# On the Origins of Indirect Jumps in Embedded Software

Ariane Nicolas, Ronan Lashermes, Isabelle Puaut, Erven Rohou

# On the Origins of Indirect Jumps in Embedded Software

Ariane Nicolas
Univ Rennes - Inria - CNRS - IRISA
Rennes, France
ariane.nicolas@inria.fr

Ronan Lashermes
Rambus
Rotterdam, Netherlands
rlashermes@rambus.com

Isabelle Puaut
Univ Rennes - Inria - CNRS - IRISA
Rennes, France
isabelle.puaut@irisa.fr

Erven Rohou
Univ Rennes - Inria - CNRS - IRISA
Rennes, France
erven.rohou@inria.fr

## Abstract

Indirect control-flow transfers complicate control-flow graph (CFG) construction, thereby reducing the precision of static analyses and control-flow integrity (CFI) mechanisms in embedded systems. While previous work has primarily focused on resolving indirect jump targets, comparatively little attention has been devoted to understanding the reasons behind their generation. This paper presents a systematic empirical study of the origins of indirect jumps in compiled binaries. We introduce a taxonomy that characterizes the programming constructs and compiler transformations responsible for their generation. Our analysis encompasses C, C++, Fortran, and Rust programs compiled with GCC and LLVM at multiple optimization levels, targeting the 32-bit RISC-V instruction set. We then quantify the prevalence of each identified category over representative benchmarks and analyze differences across programming languages and compilation configurations. By clarifying the origins of indirect control transfers, this work provides insight into their impact on CFG precision and the static analysis of embedded software.

***CCS Concepts:*** • **Software and its engineering** → **Automated static analysis**; **Compilers**; **Language features**; • **Computer systems organization** → Embedded systems.

***Keywords:*** Indirect jumps, Static Analysis, Compilers, Embedded Systems, RISC-V

**ACM Reference Format:**
Ariane Nicolas, Ronan Lashermes, Isabelle Puaut, and Erven Rohou. 2026. On the Origins of Indirect Jumps in Embedded Software. In *Proceedings of the 27th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '26), June 15–16, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3814943.3816173


*LCTES '26, Boulder, CO, USA*

ACM ISBN 979-8-4007-2721-4/2026/06
https://doi.org/10.1145/3814943.3816173

## 1 Introduction

Embedded software is commonly subjected to static analyses to validate its correctness and provide guarantees such as worst-case execution time (WCET) or worst-case energy consumption (WCEC). In addition to these compile-time guarantees, runtime security mechanisms may be deployed to enforce execution integrity and resilience against attacks or faults. In particular, control-flow integrity (CFI) mechanisms ensure that program execution cannot deviate from its intended control flow.

A common prerequisite for both static analyses and CFI mechanisms is the availability of an accurate control-flow graph (CFG) of the program, representing all valid execution paths. For instance, static WCET estimation relies on exploring the CFG to identify the path of maximal execution cost, while some CFI mechanisms enforce execution to follow only valid paths in the CFG. Consequently, the precision of the CFG is crucial for both static analysis and security enforcement in embedded systems.

However, constructing a precise CFG is particularly challenging in the presence of indirect control-flow transfers, also known as indirect jumps [6]. Unlike direct branches, these instructions transfer control to an address stored in a register rather than to a statically encoded target. While they enable dynamic program behavior, their targets may vary at runtime and are often difficult to determine statically.

In presence of indirect jumps, CFG construction typically relies on over-approximations, such as grouping potential targets into equivalence classes [17]. Such approximations reduce the precision of the resulting CFG, potentially degrading the accuracy of static analyses and weakening the guarantees provided by CFI mechanisms.

While the analysis of indirect jumps has been widely explored in the contexts of static analysis and security, prior work has primarily focused on resolving their targets [3, 7, 9, 10, 19], rather than understanding the causes of their generation. There is currently no systematic investigation on their

origins, and specifically on the programming constructs and compiler transformations causing their generation during compilation. As a result, embedded software developers lack a clear understanding of when and why indirect jumps are introduced.

This paper fills this gap through a systematic empirical study of the generation of indirect jumps. Our goal is to provide a fine-grained classification of their origins and to quantify their prevalence in real-world software. We conduct our analysis across multiple programming languages (C, C++, Fortran, and Rust), compilers (GCC and LLVM), and optimization levels (-O0 to -O3, -Os, and -Oz).

The RISC-V instruction set architecture (ISA) was selected as the compilation target due to its growing adoption in embedded systems, along with its well-defined specification and fixed-size instruction encoding. This study is restricted to user-level (unprivileged) code; as such, the `ecall` system call instruction is not considered.

The contributions of this paper are twofold:

- We provide a taxonomy of the origins of indirect jumps, by identifying the programming constructs and compiler transformations causing their generation in RISC-V binaries.
- We provide an extensive empirical analysis of the frequency of indirect jumps across different programming languages, compilers, and optimization levels.

By clarifying the origins of indirect jumps, this study aims at giving keys for more informed development practices in the context of embedded systems, to support the construction of more precise control-flow graphs.

The remainder of this paper is organized as follows: Section 2 presents our experimental protocol. Section 3 details our taxonomy of indirect jump origins with illustrative examples. Section 4 provides quantitative results across languages, compilers and optimization levels. Section 5 discusses the scope and limitations of our study, analyzes its implications, and outlines directions for future work. Section 6 reviews existing work. Finally, Section 7 concludes this paper.

## 2 Experimental Protocol

This section presents our experimental protocol to identify indirect jumps origins. Our selected target is first described, followed by a description of the analyzed benchmarks and compilation setup. Finally, our classification methodology is detailed.

### 2.1 Target

The RISC-V Instruction Set Architecture (ISA) is built around a base instruction set that can be extended through standard extensions. The exact RISC-V target for which we are compiling our benchmarks is the 32-bit `riscv32-unknown-linux-gnu` architecture, which corresponds to the RV32I base ISA with the MAFD extensions, which are the `M` for *multiplications* and divisions, `A` for *atomics*, and `F` and `D` for simple precision and double precision floating point values. We chose not to use the `C` (Compressed instructions) extension, as it only impacts the instruction size.

The RISC-V architecture defines 32 integer registers, denoted `x0` to `x31`, which can also be referred to by their conventional names as specified in the RISC-V Application Binary Interface (ABI). In particular, register `x0`, named `zero` in the ABI, is hard-wired to zero. Register `x1`, or `ra`, serves as the standard return address register for function calls, although register `x5`, or `t0` may be used as an alternative return address register. The special-purpose register `pc` (program counter) is updated implicitly during instruction execution.

The RISC-V ISA supports two main classes of control-flow transfer instructions:

- Conditional branch instructions, which redirect control flow when a specified condition is satisfied and otherwise continue with sequential execution.
- Unconditional branch instructions, referred to as *jumps*, which always transfer control flow when executed.

Additionally, while branch instructions are always *direct* in RISC-V (transfer the control to a statically known destination), jumps can be differentiated between *direct* and *indirect* jumps, for which the destination address is stored in a register.

In the RISC-V ISA, indirect jumps are implemented through a single instruction, `jalr` (*jump and link register*), which follows the format `jalr rd, offset(rs1)`.

The `jalr` instruction modifies the value of register `pc`, redirecting the control flow to a new address, computed by adding the sign-extended immediate value `offset` to the value contained in register `rs1`. When used for a function call, the destination register `rd` is set to the return address of the function (`pc + 4` in the absence of the compressed extension). Using register `zero` as the destination register (`rd`) indicates that the `jalr` instruction performs a jump rather than a function call.

