introduce either differences between the copies, or side-effects. There are many ways to accomplish this, one of which is shown in Figure 5.10 (compare to original: Figure 3.10).

```
define i32 @f(i32 %x, i32 %y) #0 {
  %x1 = call i32 (i32, ...) @llvm.tracing_opaqueio.i32(i32 %x)
  %z1 = add i32 %x1, %y
  %y2 = call i32 (i32, ...) @llvm.tracing_opaqueio.i32(i32 %y)
  %z2 = add i32 %x, %y2
  %eq = icmp eq i32 %z1, %z2
  br i1 %eq, label %bb.ok, label %bb.err

bb.ok:
  ret i32 %z1
bb.err:
  call void () @abort()
  unreachable
}
attributes #0 = { noinline optnone }
```

Figure 5.10: Avoid optimizations during lowerings: hardened IR by opacifying operands

Ideally, I would hide the addition in wrappers to protect the operation itself. However, in their current form wrappers unfold during instruction selection, which still allows the DAG data structure to merge the unfolded nodes.[10]

A symmetric application of pure opacification wouldn't work for this use case as it would introduce neither variations nor side-effects, and thus would still be optimized by CSE. The opacification must at least be non-deterministic.

5.3.4. Map source variables to registers

This next example is from Section 3.4.4; the goal was to get the CFI signature variable `CoT` mapped to a consistent register. As discussed previously, this can't go through the standard promotion-to-register optimization (`mem2reg`), because it takes place on LLVM IR, which uses the SSA form, and thus breaks up the variable's live ranges into separate values (after the register allocator may do anything).

Tracing LLVM's solution to this issue is to keep the loads and stores to `CoT` intact during the IR stage and promote to a register before register allocation, just after the Machine IR representation exits its own SSA form.

Because not all features have clean user-facing interfaces yet, the hardened code for this example is identical to Figure 5.5. The feature is triggered by tracing the writes to `CoT` and then the late promotion pass is hardcoded and picks up on it.

With this protection, accesses are handled as follows at each abstraction level:

- In the middle-end (LLVM IR), the writes are side-effects and `CoT` stays on the stack;
- Same during instruction selection;
- After instruction selection, the program (now in Machine IR) undergoes SSA-based optimizations, while the variable is still on the stack;
- Finally, just before register allocation, the variable is promoted to a single virtual register for the entire function.[11]

The final assembly code is identical to Figure 5.6, with `CoT` being allocated to `s0` for the entire function.[12]

5.3.5. Cleanup sensitive registers

The example in Section 3.4.5 is concerned with cleaning up all the sensitive data from a function's local registers when it exits. Doing it at the source level like the attempt from Figure 3.13 fails for many reasons, not least of which is that `T1i = 0`, even if kept, only defines a new live range and may not erase previous values of `T1i`.

A traditional approach to solve this problem would be to discover sensitive variables in a back-end data-flow analysis pass. Tracing LLVM instead invites annotating the sensitive

[10] In fact, this is also the reason why the solution opacifies `x` in the first addition and `y` in the other, as the current expansion of the `opaqueio` intrinsic is a mergeable DAG node, which is likely a bug.

[11] This doesn't technically guarantee that it will be allocated a to a single physical register; individual allocators can differ. These variations can be bypassed with aggressive hints or an early allocation pass.

[12] Getting the same output is mostly a coincidence. The first example would still work if the live ranges were allocated separately; I just reused the same register promotion pass for simplicity.

data through a suitable type, essentially maintaining this data-flow information during each compilation stage, as shown in Figure 5.11.

```
int compare_arrays_2(int ! __attribute__((trace(dataflow))) *T1, int *T2, unsigned N) {
  int equal = true;
  for(unsigned i = 0; i < N; i++) {
    if(__builtin_tracing_untrace(T1[i] + 1) != T2[i]) {
      equal = false;
      IO(equal); // Force a real conditional.
    }
  }
  return equal;
}
```

Figure 5.11: Cleanup sensitive registers: Hardened source with data-flow tracing

The key to this solution is the `trace(dataflow)` attribute on `T1`'s pointed type, indicating that all data loaded from this pointer should be traced through the type taint until the back-end. After that, a hardcoded back-end pass following register allocation resets all registers used to store traced values at the function exit, yielding the code in Figure 5.12 where the registers used to load sensitive data (a4) and de-obfuscate it (a5) are properly cleaned up when leaving.

```

compare_arrays_2:
  beqz    a2, .trivial
  li      a6, 1
  li      a7, 1
  j       .loopbody
.loop:
  addi    a2, a2, -1
  addi    a1, a1, 4
  addi    a0, a0, 4
  beqz    a2, .end
.loopbody:
  # Load into a4
  lw      a4, 0(a0)
  lw      a3, 0(a1)
  # +1 into a5
  add     a5, a4, a6
  beq     a5, a3, .loop
  li      a7, 0
  j       .loop
.trivial:
  li      a7, 1
.end:
  mv      a0, a7
  # Clean them up both
  li      a4, 0
  li      a5, 0
```

Figure 5.12: Cleanup sensitive registers: hardened assembly after tracing dataflow

The two other source changes work around current limitations of data-flow tracing:

- I drop the traced type in the comparison because a desirable front-end simplification needed to compile down to a clean beq/bne isn't yet performed on traced types;
- The I/O forbids the branchless rewriting of the check, which happens after the taint is lost.[13]

Algorithm 2 summarizes the back-end pass which inserts the function-exit cleanup of registers that have held sensitive values. It iterates on traced registers tracked down through the

[13] Currently the taint is type-based and disappears after instruction selection. We could keep it until register allocation (which is after the offending Machine IR optimization pipeline) using a dedicated register class.

Algorithm 2 Machine IR sensitive register cleanup pass

Input: A Machine IR function `MF` in non-SSA form
Output: Adds sensitive register cleanup instructions in all return paths in `MF`

```
Cleanups ← {}
for each virtual register VirtReg in MF do
    if VirtReg is traced and has been allocated to some PhysReg then
        Add PhysReg to Cleanups
for each returning Machine Basic Block MBB in MF do
    Terminator ← first terminator instruction of MBB
    Alive ← registers used between Terminator and the end of MBB
    for each PhysReg in Cleanups − Alive do
        insert "li PhysReg, 0" in MBB before Terminator, with flag FrameDestroy
```

taint and adds instructions to zero them out on function exit. (The FrameDestroy flag marks them as epilogue code, avoiding certain optimizations.) The associated intermediate security properties from Figure 3.2 (structural and semantic properties) around this pass are as follows:

- Prior to the early register allocation, all virtual registers in the function that depend on sensitive inputs (in the sense of `trace(dataflow)`) are marked as traced.
- After early register allocation, they are all allocated and none are spilled (allowing the compiler to fail if more live values are traced than can be held in registers).
- After the cleanup pass, all physical registers that at some point hold a sensitive value are zero (or a return value) at every return point.

This approach has a higher upfront cost compared to late data-flow analysis (as it needs to track through multiple layers), but it has two unique benefits:

1. The initial type annotations are complete (unless the source mixes pointers to sensitive and non-sensitive data). This completeness could be carried down to lower levels, providing complete annotation information at low levels, unlike a data-flow analysis which is normally incomplete. (Preserving the completness is not automatic because the compiler may erase types in some internal representations.)
2. Tracing the taint down avoids the precarious work of finding taint sources on heavily transformed code, where source pointers and allocations are hard to identify.

5.3.6. Split register allocation

The final example uses the same array comparison program, but this time the goal is to split the registers allocated to sensitive and non-sensitive data. At the source level, nothing changes compared to Figure 5.11, I still only annotate the sensitive data by tracing the dataflow of the reads from `T1`.

The difference is in the back-end pass; rather than adding zeroing instructions after register allocations, this time I preallocate sensitive data to temporary registers (`t1`, `t2`, ...) before the full register allocation takes place, leading to the code in Figure 5.13.

This "pre-allocation" can take many forms, of which I tested two:

```
compare_arrays_2:
  beqz    a2, .trivial
  li      a4, 1
  li      a3, 1
  j       .loopbody
.loop:
  addi    a2, a2, -1
  addi    a1, a1, 4
  addi    a0, a0, 4
  beqz    a2, .end
.loopbody:
  # Uses a separate register, t1
  lw      t1, 0(a0)
  lw      a5, 0(a1)
  # Uses a separate register, t2
  add     t2, t1, a4
  beq     t2, a5, .loop
  li      a3, 0
  j       .loop
.trivial:
  li      a3, 1
.end:
  mv      a0, a3
  ret
```

Figure 5.13: Split register allocation: hardened assembly after preallocation

1. specifying an allocation order hint for the standard register allocator;
2. using a complete pre-pass that allocates only sensitive registers (used here).

The pre-pass approach enables reserving the registers used for sensitive data so they can't be repurposed for non-sensitive data while dead.

5.4. Application to a full PIN verification program

Having now demonstrated the use of Tracing LLVM for elementary tasks on small examples, let's consider a more complete application with a complex task (still on a small program). The example is of a typical smart card PIN verification function `verifyPIN` [Dur+16]; it was chosen to exemplify many of the typical problems faced by countermeasures in a short function and incorporates parts of previous examples. I'll show that the different features showcased in the previous examples compose well to produce a predictable, fine-tuned assembler program.

For simplicity of exposition, all the countermeasures considered will be implemented on source code, even the ones that would usually be on intermediate programs (the argumentation accounts for the natural placement).

5.4.1. Initial source code

The initial code for `verifyPIN` is shown in Figure 5.14. Its inputs are the number of attempts left (`g_ptc`) and the PIN codes to compare (`g_userPin`, `g_cardPin`); its outputs are the authentication result and an update to the attempt count. The code already features protections against common attacks; ignoring them for now, the function simply checks if there are any attempts left (line 12), and if so compares all four digits one by one (lines 14–17). If the input PIN is incorrect, the number of attempts left is decremented (line 19).

Smart cards commonly need to ward off multiple types of security threats, from general safety to random attacks to targeted attacks. For this example, let's assume the following threats and associated *security requirements* $\langle R_1 \rangle$ to $\langle R_5 \rangle$:

```c
typedef int BOOL;
enum { TRUE = 0xAA, FALSE = 0x55 };
extern unsigned char g_ptc, g_userPin[4], g_cardPin[4];
/* Increment the Step Counter and check for consistency. */
#define SCI_CHECK(N) if(++SC != (N)) countermeasure()
/* Read GL twice, check for consistency, and return the value. */
#define READ(GL) ({ a = GL; b = GL; if(a != b) countermeasure(); a; })

BOOL verifyPIN(void) {
  register BOOL valid = TRUE;
  register int SC = 0, a, b;
  if(READ(g_ptc) == 0) return FALSE;

  SCI_CHECK(1); if(READ(g_userPin[0]) != READ(g_cardPin[0])) valid = FALSE;
  SCI_CHECK(2); if(READ(g_userPin[1]) != READ(g_cardPin[1])) valid = FALSE;
  SCI_CHECK(3); if(READ(g_userPin[2]) != READ(g_cardPin[2])) valid = FALSE;
  SCI_CHECK(4); if(READ(g_userPin[3]) != READ(g_cardPin[3])) valid = FALSE;

  SCI_CHECK(5); g_ptc -= (valid == FALSE);
  SCI_CHECK(6); a = b = 0;
  return valid;
}
```

Figure 5.14: Source code of an attempt at a secure `verifyPIN` function.

- Modifying data in transit from memory [Tal+21]. Here we only deal with reads; for a detailed analysis of write corruption, see NEMESIS [DSL17].
  → $\langle R_1 \rangle$ Duplicate memory reads and check consistency for sensitive values.
  → $\langle R_2 \rangle$ Keep all sensitive values not required to be in memory in registers.
- Skipping instructions [Bar+12] or sequences of instructions, one of the conceptually-easiest ways to bypass the PIN code comparison.
  → $\langle R_3 \rangle$ Do not accept the PIN if an entire digit check is skipped.
- Leakage of secrets or values derived from secrets to other functions, which are not assumed to be secure or trusted. Only when there are no leakages can one study the security of the `verifyPIN` function as a single unit.
  → $\langle R_4 \rangle$ Clean CPU registers of all sensitive data when the function returns.
- Potential bit flips, which can happen accidentally even without an attacker.
  → $\langle R_5 \rangle$ Use 8-bit *hardened booleans* that use values `0x55` (false, 85) and `0xaa` (true, 170) which can't easily be corrupted by bit-flips [Dur+16].

These requirements are representative of the variety of protections found in literature: $\langle R_1 \rangle$ and $\langle R_3 \rangle$ are data-flow and control-flow integrity respectively; $\langle R_4 \rangle$ is a common concern originating in side-channel attacks; $\langle R_5 \rangle$ is a standard precaution; and $\langle R_2 \rangle$ illustrates a non-trivial compiler-related constraint.

When building a complete system, these requirements would be formalized based on tested attack models [DBP23] and accompanied by either simulated experiments or proofs [MDG24] that they indeed block attacks. These efforts are outside the scope of the example though;

```
clang --target=riscv32 -march=rv32gc -mabi=ilp32d -O1 -S pin.c
```

```
verifyPIN:
  lui   a0, %hi(g_ptc)
  # Double reads are gone
  lbu   a1, %lo(g_ptc)(a0)
  beqz  a1, .return_FALSE
  lui   a0, %hi(g_userPin)
  addi  a2, a0, %lo(g_userPin)
  lbu   a6, %lo(g_userPin)(a0)
  lui   a3, %hi(g_cardPin)
  addi  a4, a3, %lo(g_cardPin)
  lbu   a7, %lo(g_cardPin)(a3)
  # Not the SCI order at all
  lbu   t0, 1(a2)
  lbu   t1, 1(a4)
  lbu   t2, 2(a2)
  lbu   a5, 2(a4)
  lbu   a2, 3(a2)
  lbu   a4, 3(a4)
  # No individual checks
  xor   a0, a6, a7
  xor   a3, t0, t1
  xor   a5, t2, a5
  xor   a2, a2, a4
  or    a2, a2, a5
  or    a0, a0, a3
  or    a3, a2, a0
  snez  a2, a3
  li    a0, 0xaa
  beqz  a3, .valid
  li    a0, 0x55
.valid:
  sub   a1, a1, a2
  lui   a2, %hi(g_ptc)
  sb    a1, %lo(g_ptc)(a2)
  # t0, t1, t2, a3, a4, a5 leak
  # Zero-cleanup gone
  ret
.return_FALSE:
  li    a0, 0x55
  ret
```

Figure 5.15: Assembly output from compiling verifyPIN with optimizations (-O1)

here I assume that $\langle R_1 \rangle$ to $\langle R_5 \rangle$ are adequate for the case at hand and focus on getting the compiler to generate appropriate assembler code.

5.4.2. Countermeasures inserted and failed initial build

verifyPIN incorporates multiple countermeasures in an attempt to satisfy our requirements. Normally these would be performed at IR or back-end level, but for ease of presentation I have modeled them at the source level here. Compiling with LLVM yields the 32-bit RISC-V assembly code reported in Figure 5.15, in which it appears that none of them really worked:

1. Reads from globals were duplicated using the READ macro[14] (line 7) which evaluates a global twice and aborts by calling countermeasure in case of mismatch. But there are no double loads in Figure 5.15 (no call to countermeasure and only 8 lbu instructions accessing the PINs), so $\langle R_1 \rangle$ isn't satisfied.
2. verifyPIN attempts to keep variables in registers with the register qualifier (line 10). But while register was first intended as a hint for register allocation, it has long since ceased to affect code generation (as compilers now have smarter register allcation algorithms). It now remains only as a constraint on the uses of the variable [C23, 6.7.2§12, note 128], similar to const.[15] $\langle R_2 \rangle$ is still satisfied but just by chance.
3. The control flow was protected with Step Counter Incrementation (SCI) at source lines 14–17, with the step counter SC being incremented and checked after important steps. But

[14] The macro uses GCC's statement-expression extension; this is only for brevity.

[15] In fact, register was deprecated in C++11 [C++11, §D.2], removed in C++17 [C++17, §5.11].

in Figure 5.15, the SCI checks are gone and all digits are even loaded at once (lines 8–18) before anything is compared, so $\langle R_3 \rangle$ isn't satisfied either.

4. Both sensitive variables a and b (which hold copies of card PIN digits) were cleaned when leaving the function (line 20). But in the assembler code, there is no zeroing in the return block and the function returns while t0, t1, t2, a3, a4, and a5 still hold values related to the card PIN, violating $\langle R_4 \rangle$.[16]
5. valid is reduced to a single-bit boolean in the IR (LLVM's Intermediate Representation—not shown), then replaced with an or–xor chain that accumulates bitwise differences between the user and card PIN. While not as weak as a standard boolean, this is still not a hardened boolean, invalidating $\langle R_5 \rangle$.

Unsuprisingly, problems that we've seen before have only compounded.

5.4.3. Secure build using Tracing LLVM features

The verifyPIN function can be fixed with a combination of basic Tracing LLVM features and some of the dedicated/hardcoded countermeasures showcased in previous examples. To facilitate the discussion of how each hardening decision contributes to the final program, let's look right away at the secure assembly code that we'll end up with, shown in Figure 5.16.

verifyPIN_secure spans lines 1–50. The code for the digit checks starting at line 16 is extremely regular[17], so I factored it out for brevity through a CHECK_DIGIT "macro" at line 52, parameterized by the step counter, digit number, and label of the next check. Throughout the function, the boolean valid is allocated to a0, and the step counter SC to a1 (except line 25 where it briefly switches with a2). For ease of reading I also configured the early register allocator to allocate traced values (tainted by the source dataflow attribute) starting at a5.[18]

In this version the shortcomings of the original build are all fixed. The CHECK_DIGIT sequence strictly follows the ordering set by SCI checks. valid is back to being a hardened boolean, always holding either of the constants TRUE (170) or FALSE (85). The double reads all remain, as seen by the lbu/lbu/bne triplets lines 7, 61 and 66. Finally, both registers that touch sensitive variables, a5 and a6, are cleaned up at the end of the function at line 46.

This is achieved by the following changes. First, the data-flow of g_cardPin is traced, enabling the cleanup countermeasure from Section 5.3.5, which in turns makes the a=b=0 unneeded.

```
unsigned char ! __attribute__((trace(dataflow))) g_cardPin[4];
/* also run the cleanup countermeasure which zeros all registers
   tainted by trace(dataflow) in the back-end (hardcoded) */
```

The step counter order is enforced as in Section 5.3.2 by tracing the writes to both valid and SCI, enforcing the update order of both these variables. The value of the step counter SC is also protected against constant propagation by opacifying its value at every step.

[16] Which values leak varies a lot with every build; this one is particularly egregious. But the PIN can almost always be revealed from the leaks by an adversary in a couple of attempts.

[17] Unlike the leaks, this was consistent across all inspected builds.