### 2.2 Benchmarks

To obtain a diverse and representative set of real-world software systems for analysis, we selected the SPEC CPU 2017 benchmark suite as our base dataset [16]. This benchmark suite includes multiple projects written in different compiled languages, covering a range of program sizes and application domains. We retained only the projects that could be successfully built for our target RISC-V 32-bit platform, which resulted in the exclusion of one benchmark, *Perl*, whose SPEC CPU 2017 implementation is officially not portable to RISC-V. Overall, the benchmark suite enabled the analysis of programs written in C, C++, and Fortran. We additionally analyzed the GNU C Library (glibc), as it is widely used and contains exotic code (e.g., handwritten assembly).

As no official benchmark suite is currently available for the Rust language, a dedicated set of benchmarks was assembled. The selection prioritized large-scale projects in order to promote diversity in indirect jump usages. The resulting Rust benchmark set includes *coreutils 0.4.0*, *ripgrep 15.0.0*, and *cargo 0.93.0*.

The selected benchmarks are presented in Table 1, along with their programming language and source code size in KLOC (thousands of lines of code). Benchmark entries that do not originate from the SPEC CPU2017 suite are highlighted in gray.

**Table 1.** Presentation of the analyzed benchmarks

| Benchmark name | Language | KLOC |
|---|---|---|
| lbm | C | 1 |
| mcf | C | 3 |
| nab | C | 24 |
| xz | C | 33 |
| x264 | C | 96 |
| imagick | C | 259 |
| glibc | C | 960 |
| gcc | C | 1 304 |
| namd | C++ | 8 |
| deepsjeng | C++ | 10 |
| leela | C++ | 21 |
| omnetpp | C++ | 134 |
| povray | C++, C | 170 |
| parest | C++ | 427 |
| xalan | C++ | 520 |
| blender | C++, C | 1 577 |
| exchange2d | Fortran | 1 |
| bwaves | Fortran | 1 |
| fotonik3d | Fortran | 14 |
| roms | Fortran | 210 |
| cactusBSSN | C++, C, Fortran | 257 |
| cam4 | Fortran, C | 407 |
| wrf | Fortran, C | 991 |
| ripgrep | Rust | 750 |
| coreutils | Rust | 1 311 |
| cargo | Rust | 1 611 |

## 2.3 Cross-Compilation

The benchmarks are cross-compiled for the RISC-V 32-bit target using both LLVM and GCC compilers whenever possible, across the optimization levels listed in Table 2. Compilation is performed with dynamically linked libraries. For Rust binaries, C libraries are dynamically linked, whereas other Rust *crates*[1] are statically embedded, as the target platform does not support dynamic Rust libraries (`dylib`).

Flang, the Fortran frontend of LLVM, does not yet support our selected RISC-V target; therefore, Fortran projects were compiled using GCC only. Conversely, no official GCC frontend is available for Rust, so Rust projects were compiled using *rustc*, which is based on LLVM.

[1]In Rust, a crate is the fundamental compilation and distribution unit of a program.

Additionally, glibc is only compiled using GCC, and, as stated in the documentation, it must be optimized to some level, thus cannot build for `-O0` [2].

Considering all available compiler and optimization configurations, the cross-compilation step produces twelve distinct binaries for each benchmark written in C and C++, six binaries for benchmarks written in Fortran and Rust, and five binaries for glibc.

**Table 2.** Levels of optimization

| Optimization level | Description |
|---|---|
| 0 | No optimization |
| 1 | Moderate optimization |
| 2 | Extensive optimization |
| 3 | Full optimization |
| s | Optimize for code size |
| z | Optimize aggressively for code size |

## 2.4 Identification Methodology

This section describes our classification process for tracing indirect jumps back to the programming constructs or compiler transformations responsible for their generation.

Our study combines disassembled code and source code analysis. Moreover, since the contribution focuses on the taxonomy of indirect jump origins, the classification process primarily relies on lightweight automated analyses, complemented by manual inspection when a deeper examination is required.

As shown in Figure 1, our classification methodology is organized into several successive identification passes, each described in the following subsections and illustrated with a brief example. As indirect jump origins are detailed in the next section, the examples provided here are purely illustrative.

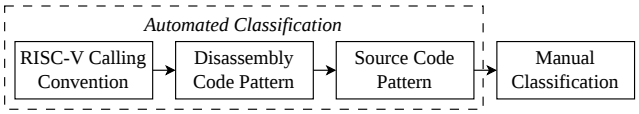


**Figure 1.** Classification Methodology.

**2.4.1 Use of RISC-V Calling Convention.** First of all, the RISC-V calling convention, defined in its application binary interface (ABI), provides an initial filter to determine indirect jumps origins. As stated in the RISC-V specification, the `jalr rd, offset(rs1)` instruction can have different behaviors depending on the value of its arguments:

- **Indirect function call,** with `rd` = `ra` or `t0`. The return register `rd` is properly set (`rd` ≠ `zero`), the instruction corresponds to a function call, also denoted by the pseudo-instruction `call`.

[2]See https://sourceware.org/glibc/wiki/FAQ#Why_do_I_get:.60.23error_.22glibc_cannot_be_compiled_without_optimization.22.27.2C_when_trying_to_compile_GNU_libc_with_GNU_CC.3F

- **Return,** with `offset=0`, `rs1` = `ra` or `t0` and `rd` = `zero`. The return register is not set, through the use of destination register `zero`, and the control flow is redirected to a previously stored return address. This corresponds to a function return, denoted by the pseudo-instruction `ret`.
- **Indirect jump,** with `rd` = `zero` and `rs1` ≠ `ra` or `t0`. No return address is set and the control flow does not jump to a previously saved location. This corresponds to a regular indirect jump. This is denoted by the pseudo-instruction `jr` (*jump register*).

Additionally, as RISC-V instructions use fixed-size encoding, direct jump instructions are limited to targets within a ±1 MiB range of the pc. Due to the limited range of direct jump instructions, reaching distant targets requires a two-instruction sequence based on indirect jumps. A 32-bit target address is constructed by first loading the 20 upper bits using either an `auipc` or `lui` instruction, and then completing the address with the 12 lower bits using `jalr`, which actually performs the jump. The combination of these two instructions constitutes a fourth type of indirect jump specific to fixed-size ISAs: **long-range jumps**.

The disassembled code is scanned to identify `jalr` instructions. Analysis of the instruction operands is sufficient to distinguish between returns, calls, and jumps. Returns are directly classified, whereas the origins of calls and jumps are further refined through more detailed analysis, as described below.

#### 2.4.2 Use of Pattern Matching on Disassembled Binary.

The origins of some indirect jumps can be identified easily in disassembled code as they generate typical instruction sequences. This is achieved through a backward analysis of the instructions preceding the jump, matching them against known *patterns*.

To illustrate this approach, we can consider the example of long-range jumps which produce a distinctive two-instruction sequence. If a `jalr` instruction is identified, preceded by an `auipc` or `lui` with matching registers, it is automatically classified as a *long-range jump*.

#### 2.4.3 Use of Source Code Information.

If no recognizable pattern is found in the disassembled binary, the analysis proceeds at the source code level. The source code line corresponding to the jump instruction is first obtained using the `addr2line` command from the GNU Binutils, which maps machine instruction addresses to source code locations using symbol and debugging information. The source line is then analyzed through pattern matching.