[18] This leads to a few inefficient moves like mv a5, a0 that should ideally be optimized.

```
clang --target=riscv32 -march=rv32gc -mabi=ilp32d -g -ftracing -O3 -S pin-s.c
```

```
verifyPIN_secure:
  addi  sp, sp, -16
  sw    ra, 12(sp)
  li    a0, 170     # valid = TRUE
  # READ(g_ptc)
  lui   a3, %hi(g_ptc)
  lbu   a2, %lo(g_ptc)(a3)
  lbu   a3, %lo(g_ptc)(a3)
  bne   a2, a3, .countermeasure
  # If zero, return FALSE
  beqz  a2, .no_attempts_left
  li    a1, 0       # SC = 0 (sunk)

# All four blocks are identical and
# factored in the CHECK_DIGIT "macro"
.d0: CHECK_DIGIT(1, +0, .d1)
.d1: CHECK_DIGIT(2, +1, .d2)
.d2: CHECK_DIGIT(3, +2, .d3)
.d3: CHECK_DIGIT(4, +3, .decide)

.decide:
  # SCI_CHECK(5):
  mv    a5, a0
  # <Opaque I/O touches a1>
  addi  a2, a1, 1
  li    a1, 5
  bne   a2, a1, .countermeasure
  mv    a5, a0

  # g_ptc -= (valid == FALSE):
  lui   a2, %hi(g_ptc)
  lbu   a3, %lo(g_ptc)(a2)
  addi  a4, a0, -85
  seqz  a4, a4
  sub   a3, a3, a4
  sb    a3, %lo(g_ptc)(a2)
  # <Opaque I/O touches a1>
  addi  a1, a1, 1   # SC += 1
  li    a2, 6       # SCI_CHECK(6)
  beq   a1, a2, .return
.countermeasure:
  call  countermeasure
.no_attempts_left:
  li    a0, 85
.return:
  li    a5, 0       # Cleanup a5
  li    a6, 0       # Cleanup a6
  lw    ra, 12(sp)
  addi  sp, sp, 16
  ret

CHECK_DIGIT(_N, _OFF, _NEXT_LABEL):
  mv    a5, a0
  # <Opaque I/O touches a1>
  addi  a1, a1, 1   # SC += 1
  mv    a2, a1
  li    a3, _N      # SCI_CHECK(_N)
  bne   a1, a3, .countermeasure
  # READ(g_userPin[_OFF]):
  lui   a3, %hi(g_userPin+_OFF)
  lbu   a2, %lo(g_userPin+_OFF)(a3)
  lbu   a3, %lo(g_userPin+_OFF)(a3)
  bne   a2, a3, .countermeasure
  # READ(g_cardPin[_OFF]):
  lui   a3, %hi(g_cardPin+_OFF)
  lbu   a6, %lo(g_cardPin+_OFF)(a3)
  lbu   a5, %lo(g_cardPin+_OFF)(a3)
  bne   a6, a5, .countermeasure
  # If different, valid = FALSE:
  beq   a6, a2, _NEXT_LABEL
  li    a0, 85      # valid = FALSE
```

Figure 5.16: Secure `verifyPIN` function written and compiled with Tracing LLVM (`-O3`)

```
BOOL ! __attribute__((trace(writes))) valid;
int ! __attribute__((trace(writes))) SC;
#define SCI_CHECK(N) { \
    SC = __builtin_tracing_opaqueio(SC, valid) + 1; \
    if(SC != N) countermeasure(); }
```

One last hardening pass is also needed: duplicating all accesses to globals. I could do this with a run-of-the-mill LLVM IR duplication pass using wrappers to hide duplicate accesses so they're not eliminated. However, the process can be expedited here by keeping the `READ()`

macro and making the globals of interest volatile, since preserving accesses is the exact effect of `volatile` [C23, 5.1.2.4§2].

```
unsigned char volatile g_ptc, g_userPin[4];
unsigned char ! __attribute__((trace(dataflow))) volatile g_cardPin[4];
```

In order to strictly follow the separation of concerns from Figure 3.1 the countermeasures should use higher-level source annotations (which the end-developer can use) and make the type and qualifier changes in the front-end (which is the security engineer's responsibility). I'm leaving this out for the sake of not adding even more layers to the explanation.

5.4.4. Discussion

`verifyPIN_secure` showcases how internal compiler guarantees can compose to great effect. Notice how the function is guaranteed to compile to four independent digit tests (because they're separated by step counter checks), which must use conditional branches that assign the 8-bit hardened boolean `FALSE` to `valid` (because `valid=FALSE` is traced and is conditional in the source), and this assignment will be a `mv` or `li` (because traced values are promoted to registers). This subtle balance between following the source and optimizing wouldn't be feasible with only outside control of the compiler (such as disabling select optimizations).

5.5. Chapter conclusion

This practically-oriented chapter went through Tracing LLVM from a countermeasure developer's point of view: the project, its main features (relating to opacification, lowerings control and tracing), direct applications, and a complex PIN verification example to tie them all together. I showed how the tools can already be employed to produce fine-tuned assembly code for security purposes, all without sacrificing basic optimizations. The set of primitives in Tracing LLVM isn't intended to be complete (in a theoretical sense); this is just a snapshot of the current version, which I hope to improve in the future.

The choice of primitives and their implementation in LLVM, from the compiler developer's point of view, is not random and hides research questions that I will now address in Chapter 6.

Implementation and integration of Tracing LLVM 6

Papers that implement countermeasures with compiler support, such as those discussed in Section 2.3, generally don't dwell on compiler integration, mostly focusing on when to harden and how to defeat optimizations. There are nonetheless significant research questions associated with the effort, mostly in accounting for compiler behaviors that might threaten the countermeasure (which is often considered) and maintaining these implementations (which is often not). As we've seen in Chapter 5, many of Tracing LLVM's security features are guided by, if not designed for, LLVM's internal details.

This chapter will delve into the implementation constraints arising from this integration of an unconventional security feature in an existing production toolchain. Section 6.1 sets the stage with a perimeter of what's possible or not and the targets set for Tracing LLVM. Section 6.2 discusses existing non-functional features in LLVM and whether we can rely on them for security purposes. Section 6.3 details the techniques used or considered to extend language representations in Tracing LLVM. Finally, Section 6.4 documents all the changes made in Tracing LLVM compared to the baseline LLVM 17.

Research questions

→ Can existing non-functional features be relied upon for security?
→ How to extend LLVM without requiring extensive oversight of optimizations?

6.1. Working in LLVM

In a project the size of LLVM (about 35 million lines of code), understanding the details of every transformation is out of the picture, so the data structures, invariants, and API contracts quickly become more important than the code. The extended semantics for security and tracing are not planned in LLVM's design, but many individual extensions rely on features that are. Correctly fitting security pieces within existing structures and contracts is the difference between a one-off prototype and a piece of software with potential for future maintenance.

6.1.1. Constraints and targets

LLVM is both huge and quickly evolving (over 2000 authors and 37k commits in 2024[1]), so changes that would require adapting large numbers of passes and lowerings are simply not feasible. Thus, my focus is on changes that:

[1] https://www.phoronix.com/news/LLVM-Code-Activity-2024

1. Use documented extension points (for long-term maintainability);
2. Are handled predictably (e.g. ignored) by existing passes without intervention;
3. Facilitate validation, such as by triggering explicit compiler errors if not properly integrated in the compilation flow.

Not changing existing passes means we have to live within their only strong guarantee of preserving semantics. Opaque extensions with implementation-defined semantics work well here as the requirement to preserve any possible implementation is a strong constraint that's easy to exploit. In fact, Tracing LLVM changes almost no optimizations; all of the work is concentrated on representations and lowerings (a full list of changes is given in Section 6.4).

Obviously, the usual software engineering concerns apply. As a representative example for this whole category, dataflow-traced types should propagate through binary operations (see Section 5.2.5). Each binary operation in C is overloaded and traced types should propagate transparently through each overload. Annotating each case would result in dozens of insertions which would be difficult to test and maintain (as future versions are likely to move cases around or add new ones). Instead, Tracing LLVM unwraps the traced types from binary operations' operands before starting the case analysis, and puts them back later[2]. This reduces the amount of new code and the maintenance effort, as I only need to watch for one entry point for all binary operations in future versions.

As far as verification goes, like most works dealing with production tools like LLVM, the correctness of Tracing LLVM is not formally proven. Guarantees derived from the preexisting assumption that the compiler preserves observable behaviors and internal API contracts are considered strong; other properties might be assumed, in which case the goal is to check them dynamically during compilation. Chapter 7 goes into more detail about these assumptions and the role of empirical validation.

6.1.2. Maintenance and research tools

There's no two ways about it: maintaining software is difficult, and maintaining research software doubly so (in part due to institutional and publishing culture). There's also little reason to expect a tool like Tracing LLVM to be merged upstream, for most of the reasons already mentioned in Section 3.3.3. Available tooling still makes a significant difference for users, so I kept medium-term maintenance a primary target while developing Tracing LLVM.

This concern over longevity motivated me to port chosen features from Vu's *llvm-property-preserving*, namely the opacification system (with side-effects) and most of the observation logic and debug information (with limited testing yet however). These mechanisms are extremely useful for security, as I've demonstrated at length in Chapter 5, and thus very valuable to preserve in a readily-available form. For this port, I generated a diff between Vu's final code and the latest upstream commit merged into it (about 2000 lines on top of LLVM 12), analyzed it, and ported each section to LLVM 17. Despite jumping straight through 5 major versions, only minimal adjustments were needed to fit Vu's extensions in LLVM 17.

Tracing LLVM also comes with documentation, including a description of *llvm-property-*

[2] See `clang/lib/Sema/SemaExpr.cpp`, the call to `analyzeTracedOp1`; or follow this versioned link.

preserving's internal implementation (which was previously undocumented):

→ DOC: Tracing LLVM at `https://gricad-gitlab.univ-grenoble-alpes.fr/tracing-llvm/doc`
→ DOC: Property-preserving LLVM at `misc/llvmpp.md`

When it comes to the technical details, improving maintainability in Tracing LLVM mostly involves relying on guarantees from program structure, compared to checking that optimizations don't perform undesirable transformations, as there's a lot less code defining structures than there is code optimizing them. Reusing existing non-functional features would also be a great way to reduce the amount of new logic, but sadly the opportunities there are limited.

6.2. Existing non-functional features in LLVM

The tracking of non-functional information for Tracing LLVM's isn't entirely new in the LLVM codebase; there are already such cross-abstraction features in the compiler. Using them for Tracing LLVM isn't straightforward, however.

6.2.1. Debug information

By far the closest feature to tracing, debug information has the express purpose of tracking the relationship between source code and all intermediate compiler stages down to assembly. The most obvious occurrence is the mapping of control points to source lines, which occurs at all levels of abstraction. Some features of Tracing LLVM could in principle be handled through debug information, such as tracing writes to variables. However, this is nearly impossible in practice due to a significant difference in approach:

- Debug information is functionally transparent, optional, and offers a best-effort description of the program without ever preventing the compiler from optimizing as it sees fit;
- Meanwhile, tracing for security countermeasures is not optional (when used) and purposefully inhibits optimizations as needed to ensure that simple relationships between source and intermediate programs exist (e.g. all assigned values being computed even if redundant).

Using debug information for security purposes is made even harder in practice by the fact that LLVM handles it through *metadata*, which can be removed at anytime by API contract. It's actually even worse as *"a transformation is required to drop any metadata attachment that it does not know or know it can't preserve"*[3].

Interestingly, debug information reveals the existence of *hidden lowerings*: transformations not classified as lowerings but that do carry a change of abstractions. For instance, the LLVM IR register-promotion pass `mem2reg` identifies stack allocations that are only read or written directly and never have their address taken, then replaces each write with the definition of a new SSA temporary. Figure 6.1 shows the debug tracking for an assignment `a = x+y`.

In the original code, an allocation is performed on the stack using the `alloca` instruction, resulting in the pointer `%a`. Immediately after the allocation, the LLVM primitive `llvm.dbg.declare` is used to associate the pointer `%a` with the source variable "a" (referred to by the metadata

[3] `https://llvm.org/docs/LangRef.html#metadata-nodes-mdnode`

```
; original code
  %a = alloca i32, align 4
  call void @llvm.dbg.declare(metadata ptr %a, metadata !18, metadata !DIExpression())
  %add = add nsw i32 %x, %y
  store i32 %add, ptr %a, align 4

; after mem2reg
  %add = add nsw i32 %x, %y
  call void @llvm.dbg.value(metadata i32 %add, metadata !18, metadata !DIExpression())

; debug details (for both)
!18 = !DILocalVariable(name: "a", scope: !9, file: !1, line: 12, type: !12)
!9  = !DISubprogram(...) ; the function "f"
!12 = !DIBasicType(...)  ; the type "int"
```

Figure 6.1: Lowering of debug information through the mem2reg optimization

!18). This is performed only once and tells the debugger to look at the stack to know the current value of the source variable “a” at any control point in the function.

After register promotion, there is of course no longer a stack allocation. Instead, the value of “a” is communicated to the debugger via an imperative update after each assignment. After the addition, the primitive llvm.dbg.value indicates that %add is the new value for “a” (again named as !18), and there will be one such call for each assignment.

In essence, mem2reg lowers the debugging abstraction from defining values by location to defining them by the stateful execution of debug assignments. This erodes the relationship between source and intermediate programs that we needed for tracing in examples such as “Sequencing at variable writes” (Section 5.3.2). This constitutes a lowering because we are changing abstractions, even though it is “hidden” by the fact that both source and target abstractions (location and explicit assignment) are housed in the same language (LLVM IR).

6.2.2. Inline assembly

The main abstraction-breaking feature in clang, pioneered by GCC, is *inline assembly* [GCC25], which allows the insertion of arbitrary assembly code into C programs (... and all intermediate languages). I briefly discussed inline assembly in Section 2.4.2, but without going into its implementation details.

Extended inline assembly blocks are opaque input-to-output sequences of assembler code. Consider the documentation’s introductory example (converted to RISC-V below).

```
int src = 1, dst;
/*  [code]                        : [outputs] : [inputs] */
asm("mv %0, %1; addi %0, %0, 1" : "=r"(dst) : "r"(src));
return dst;
```

The code is an opaque string[4] consisting of assembly code where operands interfacing with surrounding C code can be specified with numbered tokens `%0`, `%1`, etc. The associated operands are given, in numbering order, in the "outputs" and "inputs" sections. Each operand consists of *constraints* (given as a string) and a C lvalue or rvalue. In the example:

- `%0` is an output; it has the constraint `"=r"`, meaning it will be written to (=) and the instruction operand should be a general-purpose register (r). The output is the lvalue `dst`, so the value of the register chosen for `%0` will be written back to `dst` after the statement.
- `%1` is an input; the constraint `"r"` indicates that this operand must also be a register, and it must be preloaded with the value of `src`.

The feature goes much further than this basic example, including the ability to specify side-effects (using the `volatile` qualifier—by default a block with inputs and outputs is considered pure), registers clobbered for temporary computations, complex constraints for controlling the types of operands generated, labels that the block can jump to, and more.

Inline assembly is the archetype of a feature that provides the user with fine control over the generation of target code, while surviving all compilation stages and interfacing effectively with the rest of the program. Its clear limitation is that it takes assembler code directly, so the code itself is target-dependent and can't undergo any automatic lowering. It can however be used directly by applications and might be used internally by Tracing LLVM in the future for gadgets whose assembler code can be hardcoded.

6.2.3. Hacks

And finally, as one would expect from a complex production tool, there are spots in which LLVM breaks its own rules (abstractions) and incorporates low-level data in high-level transformations. One giant hack I had to contend with while writing the fetch skips hardening countermeasure is the static branch relaxation pass[5]. I would like to detail it here because it is emblematic of model-against-implementation friction and justifies the non-formal approach to interfacing with LLVM.

Branch relaxation is the process of selecting the best available type of conditional branch code based on the distance to the target:

- If the target is close, one can branch with a single conditional branch, which in RISC-V have an 12-bit immediate providing a range of ±4096 bytes[6]:

```
  beqz    a0, .zero
  # non-zero case...
.zero:
  # zero case...
```

- But if the target is further than the range of a branch immediate, a full-range unconditional jump instruction must be used, which requires an extra basic block.

[4] Not even parsed in GCC; syntax-checked in clang (both estimate the size for relaxation)

[5] Found in `llvm/lib/CodeGen/BranchRelaxation.cpp`, at least in LLVM 17.

[6] The least significant bit is always 0 due to code alignment and is not represented.

```
  bnez    a0, .nonzero
  auipc   a1, %hi(.zero)
  jr      %lo(.zero)(a1)
.nonzero:
  # non-zero case...
.zero:
  # zero case...
```

Finding the shortest combination of instructions that satisfies range requirements is not as trivial as it may seem, for two reasons.

- First, because of linker relaxation (basically a linker optimization that may remove code during linking[7]), the true final size of the program is only known by the linker, so the assembler and compiler can only work with upper bounds.
- Second, the information needed to optimize is only available very late. In general, only the linker (which is responsible for laying out code sections) knows what function comes before what other and how far jumps are from their targets. The assembler or object code emitter may know for branches to the same section from the same compilation unit. Higher-level code which doesn't know the exact instruction sequences is almost blind.

It follows that structurally, only the linker can find optimal instruction sequences. However, because not all linkers support relaxation, LLVM still attempts to optimize short jumps within a given function, which is the task of the *static branch relaxation* pass.

This pass runs on Machine IR, in a state where there are still unexpanded pseudo-instructions, unassembled inline assembly, and no strict ordering of basic blocks within functions. In order to still evaluate ranges, the pass asks the back-end for upper bounds on the size of instructions and then imitates part of the layout process of the code emitter.

This pass is quite brittle as it imports later abstractions (such as code size) earlier in the compiler and produces invalid programs if the layout predictions or code size heuristics are incorrect. It also prevents the insertion of new code past static branch relaxation as that could invalidate short jumps which the linker cannot expand back. Looking back at the fetch skips countermeasure from Chapter 4, which relies on linker relocations and has its placement constrained by the static branch relaxation pass, such complex interactions hamper (if not outright discourage) formal validation of the implementation.

6.3. Techniques to extend or constrain representations

With the scale of LLVM, it's not reasonable to keep tabs on all optimizations to prevent certain transformations or avoid code patterns from being used. The clearly superior option is to modify the language in such a way that the existing guarantees of semantic preservation and API compatibility extend to fulfill the new requirements. This section documents the methods used in Tracing LLVM and some that were considered but not used yet.

[7] It's actually a bit worse than this, as cross-section relaxation can invalidate previous relaxations and cause non-converging oscillations. See e.g. `https://github.com/llvm/llvm-project/pull/142899`.

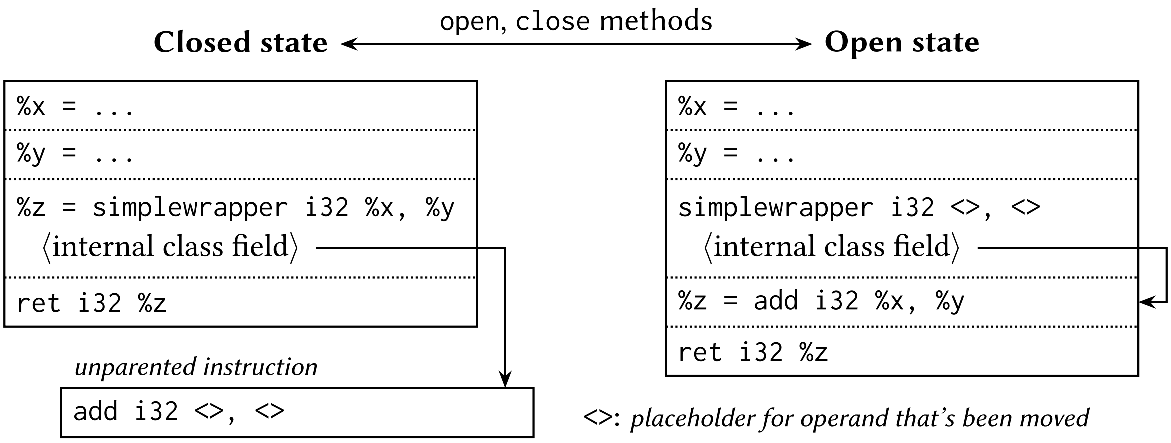


Figure 6.2: Wrapper instructions in the context of a basic block.

6.3.1. Intended language extensions

Some features are open by design of either language or implementation. For instance, clang has a modular type system. Its types more or less constitute an inductive family and have a common interface that's easily extended. Even though in principle C has a finite set of types, there are sufficiently many built-ins, target-specific or compiler-specific extensions that the implementation leaves it completely open. This is how traced types (from Section 5.2.3) are implemented.

Similarly, LLVM IR leaves its set of instructions open. It has close to 70, and manages them through a shared interface (the base `Instruction` class) if only to keep the implementation clean. The assumptions on new instructions are minimal, being given either by SSA (e.g. data-flow) or the generic interface (e.g. memory behavior, side-effects, etc.). This is how wrappers (from Section 5.2.2) and a number of builtin-style instructions are implemented.

6.3.2. States allowed by API not exploited by implementation

Extensions can also rely on program structures allowed by API contract even if not intended for semantic purposes. For instance, LLVM IR instructions are normally placed in basic blocks, and so the `Instruction` class has a method `getParent` returning a pointer to a `BasicBlock`. However, instructions sometimes have no parent; when created, moved, or before being destroyed. Operations that don't require the full block context are thus legal and usable on unparented instructions, which I rely on in wrappers.

As a reminder, wrappers hide one or more instructions from LLVM, exposing only their data-flow (see Section 5.2.2). I mentioned that the example below was a single `simplewrapper` instruction with a hidden `add`:

```
%z = simplewrapper i32 1 2 closed   ; 1 hidden instruction, 2 arguments
   add i32 %x, %y                   ; hidden to LLVM
```

The memory representation of this instruction is shown in Figure 6.2 (left) in the context of a parent block: the linked `add` is unparented and only reachable through an internal field of the wrapper. This also shows why its operands must be moved to the wrapper: otherwise it'd be reachable through SSA's def-use and use-def chains.

Naturally, the linked instruction of a wrapper can't be used properly without its operands. Wrappers can be *opened* to expose the linked instruction in the parent block and give it its operands back (Figure 6.2, right). This is used at key moments such as for lowering and printing, which is why the code shown above has operands on the linked instruction (the `closed` keyword denotes the intended state).

Another example of this extension method is the Selection DAG representation of wrappers, where the linked instruction is not lowered and instead referenced by a generic DAG node that points to an arbitrary IR value (as was shown in Figure 5.3). This type of reference isn't normally used for code, but serves as a good opacifier.

In both cases, the linked instruction is hidden by a strong constraint: the first through the internal implementation of a subclass, and the second through a node type that escapes the instruction selection "language" (DAG). It is extremely unlikely for optimizations to discover and modify the hidden code.

6.3.3. Constrained shapes within existing languages

Some intermediate languages have features that structurally constrain the program. For instance, the Selection DAG provides *glue* edges that force instructions to be scheduled as a block, not only enforcing their relative order but also preventing any instructions from being scheduled in-between[8]. Normally it's used to enforce ordering when there are implicit dependencies (such as writing arguments to registers before calling a function), but this mechanism could be used to lower type wrappers (as shown in Listing 5.4) until machine IR by preventing untraced values from escaping in the DAG.

A similar mechanism, called *bundles*, is available in Machine IR (in the back-end; recall Figure 5.2). It's intended for Very Long Instruction Word (VLIW) architectures which emphasize parallelism by static scheduling, but results in practice in a lot of scheduling constraints for optimizations, which security code could again use to avoid undesired optimizations.

6.3.4. Divert existing features

Existing features can also be "abused" for their operational effect rather than their intended semantics. For instance, LLVM has *convergence semantics*[9] for parallel execution environments. Operationally, calls to convergent builtins cannot be made control-dependent on any new values, and in general (in the absence of additional information) they cannot be hoisted or sunk across branches. This type of guarantee can be useful for control-flow checks, as in general the purity of signature updates and checks leaves them open to many optimizations.

The long list of attributes supported by clang and LLVM[10] is also a better entry point for maintainability that custom compiler logic, featuring optimization blockers (`nomerge`,

[8] Until Machine IR is emitted, that is.

[9] `https://llvm.org/docs/ConvergentOperations.html`

[10] `https://clang.llvm.org/docs/AttributeReference.html`

noduplicate, no-builtin), front-end automation (e.g. cleanup), validation mechanisms (noescape, noderef), and even tracing of some kinds (annotate, preferred-type).

6.4. Summary of changes made to LLVM

For the sake of completeness, I'll now briefly list all the changes Tracing LLVM makes to structures, lowerings and passes that we've encountered in this thesis.

- In the front-end (clang), I add traced types with two attributes dataflow and writes, as well as typing and code generation rules for traced code.
- In the middle-end (LLVM IR), I add traced types, intrinsics for manipulating them, and wrapper instructions.
- In the instruction selector (Selection DAG), I add traced scalar types with integer promotion rules, and a hook for manually lowering wrappers.
- In the back-end IR (Machine IR), I add two passes:
  - The late register-promotion pass which promotes traced stack locations into registers;
  - The "early" register allocation for only traced values.[11]
- I also port over a side-effecting version of Vu's observations [Vu21, §4] from LLVM 12, since they fit neatly in the design.

There were two instances where I had to modify optimizations that violated requirements:

1. I modified InstCombine, the IR peephole rewriter, so it doesn't match loads and stores of traced writes; otherwise LLVM likes to change the access type and compensate with a bitcast. This affects loads and stores of trace(dataflow) data, which are currently not protected by wrappers. (5 LoC)
2. I altered the RISC-V Machine IR copy propagation pass to not eliminate instructions in stack-frame destruction code (marked with the FrameDestroy attribute), so the cleanup countermeasure from Algorithm 2 can protect its cleanup code. (2 LoC)

There will likely be more in the future while the implementation irons itself out.

6.5. Chapter conclusion

The features in Tracing LLVM grew both from the needs of security countermeasures and the feasibility of LLVM extensions. Unsurprisingly, the design approach really differs between a demonstration with a clearly-delimited scope such as would be found in a single research paper, and compiler support that can be reliable over time. This is a standout feature in Vu's thesis, and hopefully this one as well.

In this chapter I explained why Tracing LLVM doesn't rely much on LLVM's integrated non-functional features and discussed the main extensions points that I relied on to implement security extensions without ruling out the maintainability of the implementation.

[11] I intended for this pass to occur before the Machine IR optimizer (which would have been earlier), but this didn't work out because the optimizer only works on the SSA flavour of Machine IR.

→ Can existing non-functional features be relied upon for security?

This really depends on each feature. Debug information is a lost cause for tracing, since it constrains nothing and disappears too frequently. Inline assembly is great at everything it offers, as long as the assembly code can be provided in advance. The overall usability of existing non-functional features is mixed. Fortunately, there are other functional features and API contracts that we can rely on for custom logic.

→ How to extend LLVM without requiring extensive oversight of optimizations?

By encoding the security requirements in the program structure or language semantics. The best option is always for transformations that violate non-functional requirements to be functionally invalid, which is why opacification works so well. The coverage from such structural or semantic invariants will never be complete though, so a healthy dose of dynamic checks and assertions to go with them is a minimum.

7 Outline of guarantees and validation methods

Chapter 6 mentioned that formal proofs weren't realistic for validating custom logic in such a large and organic compiler as LLVM. This begs the question as to what mix of simplified formalism and experimental validation is achievable, and if any design compromises are required. This brief chapter will establish some practical bases, analyze theoretical threats against opacification primitives, and finish with Tracing LLVM's current (unchecked) assumptions about compiler behavior.

Section 7.1 provides brief context as to how experimental validation and formal derivation would work together. Most of the chapter, Sections 7.2 to 7.4, is dedicated to the guarantees that opacification mechanisms can provide. These sections summarize Vu's assumptions out of his original formalization [Vu21, 5] then enumerate a fairly long list of theoretical threats that rule out semantic guarantees. Complexity arises from having multiple opacification primitives; subtleties on the notions of dependence, preservation, and traces; weaknesses that threaten some but not all approaches; and applications that don't directly follow from base guarantees. Table 7.11 summarizes the entire effort.

Since Tracing LLVM does not yet implement dynamic validation features, I will only briefly discuss the considerations that are in the direct continuity of Chapter 6 in matters of implementation and maintainability.

Research questions

→ What constraints should experimentally-verified guarantees satisfy?
→ What guarantees can or can't be extracted from opacification primitives?

7.1. Scope and tools for validation

There are several competing goals to accomplish in validating security primitives:

1. Rely as much as possible on formal inference as opposed to experimental validation;
2. Formulate experimental assumptions along API contracts as much as possible;
3. Aim for experimental assumptions that can be checked in complete ways.

With a certified compiler like CompCert [Ler09], one could focus entirely on goal (1), but unsurprisingly that's not an option with LLVM, which prioritizes practical output and leaves some semantic details unformalized[1] (one example is the unclear rules around pointer provenance

[1] Its presumed successor MLIR has so far not improved on this trait either (regrettably).

and escaping). Interestingly though, once the baseline is established that *some* assumptions must be experimentally validated, goals (1)–(3) create some tension.

For example, consider a countermeasure that runs some setup and wind-down code at precise moments at the entry and exit of functions. Maximizing formal inference suggests that related back-end passes such as the prologue and epilogue inserter should be certified compatible with the countermeasure. However, it's just as effective to perform translation validation and raise alarms if the expected placement of setup or wind-down code is violated. As it turns out, the prologue and epilogue inserter is a complex Machine IR pass which spans about 1600 LoC and calls into many target-dependent functions for manipulating code, including hooks for custom target logic at multiple points in the process[2]. The certification effort would be considerable, likely require simplifications, and wouldn't trivially port over to future versions or other back-ends. Hence, minimal experimental assumptions isn't always the best option.

Goal (3) is the requirement for loosening formal verification—experimental assumptions should be validated in *complete* ways as much as possible. A typical source of incompleteness might be from data-flow analysis. In the "Cleanup sensitive registers" example (Section 5.3.5), I discussed how tracing types had the potential to provide complete sensitivity information, as opposed to performing a back-end data-flow analysis, which is normally not complete[3]. An incomplete check means that property violations might go unnoticed, which is not good.

Goal (2) is pretty straightforward; it's much easier to enforce and check assumptions that are expressed along API contract lines. For instance, checking behavior on register allocators is easier for allocators based on the `RegAllocBase` class, which keeps a one-to-one record of the mapping between virtual and physical registers[4] and rewrites the program at the end. This allows checks to be performed in-between passes. By comparison the default register allocator for unoptimized builds, `RegAllocFast`, makes decisions on-the-fly and may assign different occurrences of the same virtual register to different physical registers, leaving no trace of its decisions unless the pass is modified directly.

Figure 7.1 visualizes how empirical behavioral guarantees from LLVM could be strengthened into formal guarantees for countermeasures:

- First run an **experimental filter** to make sure that LLVM conforms to a simplified formal model consisting of all the experimental assumptions. The point is to error out and fail the compilation whenever an assumption is violated (acting like translation validation).
- Then **countermeasure constraints** triggered by security features in the compiler can provide guarantees expressed in terms of the simplified formal model, building up to a security proof for the countermeasure.

These steps would ideally be the (separate) responsibilities of compiler and security engineers as originally depicted on Figure 3.1, splitting the validation in exactly the same way as countermeasure design and implementation.

[2] See `llvm/lib/CodeGen/PrologEpilogInserter.cpp`; the main pass function `runOnMachineFunction` calls two arbitrary-logic hooks `processFunction*` given by the target-specific `TargetFrameLowering` class.

[3] This isn't a matter of the analysis' quality; intermediate compiler stages might rewrite the program in a way that mixes sensitive and non-sensitive data.

[4] If the allocator decides to split a live range, a new virtual register is created.

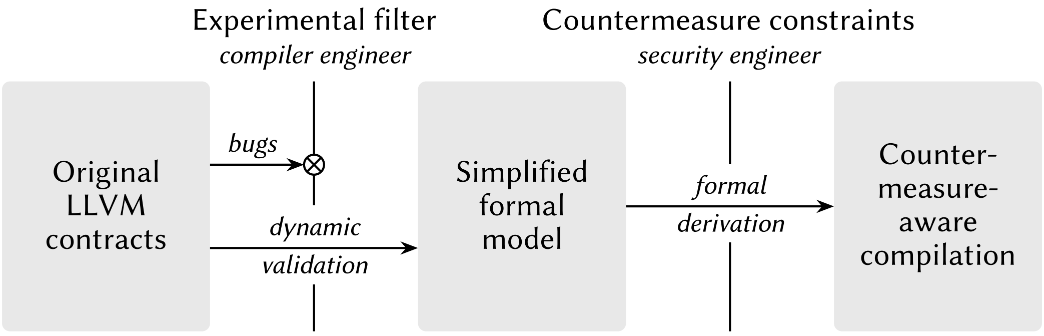


Figure 7.1: Model for a gradual strengthening of the behavioral guarantees of LLVM

One important step that Tracing LLVM already takes is to define security at every intermediate step (recall the security property lowering of Figure 3.2). To no one's surprise, in a heavily pass-based compiler such as LLVM, isolating the passes that violate security properties reveals bugs effectively compared to just validating the assembler output. Bugs up to the current version of Tracing LLVM were found this way by manual bisection.

7.2. Properties desired from opacification

Let's now take a look at the experimental or formal guarantees one could provide for opacification features. As a reminder, opacification features (described in Sections 5.2.1 to 5.2.3) are implementation-defined, which means the compiler must preserve all possible implementations. This strong property is easy to reason with: a transformation can be proven to violate opacity just by exhibiting an implementation on which it's functionally invalid.

In the following sections, I'll consider the following five primitives:

- Three **opacification functions** (Section 5.2.1) to opacify values, written `OPACIFY()`, with three possible degrees of strictness:
  1. A completely pure function (the least strict);
  2. A non-deterministic function that doesn't carry a side-effect;
  3. A side-effecting function (the most strict).
- **Wrapper instructions** (Section 5.2.2) to opacify computations;
- **Traced types** (Section 5.2.3) to opacify types and their representations.

Let's start by reviewing Vu's theory of opacification and the guarantees that it expects to enforce when the source program is transformed by the compiler, either by specification, or as a consequence thereof. This is a very quick summary of Chapter 5 of his thesis [Vu21], which is formalized on a Mini-IR language similar to LLVM IR; for the sake of presentation I will write the examples in C. Starting with the basics:

→ Opacification is a function that takes one or more arguments and returns an unspecified value of the first argument's type. The runtime implementation returns the first argument and ignores any others. [Vu21, 5.2.3.5]

→ Opacification has no side-effect, no memory effect, and is deterministic [Vu21, 5.3.2.1].

This means that Vu's opacification function is pure (the least strict kind). This choice is intentional, as this allows opacification calls to be merged with other identical calls for efficiency [Vu21, p. 67], removed if dead, hoisted out of loops, etc.

The basic guarantee in all transformations is that calls to the opacification function can return any value of the appropriate datatype, and thus, every dependent expression must be evaluated. The only requirement for this is that the downstream expression must have at least two possible dynamic values (the *opaque value set*) so that the compiler can't constant-fold it.

> "If the opaque value set (...) of possible values used or read by [an instruction] i_k (which is on a dependence relation with the value defined by [an instruction] i_j) contains at least two elements, then the compiler has no other way to compute this value used or read by i_k than using the opaque value defined by i_j (...) and the presence of i_k actually requires the presence of i_j." — [Vu21, 5.2.2] *(some formal notations elided)*

For context, the *dependence relation* used here covers both data and control dependencies, so i_k must either (indirectly) use the value produced by i_j (as specified by the execution traces) or be on a conditional path which depends on it (as given by the program's CFG). All preservation guarantees are obviously predicated on the entire construction not being dead, which Vu ensures by sinking dependent values in I/O operations. In the following, I'll always assume that outputs are alive, which gives us our first expected property (assumed):

→ $(\mathbf{P_1})$: The dependence between an opaque instruction that can take multiple dynamic values and a downstream dependency must be preserved in all traces.

Vu leverages this to formalize the notion of *opaque chain*, which is a chain of dependent computations, starting and ending with opacification calls, such that all intermediate values have an opaque value set of size at least 2. Again, I'll focus on examples where the chain's output is alive. This implies that the entire chain must be computed because of the dependence:

> The control- and data-dependence restrictions [that all opaque value sets have at least 2 elements] serve as "information-carrying" guarantees: the compiler does not have enough information about the possible paths dependent on an opaque value or on the processing of opaque values to break an opaque chain into distinct dependence chains. — [Vu21, p. 81]

This gives us two additional expected properties (derived):

→ $(\mathbf{P_2})$: Live opaque computations must be evaluated in all traces;
→ $(\mathbf{P_3})$: Dependencies in opaque chains constrain the evaluation order in all traces.

The liveness follows from transitively applying $(\mathbf{P_1})$ starting at the live output of the chain, while the sequencing follows from the definition of the dependence relation, whose data- and control-flow cases are all ordered by time [Vu21, p. 72].

The final guarantee laid out by Vu (assumed) is that opaque expressions must be evaluated on their original input value as given by the source program, as the compiler can't characterize the behavior of opaque computations on any other inputs.

> (...) a valid program transformation has to preserve any value used in the opaque expression, as proceeding with downstream computation would otherwise involve some form of unauthorized guessing of the opaque expression's behavior. — [Vu21, p. 79]

→ $(\mathbf{P_4})$: Opaque calls must be evaluated on their original inputs.

In addition, the countermeasures on which the method is evaluated [Vu21, 5.4] showcase a few direct applications which are not fundamental properties of opacification but rather constraints that it helps enforce. The masking example, for instance, uses opacification to prevent the compiler from reassociating exclusive-or operations:

```
key = OPACIFY(OPACIFY(key ^ new_mask) ^ old_mask);
```

But the security objective for this statement is much broader; we need `key ^ old_mask` to never be evaluated. I'll stick to that formulation even though Vu didn't target this property directly.

→ $(\mathbf{A_1})$: No evaluation of expressions not given in source.

And finally, the constant-time selection example uses opacification to prevent the compiler from re-identifying branching operations.

```
uint32_t ct_sel_val1(uint32_t x, uint32_t y, bool b) {
  signed m = OPACIFY(0 - b);
  return OPACIFY((x & m) | (y & ~m));
}
```

→ $(\mathbf{A_2})$: Preserve the constant-time property on pure expressions.

As it will turn out, all of these properties can be violated in one way or another by legal compiler transformations on the pure version of the opacification function. To be clear, the threats described below are mostly theoretical and can generally be trusted to not arise in practice, meaning e.g. translation validation (if possible) is a satisfying solution. However, this means they should be checked on the experimental side of Figure 7.1. This also explains my more conservative implementation of the opacification function.

7.3. Transformations weakening opacification guarantees

This section is a listing of some unusual transformations that defeat the properties listed previously. All of these are of course legal (in C, or for the ones dependent on the cardinality of booleans, in LLVM IR at least), but they're specially crafted and range from "unlikely to happen" to "just a theoretical point". In the examples below, the arrow comment "->" describes that the code above can be legally rewritten into the code below.

Simplifying identities. Identities such as absorbing elements can eliminate dependencies on *some* paths, based on dynamic values. For instance, bitwise *and* with zero is always zero:

```
a = OPACIFY(b) & c;
// ->
if(c == 0)
  a = 0; // b not even computed, let alone opacified
else
  a = OPACIFY(b) & c;
```

Listing 7.2: New control flow breaking opaque chains with pure opacification

With pure or non-deterministic opacification, not only is `OPACIFY(b)` dead, but `b` as well, violating $(\mathbf{P}_2)$. A side-effecting opacification would require the call to be kept but the dependency can still be broken, and the ordering along with it.

```
a = OPACIFY(b) & c;
// ->
if(c == 0) {
  a = 0;
  OPACIFY(b); // side-effect unused, b computed out-of-order
}
else
  a = OPACIFY(b) & c;
```

Listing 7.3: New control flow breaking opaque chains with side-effecting opacification

This violates $(\mathbf{P}_1)$ and $(\mathbf{P}_3)$ for all three types of opacification functions.[5]

Obviously expanding a bitwise operator this way makes little sense performance-wise. There are, however, a couple of native operators in C that perform non-trivial tasks and can be compiled in smart ways: namely, selection, division and modulus, assignment, passing, and lvalue-to-rvalue conversion of large structures (all of which can generate a copy), and most arithmetic and bitwise operations on large types. For instance, on some architectures without hardware division GCC has a `call-table` division strategy that implements division with a table of reciprocals for small divisors (using the well-known double-precision multiplication/bitshift method) and a full division algorithm as fallback. Opacification alone cannot prevent the compiler from selecting such an implementation.