The most straightforward example of this process is for *jump tables* (arrays containing target addresses for jumps). Depending on the programming language, if the source code line contains the keyword `switch` (C, C++), `match` (Rust) or `select` (Fortran), it is classified as a jump through a *jump table*.

#### 2.4.4 Manual Classification.

When automated classification fails to detect the origin of a jump, its address is recorded for future manual analysis. Manual analysis relies on both disassembled binaries and source code to further characterize previously unclassified jumps.

For instance, certain indirect jumps originate from code defined within function-like or object-like macros. In C and C++, such macros are expanded at compile time, but their internal structure is opaque to debugging tools. As a result, `addr2line` returns only the macro invocation site rather than the exact line of the indirect jump. A manual inspection of the macro expansion is required to determine the origin of the jump.

## 3 Classification of Indirect Jumps Origins

This section presents the proposed taxonomy of indirect jump origins. Although multiple languages were analyzed, many similarities were observed across them, leading to a unified classification. When relevant, language-specific characteristics are highlighted. When applicable, we also indicate if a characterization is specific to the RISC-V architecture.

Figure 2 shows a summary of identified indirect jumps origins, grouped by the jump purpose.

The remainder of this section details each element of the proposed classification, with one subsection per broad category based on the RISC-V ABI: returns, long-range jumps, indirect calls, and indirect jumps. The origins of indirect calls are presented in the order in which they were identified during the classification process. In addition, each identified origin follows a uniform presentation, beginning with a summary table containing the following information:

- *Mode of identification*, according to our classification methodology (ABI, ASM pattern, source code pattern, or manual classification);
- *Language(s)* in which they can be found;
- *Source* (developer or compiler), indicating whether the construct results from a developer's choice (i.e., a programming pattern allowed by the language) or is generated by the compiler.

A detailed classification including how each indirect jump origin was identified is provided in Appendix A.

### 3.1 Origins of Returns

| ABI | All languages | Compiler |
|---|---|---|

Regardless of the programming language, a *return* instruction (`jalr zero,0(rs1)`, with `rs1` = `ra` or `t0`) is generated when a function or method completes. It performs an indirect jump to the return address previously stored in `rs1` by the *call* pseudo-instruction, transferring control back to the call site. The source constructs that trigger this instruction vary

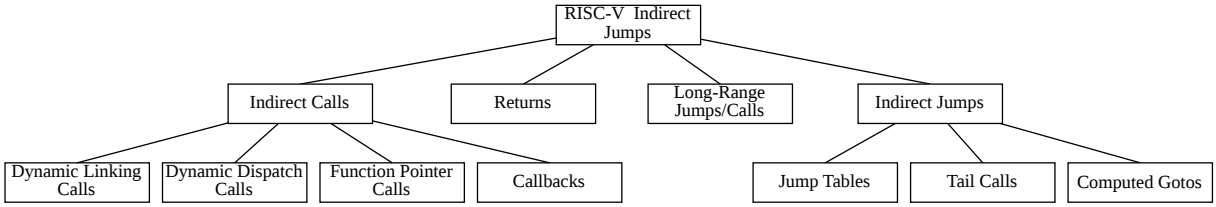


**Figure 2.** Indirect Jumps Categories.

across languages. In C, C++, and Rust, return instructions are generated when encountering the `return` keyword or when execution reaches the end of a function or method. In Fortran, they are produced at the `end subroutine` or `end function` statements. Additionally, in Rust, the value of the final expression is returned implicitly when the last expression of a block appears without a trailing semicolon.

### 3.2 Origins of Long-range Jumps/Calls

| ABI | All languages | Compiler |
|---|---|---|

As explained in Section 2.4.1, the RISC-V ISA allows jumps beyond the range of direct jump instructions through a two-instruction sequence: a `lui` or `auipc` instruction followed by an indirect jump `jalr`. We refer to this sequence as *long-range jumps*.

Consequently, many indirect jumps can appear in large RISC-V binaries. However, their targets are known at compile time, making them a distinct category of indirect jumps specific to fixed-size ISAs such as the RISC-V.

### 3.3 Origins of Indirect Calls

Indirect calls occur when the address of the function being called is only known at runtime. This situation most commonly arises from the use of function pointers, which allow function addresses to be stored in variables. The following paragraphs further refine the origins of indirect calls.

#### 3.3.1 Dynamically Linked Libraries.

| ASM pattern | All languages | Compiler |
|---|---|---|

Indirect function calls may arise from the use of dynamically linked library functions, whose addresses are not known at compile time. During compilation, a small code sequence —commonly referred to as a *stub*—is generated for each library function. These stubs are grouped in a dedicated region of the binary called the procedure linkage table (PLT). At runtime, when a library function is called for the first time, control is transferred through the PLT to the dynamic linker, which resolves the address of the requested function within the shared library. This address is then stored in a data region of the binary known as the global offset table (GOT). On subsequent calls, execution is redirected to the corresponding PLT stub, which retrieves the function address from the GOT and performs the indirect call. An example of a stub generated for the `printf` function from glibc is shown in Figure 3.

```
118a0 <printf@plt>:
118a0:   auipc t3,0xd34
118a4:   lw t3,1896(t3) # gets the address from the GOT
118a8:   jalr t1,t3     # performs the indirect call
118ac:   nop
```

**Figure 3.** Generated PLT stub for `printf`, obtained with `objdump`

#### 3.3.2 Dynamic Dispatch.

| ASM pattern | C++, Rust | Developer |
|---|---|---|

The concept of *polymorphism* allows the definition of abstract interfaces that can be implemented by multiple concrete types. Such interfaces may declare abstract methods that each implementation must provide. When a method is invoked through a reference typed as the interface, the implementation to execute is determined at runtime based on the concrete type of the underlying object. This mechanism is known as *dynamic dispatch*.

In C++, this mechanism is supported through the use of *virtual methods*. Consider the example of dynamic dispatch shown in Figure 4. A `Base` class is defined, containing two virtual methods `meth1` and `meth2`, followed by a `Derived` class that inherits from `Base` and overrides the `meth1` method.

```
class Base {
public:
    virtual void meth1() {
        printf("Base method 1\n");
    }
    virtual void meth2() {
        printf("Base method 2\n");
    }
};

class Derived : public Base {
public:
    void meth1() override {
        printf("Derived method 1\n");
    }
};

// Function taking a Base reference
void callMeth1(const Base& obj) {
    obj.meth1();  // dynamic dispatch happens here
}
```

**Figure 4.** Example of dynamic dispatch in C++

The compiler generates a hidden variable for each class, called the *virtual table* (vtable). This `vtable` is implemented

as an array of function pointers, each pointing to a method of the class. By default, the `vtable` is inherited from the parent class, and pointers are updated when methods are overridden. The state of the `vtables` for our example is illustrated in Figure 5.

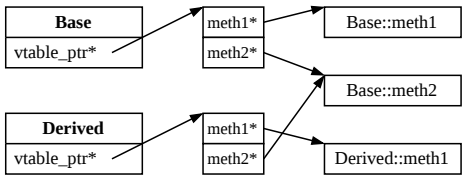


**Figure 5.** Virtual table mechanism.

At runtime, a virtual method can be invoked through a reference to the parent class, as illustrated by the `callFunc1` function in our example. The address of the appropriate concrete method implementation is obtained through a `vtable` lookup, resulting in an indirect call.