This lack of control is why constant-time programs define their own selection function in the first place. But C is ill-equipped for redefining division or large arithmetic, having no direct access to CPU instructions, carry flags, etc. (Besides, performance would be terrible.)

By comparison, wrappers are designed specifically to provide control over *operations*. They completely bypass the rewriting problem by preserving the source-provided operation until the compiler reaches a level of abstraction at which the desired implementation can be substituted. It can also enforce ordering in general by using the implied ordering for all side-effects.

Rewriting pure operations. Using the language's native logical and arithmetic operations enables the compiler, no matter the operands, to rewrite instruction patterns via peephole optimizations, fold constants, generate more constants by specializing functions (the interprocedural `function-specialization` pass in LLVM), and otherwise choose alternative implementations of operations.

For instance, here is a legal transformation of a constant-time selection function that breaks the constant-time property expected by $(\mathbf{A}_2)$ while likely improving performance.

[5] Preventing the duplication of the side-effecting `OPACIFY(b)` still leaves speculation (see below).

```
uint32_t ct_sel_val2(uint32_t x, uint32_t y, bool b) {
  signed m = OPACIFY(1 - b);
  return OPACIFY(x * (1 - m) | y * m);
}
// ->
uint32_t ct_sel_val2(uint32_t x, uint32_t y, bool b) {
  signed m = OPACIFY(1 - b);
  uint32_t z = (m == 0) ? x : x * (1 - m) | y * m;
  return OPACIFY(z);
}
```

Listing 7.4: Legal non-constant-time transformation of a selection function

Here we can eyeball that the x path takes a single instruction (just the branch[6]), while the y path takes 5 (non-taken branch and 4 computations), compared to the base path of 4 for both. It stands to reason that the branching version may in fact be faster on real hardware (misprediction delays on one hand, multiplier latency on the other), making this optimization the one threat in this list that appears realistic.

One could opacify all sub-expressions to avoid triggering complex rewrite rules, but this still allows unfoldings of single operations, which broke $(\mathbf{P_1})$ in “Simplifying identities”.

Speculation (static). Sequence breaks can happen due to speculation, where the compiler guesses a value and eagerly evaluates downstream (pure) uses of it. If the speculated guess is right, downstream uses will have been evaluated before the value’s original expression, bypassing the data-flow dependency. It doesn’t matter whether the original expression has a side-effect or not as long as the downstream uses are pure.

```
a = OPACIFY(b) + c;
// ->
a = b + c; // speculate that OPACIFY(b) = b
ob = OPACIFY(b);
if(ob != b)
  a = ob + c;
```

Listing 7.5: Breaking side-effect dependency order on speculated paths

This breaks $(\mathbf{P_3})$ for all versions of the opacification function, although it should be easier to control or avoid in practice in the side-effecting case (which is one of the assumptions Tracing LLVM makes; see Section 7.5).

Note that this is *not* hardware speculation but static “code” speculation, and the threat remains even when executing on an in-order core. Conversely, this threat pops up everywhere on out-of-order cores even if the code doesn’t itself speculate, which is why Spectre and its variations are so hard to rein in.

[6] On RISC-V the first parameter and return value are both in a0, so no move needed.

Tabulating the opacification function. If the opacification function is deterministic, the compiler can tabulate any call site where the input is statically limited to a few values, such as a boolean or a computation based on one. This could be an issue with the constant-time selection example [Vu21, 5.4.1.4], where the opacified expression `0 - b` can only take two values (`0` and `-1`) since `b` is a boolean.

```c
uint32_t ct_sel_val1(uint32_t x, uint32_t y, bool b) {
  signed m = OPACIFY(0 - b);
  return OPACIFY((x & m) | (y & ~m));
}
// ->
uint32_t ct_sel_val1(uint32_t x, uint32_t y, bool b) {
  signed om1 = OPACIFY(-1), o0 = OPACIFY(0);
  signed m = b ? om1 : o0;
  return OPACIFY((x & m) | (y & ~m));
}
```

Listing 7.6: Breaking evaluation order by tabulating an opacify with two values

While not showcased here, this easily violates the ordering constraint in $(\mathbf{P_3})$ as the now-constant opacifications can be air-lifted pretty much anywhere in the program. For instance, if `ct_sel_val1` gets inlined into a loop, `om1` and `o0` can be hoisted out of the loop, allowing the opacifications to occur *before* the loop-dependent values of `b` are evaluated.

And while the performance benefits are dubious, the same optimization could legally be performed for larger sets with an array, in the common case where the input value is bounded. This loses the constant-time property $(\mathbf{A_2})$ on an opacification through a memory access.

```c
OPACIFY(x & 15);
// ->
const int opc_table[16] = { OPACIFY(0), ..., OPACIFY(15) };
opc_table[x & 15];
```

Listing 7.7: Breaking evaluation order and constant-time by tabulating an opacify

A jump table with a `switch` is even worse, both performance-wise and security-wise:

```c
OPACIFY(x & 15);
// ->
switch(x & 15) {
case 0: OPACIFY(0); break;
case 1: OPACIFY(1); break;
...
case 15: OPACIFY(15); break;
}
```

Listing 7.8: Breaking evaluation order and constant-time by dispatching an opacify

This has the extra effect of breaking the constant-time property $(\mathbf{A_2})$ even for a non-deterministic or side-effecting opacification function since it doesn't compute any unneeded opacification.

Global specializations. The ability for opacifications to be hoisted, merged, and otherwise optimized on a global scale leaves Vu with a very weak mapping between opacification calls in source and transformed programs. For instance, the entire program could be duplicated and specialized for the four possible implementations of boolean opacification:

- The constant implementations where `OPACIFY(b)` is `true` or `false`;
- The bijective implementations where `OPACIFY(b)` is `b` or `!b`.

A single check of `OPACIFY(true)` and `OPACIFY(false)` at the program entry is enough to select the correct one and prune any boolean opacification from the entire program. Any source opacification of a boolean must then be mapped to this initialization check, which still constitutes a dependency, but ruins any notion of ordering within opaque chains by putting all boolean opacifications completely out-of-order, violating $(\mathbf{P}_3)$.

Function structure inference. The most straightforward property in this discussion is $(\mathbf{P}_4)$, the fact that an opaque call with a given value v can't be replaced by an opaque call with any other value. To avoid any of the tricks discussed in "Simplifying identities", assume we opacify a boolean and immediately sink it into an I/O on exactly one control flow path for all inputs. If the opacification function is pure, the compiler can still—amazingly!—abuse it to invalidate this expectation.

```c
// Constant-time array access function: ct_array_read(T, i, N) = T[i]
int ct_array_read(int *T, unsigned i, unsigned N) {
  int Ti = 0;
  for(unsigned j = 0; j < N; j++)
    Ti += T[j] * (int)OPACIFY(j == i);
  return Ti;
}

if(!OPACIFY(i >= N) && (Ti = ct_array_read(T, i, N)) > 0)
  IO(OPACIFY(Ti == Tj));
// ->
if(!OPACIFY(i >= N) && (Ti = ct_array_read(T, i, N)) > 0)
  IO(!OPACIFY(Ti != Tj));
// or...
if(!OPACIFY(i >= N) && (Ti = ct_array_read(T, i, N)) > 0)
  IO(OPACIFY(false) ^ (Ti == Tj));
```

Listing 7.9: Absurd yet legal contextual inference of a pure opacification function

In this program, the compiler could infer most of the structure of the boolean opacification function due to contextual information. Specifically, for the I/O to be reachable,

- `OPACIFY` must return `false` for some inputs, or the first half of the condition can't be fulfilled;
- `OPACIFY` must also return `true` for some inputs, otherwise the constant-time array access function can only return 0, and thus the second half of the condition cannot be fulfilled.

As boolean opacification is a finite function of just two elements, returning both values deterministically means that `OPACIFY(true)` and `OPACIFY(false)` are opposite of each other.

This reveals the structure of the function, leading to the identities illustrated above. And thus, $(\mathbf{P}_4)$ can also be violated for a pure implementation. This also breaks $(\mathbf{P}_2)$ (liveness) indirectly.

Now, in the example above if the I/O is reached, the trace leading up to it must contain calls to both `OPACIFY(true)` and `OPACIFY(false)`, so it could be argued that the original opacification maps to one of these. I theorize that even this can be avoided by making the program conditionally valid on boolean opacification not being constant.

```c
/* Example with the forward-progress assumption */
IO(OPACIFY(b));
while(OPACIFY(b1) == OPACIFY(b2)) { /* ... pure code ... */ }

/* Example with undefined behavior */
IO(OPACIFY(b));
*(OPACIFY(b1) != OPACIFY(b2) ? &myint : (int *)NULL) = 73;
```

Listing 7.10: Global validity assumptions revealing a pure opacification function

These examples highlight the compiler's assumptions that loops with non-constant condition expressions and a pure body terminate [C23, 6.8.6.1§4][7], and that no undefined behavior occurs on any reachable paths. This again reveals the structure of the opacification function. In that case, the I/O could be reached without ever evaluating `OPACIFY()` with the intended value of `b`, bypassing $(\mathbf{P}_4)$ entirely. These concerns frequently come up in static analysis.

Insertion of arbitrary code. The requirement in the first application $(\mathbf{A}_1)$, that expressions not specified in the source program shouldn't be evaluated, is of course unrealizable as is, since the compiler can always insert dead code wherever it pleases. Opacifying expressions doesn't help with this as the compiler can still freely manipulate the program variables involved, in this case, in the masking scheme.

While this is mostly a theoretical exercise, traced types do constitute an extra barrier to this violation, as the values' representation is hidden and can only be manipulated by wrappers that are themselves traced. Unexpected insertions of traced code would trigger monitoring systems, thus detecting and handling this vulnerability as a bug.

7.4. Summary of expectations and threats for opacification

Table 7.11 summarizes the threats listed above against each base property or application for each type of opacification method. Overall, using a pure opacification function leaves quite a few theoretical angles of attack. Relying on side-effects is the universal but not very subtle fix, as everything can be constrained by the implicit chaining of side-effects as long as they're attached to the right program element. Wrappers help in multiple instances by attaching side-effects to operations instead of values. Traced types are occasionally useful for opacification, although their core value is for tracing, as was discussed in Section 5.2.3.

[7] The standard implies that this is valid even if the loop is not reached!

Base property	**Method**	**Threat identified**	**Risk level**
($\mathbf{P}_1$) Dependencies preserved	Pure	Identities	Unlikely
	Non-deterministic	Identities	Unlikely
	Side-effecting	Identities	Unlikely
	Wrappers	–	–
($\mathbf{P}_2$) Live opaques evaluated	Pure	Identities	Unlikely
		Function inference	Theoretical
	Non-deterministic	Identities	Unlikely
	Side-effecting	–	–
($\mathbf{P}_3$) Evaluate in dependency order	Pure	Identities	Unlikely
		Tabulate	Unlikely
		Speculation	Unlikely
		Global optimization	Theoretical
	Non-deterministic	Identities	Unlikely
		Speculation	Unlikely
	Side-effecting	Identities	Unlikely
		Speculation	Unlikely
	Wrapper	–	–
($\mathbf{P}_4$) Input values preserved	Pure	Function inference	Theoretical
	Side-effecting	–	–
Application	**Method**	**Threat identified**	**Risk level**
($\mathbf{A}_1$) Only source expressions	Pure	Arbitrary additions	Theoretical
	Non-deterministic	Arbitrary additions	Theoretical
	Side-effecting	Arbitrary additions	Theoretical
	Traced type	Add traced code	Bug
($\mathbf{A}_2$) Preserve constant-time	Pure	Tabulate	Unlikely
		Jump table	Unlikely
		Peephole	Realistic (case-by-case)
	Non-deterministic	Switch table	Unlikely
		Peephole	Realistic (case-by-case)
	Side-effecting	Switch table	Unlikely
		Peephole	Realistic (case-by-case)
	Wrapper	–	–

Table 7.11: Theoretical threats to properties and applications of opacification

7.5. Assumptions underlying the design of Tracing LLVM

So far, most of the violations observed in Tracing LLVM arise not from incorrect assumptions (as the threats are still theoretical), but from incorrect integration. The code published with this thesis still doesn't account for all of LLVM's representation details and its failures are usually because protections aren't end-to-end (e.g. wrappers don't survive past the instruction selector) or because I missed semantic details, leading to bugs.

Still, for the sake of a direct comparison with Section 7.2, let's briefly review the (currently unchecked) assumptions on which Tracing LLVM bases its security. Tracing LLVM's primitives are largely based on side-effects, which means they rely mostly on the opacity and implicit ordering associated with side-effects. A combination of value opacification (with the opacification function) and operation opacification (with wrappers) can deal with most of the threats described in Section 7.3. This leaves three primary assumptions:

- **Preservation of side-effect traces:** all side-effects on reachable paths must be preserved, and they must execute in the order specified by the source code. This is always true if the compiler is not bugged.
- **Implementation-chosen order for unsequenced side-effects:** not all side-effects are ordered in C, so I assume clang enforces an arbitrary but consistent order (details below).
- **Side-effect user dependency:** the assumption that uses of values returned by side-effects are evaluated after the side-effect takes place. This essentially assumes no speculation on values returned by side-effects.

Variations of the properties from Section 7.2 can then be derived from these assumptions:

→ $(\mathbf{P}_1)$: Uses of a value (even not opaque) in a side-effecting downstream dependency must be preserved in all traces.
  - The original $(\mathbf{P}_1)$ has an opaque input value and a pure user. By making the user side-effecting (using a wrapper), this becomes identical to $(\mathbf{P}_4)$.

→ $(\mathbf{P}_2)$: Live opaque side-effects must be evaluated in all traces
  - This follows from the preservation of side-effect traces.

→ $(\mathbf{P}_3)$: Dependencies in opaque chains constrain the evaluation order in all traces.
  - For chained side-effects, this follows from the preservation of traces. Pure computations in-between side-effects are synchronized by the user-dependency assumption.

→ $(\mathbf{P}_4)$: Opaque side-effects must be evaluated on their original inputs.
  - Always guaranteed because side-effects' arguments are part of the execution trace.

The side-effect sequencing assumption comes from a complication I didn't address, which is that the strict ordering of side-effects is not entirely specified in C and there are version differences. A question arises if multiple side-effects occur in-between two sequence points. A typical example would be multiple volatile accesses, or multiple side-effecting calls in different arguments of a function invocation.

```
volatile int x = 2, y = 4;
x + x;                      /* Unspecified in C99, UB in C11 */
x + y;                      /* Unspecified in C99 and C11 */
f(getchar(), getchar());   /* Unspecified in C99, maybe UB in C11 */
```

In such a situation the ordering of the side-effects is unspecified, both in C99 [C99, 6.5§3] and in C11 [C11, 6.5§3]. However, C11 makes it undefined behavior to have unsequenced side-effects on a single scalar variable [C11, 6.5§2], which in this case affects x+x and may affect double-getchar depending on its implementation.

I'm not aware of any LLVM IR instruction that can carry two source side-effects, so the lowering of the C AST to LLVM IR should fix an order implicitly, which must then be preserved. This means this problem could be solved simply by over-specifying and enforcing a canonical (implementation-defined) order in clang. So far I've been assuming that it follows one already.

The hardest assumption to check would be the side-effect user dependency; I believe this can be verified by data-flow analysis in the future.

7.6. Chapter conclusion

The main tool at our disposal to limit compiler interference is opacification. It's a very convincing option, being both easy to define (through new functions, instructions, types...) and easy to reason about (by instantiating opaque program elements with different implementations). However, it doesn't automatically overcome the semantic subtleties of the many languages involved in the toolchain, and there are many unlikely but legal transformations that could break desirable properties of opacification. Extracting guarantees will thus require a combination of stricter primitives (such as relying more on side-effects, which sacrifices some optimization potential) and experimental validation (which sacrifices some predictability).

→ What constraints should experimentally-verified guarantees satisfy?

In my opinion, the main goal is for these guarantees to be reliable when used in the real world, meaning they must be *complete* in their checks and should be *maintainable* in their integration with the compiler. Reducing the amount of code covered by experimental validation is desirable of course, but secondary.

→ What guarantees can or can't be extracted from opacification primitives?

In practice, LLVM will behave nicely if it doesn't know anything about a function, instruction, or type, so "reasonable" expectations will be met. Most of the time. When it comes to pure theory there are many unlikely transformations that can violate weaker forms of opacification, meaning that whatever guarantees can be extracted will likely come either from experimental validation or from a long chain of side-effecting statements.

Conclusion and future work 8

In this thesis, I set out to improve the reliability of software components of security countermeasures in two ways: aiming for the lowest-level fault models available, and enriching compilers with security tooling and preservation guarantees.

For dealing with low-level fault models, I showed in Chapter 4 that a software/hardware co-designed approach could resolve the threat of fetch skips, with a reasonable cost in complexity and time performance. Key to this effort, the countermeasure *provably* defeats all single-fault attacks and most multi-fault attacks as well. This high level of security combined with lower implementation costs than hardware countermeasures are important qualities of co-design.

As far a secure compilation goes, I first demonstrated in Chapter 3 that compilers are at the heart of software countermeasures (pretty much regardless of abstraction level) while also being unpredictable and difficult to control (beyond just optimizations) as a result of their complex design that only considers functionality. This makes secure compilation an area where improvements would propagate to the entire design and verification process for countermeasures, which is exciting!

I then detailed the design, implementation, and some validation elements of Tracing LLVM, my LLVM extension for security-aware compilation, in Chapters 5 to 7. Tracing LLVM currently provides basic features for program opacification, controlling lowerings, and tracing elements from the source program down to back-end level. These already enable some fine control of code generation for security purposes, which I tested and combined on small programs, up to a variation of the classical PIN verification function with multiple layered countermeasures.

Future work. There are many perspectives on both axes. The design space for co-designed countermeasures remains mostly unexplored; Chapter 4 demonstrated just one countermeasure where software and hardware supported each other. Compiler back-ends can be tuned in many more ways, for instance by controlling register allocation to limit spilling, by communicating information to hardware about sensitive values held in registers, or by avoiding indirect jumps. (Some of these would additionally benefit from tracing support!) Proof methods for these co-designed systems also deserve attention, as most modeling techniques focus either on software or hardware, but rarely both together. After all, the approach used in my operational semantics of assembler with fetch skips is unlikely to generalize well to other hardware aspects that aren't a direct function of executed code (like predictors with hidden state). This work could likely fill another thesis or two.

Security-aware compilation is where the clearest directions are, however. The current version of Tracing LLVM opens up a lot of directions for improvement:

- **Dynamic and runtime validation:** I discussed dynamic validation in Chapter 7; this can also be paired with validation at runtime, such as with a debugger. Immediate improvements in this department include porting over Vu's debugging tools, verifying unchecked assumptions during compilation, and adding test tools to verify the structural and semantic properties from Figure 3.2 on intermediate programs for unit testing.
- **Improved security primitives:** I showed a number of opacification and tracing primitives in Chapter 5. These provide basic functionality but there are a lot more elements worth tracing (such as memory accesses and conditional control flow) and more program elements that would benefit from opacification (such as control flow). There's also a strong industry incentive to maintain performance, which I've reduced by some unevaluated amount by switching back to side-effects from Vu's pure opacification function; a relevant follow-up project would be to quantify that and improve on it.
- **Tracing query interfaces:** Purely on the practical side, the tracing system still lacks a clear user-facing interface for programmatically querying and manipulating traced elements in the program. This falls in a general category of engineering improvements to encourage real-world usage.
- **Scaling up countermeasure and programs:** Chapter 3 envisioned a split between compiler design and countermeasure design. However, Tracing LLVM has only been tested so far on simple programs and countermeasures (where the evaluation is reading the intermediate and assembly outputs). To achieve the split and relieve security engineers from worrying about secure compilation, Tracing LLVM should be tested at the same scale as complex software countermeasures. That's likely also worth a thesis of its own.