A call through a `vtable` follows a recognizable pattern: it begins by loading the `vtable` address from the object, then loading the appropriate method pointer using an offset, and finally performing an indirect call to the resolved method address. In disassembled code, this pattern is typically translated into a sequence of three instructions: two loads followed by an indirect call. An example of the sequence generated by GCC or LLVM can be found in Figure 6.

```
lw a0, 56(sp)    // a0 = vtable's address
lw a1, 12(a0)    // a1 = method's address
jalr ra, 0(a1)   // indirect call
```

**Figure 6.** Machine code generated for a vtable call.

In Rust, dynamic dispatch is achieved through the concept of a *trait*, which acts as an interface and can be implemented by multiple *structs*[3]. A trait can be passed as a function or method argument using a *dynamic reference* via the `dyn` keyword. Calls through dynamic references are performed via a vtable, and produce the same instruction sequence as the one presented in Figure 6.

Rust additionally provides the concept of *closure*, anonymous functions written inline, that can capture variables from their surrounding scope. In Figure 7, a closure is defined in function `main`, and captures the variable `factor`. Closures can be stored under a `Fn`, `FnMut` or `FnOnce` trait, to be for instance passed in argument. This mechanism is shown in Figure 7, where a dynamic reference of the closure is passed to the `callClosure` function. Invoking such closures results in dynamic dispatch, and generates the instruction pattern presented in Figure 6.

### 3.3.3 Function Pointers.

| Source Code Pattern | All languages | Developer |
|---|---|---|

[3]In Rust, a *struct* is a composite data type that groups related data fields.

```
fn callClosure(func: &dyn Fn(i32) -> i32, x: i32) -> i32 {
    func(x) // dynamic dispatch
}

fn main() {
    let factor = 3;
    let closure = |n| n * factor;
    let result = callClosure(&closure, 5);
}
```

**Figure 7.** Example of a closure in Rust.

In many programming languages, the address of a function can be stored in a variable, forming a *function pointer*. In C++, function pointers are declared using pointer syntax with the `*` operator. In Rust, function pointers can be obtained by casting plain functions or non-capturing closures to the `fn` type. Function pointers can serve several common purposes, that are listed below.

***Function pointers within structures.*** Function pointers may be stored within structures. Through this mechanism, they can act as methods for their enclosing object, facilitating direct invocation from the structure instance.

***Arrays of function pointers.*** Function pointers can also be stored in arrays. This construct is typically employed to iterate over multiple functions sharing the same signature, or to dynamically select the function to invoke based on an index, similarly to a jump table mechanism.

***Void pointers.*** In C and C++, casting a function pointer to `void*` can be used to hide the function type, a technique commonly referred to as *type erasure*. This enables passing function pointers regardless of their type. Before invocation, however, the pointer must be cast back to its original type.

***Type-erased callable wrappers (C++ only).*** C++ provides the `std::function` class, a generic function wrapper capable of storing any callable entity of C++, such as function pointers, method pointers, lambda expressions, or functor objects. This class relies on internal type erasure based on `void*` pointers, hiding the concrete type of the stored callable while preserving a uniform interface. Consequently, invoking a callable stored in a `std::function` results in an indirect call.

### 3.3.4 Callbacks.

| Manual classification | All languages | Developer |
|---|---|---|

When a function pointer is given as an argument to a function, delegating its invocation to this function, it is referred to as a *callback*. This construct enables to implement handler-based designs, for instance to react to user interaction in graphical interfaces.

A classical example is the `qsort` function from the C standard library, which sorts an array according to a comparison function given in argument. When calling this given function, an indirect call is generated.

A callback can be implemented through different language constructs:

- Function pointers in C, C++ and Rust,
- Closures under `Fn`, `FnMut` or `FnOnce` traits in Rust, resulting in dynamic dispatch,
- Callables stored in the `std::function` type in C++,
- Functions declared with the `external` keyword in Fortran.

### 3.4 Origins of Indirect Jumps

Indirect jumps redirect the control flow to a destination determined at runtime, without setting a return address. Several patterns causing their generation have been identified.

#### 3.4.1 Jump Tables.

| Source code pattern | All languages | Compiler |
|---|---|---|

Jump tables are generated by compilers to optimize long sequences of conditional branches, such as those produced by switch-like constructs. When encountering the keyword `switch` in C or C++, `match` in Rust, or `select` in Fortran, the compiler evaluates whether the set of case values is sufficiently dense. If so, it emits a *jump table*, which contains the addresses of the code blocks associated with each case. The switch input value is then used as an index to retrieve the corresponding case address from the table.

This optimization results in a characteristic instruction sequence, as illustrated in Figure 8. The input value `a1` is first multiplied by the instruction size in bytes to compute the byte offset within the jump table. The base address of the jump table is then computed using a combination of `lui` and `addi` instructions. The target address is obtained by adding the offset to the table base address, after which the effective jump destination is loaded from memory. Finally, an indirect jump is performed.

```
slli  a1, a1, 2    // offset = input * 4
lui   a2, 0xc4
addi  a2, a2, 84   // base jump table address
add   a2, a2, a1   // case address = base + offset
lw    a3, 0(a2)    // a3 = *(case address)
jr    a3           // indirect jump
```

**Figure 8.** Machine code generated for a jump table by GCC

The exact generated instruction pattern and ordering may vary across compilers, which motivated us to identify jump tables at the source-code level.

During our experiments, we observed that both GCC and LLVM may also generate jump tables from large `if-elif-else` sequences involving comparisons between the same variable and constant values. Additionally, when a loop body in C++ only contains a switch statement, the compiler may optimize it as a jump table, in which each case transfers control back to the beginning of the loop.

#### 3.4.2 Indirect Tail Calls.

| ASM pattern + Source code pattern | All languages | Compiler |
|---|---|---|

When the final statement of a function is a call to another function, the compiler may generate a `jump` instruction instead of a `call` instruction directly followed by a return. This optimization, known as a *tail call optimization*, allows to keep the caller's stack frame instead of allocating a new one.

If the last statement in a function is an indirect call (for instance through a function pointer or dynamic dispatch), the resulting jump is likewise indirect. An example of situation where an indirect tail call would be generated is presented in Figure 9.

```
int func1(int x, int (*fptr)(int)) {
    x = x * 2;
    return fptr(x); // <- call optimized as an indirect jump
}
```

**Figure 9.** C code eligible for tail call optimization

Before jumping to the called function, the caller's stack frame must be deallocated by incrementing the stack pointer. A tail call optimization thus generates a characteristic sequence of an `addi` instruction incrementing the stack pointer, followed by an indirect jump.

When the called function has the same stack frame layout as the caller—meaning it has the same return type and its arguments occupy the same space—the optimization is referred to as a *sibling call*. In particular, when a function invokes itself in tail position, it is known as a *recursive tail call*.

#### 3.4.3 Computed Gotos.

| Manual classification | C, C++, Fortran | Developer |
|---|---|---|

In C, C++ and Fortran, the `goto` statement allows to perform an unconditional jump to a labeled location. Fortran additionally provides a *computed goto* construct, that allows to select a label from a list, based on the value of a given integer. This mechanism is equivalent to a jump table, where an index is used to access a table entry and transfer control to the corresponding address.

In C and C++, a non-standard GCC extension known as *labels as values* provides similar functionality. This extension allows the address of a label to be obtained using the `&&` operator and enables indirect jumps to that label address (see Figure 10). Depending on the compiler version, a label address may even be passed as a function argument, allowing jumps to labels located in other functions.

During our investigations, this construct has only been encountered within `glibc`.

```
// computed goto code from glibc, vfprintf-internals.c
void jump(unsigned char f, void **table) {
    void *ptr;
    ptr = table[f];
    goto *ptr;
}
```

**Figure 10.** Example of a computed goto source code in GCC

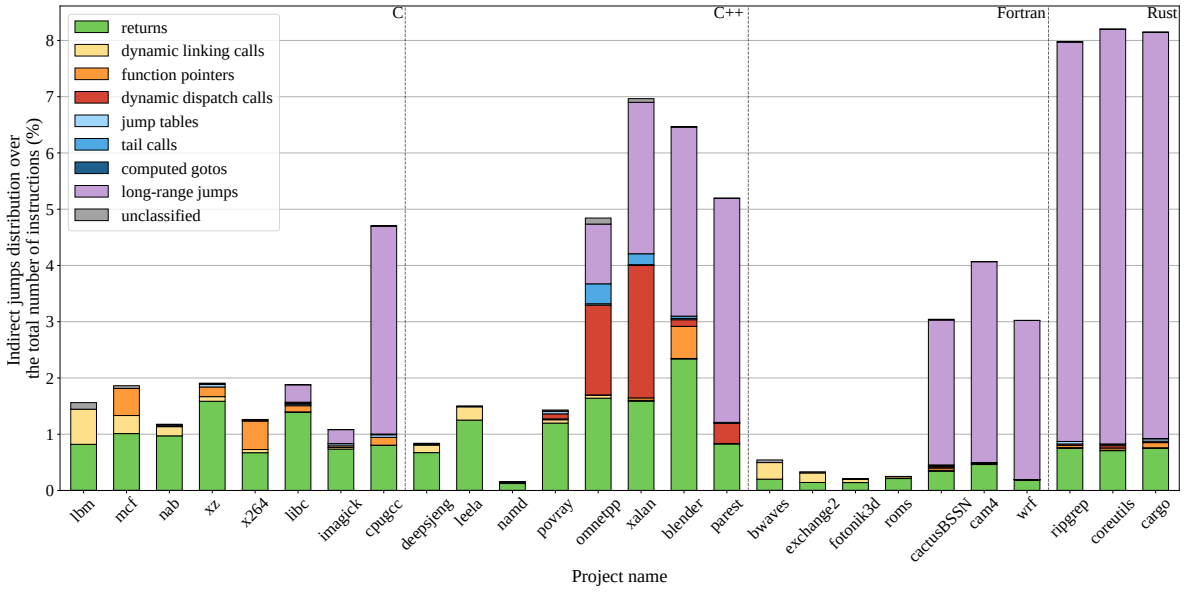


**Figure 11.** Distribution of indirect jumps over the total number of instructions at -O2.

## 4 Proportion of Indirect Jumps

This section presents the results of our empirical study, reporting the proportion of indirect jumps detected statically in RISC-V binaries. The objective is to highlight the differences in the proportions of generated indirect jump across programming languages, compilers, and optimization levels.

First, a single compilation configuration is considered to enable comparisons across programming languages. We then focus specifically on indirect calls, and finally examine the impact of compiler optimization levels.

### 4.1 All Indirect Jumps

To emphasize the differences between programming languages, we first restrict our analysis to a single compilation configuration. We select optimization level `-O2` (see Table 2), as it is commonly used in production environments. For the following graphs, studied benchmarks are compiled with GCC for C, C++ and Fortran projects, and with Rustc (LLVM) for Rust projects.

Figure 11 presents the stacked distribution of indirect jumps origins per compiled benchmark. For each origin, the reported percentage corresponds to the number of jumps divided by the total number of instructions in the binary.

Origins of indirect jumps are distinguished by color: long-range jumps in purple, returns in green, indirect calls in warm colors and indirect jumps in blue tones. Benchmarks are grouped by programming language, and for a given language are ordered by the total number of instructions in the binary.

A notable observation concerns larger binaries, which exhibit a higher percentage of long-range indirect jumps, which is expected since they are generated to replace direct jumps whose target address lies outside the ±1 MiB range relative to the `pc`.

This effect is particularly pronounced in Rust benchmarks, where long-range jumps account for up to 7 % of all generated instructions. This behavior is explained by Rust's default linker configuration, which does not enable *relaxation*. During compilation, calls are conservatively emitted as long-range indirect jumps. If their target falls within the reachable range, they may be "relaxed" by the linker into direct jumps. As this optimization is disabled by default in Rust's linker, most long-range jumps remain unchanged in the final binary. When linker relaxation is explicitly enabled, their proportion decreases significantly, reaching approximately 3–4 %, which is consistent with the values observed in the other benchmarks.

Moreover, we can observe that smaller benchmarks exhibit limited diversity in indirect jump categories, mainly comprising returns, dynamic linking calls, and occasionally jump tables. Larger projects require more abstractions such as interfaces or callbacks, consequently generating a broader variety of indirect jumps.

It is also worth noting that, excluding long-range jumps, the overall proportion of indirect jumps across all categories remains relatively low.

Finally, some jumps remain unclassified, mainly due to missing debug information in certain benchmarks, which prevented automated or manual analysis at the source code level.

### 4.2 Indirect Calls

We now focus specifically on indirect calls to analyze the impact of language paradigms and developer choices.

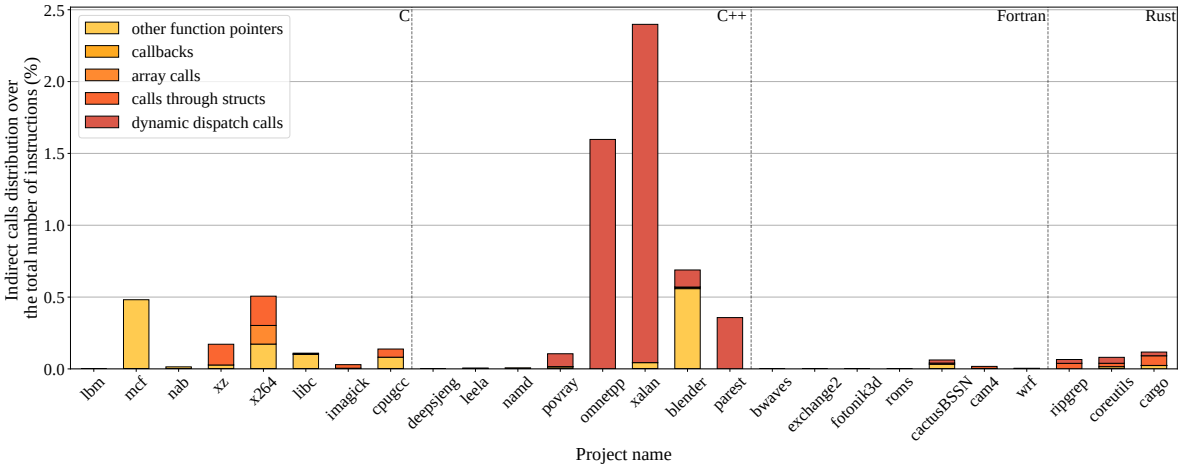


**Figure 12.** Distribution of indirect calls over the total number of instructions at -O2.

Figure 12 follows the same presentation as Figure 11: for each compiled project, a stacked distribution of indirect calls categories is reported. Dynamic linking calls are not represented, as they do not bring information about developer-introduced indirect calls. The *function pointers* category is decomposed into four subcategories: *calls through structs*, *calls through arrays*, *callbacks*, and *other function pointers*, which represents any other use of function pointers.