And of course, the research questions of Chapter 6 on software engineering remain, relating to integration and maintainability that can be tested along the way.

Some of the bibliography relating to split compilation and traceability isn't limited to security; other possible targets include performance, worst-case execution time or to facilitate compiler engineering. It would be an interesting open direction to investigate if Tracing LLVM's increased control over compilation can help with these other non-functional requirements.

With that, I've reached the end of this security- and compiler-packed thesis. Onwards for more science!

Proof of the fetch skips hardening theorem A

This appendix completes the formalization and proof of Chapter 4. Section A.1 provides the full instruction semantics for Xccs, and Section A.2 proves the security theorems.

A.1. Detailed instruction semantics

Let's start by properly defining the semantics of instructions, which were omitted from the main text. For this, we have to give the functions:

$$\begin{aligned}
[\![\cdot]\!] &: (\mathrm{PC}, \sigma, \alpha) &&\mapsto (\mathrm{PC}', \sigma', \alpha') \text{ *or* } \bot \text{ *or* } \mathsf{end}(\alpha') \\
[\![\cdot]\!]_{ccs} &: (\mathrm{PC}, \sigma, \alpha, d) &&\mapsto (\mathrm{PC}', \sigma', \alpha') \text{ *or* } \bot \text{ *or* } \mathsf{end}(\alpha')
\end{aligned}$$

The two functions are similar in nature; $[\![\cdot]\!]_{ccs}$ is used when re-running Xccs instructions after their 32-bit checksum value (passed as 4th argument) has been found.

When giving instruction semantics, we assume assembler notations are encoded following the RISC-V ISA [RV1], into 16-bit values for mnemonics starting with "`c.`", 32-bit otherwise. "`i`" implicitly refers to the argument to $[\![\cdot]\!]$ and $[\![\cdot]\!]_{ccs}$. Unspecified members in the output σ'/α' are implicitly kept from the input σ/α.

Instructions that are not jumps, traps, or Xccs instructions have their natural semantics, with two small changes: they cannot run when CCSPROT is set to a non-zero address, and they update CCS as a side-effect. We illustrate this category with the add and c.add instructions, and the c.nop instruction that we use as padding while hardening.

$$[\![(\texttt{c.})\texttt{add r}_\texttt{d}\texttt{, rs}_1\texttt{, rs}_2]\!](\mathrm{PC}, \sigma, \alpha) =
\begin{cases}
\bot \text{ if } \sigma.\texttt{CCSPROT} \neq 0 \\
(\mathrm{PC} + \|\texttt{i}\|, \sigma', \alpha') \text{ otherwise, with} \\
\quad \sigma'.\texttt{CCS} = \sigma.\texttt{CCS} + \texttt{realign}(\mathrm{PC}, \texttt{u}_{32}(\texttt{i})) \\
\quad \alpha'[\texttt{r}_\texttt{d}] = \alpha[\texttt{rs}_1] + \alpha[\texttt{rs}_2]
\end{cases}$$

$$[\![\texttt{c.nop}]\!](\mathrm{PC}, \sigma, \alpha) =
\begin{cases}
\bot \text{ if } \sigma.\texttt{CCSPROT} \neq 0 \\
(\mathrm{PC} + \|\texttt{i}\|, \sigma', \alpha) \text{ otherwise, with} \\
\quad \sigma'.\texttt{CCS} = \sigma.\texttt{CCS} + \texttt{realign}(\mathrm{PC}, \texttt{u}_{32}(\texttt{i}))
\end{cases}$$

The CCS update adds the instruction's opcode to the CCS register, adjusted for PC alignment:[1]

$$\texttt{realign}(\mathrm{PC}, \texttt{i}) = \begin{cases} \texttt{i} \text{ if } \mathrm{PC} \equiv 0\ [4] \\ 2^{16}\texttt{LSH}(\texttt{i}) + \texttt{MSH}(\texttt{i}) \text{ otherwise} \end{cases}$$

Trapping instructions include c.ebreak which we treat as non-recoverable, and ecall which invokes a syscall whose number is specified in register a7 (x17). We treat an invocation of syscall __NR_exit (1) as program termination.[2]

$$\llbracket \texttt{c.ebreak} \rrbracket(\mathrm{PC}, \sigma, \alpha) = \bot$$

$$\llbracket \texttt{ecall} \rrbracket(\mathrm{PC}, \sigma, \alpha) = \begin{cases} \texttt{end}(\alpha) \text{ if } \alpha[17] = 1 \text{ and } \sigma.\texttt{CCSPROT} = 0 \\ \bot \text{ otherwise} \end{cases}$$

Jump instructions are modified the most by Xccs. Here we show jal (*jump-and-link*), which in RISC-V is used for unconditional jumps and function calls, and beq (*branch-if-equal*), a conditional branch. The extension to jalr (which is used to return from functions) is immediate by replacing the target PC + imm of jal with the value of a register in α.

$$\llbracket \texttt{jal } \texttt{r}_\texttt{d}\texttt{, imm} \rrbracket(\mathrm{PC}, \sigma, \alpha) = \begin{cases} \bot \text{ if } \sigma.\texttt{CCSPROT} \neq \mathrm{PC} \\ (\mathrm{PC} + \texttt{imm}, \sigma', \alpha') \text{ otherwise, with} \\ \quad \sigma'.\texttt{CCS} = 0 \\ \quad \sigma'.\texttt{CCSPROT} = 0 \\ \quad \sigma'.\texttt{CCSD.JO} = 0 \\ \quad \alpha'[\texttt{r}_\texttt{d}] = \mathrm{PC} + \|\texttt{i}\| + 2 \times \sigma.\texttt{CCSD.JO} \end{cases}$$

$$\llbracket \texttt{beq } \texttt{rs}_1\texttt{, } \texttt{rs}_2\texttt{, imm} \rrbracket(\mathrm{PC}, \sigma, \alpha) = \begin{cases} \bot \text{ if } \sigma.\texttt{CCSPROT} \neq \mathrm{PC} \\ (\mathrm{PC} + \texttt{imm}, \sigma', \alpha) \text{ if } \alpha[\texttt{rs}_1] = \alpha[\texttt{rs}_2] \text{, with} \\ \quad \sigma'.\texttt{CCS} = 0 \\ \quad \sigma'.\texttt{CCSPROT} = 0 \\ \quad \sigma'.\texttt{CCSD.JO} = 0 \\ (\mathrm{PC} + 4, \sigma', \alpha) \text{ otherwise, with} \\ \quad \sigma'.\texttt{CCS} = \sigma.\texttt{CCS} + \texttt{realign}(\mathrm{PC}, \texttt{u}_{32}(\texttt{i})) \end{cases}$$

There are two changes compared to traditional jumps: (1) it is only allowed when CCSPROT is set to PC; (2) it accounts for the *jump offset* (JO) field of CCSD to jump over trap barriers after function calls. The jump also prepares the execution of the next block by resetting CCS and other registers to 0. We do not allow the compiler to compress jal into the 16-bit version c.jal to make sure the next block is also aligned.

Conditional branches are similar, except that if the branch is not taken the block continues and in particular CCS is updated rather than reset.

The last type of instruction is Xccs instructions. Because these have both a 32-bit opcode and a 32-bit argument, they cannot be executed in a single step. A "delay slot" mechanism with σ.CCSDS and the Checksum-Delay-Slot rule is used to solve this issue in two steps. At the

[1] Computing the sum of realigned instruction opcodes ends up being equivalent to summing the values returned by fetch rules in the program, see Lemma 4. I choose to update during decoding because of the intuition that a CCS update during the fetch cycle would be impacted by the fault.

[2] We provide this exit mechanism to have a reasonable program model, but in practice the exit() function is in the non-protected libc, so we don't actually worry about protecting the last block.

first step, the Xccs opcode will be fetched and recorded in $\sigma.\mathtt{CCSDS}$. At the second step, the checksum value will be fetched, and passed as argument to the appropriate semantics function $[\![\sigma.\mathtt{CCSDS}]\!]_{ccs}$.

The first step which records the opcode in CCSDS proceeds as follows:

$$[\![\mathtt{ccs(b)}]\!](\mathrm{PC},\sigma,\alpha) = [\![\mathtt{ccscall(b)}\ \mathtt{N}]\!](\mathrm{PC},\sigma,\alpha) =$$
$$\begin{cases} \bot \text{ if PC is unaligned or } \sigma.\mathtt{CCSPROT} \neq 0 \\ (\mathrm{PC} + \|\mathtt{i}\|, \sigma', \alpha) \text{ otherwise, with} \\ \quad \sigma'.\mathtt{CCS} = \sigma.\mathtt{CCS} + \mathtt{realign}(\mathrm{PC}, \mathtt{u}_{32}(\mathtt{i})) \\ \quad \sigma'.\mathtt{CCSDS} = \mathtt{i} \end{cases}$$

The second step, which always uses the Checksum-Delay-Slot rule, reads back from CCSDS and calls the $[\![]\!]_{ccs}$ function by passing the checksum d just fetched from memory as an extra 4th argument. All 4 Xccs instructions have similar semantics; the -call variants set $\mathtt{CCSD.JO} = \mathtt{N}$ and the -b variants flip the least significant bit of the checksum value before comparing. We factor them as follows (d is a checksum value):

$$\mathsf{check}(\mathtt{mask} : \mathtt{u}_{32}, \mathtt{JO} : \mathtt{u}_5)(\mathrm{PC},\sigma,\alpha,d) =$$
$$\begin{cases} \bot \text{ if } \sigma.\mathtt{CCS} \neq (d \oplus \mathtt{mask}) \text{ or } \sigma.\mathtt{CCSPROT} \neq 0 \\ \bot \text{ if } d \text{ is not a valid checksum literal} \\ (\mathrm{PC} + 4, \sigma', \alpha) \text{ otherwise, with} \\ \quad \sigma'.\mathtt{CCSPROT} = \mathrm{PC} + 4 \\ \quad \sigma'.\mathtt{CCSDS} = 0 \\ \quad \sigma'.\mathtt{CCSD.JO} = \mathtt{JO} \end{cases}$$

$$\begin{aligned} [\![\mathtt{ccs}]\!]_{ccs} &= \mathsf{check}(0,0) \\ [\![\mathtt{ccsb}]\!]_{ccs} &= \mathsf{check}(1,0) \\ [\![\mathtt{ccscall}\ \mathtt{N}]\!]_{ccs} &= \mathsf{check}(0,\mathtt{N}) \\ [\![\mathtt{ccscallb}\ \mathtt{N}]\!]_{ccs} &= \mathsf{check}(1,\mathtt{N}) \end{aligned}$$

The “valid checksum literals” test closes attack that feed an incorrect checksum value by injecting a fault between the Xccs opcode and its checksum parameter. We avoid this by preventing a 32-bit value d from appearing verbatim as a checksum value if it decodes as a jump instruction, an Xccs instruction, or a pair of `c.ebreak`. The countermeasure gets rid of such values by flipping their LSB, which is shown to be correct in Lemma 1.

A.2. Hardening algorithm and proof of security

A.2.1. Additional definitions on blocks

The start address of a block bb is called blockAddr(bb). Since instructions in the same block are loaded contiguously, each block spans the interval

$$\mathtt{blockSpan(bb)} = \Big[\mathtt{blockAddr(bb)},\ \mathtt{blockAddr(bb)} + \sum_{\mathtt{i}\in\mathtt{bb}} \|\mathtt{i}\|\Big).$$

Programs being well-formed means that no two blocks’ spans intersect and every jump points to the blockAddr of some block.

Literals	Last bits	Note about flipped value
`c.jal`, `c.j`, `c.beqz`, `c.bnez`	`xxxxx.01`	RVC quadrant `00` contains no jumps and cannot be extended
`c.jr`, `c.jalr`, `c.ebreak`	`00000.10`	`00000.11` is a 32-bit load
Xccs instructions	`00010.11`	Quadrant `10` only has invalid literals at `00000.10`, and no extensions
`b*` (conditional branches)	`11000.11`	
`jalr`	`11001.11`	
`jal`	`11011.11`	

Figure A.1: Low bits of invalid checksum literals

A.2.2. Feasibility of Algorithm HARDEN

Algorithm 1 relies on the ability to get rid of invalid checksum literals by flipping their Least Significant Bit (LSB). We still have to show that this indeed works.

Lemma 1 (Invalid checksum literals can be avoided).
If d : u32 is an invalid checksum literal, then $(d \oplus 1) + 2^{14}$ is a valid checksum literal.

Proof. Invalid checksum literals are jumps, Xccs instructions, and pairs of `c.ebreak`. The low 7 bits of each invalid literal is listed in Table A.1; note how flipping the LSB never yields a new pattern that's present in the table.

It is important that the validity of the checksum literal can be inferred from the low 7 bits only. This is because flipping the LSB of the checksum literal also comes with an update to the Xccs opcode (replacing `ccs` by `ccsb` or `ccscall` by `ccscallb`). This update amounts to flipping bit 14 of the opcode, thus affecting bits 14–31 of the checksum value. Since bits 0–6 are not affected, the final checksum literal is still valid even after accounting for this change. □

A.2.3. Structure of hardened programs

The security property we want to establish is that injecting a single fault in the execution of a hardened program leads to termination with $\perp$ before the end of the current block. This relies on a particular structure for blocks, which we also use to formalize the hardening process.

Definition 5.
A source block is a block whose sequence of instructions consists of:

- *0 or more* **straight** *(*`add`, `ebreak`, `ecall`*...) or* **conditional branch** *(*`beq`*...) instructions, excluding* `c.ebreak` *and Xccs instructions; followed by*
- *One* **unconditional jump instruction** *(*`jal`*...).*

Unlike classical definitions, we do not count conditional branches as block terminators. This improves the correspondence between blocks and checksum regions; in particular, it ensures the invariant that `CCS` resets to 0 at the start of every block.

Definition 6.
A hardened block is a block hb *obtained by hardening a source block, i.e.* hb $=$ HARDEN(sb, N) *for*

some source block sb*.* HARDEN *is the same Algorithm 1 as before.*

The hardening algorithm processes instructions individually. All the original instructions are kept. Before each jump, a checksum check is added by procedure addChecksum(), in the form of an Xccs instruction (ccscall for function calls, ccs otherwise) followed by the reference checksum value. In order to close certain attack vectors, the checksum value must not decode as a jump or Xccs instruction; if that happens, the LSB of the checksum value is flipped and the Xccs instruction is replaced by its -b variant. Finally, a barrier of c.ebreak instructions is added at the end.

Now, conditional branches in the middle of a block allow for early exits, but don't reset the checksum. The idea is that upon exiting the block, the expected checksum value is always the sum of all lines from the beginning of the block to the exit instruction. It will nonetheless be useful to split the blocks at each exit point to help formalization.

Lemma 2 (Structure of source blocks).
A source block sb *can be uniquely decomposed into $m \geq 0$ early and one final section, as*

$$\mathsf{sb} = \mathsf{se}_1 + \ldots + \mathsf{se}_m + \mathsf{sf} \quad \text{(“+” is concatenation)}$$

where

- *each* se_k *(source early, $1 \leq k \leq m$) section consists of straight instructions followed by one conditional branch;*
- *the* sf *(source final) section consists of straight instructions and the block's unconditional jump.*

Proof. Because there is exactly one conditional branch per se_k and none other, m must be the number of conditional branches in sb. The grouping is straightforward from here. □

Lemma 3 (Structure of hardened blocks).
A hardened block $\mathsf{hb} = \mathsf{HARDEN}(\mathsf{se}_1 + \ldots + \mathsf{se}_m + \mathsf{sf}, N)$ *can be uniquely decomposed into 4-aligned sections*

$$\mathsf{hb} = \mathsf{he}_1 + \ldots + \mathsf{he}_m + \mathsf{hf}$$

where

- *each* he_k *(**hardened early**, $1 \leq k \leq m$) section consists of:*
  1. *the straight instructions of* se_k*;*
  2. *an optional* c.nop *or* nop*, in a way that* $1)$ *and* $2)$ *combined are not empty;*
  3. *a 4-aligned* ccs *or* ccsb*;*
  4. *a 4-aligned 32-bit checksum value;*
  5. *the conditional branch of* se_k*;*

- *the* hf *(**hardened final**) section consists of:*
  1. *the straight instructions of* hf*;*
  2. *an optional* c.nop *or* nop*, in a way that* $1)$ *and* $2)$ *combined are not empty;*
  3. *a 4-aligned* csscall *or* ccscallb *if the terminator is a call, a* ccs *or* ccsb *otherwise;*
  4. *a 4-aligned 32-bit checksum value;*
  5. *the terminator of* sf*;*

6. *$2N + 4$ `c.ebreak` instructions.*

Proof. Decomposition: The main (**for** `i`) loop in HARDEN is a morphism for concatenation. Let he_k/`hf` the hardened sequences corresponding to se_k/`sf` in the input; structurally, we get $\mathsf{hb} = \mathsf{he}_1 + ... + \mathsf{he}_m + \mathsf{hf}$.

Section contents: Straight instructions are clearly preserved by HARDEN. For both types of sections, items 2 through 5 are generated by the handling of the section's jump by HARDEN. The trap barrier is always added at the end of the block; we can count it towards `hf`.

Alignment: First, all sections have a length that is a multiple of 4. This constraint is ensured by adding a c.nop before each section's branch if the straight instructions finish on an unaligned address. The following Xccs instruction, checksum value, and branch instruction all occupy 4 bytes each. Then, because the block hb itself is 4-aligned and the sections follow each other in memory, each section individually must be 4-aligned. □

Finally, we give an intuitive view of the checksum values by proving that the per-instruction summing process with realignment is equivalent to summing 4-aligned lines. We can do this by comparing both computations on the underlying sequences of $\mathsf{u16}$ values.

Lemma 4 (`realign` sum computes the sum of fetched lines).
Let $h = (h_0, ..., h_{2n-1}) : \mathsf{u16}^{2n}$ a sequence of $2n$ $\mathsf{u16}$ ("halfwords") and $(w_0, ..., w_{n-1}) : \mathsf{u32}^{n}$ a sequence of $\mathsf{u32}$ ("words") whose concatenation is h, i.e. such that $w_i = h_{2i} + 2^{16} h_{2i+1}$.

Let $(\mathtt{i}_0, ..., \mathtt{i}_{m-1})$ a sequence of instructions, and write $\mathtt{offset}_j = \frac{1}{2}\sum_{k=0}^{j-1} \|i_k\|$ the offset of instruction j in the sequence (in 16-bit units). Assume the concatenation of this sequence is also h, i.e. $\mathtt{offset}_m = 2n$ and for all j,

- $\mathtt{i}_j = h_{\mathtt{offset}_j}$ *if $\mathtt{i}_j$ is a 16-bit instruction;*
- $\mathtt{i}_j = h_{\mathtt{offset}_j} + 2^{16} h_{\mathtt{offset}_j+1}$ *otherwise.*

Then,

$$\sum_{j=0}^{m-1} \mathtt{realign}(2 \cdot \mathtt{offset}_j, \mathsf{u32}(\mathtt{i}_j)) = \sum_{i=0}^{n-1} w_i.$$

Proof. First, rewrite every instruction's realigned contribution in terms of h_i; we have

$$\forall j,\ \mathtt{realign}(2 \cdot \mathtt{offset}_j, \mathsf{u32}(\mathtt{i}_j)) = \sum_{i=\mathtt{offset}_j}^{\mathtt{offset}_{j+1-1}} 2^{16(i \bmod 2)} h_i.$$

We can show this by developing the RHS in all four cases:

- $\mathtt{i}_j$ is 16-bit, $\mathtt{offset}_j \bmod 2 = 0$: $h_{\mathtt{offset}_j} = \mathtt{i}_j$
- $\mathtt{i}_j$ is 16-bit, $\mathtt{offset}_j \bmod 2 = 1$: $2^{16} h_{\mathtt{offset}_j} = 2^{16}\mathtt{i}_j$
- $\mathtt{i}_j$ is 32-bit, $\mathtt{offset}_j \bmod 2 = 0$: $h_{\mathtt{offset}_j} + 2^{16} h_{\mathtt{offset}_j+1} = \mathtt{i}_j$
- $\mathtt{i}_j$ is 32-bit, $\mathtt{offset}_j \bmod 2 = 1$: $2^{16} h_{\mathtt{offset}_j} + h_{\mathtt{offset}_j+1} = 2^{16}\mathsf{LSH}(\mathtt{i}_j) + \mathsf{MSH}(\mathtt{i}_j)$

Now, by summing from $j = 0$ to $m - 1$, we get

$$S := \sum_{j=0}^{m-1} \mathtt{realign}(2 \cdot \mathtt{offset}_j, \mathtt{u}_{32}(\mathtt{i}_j)) = \sum_{i=0}^{\mathtt{offset}_{m-1}} 2^{16(i \bmod 2)} h_i.$$

Recalling that $\mathtt{offset}_m = 2n$, this equivalent to unfolding w_i into a sum of $\mathtt{u}_{16}$:

$$S = \sum_{i=0}^{2n-1} 2^{16(i \bmod 2)} h_i = \sum_{i=0}^{n-1} w_i. \qquad \square$$

A.2.4. Program state upon leaving a hardened block

We will now show that the counter-measure ensures good properties in the execution of hardened blocks, namely that they can only be exited by a legitimate jump instruction and while validating the checksum associated to that jump.