In C projects, the category *calls through structs* appears in the majority of projects, illustrating the common use of function pointers as methods within structures to emulate object-oriented programming features.

In C++, this presence of *calls through structs* is replaced by *dynamic dispatch calls*, reflecting the polymorphism inherent to the object-oriented paradigm of C++. The proportion of dynamic dispatch calls can represent a significant portion of the generated code, accounting for up to 2.4 % of all instructions in project Xalan for instance.

Fortran projects exhibit a notably lower proportion of indirect calls compared to other languages. Even in larger projects, the diversity of indirect calls is primarily attributable to the inclusion of C libraries in the compiled binaries.

Similarly to Fortran projects, Rust projects present a small overall proportion of indirect calls, but with greater diversity than Fortran, including both *dynamic dispatch calls* and *calls through structs*. This can be explained by the use of dynamic traits that produce vtable dispatch, combined with the presence of C libraries which introduce *calls though structs*. Overall, the proportion of indirect calls in Rust projects remains limited. This can be explained by Rust's monomorphization strategy, which favors static dispatch rather than dynamic dispatch.

Across all languages, the proportion of generated indirect calls shows no correlation with the binary size. This observation indicates that their generation is driven by language paradigms and developer choices rather than by the number of generated instructions.

### 4.3 Optimization Levels

As indirect jumps may be introduced by the compiler optimization passes, our focus here is on the impact of optimization levels and compiler choice. Overall, our experiments reveal very limited differences between the two studied compilers in the number of generated indirect jumps, except for a few optimization cases discussed below. We present the observed trends, for which exact numerical values are not required. The corresponding graphs are available in Appendix B for readers interested in the detailed results.

***Dynamic Linking Calls.*** Indirect calls introduced by dynamic linking of libraries are independent of compiler optimization levels, therefore their number remains stable across optimization configurations. This number is also relatively consistent across benchmarks, about a few hundreds regardless of the language or the benchmark size. The only notable exception is `cargo`, our largest benchmark, which contains more than six hundreds dynamic linking calls, reflecting its broader reliance on external dependencies.

***Jump tables.*** When compiling with LLVM, indirect jumps generated by jump-table optimizations are already observable from optimization level `-O0`, whereas with GCC they appear only from level `-O1`.

This difference originates from the distinct treatment of certain optimizations by the compilers. In GCC, the `-O0` optimization level performs virtually no optimization, mostly preserving the source code structure. In contrast, independently of the selected optimization level, LLVM applies certain transformations in the compiler backend.

Despite this difference, both compilers provide the `-fno-jump-tables` option, which disables the generation of jump tables regardless of the optimization level.

***Tail calls.*** Tail call optimization is enabled from optimization level `-O1` level in LLVM, and `-O2` in GCC. We noticed that Rust projects exhibit a relatively low percentage of tail calls compared to other languages. This can be attributed to Rust's systematic invocation of destructors for local variables when they go out of scope, which requires to return to the caller function in order to release local resources. This mechanism prevents the application of tail call optimization in many cases.

As for jump tables, tail call optimization can be disabled in both compilers using the `-fno-optimize-sibling-calls` option.

## 5 Discussion

This section discusses the limitations of our empirical study, examines the implications of our findings, and outlines potential directions for future work.

### 5.1 Study Scope

First, our study is purely static. A complementary runtime analysis would allow to determine instructions from which categories are most frequently executed, and therefore have the greatest impact.

Second, the analysis is restricted to a 32-bit RISC-V unprivileged architecture. As a result, architecture-specific constructs present in other ISAs may not be reflected in our taxonomy. Similarly, by focusing on the unprivileged subset, we do not consider control transfers related to privilege-level changes, such as interrupt return instructions (`mret`, `sret` in RISC-V).

Third, although the selected benchmarks were intended to be representative, they may not provide exhaustive coverage of indirect jump usages. This limitation is particularly significant for Rust, for which no widely recognized benchmark suite is currently available.

Finally, while the selected languages are widely used in embedded systems, other compiled languages may exhibit distinct patterns. Notably functional languages such as Haskell may present specific tail calls constructs.

### 5.2 Implications of Indirect Jumps Origins

Our taxonomy allows to differentiate how the origin of each jump impacts CFG construction.

Long-range jumps are introduced for architectural reasons and have statically known targets. Similarly, jump tables and many computed gotos constructs rely on a finite and statically-defined set of potential targets. Dynamic dispatch mechanisms and function pointers with known signature also restrict the targets to functions matching a specific type. While they may enlarge the CFG, these constructs still have a limited set of targets that can be determined by state-of-the-art point-to analyses.

In contrast, constructs relying on type erasure, such as functions cast with `void*`, or fully computed gotos addresses, may be impossible to resolve statically, even with access to source code. In such cases, the target is intentionally obscured, significantly complicating accurate CFG construction. It is interesting to note that such opaque mechanisms are disallowed in safe Rust. From a security/WCET point-of-view, these indirect jumps are not desirable as obfuscation is in opposition with the stated goal of easy CFG reconstruction. This suggests that language design choices directly influence the ability to construct an accurate CFG.

### 5.3 Future Work

The identification of potentially opaque constructs leads to the question of how their presence could be managed to improve both the security and the analyzability of embedded software.

One direction consists in promoting development guidelines that discourage the use of programming patterns generating indirect control-flow transfers. For instance, one could advise against using function pointers. Certain compilation options could be forbidden to limit the generation of specific patterns (e.g., disabling jump tables), and static linking could be imposed to reduce indirect jumps introduced by dynamic linking. Such recommendations could be integrated in already established coding standards such as the MISRA guidelines for critical systems [12].

A more radical approach would consist in eliminating indirect jumps at the architectural level, by removing the indirect jump instruction from the ISA, and enforcing compilation strategies to replace them by safer constructs. This would require alternative mechanisms to preserve language expressiveness, and may incur performance overheads. Furthermore, some programs may fail to compile if they rely on opaque indirect transfers.

This potential elimination of indirect jumps is out of the scope of this article and has to be left for future works. Nevertheless, reducing unnecessary indirect transfers could significantly improve CFG precision.

## 6 Related Works

Indirect control transfers have been investigated in several research domains.

In the context of CFG reconstruction, existing approaches primarily aim at resolving or over-approximating their possible target sets in order to improve CFG accuracy [4, 7, 8, 14, 19, 20].

Similarly, CFI research has extensively addressed indirect transfers, as they represent a primary attack surface for control-flow hijacking [1, 2, 18]. Most CFI mechanisms aim to constrain indirect jumps to only target a set of legitimate destinations.

At the microarchitectural level, research on indirect branch prediction focuses on predicting the runtime targets of indirect control transfers to improve processor performance [5, 11, 15].

Across all these domains, the focus is on resolving, predicting, or constraining the targets of indirect transfers, rather than examining the causes of their generation. To the best of our knowledge, no prior work has systematically investigated the origins of indirect jumps.

## 7 Conclusion

In this paper, we presented a taxonomy of indirect jumps origins, detailing the programming constructs and compiler transformation leading to their generation at compile time. This taxonomy was complemented by an empirical analysis of indirect jumps proportions across different programming languages, compilers, and optimization levels. Our study provides a deeper understanding of indirect jumps and their impact on CFG construction. Future works will explore strategies to more effectively manage opaque indirect transfers and mitigate their impact on program analysis.

## Data Availability Statement

The classification tool presented in this paper is available as an artifact at [13].

## Acknowledgments

This work benefited from the financial support of the French Agence Nationale de la Recherche (ANR) for the FAIR project, under reference n°ANR-25-CE39-5360.

## A Detailed Classification

In this section, the classification methodology for each indirect jump origin is detailed in Table 3. Additionally, a flow chart is provided in Figure 13, allowing to follow our complete process.

## B Evolution on Optimization Levels

This section presents the variation of several indirect jumps categories across optimization levels. Respectively, dynamic linking calls are represented on Figure 14, jump tables on Figure 15, and tail calls on Figure 16. To facilitate readability, the graphs only feature projects of more than 30 KLOC. As the proportion of generated jumps for some categories may vary a lot between benchmarks, some graphs are displayed on a logarithmic scale.

**Table 3.** Detailed Indirect Jumps Classification

| Type of jump | Origin | RISC-V Calling Convention | ASM pattern | Source Code Pattern | Manual Classification |
|---|---|---|---|---|---|
| Return | Function return | `jalr zero 0(rs1)`, with `rs1` = `ra` or `t0` | - | - | - |
| Long-range Jump | Direct jump off-limits | - | `auipc/lui`<br>`jalr` | - | - |
| Indirect Call<br>`jalr rd offset(rs1)`, with `rd` = `ra` or `t0` | Dynamic Linking Call | - | `auipc`<br>`lw`<br>`jalr`, with `rs1` = `t1` | - | - |
| | Vtable Call | - | `lw`<br>`lw`<br>`jalr` | - | - |
| | Call through Struct | - | - | `struct.func(`<br>`struct->func(` | - |
| | Array Call | - | - | `array[index](`<br>or `(*array[index])(` | - |
| | Function Pointer | - | - | `func(`<br>or `(*func)(` | - |
| | Callback | - | - | - | Verify if the pointer is given in argument |
| Indirect Jump<br>`jalr zero offset(rs1)`, with `rs1` ≠ `ra` or `t0` | Tail Call | - | `addi sp,sp,offset`<br>`jalr` | `return func(`<br>or `func(`<br>+ next line = `return;` | When needed if undetected by source-code pattern matching |
| | Computed Goto | - | - | `goto *ptr` | - |
| | Jump Table | - | - | `switch`<br>or `match`<br>or `select` | When needed if undetected by source-code pattern matching |