In the following, we assume a program P and a successful execution $e = [s_1, ..., s_{|e|}]$, with each step being written $s_i = \langle \mathrm{PC}_i, \rho_i, \delta_i, \sigma_i, \alpha_i \rangle \to r_i$, which structurally implies that

$$\begin{cases} r_i = \langle \mathrm{PC}_{i+1}, \rho_{i+1}, \delta_{i+1}, \sigma_{i+1}, \alpha_{i+1} \rangle \text{ if } i < |e|; \\ r_{|e|} = \mathsf{end}(\alpha_{|e|}). \end{cases}$$

The first step in the proof is to show that the trap barrier prevents the end of a hardened block from being reached, meaning that any exit must happen through a jump instruction.

Lemma 5 (Hardened blocks must be exited by jumps).
Let hb *be a hardened block of P, and assume the execution enters* hb, *meaning that there exists t ("top") such that $PC_t = \mathsf{blockAddr}(\mathsf{hb})$.*

Let n be the number of instructions executed without leaving hb, *i.e. the largest number such that the following all hold:*
1. $t + n \leq |e|$;
2. $\forall i \in [t, t+n),\ PC_{i+1} \in \mathsf{blockSpan}(\mathsf{hb})$;
3. $\forall i \in [t, t+n),\ s_i$ *is not a jump instruction.*[3]

Then, either

- $t + n = |e|$ *(hence s_{t+n} is a successful termination by an invocation of the* `exit` *syscall), or*
- $t + n < |e|$ *and s_{t+n} is a jump instruction.*

Proof. By contradiction. An exit not covered by the lemma statement must be with $t+n < |e|$ and s_{t+n} not a jump instruction. As such, there must be no jump instruction in the entire sub-sequence $s_t \dots s_{t+n}$.

For a non-jumping step s_i, we always have $\mathrm{PC}_{i+1} > \mathrm{PC}_i$. This is because the only two changes

[3] For now we consider a forged jump that jumps back somewhere into the block as "leaving" the block; we will later show that this cannot happen.

to PC are the update by the instruction's semantics (which always adds $\|\texttt{i}\| > 0$) and the potential skip from the S32(k) fetch rule (which is $4k \geq 0$).

Therefore, $(\mathrm{PC}_i)_{t \leq i \leq t+n}$ is a strictly increasing sequence, which means that every address in blockSpan(hb) is part of exactly one $[\mathrm{PC}_i, \mathrm{PC}_{i+1})$ interval (i.e. there is exactly one step that consumes it as part of its execution).

So then, we can look at hb's ending, which has the following 4-aligned sequence (where each line covers 4 bytes of code):

```
    ccs/ccsb/ccscall/ccscallb
    <checksum value>
    j32
L:  c.ebreak; c.ebreak # (repeat N+2 times)
```

The j32 marker represents the jump instruction, which is always a 32-bit instruction. The trap barrier starts at address L and contains $2N + 4$ c.ebreak instructions, therefore the block ends at $\mathsf{L} + 4N + 8$.

From the previous argument, there is exactly one step s_i such that address L lies in the interval $[\mathrm{PC}_i, \mathrm{PC}_{i+1})$, meaning that the line at address L is loaded or skipped during the execution of step s_i. Step s_i starts at PC_i, which can be written in a unique way as either $\mathrm{PC}_i = \mathsf{L} - 4k$ or $\mathrm{PC}_i = \mathsf{L} - 4k - 2$ depending on its alignment.

Now by analyzing all cases for the value of k and alignment of PC, we can show that the execution always crashes at or shortly after s_i, completing the contradiction argument.

- $k > 0$. This case is only possible if s_i uses the fetch rule S32(k') because $\mathrm{PC}_{i+1} > \mathsf{L}$, yet all other options for non-jump instructions result in $\mathrm{PC}_{i+1} \leq \mathrm{PC}_i + 4$. Since $k' \leq N$, the fetch does not reach beyond the trap barrier and returns a pair of c.ebreak instructions. These get consumed by s_i's step rule, which we can identify from the alignment of PC.
  - PC aligned: s_i must be using either Aligned-16 or Checksum-Delay-Slot. In the first case, the first c.ebreak fetched by s_i is executed, leading to a trap. In the second case, the checksum check also traps because double c.ebreak is not a valid checksum literal.
  - PC unaligned: since there was a fetch, Unaligned-32 is being used. A 32-bit instruction is composed from MSH(δ_i) and the first c.ebreak. Recall that s_i is not a jump, so this instruction either crashes or leads into s_{i+1} with PC_{i+1} unaligned. In this case, s_{i+1} will use Unaligned-16 and run the second c.ebreak, trapping as claimed.
- $k = 0$. In this case, PC_i must be either L or $\mathsf{L} - 2$.
  - $\mathrm{PC}_i = \mathsf{L}$ (aligned). Here, s_i uses one of Aligned-16, Aligned-32, or Checksum-Delay-Slot, all of which require a fetch that can only return $\rho_i = $ j32 (if S&R32 is used) or a pair of c.ebreak. Using Aligned-16 and Aligned-32 would either crash from running a c.ebreak or contradict the hypothesis that s_i is not a jump. Using Checksum-Delay-Slot would also crash because both possible fetch results are invalid checksum literals.
  - $\mathrm{PC}_i = \mathsf{L} - 2$ (unaligned). Because there is a fetch, s_i must be using rule Unaligned-32. Much

like the similar case in $k > 0$, the instruction recomposed from $\mathtt{MSH}(\delta_i)$ and $\mathtt{LSH}(\delta_{i+1})$ (δ_{i+1} is the δ' from the rule) cannot be a jump, so if it doesn't crash execution continues to s_{i+1} with the new $\mathtt{LSH}(\delta_{i+1})$.

If s_i uses fetch rule NoFault or S32(k'), then δ_{i+1} is a pair of c.ebreak, therefore s_{i+1} will use Unaligned-16 and crash running the second c.ebreak.

Execution only proceeds past s_{i+1} if s_i uses fetch rule S&R32, in which case s_{i+1} is a repeat of s_i 4 bytes later ($\mathrm{PC}_{i+1} = \mathtt{L} + 2$). The same analysis as s_i applies, with two changes. First, S32(k') now reaches 4 bytes further, up to $\mathtt{L} + 4 + 4N$, which hits the last 4 bytes of the barrier. This shows why $2N + 4$ c.ebreak are required. Second, using S&R32 no longer succeeds because ρ_{i+1} is now also a pair of c.ebreak. Therefore, s_{i+2} crashes.

This guaranteed crash implies that a successful execution can only exit a protected block in one of the ways described by the theorem statement. □

The next objective is to show that the sequence of Xccs instruction, checksum and jump instruction is secure in that only *legitimate* jumps can be used to exit a hardened block. A key part of this is that only original instructions must be executed around the time of the jump, not forged instructions. This synchronization is guaranteed by the fact that an Xccs instruction is needed to jump, and Xccs instructions *cannot* be forged.

Lemma 6 (Xccs instructions can be repeated but not forged). *Let* $\mathsf{hb} = [\mathtt{i}_1, \ldots, \mathtt{i}_{|\mathsf{hb}|}]$ *be a hardened block and* $s = \langle PC, \rho, \delta, \sigma, \alpha\rangle \to \langle PC', \rho', \delta', \sigma', \alpha'\rangle$ *a step that successfully runs an Xccs opcode within* hb*, i.e. such that* $[PC, PC') \subseteq \mathtt{blockSpan}(\mathsf{hb})$*. Further assume that we don't repeat upon landing in* hb*, i.e. either* s *doesn't use rule* S&R32 *or* $PC \geq \mathtt{blockAddr}(\mathsf{hb}) + 4$.

Then, the instruction executed by s *is an original instruction of* hb*, in the sense that there is an Xccs instruction* $\mathtt{i}_j \in \mathsf{hb}$ *(with address* $\mathtt{L} = \mathtt{blockAddr}(\mathsf{hb}) + \sum_{k=0}^{j-1} \|\mathtt{i}_k\|$*) such that one of the following is true:*

1. *s uses fetch rule* NoFault *and* $PC = \mathtt{L}$*;*
2. *s uses fetch rule* S32(k) *and* $PC = \mathtt{L} - 4k$*;*
3. *s uses fetch rule* S&R32 *and* $PC = \mathtt{L} + 4$.

Proof. Xccs instructions trap when executed with unaligned PC, so they can only be run by step rule Aligned-32. This rule always uses the value δ returned by a fetch as an opcode, which is a 4-aligned value found in program memory at an address that depends on the fetch rule:

- With NoFault, δ is the value at PC;
- With S32(k), δ is the value at $\mathrm{PC} + 4k$;
- With S&R32, δ is the value at $\mathrm{PC} - 4$.

Given the hypotheses and the length of the trap barrier, this value must be a 4-aligned value within hb, which can intersect the instruction sequence in three ways:

1. δ matches a 32-bit instruction $\mathtt{i}_j \in \mathsf{hb}$: then the lemma is obviously true.

2. $\texttt{LSH}(\delta)$ is the last two bytes of an instruction $\texttt{i}_j \in \mathsf{hb}$ and $\texttt{MSH}(\delta)$ is the first two bytes of $\texttt{i}_{j+1}$. This is impossible because δ is an Xccs opcode so $\texttt{MSH}(\delta) = 0$. The 16-bit zero value is reserved in the RISC-V ISA as an invalid opcode, thus $\texttt{i}_{j+1}$ could not be a valid instruction.
3. δ matches a 32-bit checksum argument to an Xccs instruction: this is also impossible because Xccs opcodes are not valid checksum literals.

Since only case 1) is possible, s_i must indeed be executing a legitimate instruction $\texttt{i}_j \in \mathsf{hb}$ (maybe after a fault). □

Definition 7.
A state $\langle PC, \rho, \delta, \sigma, \alpha \rangle$ is called a legitimate entry into a block bb *if it satisfies $PC = \texttt{blockAddr}(\texttt{bb})$, $\sigma.\texttt{CCSPROT} = \sigma.\texttt{CCSDS} = 0$, and ρ does not decode as an Xccs instruction.*

Definition 8.
A sequence of steps $s_t \dots s_b$ ("top", "bottom") is defined as a legitimate execution of a hardened block hb *if the following conditions are met:*
- *s_t's initial state is a legitimate entry into* hb*;*
- *$\forall i \in [t, b),\ PC_{i+1} \in \texttt{blockSpan}(\texttt{hb})$;*
- *s_b is the only instruction in the sequence to be a taken jump or an invocation of the* `exit` *syscall.*

Lemma 7 (Jumps out of hardened blocks must be legitimate). *Let* hb $= [\texttt{i}_1, \dots, \texttt{i}_{|\mathsf{hb}|}]$ *a hardened block of P, and $s_t \dots s_b$ a legitimate execution of* hb *where s_b is a jump. Then:*

1. *$b \geq t + 2$ and s_{b-2} is an Xccs instruction;*
2. *The jump is legitimate, i.e. there is an instruction $\texttt{i}_j \in \mathsf{hb}$ such that $PC_b = \texttt{blockAddr}(\texttt{hb}) + \sum_{k=0}^{j-1} \|\texttt{i}_k\|$ and step s_b executes the opcode $\texttt{i}_j$.*
3. *The checksum is correct when leaving the block, i.e. $\sigma_b.\texttt{CCS} = [PC_b - 4]$ if it's a valid checksum literal, $[PC_b - 4] \oplus 1$ otherwise.*
4. *s_b's final state is a legitimate entry into another block.*

Proof. Walking back from the last few instructions, for s_b to be a jump and not trap, we must have $\sigma_b.\texttt{CCSPROT} \neq 0$. Therefore $b > t$ and s_{b-1} must use the Checksum-Delay-Slot rule, as no other type of step can *end* with $\texttt{CCSPROT} \neq 0$. This implies that $\sigma_{b-1}.\texttt{CCSDS} \neq 0$, so once again $b - 1 > t$ and s_{b-2} must execute an Xccs instruction, because CCSDS can only be non-zero for one step and no other instruction sets it.

By Lemma 6, there must be an original Xccs instruction from hb being executed by s_{b-2}, and such instructions are only found at the end of early or final sections, which have the following structure:

```
L:     ccs/ccsb/ccscall/ccscallb
L+4:   <checksum value>
L+8:   j32
L+12:  # ... next section or c.ebreak ...
```

Note that while s_{b-2} runs the opcode found in memory at address L, PC_{b-2} is not guaranteed to be L since faults might be involved. The lemma focuses on showing that even then, in all execution scenarios starting from s_{b-2} the legitimate jump at $\mathsf{L}+8$ must be taken by s_b while also passing the checksum at $\mathsf{L}+4$.[4]

First note that, as Xccs instructions set CCSPROT $=$ PC $+\ 8$ and both s_{b-2} and s_b already increment PC by 4, neither s_{b-1} nor s_b can fetch with S32(k) as that would cause PC $>$ CCSPROT in s_b and trap. The only variable after s_{b-2} is whether the S&R32 fetch rule is used to change either fetched value.

- If s_{b-2} uses either of the fetch rules NoFault and S32(k), then $\text{PC}_{b-1} = \mathsf{L}+4$ and $\text{PC}_b = \mathsf{L}+8$.

  Then, s_{b-1} uses the Checksum-Delay-Slot rule after fetching a checksum literal. Attacking the fetch with S&R32 would return the Xccs opcode of s_{b-2}, and trap because it's not a valid checksum literal. Thus s_{b-1} must fetch with NoFault, passing the checksum.

  Finally, s_b performs the jump. Fetching with S&R32 is again impossible as that would execute the checksum literal, which cannot be a jump. Therefore, s_b fetches with NoFault, and executes the intended jump at $\mathsf{L}+8$ while passing the intended checksum.

- If s_b uses the fetch rule S&R32, then $\text{PC}_b = \mathsf{L}+4$, which means that $\text{PC}_{b-1} = \mathsf{L}+8$ and $\text{PC}_b = \mathsf{L}+12$.

  This time, s_{b-1} must fetch with S&R32 to obtain the intended checksum, as a NoFault fetch would return the jump opcode, which is not a valid checksum literal.

  Then, s_b must also fetch with S&R32 to get the jump, since a NoFault fetch would the value at $\mathsf{L}+12$, which cannot be a jump.

Notice that in the second case the entire jump widget is "delayed" by 4 bytes by repeated attack with S&R32. This is still a legitimate exit for the purpose of this lemma. We will show later that in single-fault cases, this would invalidate the checksum.

Now, since the jump was executed properly, it is fairly easy to show that s_b is a legitimate entry into another block:

- The jump is to the blockAddr of another block as the program is assumed to be well-formed.
- After the jump, σ_b.CCSPROT $= \sigma_b$.CCSDS $= 0$ as a result of the semantics of s_{b-1} and s_b.
- Finally, ρ_{b+1} is either a jump, a c.ebreak, or the first line of the next section (which is never an Xccs instruction). □

A.2.5. Security guarantees

This key lemma implies that legitimate entries and exits from hardened blocks is an execution invariant. We can sum this up in the most general (multi-fault) case as every block guaranteeing its checksum must pass.

Theorem 1 (Security guarantee for multi-fault executions).
Let P a fully hardened program and $e = [s_0,\ \ldots,\ s_{|e|}]$ an execution such that

- *s_0 is a legitimate entry into a block of P;*
- *$s_{|e|}$ ends successfully, returning some* end(α).

[4] The analysis in this lemma remains correct even if rule S32(k) caused the execution of extra nop instructions, as that would reset CCSPROT to 0 and just prevent the jump altogether.

Then there exists a sequence $[\mathsf{hb}_1,\ \ldots,\ \mathsf{hb}_m]$ *of blocks of* P *such that*

1. e *can be partitioned into subsequences* $(s_{t_i} \ldots s_{b_i})_{1 \le i \le m}$ *each a legitimate execution of* hb_i*;*
2. *Each* s_{b_i} $(i \neq m)$ *is a legitimate jump of* hb_i *and* $\sigma_{b_i}.\mathsf{CCS}$ *is the correct checksum associated with that jump.*

Proof. By induction, constructing (t_i), (b_i) and (hb_i) along the way.

Assume s_{t_i} is a legitimate entry into hb_i (which is true for $i = 0$). By Lemma 5, there is b_i such that $s_{t_i} \ldots s_{b_i}$ is an execution of hb_1, with s_{b_i} leaving either by successful termination or by a jump. If s_{b_i} ends the program, we're done. Otherwise, by Lemma 7, s_{b_i} is a legitimate jump out of hb_i with $\sigma_{b_i}.\mathsf{CCS}$ passing the associated checksum, and its final state is a legitimate entry into another block hb_{i+1}. Define $t_{i+1} := b_i + 1$ and start over. □

As we've discussed previously, this doesn't completely rule out attacks, because for some blocks there exist multi-faulted paths whose checksum collides with the expected checksum. However, such paths do not exist when a single fault is injected during the execution of a block. To show this, we can go back to the sum-of-u$_{32}$ expression of the checksum and reason on the series of fetches that builds it.

Definition 9.
The fetch trace of a subsequence $s_t \ldots s_b$ *of* e *is the tuple*

$$(N, S, R, \rho_{\mathsf{entry}}, (a_i), (k_i), \mathsf{next})$$

where fetches are partitioned by rule type:

- $N \quad := \{i \mid s_i$ *uses the* NoFault *rule*$\}$
- $S \quad := \{i \mid s_i$ *uses the* $\mathsf{S32}(k)$ *rule*$\}$
- $R \quad := \{i \mid s_i$ *uses the* S&R32 *rule and* $i \neq t\}$
- $\rho_{\mathsf{entry}} := \rho_t$ *if* s_t *uses the* S&R32 *rule, 0 otherwise*

and relevant information is recorded as follows:

- *for* $i \in N \cup S \cup R$, a_i *is the address fetched by* s_i*;*
- *for* $i \in S$, k_i *is the parameter to* $\mathsf{S32}(\cdot)$ *(0 for other* i*);*
- next *gives the next step that includes a fetch, i.e.* $\mathsf{next}(i) = \min\ \{j \in N \cup S \cup R \mid j > i\}$.

Lemma 8 (Relation between checksum and fetch trace).
Let hb *a hardened block of* P *and* $s_t \ldots s_b$ *a legitimate execution of* hb*. Assume that the fetch trace of* $s_t \ldots s_{b-2}$ *(only up to the last Xccs instruction that counts towards the checksum) is recorded as* $(N, S, R, \rho_{\mathsf{entry}}, (a_i), (k_i), \mathsf{next})$.

Then fetched addresses are contiguous: $\forall i < b - 2,\ a_{\mathsf{next}(i)} = a_i + 4k_i + 4.$

Additionally, the checksum upon leaving the block is

$$\sigma_{b-1}.\mathsf{CCS} = \sum_{i \in N} [a_i] + \sum_{i \in S} [a_i + 4k_i] + \sum_{i \in R} [a_i - 4] + \rho_{\mathsf{entry}}.$$

Proof. The checksum upon leaving the block is $\sigma_b = \sigma_{b-1}$ (since CCS is not updated during checks and jumps). By instructions' semantics, σ_{b-1} is the realigned sum of the instructions executed by $s_t \dots s_{b-2}$ (i.e. the values being passed to $[\![\cdot]\!]$ or $[\![\cdot]\!]_{ccs}$ in each step rule).

Because the sequence takes no branch, each instruction i ends with PC ← PC + $\|\mathtt{i}\|$ so each new fetch queries the data immediately following the previous fetch. Accounting for skips in S32(k) means that $a_{\mathtt{next}(i)} = a_i + 4k_i + 4$ (k_i being 0 for other rules).

In addition, the sequence starts on a 4-aligned boundary and ends with a 4-aligned Xccs instruction, so the concatenation of executed instructions' encodings is exactly equal to the concatenation of values returned by fetches. Thus, by Lemma 4, σ_{b-1}.CCS is equal to the sum of values returned by fetches. These can be determined for all four categories of fetches:

- For $i \in N$, NOFAULT returns $[a_i]$;
- For $i \in S$, S32(k) returns $[a_i + 4k_i]$;
- For $i \in R$ ($i > t$), S&R32 returns ρ_i, which is always equal to $[a_i - 4]$ (because $\rho_i = [a_i]$ at the end of every step that includes a fetch);
- If s_t uses the S&R32 rule, then the fetch for s_t returns ρ_t (the last value fetched by the previous block), otherwise it's counted by N and S.

Adding these up yields the claimed formula for σ_{b-1}.CCS. □

Lemma 9 (Single faults always invalidate the checksum).
Let hb *a hardened block of P and $s_t \dots s_b$ a legitimate execution of* hb *ending in a jump. Assume the maximum number of skips allowed in a single* S32(k) *rule is $N = 1$. Then there cannot be exactly one fault attack during the execution of the block, i.e.*

$$\mathtt{Card}\ \{i \in [t, b] \mid s_i \textit{ uses } \mathtt{S32}(k) \textit{ or } \mathtt{S\&R32}\} \neq 1.$$

Proof. By Lemma 7, a legitimate exit by a jump requires an Xccs instruction at s_{b-2}, and s_{b-1} and s_b must either both use NOFAULT or both use S&R32. The second option immediately implies the theorem; this leaves the first.

Let $(N, S, R, \rho_{\mathtt{entry}}, (a_i), (k_i), \mathtt{next})$ be the fetch trace of $s_t \dots s_{b-2}$. Because s_{b-1} and s_b both use NOFAULT, the fault attack must occur in $s_t \dots s_{b-2}$, which leaves 3 options.

- s_t uses S&R32: by Lemma 8, the checksum at the end of the block is

  $$\sigma_{b-1}.\mathtt{CCS} = \sum_{i=t+1}^{b-2} [a_i] + \rho_{\mathtt{entry}},$$

  with $a_i = \mathtt{blockAddr}(\mathtt{hb}) + 4i$. Remember that the expected checksum is

  $$\sigma_{\mathtt{expected}} = \sum_{i=t}^{b-2} [\mathtt{blockAddr}(\mathtt{hb}) + 4i].$$

  The difference is $\rho_{\mathtt{entry}} - [\mathtt{blockAddr}(\mathtt{hb})] = \rho_t - [a_t]$, which is non-zero due to the condition on S&R32 (silent replacements do not count as faults). Therefore the checksum doesn't pass, contradicting the hypothesis that $s_t \dots s_b$ is a legitimate execution of hb.

- Some s_j uses S&R32 ($j \neq t$): by Lemma 8, the checksum is

$$\sigma_{b-1}.\mathsf{CCS} = \sum_{i=t}^{b-2} [a_i] - [a_j] + [a_j - 4]$$

  still with $a_i = \mathtt{blockAddr}(\mathsf{hb}) + 4i$. The difference with $\sigma_{\mathsf{expected}}$ is $[a_j - 4] - [a_j] = \rho_j - [a_j]$ which is again non-zero due to the condition on S&R32.

- Some s_j uses S32(1): still by Lemma 8, the checksum is now

$$\sigma_{b-1}.\mathsf{CCS} = \sum_{i \in N} [a_i] + [a_j + 4] = \sum_{i=t+1}^{b-2} [a_i] - [a_j],$$

  since the execution is offset by the skip at s_j, leading to $a_i = \mathtt{blockAddr}(\mathsf{hb}) + 4i + 4(i \geq j)$. The difference is $[a_j]$; for the checksum to pass we must have $[a_j] = 0$. This is impossible: for the same reasons as discussed in Lemma 6, a 4-aligned zero value cannot intersect any instructions, so it would have to be a checksum value... but then s_j would crash. This is because the execution was not faulted up to this point, so s_j must be using rule CHECKSUM-DELAY-SLOT with S32(1). As a result, s_j fetching $[a_j + 4]$, which is not a valid checksum literal (it's a 32-bit jump), leads to a crash. □

We are now finally able to prove the security property that a successful execution with no more than one fault per block has no faults at all.[5]

Theorem 2 (Security guarantee for single-fault executions).
Let P a fully hardened program and $e = [s_0, \dots, s_{|e|}]$ an execution such that

- *s_0 is a legitimate entry into a block of P;*
- *$s_{|e|}$ ends successfully, returning some end(α).*

Let $[\mathsf{hb}_1, \dots, \mathsf{hb}_m]$, $(s_{t_i} \dots s_{b_i})_{1 \leq i \leq m}$ the partition of e into block executions given by Theorem 1. If each segment $s_{t_i} \dots s_{b_i}$ uses at most one faulted fetch rule and the last segment $s_{t_m} \dots s_{b_m}$ contains no faults, then e contains no faults.

Proof. By Theorem 1, segments $i = 1$ to $m - 1$ all validate their checksums, and by hypothesis, they use at most one faulted fetch rule. By Lemma 9, they must in fact use no faulted fetch. The last block contains no fault by hypothesis. As a result, the entire execution is legitimate. □

[5] We exclude the last block since we don't protect ecall; in practice it's in non-protected libc and the entry into the exit() function is a function call that is itself protected.

Bibliography

[AA19] Misiker Tadesse Aga and Todd Austin. "Smokestack: Thwarting DOP Attacks with Runtime Stack Layout Randomization". In: *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2019, pp. 26–36. DOI: 10.1109/CGO.2019.8661202.

[Abr+21] Arnold Abromeit, Florian Bache, Leon A. Becker, Marc Gourjon, Tim Güneysu, Sabrina Jorn, Amir Moradi, Maximilian Orlt, and Falk Schellenberg. "Automated Masking of Software Implementations on Industrial Microcontrollers". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021, pp. 1006–1011. DOI: 10.23919/DATE51398.2021.9474183.

[Alm+17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. "Jasmin: High-Assurance and High-Speed Cryptography". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. New York, NY, USA, 2017, pp. 1807–1823. ISBN: 9781450349468. DOI: 10.1145/3133956.3134078. URL: https://doi.org/10.1145/3133956.3134078.

[Als+21] Ihab Alshaer, Brice Colombier, Christophe Deleuze, Vincent Beroulle, and Paolo Maistri. "Microarchitecture-Aware Fault Models: Experimental Evidence and Cross-Layer Inference Methodology". In: *16th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE. 2021, pp. 1–6.

[Als+22] Ihab Alshaer, Brice Colombier, Christophe Deleuze, Vincent Beroulle, and Paolo Maistri. "Variable-Length Instruction Set: Feature or Bug?" In: *25th Euromicro Conference on Digital System Design (DSD)*. Maspalomas, Spain. IEEE, 2022. ISBN: 978-1-6654-7405-4. DOI: 10.1109/DSD57027.2022.00068.

[Als+24] Ihab Alshaer, Brice Colombier, Christophe Deleuze, Vincent Beroulle, and Paolo Maistri. "Microarchitectural Insights into Unexplained Behaviors Under Clock Glitch Fault Injection". In: *Smart Card Research and Advanced Applications*. Ed. by Shivam Bhasin and Thomas Roche. Springer, 2024, pp. 3–22. ISBN: 978-3-031-54409-5.

[Als23] Ihab Alshaer. "Cross-Layer Fault Analysis for Microprocessor Architectures (CLAM)". Theses. Université Grenoble Alpes, Oct. 2023. URL: https://theses.hal.science/tel-04417620.

[Anc+17] Stéphanie Anceau, Pierre Bleuet, Jessy Clédière, Laurent Maingault, Jean-luc Rainard, and Rémi Tucoulou. "Nanofocused X-Ray Beam to Reprogram Secure Circuits". In: *Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2017, pp. 175–188. DOI: 10.1007/978-3-319-66787-4_9. URL: http://link.springer.com/10.1007/978-3-319-66787-4_9.

[Arc+06] Cédric Archambeau, Eric Peeters, François-Xavier Standaert, and Jean-Jacques Quisquater. "Template Attacks in Principal Subspaces". In: *Cryptographic Hardware and Embedded Systems (CHES)*. Ed. by Louis Goubin and Mitsuru Matsui. Berlin, Heidelberg: Springer, 2006, pp. 1–14. ISBN: 978-3-540-46561-4.

[Bar+12] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures". In: *Proceedings of the IEEE* 100.11 (2012), pp. 3056–3076. DOI: 10.1109/JPROC.2012.2188769.

[Bar+14a] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. "System-level Non-interference for Constant-time Cryptography". In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 1267–1279. ISBN: 9781450329576. DOI: 10.1145/2660267.2660283. URL: https://doi.org/10.1145/2660267.2660283.

[Bar+14b] Gilles Barthe, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Jean-Christophe Zapalowicz. "Synthesis of Fault Attacks on Cryptographic Implementations". In: *ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 1016–1027. ISBN: 9781450329576. DOI: 10.1145/2660267.2660304. URL: https://doi.org/10.1145/2660267.2660304.

[Bar+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. "Strong non-interference and type-directed higher-order masking". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016, pp. 116–129.

[Bar+17] Thierno Barry, Damien Couroussé, Bruno Robisson, and Karine Heydemann. "Automated Combination of Tolerance and Control Flow Integrity Countermeasures against Multiple Fault Attacks". In: *European LLVM Developers Meeting*. Saarbrücken, Germany, Mar. 2017. URL: https://hal.sorbonne-universite.fr/hal-01660160.

[Bay+13] Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. "Sleuth: Automated Verification of Software Power Analysis Countermeasures". In: *Cryptographic Hardware and Embedded Systems (CHES)*. Ed. by Guido Bertoni and Jean-Sébastien Coron. Berlin, Heidelberg: Springer, 2013, pp. 293–310. ISBN: 978-3-642-40349-1.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. "Correlation Power Analysis with a Leakage Model". In: *6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer. 2004, pp. 16–29.

[BCR16] Thierno Barry, Damien Couroussé, and Bruno Robisson. "Compilation of a Countermeasure Against Instruction-Skip Fault Attacks". In: *Workshop on Cryptography and Security in Computing Systems*. Vienna, Austria, Jan. 2016. DOI: 10.1145/2858930.2858931. URL: https://hal-cea.archives-ouvertes.fr/cea-01296572.

[BDG22] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. "Towards Effective Preservation of Robust Safety Properties". In: *37th ACM/SIGAPP Symposium on Applied Computing*. SAC '22. Virtual Event: Association for Computing Machinery, 2022, pp. 1674–1683. ISBN: 9781450387132. DOI: 10.1145/3477314.3507084. URL: https://doi.org/10.1145/3477314.3507084.

[Bel+13] Sonia Belaïd, Luk Bettale, Emmanuelle Dottax, Laurie Genelle, and Franck Rondepierre. "Differential Power Analysis of HMAC SHA-2 in the Hamming Weight

Model". In: *10th International Conference on Security and Cryptography (SECRYPT)*. Reykjavik, Iceland: Scitepress, July 2013. URL: https://inria.hal.science/hal-00872410.

[Bel+21] Nicolas Belleville, Damien Couroussé, Emmanuelle Encrenaz, Karine Heydemann, and Quentin Meunier. "PROSECCO: Formally-proven secure compiled code". In: *28th Computer Electronics Security Application Rendezvous (C&ESAR)*. Vol. 3056. ceur-ws.org. Nov. 2021, pp. 13–25. URL: https://cea.hal.science/cea-03605070.

[Bel05] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 41. DOI: 10.5555/1247360.1247401.

[Boe+23] Etienne Boespflug, Laurent Mounier, Marie-Laure Potet, and Abderrahmane Bouguern. "A compositional methodology to harden programs against multi-fault attacks". In: *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. Prague (hybrid), Czech Republic: IEEE Computer Society, Sept. 2023. DOI: 10.1109/FDTC60478.2023.00012. URL: https://hal.science/hal-04231496.

[Boh+18] Matthew Bohman, Benjamin James, Michael J Wirthlin, Heather Quinn, and Jeffrey Goeders. "Microcontroller compiler-assisted software fault tolerance". In: *IEEE Transactions on Nuclear Science* 66.1 (2018), pp. 223–232.

[Bon+23] François Bonnal, Vincent Dupaquis, Olivier Potin, and Jean-Max Dutertre. "Software-only control-flow integrity against fault injection attacks". In: *26th Euromicro Conference on Digital System Design (DSD)*. IEEE. 2023, pp. 269–277.

[C++11] JTC1/SC22/WG21. *Programming languages – C++*. ISO/IEC 14882. ISO, 2012. URL: https://www.iso.org/standard/50372.html. Document n3337 is an early draft after C++11: https://wg21.link/n3337.

[C++17] JTC1/SC22/WG21. *Programming languages – C++*. ISO, 2017. URL: https://www.iso.org/standard/68564.html. Document n4659 is the last draft of C++17: https://wg21.link/n4659.

[C11] JTC1/SC22/WG14. *Programming languages – C*. Cor. 1:2012. ISO/IEC 9899. ISO, 2011. URL: https://www.iso.org/standard/74528.html. Document n1570 is a late draft of C11: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[C23] JTC1/SC22/WG14. *Programming languages – C*. fifth. ISO/IEC 9899. ISO, 2024. URL: https://www.iso.org/standard/82075.html. Document n3220 is an early draft after C23: https://open-std.org/JTC1/SC22/WG14/www/docs/n3220.pdf.

[C99] JTC1/SC22/WG14. *Programming languages – C*. WG14 draft N1256 (with TC1–3). ISO/IEC 9899. ISO, 2007. URL: https://open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf.

[Cau+19] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. "FaCT: A DSL for Timing-Sensitive Computation". In: *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Phoenix, United States, June 2019. DOI: 10.1145/3314221.3314605. URL: https://hal.science/hal-02404755.

[Cau+20] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. “Constant-time foundations for the new spectre era”. In: *41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2020, pp. 913–926.

[CGZ+12] Jane Cleland-Huang, Orlena Gotel, Andrea Zisman, et al. *Software and systems traceability*. Vol. 2. 3. Springer, 2012.

[Cha94] Daniel Chandler. *The Sapir-Whorf Hypothesis*. 1994.

[Che+24] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W. Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. “GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers”. In: *USENIX Security*. 2024.

[Chi+12] Eduardo Chielle, Raul Sergio Barth, Angelo Cardoso Lapolli, and Fernanda Lima Kastensmidt. “Configurable tool to protect processors against SEE by software-based detection techniques”. In: *13th Latin American Test Workshop (LATW)*. IEEE. 2012, pp. 1–6.

[Chr+13] Maria Christofi, Boutheina Chetali, Louis Goubin, and David Vigilant. “Formal Verification of a CRT-RSA Implementation Against Fault Attacks”. In: *Journal of Cryptographic Engineering* 3.3 (2013), pp. 157–167. ISSN: 2190-8516. DOI: 10.1007/s13389-013-0049-3. URL: http://dx.doi.org/10.1007/s13389-013-0049-3.

[CR10] Albert Cohen and Erven Rohou. “Processor virtualization and split compilation for heterogeneous multicore embedded systems”. In: *47th Design Automation Conference (DAC)*. 2010, pp. 102–107.

[CRA06] Jonathan Chang, George A. Reis, and David I. August. “Automatic Instruction-Level Software-Only Recovery”. In: *International Conference on Dependable Systems and Networks (DSN’06)*. Philadelphia, PA, USA: IEEE, 2006, pp. 83–92. ISBN: 0-7695-2607-1. DOI: 10.1109/DSN.2006.15.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. “Template Attacks”. In: *Cryptographic Hardware and Embedded Systems (CHES)*. Ed. by Burton S. Kaliski, çetin K. Koç, and Christof Paar. Berlin, Heidelberg: Springer, 2003, pp. 13–28. ISBN: 978-3-540-36400-9.

[CSG16] David Costanzo, Zhong Shao, and Ronghui Gu. “End-to-end verification of information-flow security for C and assembly programs”. In: *SIGPLAN Notes* 51.6 (June 2016), pp. 648–664. ISSN: 0362-1340. DOI: 10.1145/2980983.2908100. URL: https://doi.org/10.1145/2980983.2908100.

[Cuo+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. “Frama-C: A software analysis perspective”. In: *International Conference on Software Engineering and Formal Methods (SEFM)*. Springer. 2012, pp. 233–247.

[CV17] Ruan de Clercq and Ingrid Verbauwhede. “A survey of Hardware-based Control Flow Integrity (CFI)”. In: *CoRR* abs/1706.07257 (2017). arXiv: 1706.07257. URL: http://arxiv.org/abs/1706.07257.

[DBP23] Soline Ducousso, Sébastien Bardin, and Marie-Laure Potet. “Adversarial Reachability for Program-level Security Analysis”. In: *32nd European Symposium on Programming (ESOP)*. Vol. 13990. Springer Nature. 2023, p. 59.

[De 19] F. De Ferrière. *A Compiler Approach to Cybersecurity.* https://llvm.org/devmtg/2019-04/slides/TechTalk-Ferriere-A_compiler_approach_to_cybersecurity.pdf. EuroLLVM. 2019.

[DPS15] Vijay D'Silva, Mathias Payer, and Dawn Song. "The Correctness-Security Gap in Compiler Optimization". In: *IEEE Security and Privacy Workshops.* 2015, pp. 73–87. DOI: 10.1109/SPW.2015.33.

[DS16] Moslem Didehban and Aviral Shrivastava. "nZDC: A compiler technique for near zero silent data corruption". In: *53rd Annual Design Automation Conference (DAC).* 2016, pp. 1–6.

[DSL17] Moslem Didehban, Aviral Shrivastava, and Sai Ram Dheeraj Lokam. "NEMESIS: a software approach for computing in presence of soft errors". In: *36th International Conference on Computer-Aided Design.* ICCAD '17. Irvine, California: IEEE Press, Nov. 2017, pp. 297–304.

[Dur+16] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. "FISSC: a fault injection and simulation secure collection". In: *Computer Safety, Reliability and Security.* Vol. 9922. Lecture Notes in Computer Science. Trondheim, Norway: Springer, Sept. 2016, pp. 3–11. DOI: 10.1007/978-3-319-45477-1_1. URL: https://hal.science/hal-03784107.

[Gau+23] Nicolas Gaudin, Jean-Loup Hatchikian-Houdot, Frédéric Besson, Pascal Cotret, Gogniat Guy, Guillaume Hiet, Vianney Lapotre, and Pierre Wilke. "Work in Progress: Thwarting Timing Attacks in Microcontrollers using Fine-grained Hardware Protections". In: *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW).* Delft, Netherlands, July 2023, pp. 1–7. URL: https://hal.science/hal-04155139.

[GCC25] GCC contributors. *How to Use Inline Assembly Language in C Code.* https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html. Accessed: 2025-04-10. 2025.

[Gei+23] Johannes Geier, Lukas Auer, Daniel Mueller-Gritschneder, Uzair Sharif, and Ulf Schlichtmann. "CompaSeC: A Compiler-Assisted Security Countermeasure to Address Instruction Skip Fault Attacks on RISC-V". In: *28th Asia and South Pacific Design Automation Conference.* ASPDAC '23. Tokyo, Japan: Association for Computing Machinery, Jan. 2023, pp. 676–682. ISBN: 9781450397834. DOI: 10.1145/3566097.3567925. URL: https://doi.org/10.1145/3566097.3567925.

[Gom+14] Kamil Gomina, Jean-Baptiste Rigaud, Philippe Gendrier, Philippe Candelier, and Assia Tria. "Power supply glitch attacks: Design and evaluation of detection circuits". In: *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST).* 2014, pp. 136–141. DOI: 10.1109/HST.2014.6855584.

[GP20] Marco Guarnieri and Marco Patrignani. *Contract-Aware Secure Compilation.* 2020. arXiv: 2012.14205 [cs.CR].

[Gra01] Graham, Paul. *Beating the Averages.* https://paulgraham.com/avg.html. Accessed: 2025-07-01. 2001.

[Gua+21] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. "Hardware-Software Contracts for Secure Speculation". In: *IEEE Symposium on Security and Privacy (S&P).* 2021, pp. 1868–1883. DOI: 10.1109/SP40001.2021.00036.

[Gut+01] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. “MiBench: A free, commercially representative embedded benchmark suite”. In: *Fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE. 2001, pp. 3–14.

[HLB19] Karine Heydemann, Jean-François Lalande, and Pascal Berthomé. “Formally verified software countermeasures for control-flow integrity of smart card C code”. In: *Computers & Security* 85 (Aug. 2019), pp. 202–224. DOI: 10.1016/j.cose.2019.05.004. URL: https://hal.sorbonne-universite.fr/hal-02123836.

[Hu+20] Wei Hu, Chip-Hong Chang, Anirban Sengupta, Swarup Bhunia, Ryan Kastner, and Hai Li. “An Overview of Hardware Security and Trust: Threats, Countermeasures, and Design Tools”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.6 (2020), pp. 1010–1038.

[Hut21] Rémi Hutin. “Compilation vérifiée et sécurisée contre les canaux cachés temporels”. Theses. École normale supérieure de Rennes, Dec. 2021. URL: https://theses.hal.science/tel-03616445.

[HWH13] Ralf Hund, Carsten Willems, and Thorsten Holz. “Practical Timing Side Channel Attacks against Kernel Space ASLR”. In: *IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 191–205.

[Jen14] Jenna Zeigen. *The Linguistic Relativity of Programming Languages*. https://2014.jsconf.eu/speakers/jenna-zeigen-the-linguistic-relativity-of-programming-languages.html. Accessed: 2025-07-01. 2014.

[KC01] Chandra Krintz and Brad Calder. “Using annotations to reduce dynamic optimization time”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2001, pp. 156–167.

[KHM20] Nermin Kajtazović, Peter Hödl, and Georg Macher. “Instrumenting Compiler Pipeline to Synthesise Traceable Runtime Memory Layouts in Mixed-critical Applications”. In: *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2020, pp. 73–78. DOI: 10.1109/ISSREW51248.2020.00040.

[Kia+21] Pantea Kiaei, Cees-Bart Breunesse, Mohsen Ahmadi, Patrick Schaumont, and Jasper van Woudenberg. “Rewrite to Reinforce: Rewriting the Binary to Apply Countermeasures against Fault Injection”. In: *58th ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 319–324. DOI: 10.1109/DAC18074.2021.9586278.

[Kim+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. “Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors”. In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), pp. 361–372. ISSN: 0163-5964. DOI: 10.1145/2678373.2665726. URL: https://doi.org/10.1145/2678373.2665726.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. “Differential power analysis”. In: *19th Annual International Cryptology Conference (CRYPTO)*. Springer. 1999, pp. 388–397.

[Koc+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *40th IEEE Symposium on Security and Privacy (S&P)*. 2019.

[Koc96] Kocher, Paul C. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *Advances in Cryptology (CRYPTO)*. Ed. by Koblitz, Neal. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113. ISBN: 978-3-540-68697-2.

[LA04] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *International symposium on code generation and optimization (CGO)*. IEEE. 2004, pp. 75–86.

[Lau+18] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. “On the Importance of Analysing Microarchitecture for Accurate Software Fault Models”. In: *21st Euromicro Conference on Digital System Design (DSD)*. Prague: IEEE, Aug. 2018. DOI: 10.1109/DSD.2018.00097. URL: https://hal.science/hal-01899800.

[Lau20] Johan Laurent. “Modélisation de fautes utilisant la description RTL de microarchitectures pour l’analyse de vulnérabilité conjointe matérielle-logicielle”. Theses. Université Grenoble Alpes, Nov. 2020. URL: https://tel.archives-ouvertes.fr/tel-03167493.

[Lem23] Matthieu Lemerre. “SSA translation is an abstract interpretation (with appendices)”. In: *Proceedings of the ACM on Programming Languages*. POPL 7 (Jan. 2023). DOI: 10.1145/3571258. URL: https://cea.hal.science/cea-03849637.

[Ler09] Xavier Leroy. “Formal Verification of a Realistic Compiler”. In: *Communications of the ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: https://doi.org/10.1145/1538788.1538814.

[Les+07] Piotr Lesnicki, Albert Cohen, Marco Cornero, Grigori Fursin, Andrea Ornstein, and Erven Rohou. “Split Compilation: an Application to Just-in-Time Vectorization”. In: *Workshop on GCC for Research in Embedded and Parallel Systems (GREPS)*. Brasov, Romania, 2007. URL: https://hal.science/hal-01257280.

[Li+20] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. “Debug Information Validation for Optimized Code”. In: *41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 1052–1065. ISBN: 9781450376136. DOI: 10.1145/3385412.3386020. URL: https://doi.org/10.1145/3385412.3386020.

[Lid+12] Jacob Lidman, Daniel J Quinlan, Chunhua Liao, and Sally A McKee. “Rose::Fttransform-a source-to-source translation framework for exascale fault-tolerance research”. In: *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN)*. IEEE. 2012, pp. 1–6.

[Lip+20] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, et al. “Meltdown: Reading kernel memory from user space”. In: *Communications of the ACM* 63.6 (2020), pp. 46–56.

[Low+20] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. “The gem5 simulator: Version 20.0+”. In: *arXiv preprint arXiv:2007.03152* (2020).

[LPR14] Hanbing Li, Isabelle Puaut, and Erven Rohou. “Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation”. In: *22nd International Conference on Real-Time Networks and Systems*. RTNS ’14. Versailles, France: Association for Computing Machinery, 2014, pp. 97–106. ISBN: 9781450327275. DOI: 10.1145/2659787.2659805. URL: https://doi.org/10.1145/2659787.2659805.

[Man+22] Noura Ait Manssour, Vianney Lapôtre, Guy Gogniat, and Arnaud Tisserand. “Processor Extensions for Hardware Instruction Replay against Fault Injection Attacks”. In: *25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. 2022, pp. 26–31. DOI: 10.1109/DDECS54261.2022.9770170.

[Man03] Stefan Mangard. “A Simple Power-Analysis (SPA) Attack on Implementations of the AES Key Expansion”. In: *5th International Conference on Information Security and Cryptology (ICISC)*. Springer. 2003, pp. 343–358.

[Mar18] Damien Marion. “Multidimensionality of the models and the data in the side-channel domain”. Theses. Télécom ParisTech, Dec. 2018. URL: https://pastel.hal.science/tel-02294004.

[MCG22] Tanmaya Mishra, Thidapat Chantem, and Ryan Gerdes. “Survey of Control-Flow Integrity Techniques for Real-Time Embedded Systems”. In: *ACM Transactions on Embedded Computer Systems* 21.4 (Oct. 2022). ISSN: 1539-9087. DOI: 10.1145/3538275. URL: https://doi.org/10.1145/3538275.

[MDG24] Sébastien Michelland, Christophe Deleuze, and Laure Gonnord. “From Low-Level Fault Modeling (of a Pipeline Attack) to a Proven Hardening Scheme”. In: *33rd International Conference on Compiler Construction*. CC 2024. , Edinburgh, United Kingdom, ACM, 2024, pp. 174–185. ISBN: 9798400705076. DOI: 10.1145/3640537.3641570. URL: https://hal.science/hal-04438994.

[MDG26] Sébastien Michelland, Christophe Deleuze, and Laure Gonnord. “Compiling Countermeasures Against Fault Attacks with Tracing LLVM”. In: *Workshop on Principles of Secure Compilation (PriSC’26)*. Rennes, France, Jan. 2026.

[Mon+23] David Monniaux, Léo Gourdin, Sylvain Boulmé, and Olivier Lebeltel. “Testing a Formally Verified Compiler”. In: *Tests and Proofs (TAP)*. Vol. 14066. Lecture Notes in Computer Science. Leicester, United Kingdom: Springer, July 2023, pp. 40–48. DOI: 10.1007/978-3-031-38828-6_3. URL: https://hal.science/hal-04096390.

[MZG24] Sébastien Michelland, Yannick Zakowski, and Laure Gonnord. “Abstract Interpreters: A Monadic Approach to Modular Verification”. In: *ACM on Programming Languages* 8.ICFP (Aug. 2024), pp. 1–28. DOI: 10.1145/3674646. URL: https://hal.science/hal-04628727.

[Noo+17] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. “Sancus 2.0: A low-cost security architecture for iot devices”. In: *ACM Transactions on Privacy and Security (TOPS)* 20.3 (2017), pp. 1–33.

[NT20] Kedar S. Namjoshi and Lucas M. Tabajara. “Witnessing Secure Compilation”. In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer, 2020, pp. 1–22. ISBN: 978-3-030-39322-9.

[ORK18] Kaan Onarlioglu, William Robertson, and Engin Kirda. “Eraser: Your data won’t be back”. In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, pp. 153–166.

[OSM02] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. “Control-flow checking by software signatures”. In: *IEEE transactions on Reliability* 51.1 (2002), pp. 111–122.

[Pad+06] Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. “Semantic Patches for Documenting and Automating Collateral Evolutions in Linux Device Drivers”. In: *Linguistic Support for Modern Operating Systems (PLOSS)*. San Jose, CA, Oct. 2006.

[Pal+11] Nicolas Palix, Gal Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. “Faults in Linux: Ten Years Later”. In: *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Newport Beach, CA, USA, Mar. 2011.

[Pes+25] Basile Pesin, Sylvain Boulmé, David Monniaux, and Marie-Laure Potet. “Formally Verified Hardening of C Programs against Hardware Fault Injection”. In: *14th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP ’25. Denver, CO, USA, 2025, pp. 140–155. ISBN: 9798400713477. DOI: 10.1145/3703595.3705880. URL: https://doi.org/10.1145/3703595.3705880.

[Pot+14] Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. “Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections”. In: *IEEE Seventh International Conference on Software Testing, Verification and Validation*. 2014, pp. 213–222. DOI: 10.1109/ICST.2014.34.

[PR13] Emmanuel Prouff and Matthieu Rivain. “Masking Against Side-Channel Attacks: A Formal Security Proof”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2013, pp. 142–159.

[Pro+17] Julien Proy, Karine Heydemann, Alexandre Berzati, and Albert Cohen. “Compiler-Assisted Loop Hardening Against Fault Attacks”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 14.4 (Dec. 2017). ISSN: 1544-3566. DOI: 10.1145/3141234. URL: https://doi.org/10.1145/3141234.

[QL11] Dan Quinlan and Chunhua Liao. “The ROSE source-to-source compiler infrastructure”. In: *Cetus users and compiler infrastructure workshop, in conjunction with PACT*. Vol. 2011. Citeseer. 2011, p. 1.

[QS01] Jean-Jacques Quisquater and David Samyde. “Electromagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards”. In: *International Conference on Research in Smart Card Programming and Security*. Springer. 2001, pp. 200–210.

[Ran23] Allison Randal. *This is How You Lose the Transient Execution War*. 2023. arXiv: 2309.03376 [cs.CR].

[RBV17] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. “Dude, is my code constant time?” In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. 2017, pp. 1697–1702. DOI: 10.23919/DATE.2017.7927267.

[Reb+01] Maurizio Rebaudengo, Matteo Sonza Reorda, Massimo Violante, and Marco Torchiano. “A source-to-source compiler for generating dependable software”.

In: *IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE. 2001, pp. 33–42.

[Rei+05] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. "SWIFT: software implemented fault tolerance". In: *International Symposium on Code Generation and Optimization (CGO)*. 2005, pp. 243–254. DOI: 10.1109/CGO.2005.34.

[RG24] Fabian Rauscher and Daniel Gruss. "Cross-core interrupt detection: Exploiting user and virtualized ipis". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2024, pp. 94–108.

[RT22] Fabrice Rastello and Florent Bouchez Tichadou. *SSA-based compiler design*. Springer, 2022.

[RV1] *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, Document Version 20191213*. Dec. 2019.

[SCA18] Laurent Simon, David Chisnall, and Ross Anderson. "What You Get is What You C: Controlling Side Effects in Mainstream C Compilers". In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018, pp. 1–15. DOI: 10.1109/EuroSP.2018.00009.

[She+21] Carlton Shepherd, Konstantinos Markantonakis, van Heijningen Nico, Driss Aboulkassimi, Clément Gaine, Thibaut Heckmann, and David Naccache. "Physical fault injection and side-channel attacks on mobile devices: A comprehensive analysis". In: *Computers & Security* 111 (Dec. 2021), p. 102471. DOI: 10.1016/j.cose.2021.102471.

[Shu+23] Amit Mazumder Shuvo, Tao Zhang, Farimah Farahmandi, and Mark Tehranipoor. "A comprehensive survey on non-invasive fault injection attacks". In: *Cryptology ePrint Archive* (2023).

[SLS19] Asanka Sayakkara, Nhien-An Le-Khac, and Mark Scanlon. "A Survey of Electromagnetic Side-Channel Attacks and Discussion on their Case-Progressing Potential for Digital Forensics". In: *Digital Investigation* 29 (2019), pp. 43–54.

[Tag11] Takanoki Taguchi. "A Prior Study of Split Compilation and Approximate Floating-Point Computations". MA thesis. INRIA-IRISA Rennes Bretagne Atlantique, June 2011. URL: https://dumas.ccsd.cnrs.fr/dumas-00636813.

[Tal+21] E. Bertrand Talaki, Mathieu Bouvier Des Noes, Olivier Savry, David Hely, Simone Bacles-Min, and Romain Lemaire. "Exposing Data Value On a Risc-V Based SoC". In: *IEEE Physical Assurance and Inspection of Electronics (PAINE)*. Nov. 2021, pp. 1–8. DOI: 10.1109/PAINE54418.2021.9707710.

[The+13] Nikolaus Theißing, Dominik Merli, Michael Smola, Frederic Stumpf, and Georg Sigl. "Comprehensive Analysis of Software Countermeasures against Fault Attacks". In: *Conference on Design, Automation and Test in Europe (DATE)*. DATE '13. Grenoble, France: EDA Consortium, 2013, pp. 404–409. ISBN: 9781450321532. DOI: 10.7873/DATE.2013.092.

[Thi+24] Jérémy Thibault, Roberto Blanco, Dongjae Lee, Sven Argo, Arthur Azevedo de Amorim, Aïna Linn Georges, Catalin Hritcu, and Andrew Tolmach. "SECOMP: Formally Secure Compilation of Compartmentalized C Programs". In: *arXiv preprint arXiv:2401.16277* (2024).

[Tol+22] Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. "Exploration of Fault Effects on Formal RISC-V Microarchitecture Models". In: *Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*. 2022, pp. 73–83. DOI: 10.1109/FDTC57191.2022.00017.

[Tri18] Alix Trieu. "Verifying constant-time implementations in a verified compilation toolchain". Thèse de doctorat dirigée par Blazy, Sandrine et Pichardie, David Informatique Rennes 1. PhD thesis. 2018. URL: http://www.theses.fr/2018REN1S099/document.

[Van+18] Jens Vankeirsbilck, Niels Penneman, Hans Hallez, and Jeroen Boydens. "Random additive control flow error detection". In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2018, pp. 220–234.

[VHM03] Rajesh Venkatasubramanian, John P Hayes, and Brian T Murray. "Low-cost on-line fault detection using control flow assertions". In: *IEEE On-Line Testing Symposium (IOLTS)*. IEEE. 2003, pp. 137–143.

[Vu21] Son Tuan Vu. "Optimizing Property-Preserving Compilation". Thèse de doctorat dirigée par Heydemann, Karine et Cohen, Albert Henri Informatique Sorbonne université. PhD thesis. Sorbonne Université, 2021. URL: http://www.theses.fr/2021SORUS435.

[Wan+17] Weijia Wang, Yu Yu, François-Xavier Standaert, Junrong Liu, Zheng Guo, and Dawu Gu. "Ridge-based DPA: Improvement of Differential Power Analysis for Nanoscale Chips". In: *IEEE Transactions on Information Forensics and Security* 13.5 (2017), pp. 1301–1316.

[Wan+22] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W Fletcher, and David Kohlbrenner. "Hertzbleed: Turning power Side-Channel attacks into remote timing attacks on x86". In: *31st USENIX Security Symposium (USENIX Security)*. 2022, pp. 679–697.

[Wen+19] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. "From Hack to Elaborate Technique—A Survey on Binary Rewriting". In: *ACM Computing Surveys* 52.3 (June 2019). ISSN: 0360-0300. DOI: 10.1145/3316415. URL: https://doi.org/10.1145/3316415.

[Who97] Benjamin Lee Whorf. "The relation of habitual thought and behavior to language". In: *Sociolinguistics: A reader* (1997), pp. 443–463.

[Win18] Hans Winderix. "Security Enhanced LLVM". MA thesis. KU Leuven, 2018. URL: https://downloads.distrinet-research.be/software/sancus/publications/winderix18thesis.pdf.

[WMP21] Hans Winderix, Jan Tobias Mühlberg, and Frank Piessens. "Compiler-assisted hardening of embedded software against interrupt latency side-channel attacks". In: *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2021, pp. 667–682.

[Woo+14] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. "The CHERI capability model: Revisiting RISC in an age of risk". In: *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014, pp. 457–468. DOI: 10.1109/ISCA.2014.6853201.

[Xia+25] Bing Xia, Yu Dong, Shihao Chu, and Chongjun Tang. “Survey of propagation strategy in taint analysis”. In: *4th International Conference on Big Data, Information and Computer Network*. 2025, pp. 760–765.

[YS18] Yuan Yao and Patrick Schaumont. “A Low-Cost Function Call Protection Mechanism Against Instruction Skip Fault Attacks”. In: *Workshop on Attacks and Solutions in Hardware Security*. ASHES ’18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 55–64. ISBN: 9781450359962. DOI: 10.1145/3266444.3266453. URL: https://doi.org/10.1145/3266444.3266453.

[Yuc+16] Bilgiday Yuce, Nahid Farhady Ghalaty, Harika Santapuri, Chinmay Deshpande, Conor Patrick, and Patrick Schaumont. “Software Fault Resistance is Futile: Effective Single-Glitch Attacks”. In: *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 2016, pp. 47–58. DOI: 10.1109/FDTC.2016.21.

[Zgh+22] Anthony Zgheib, Olivier Potin, Jean-Baptiste Rigaud, and Jean-Max Dutertre. “A CFI Verification System based on the RISC-V Instruction Trace Encoder”. In: *25th Euromicro Conference on Digital System Design (DSD)*. 2022, pp. 456–463. DOI: 10.1109/DSD57027.2022.00067.

List of Figures, Tables and Listings

Index