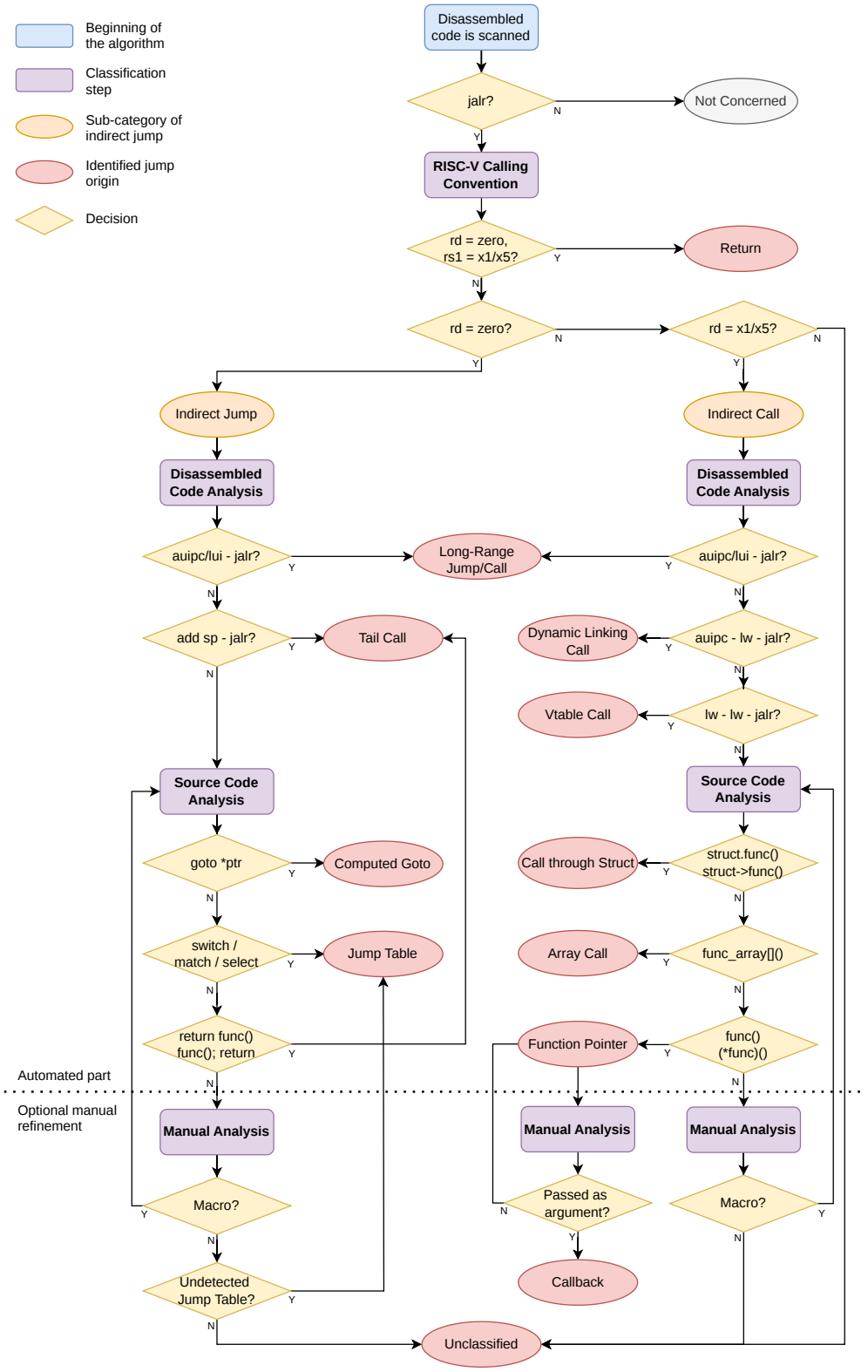


**Figure 13.** Flow Chart of the Classification Methodology.

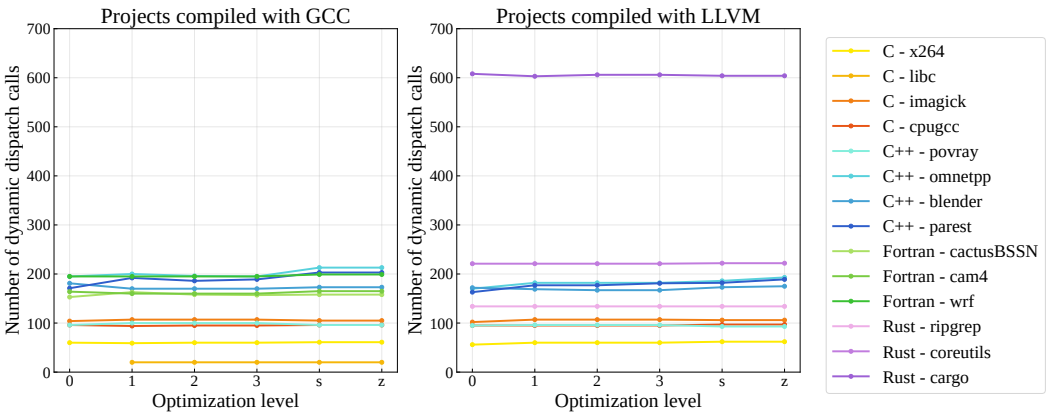


**Figure 14.** Number of generated Dynamic Linking Calls across optimization levels

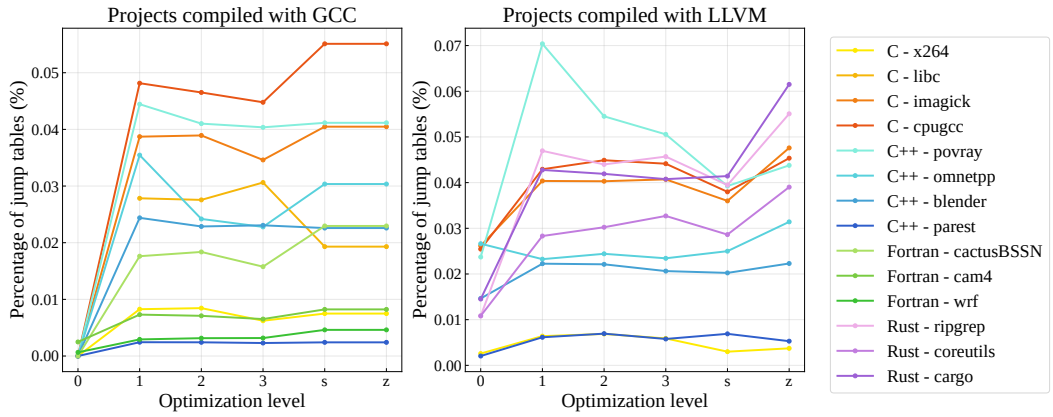


**Figure 15.** Percentage of generated Jump Tables across optimization levels

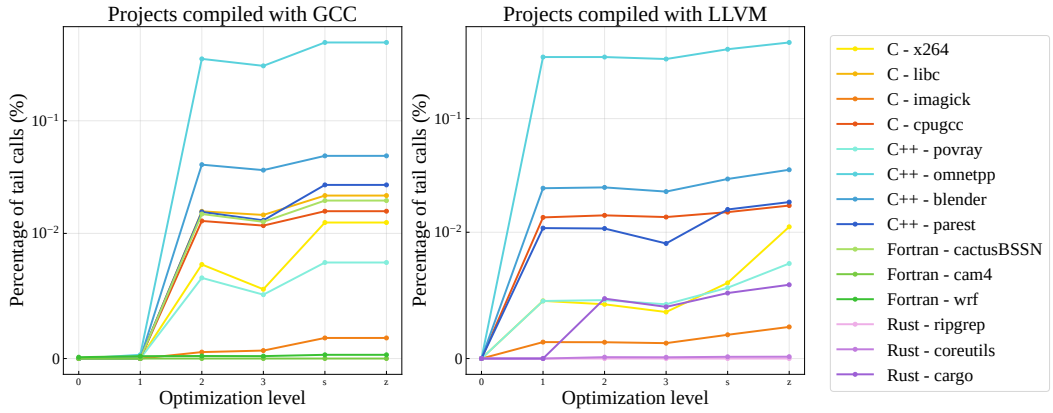


**Figure 16.** Percentage of generated Tail Calls across optimization levels

## References


[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13, 1, Article 4 (Nov. 2009), 40 pages. doi:10.1145/1609956.1609960

[2] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.* 50, 1, Article 16 (April 2017), 33 pages. doi:10.1145/3054924

[3] Alessandro Di Federico and Giovanni Agosta. 2016. A jump-target identification method for multi-architecture static binary translation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (Pittsburgh, Pennsylvania) *(CASES '16)*. Association for Computing Machinery, New York, NY, USA, Article 17, 10 pages. doi:10.1145/2968455.2968514

[4] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction* (Austin, TX, USA) *(CC 2017)*. Association for Computing Machinery, New York, NY, USA, 131–141. doi:10.1145/3033019.3033028

[5] Muhammad Umar Farooq, Lei Chen, and Lizy Kurian John. 2010. Value Based BTB Indexing for indirect jump prediction. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–11. doi:10.1109/HPCA.2010.5416659

[6] Alexandre Gonzalvez and Ronan Lashermes. 2019. A case against indirect jumps for secure programs. In *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering* (San Juan, Puerto Rico, USA) *(SSPREW9 '19)*. Association for Computing Machinery, New York, NY, USA, Article 3, 10 pages. doi:10.1145/3371307.3371314

[7] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. 2021. Refining Indirect Call Targets at the Binary Level. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/refining-indirect-call-targets-at-the-binary-level/

[8] Johannes Kinder and Dmitry Kravchenko. 2012. Alternating Control Flow Reconstruction. In *Verification, Model Checking, and Abstract Interpretation*, Viktor Kuncak and Andrey Rybalchenko (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 267–282.

[9] Dinghao Liu, Shouling Ji, Kangjie Lu, and Qinming He. 2024. Improving Indirect-Call Analysis in LLVM with Type and Data-Flow Co-Analysis. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 5895–5912. https://www.usenix.org/conference/usenixsecurity24/presentation/liu-dinghao-improving

[10] Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1867–1881. doi:10.1145/3319535.3354244

[11] Jason Mccandless and David Gregg. 2012. Compiler techniques to improve dynamic branch prediction for indirect jump and call instructions. *ACM Trans. Archit. Code Optim.* 8, 4, Article 24 (Jan. 2012), 20 pages. doi:10.1145/2086696.2086703

[12] MISRA. 2025. *MISRA C:2025 - Guidelines for the use of the C language in critical systems*. https://misra.org.uk

[13] Ariane Nicolas. 2026. Artifact for the paper "On the Origins of Indirect Jumps in Embedded Software". ACM. doi:10.1145/3747416

[14] Thomas Reinbacher and Jörg Brauer. 2011. Precise control flow reconstruction using Boolean logic. In *Proceedings of the Ninth ACM International Conference on Embedded Software* (Taipei, Taiwan) *(EMSOFT '11)*. Association for Computing Machinery, New York, NY, USA, 117–126. doi:10.1145/2038642.2038662

[15] Oliverio J. Santana, Ayose Falcón, Enrique Fernández, Pedro Medina, Alex Ramírez, and Mateo Valero. 2002. A Comprehensive Analysis of Indirect Branch Prediction. In *High Performance Computing*, Hans P. Zima, Kazuki Joe, Mitsuhisa Sato, Yoshiki Seo, and Masaaki Shimasaki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 133–145.

[16] SPEC CPU2017 2017. SPEC CPU2017. https://www.spec.org/cpu2017

[17] Gang Tan and Trent Jaeger. 2017. CFG Construction Soundness in Control-Flow Integrity. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security* (Dallas, Texas, USA) *(PLAS '17)*. Association for Computing Machinery, New York, NY, USA, 3–13. doi:10.1145/3139337.3139339

[18] Stefan Tauner and Mario Telesklav. 2021. Comparative Analysis and Enhancement of CFG-based Hardware-Assisted CFI Schemes. *ACM Trans. Embed. Comput. Syst.* 20, 5s, Article 58 (Sept. 2021), 25 pages. doi:10.1145/3476989

[19] Tianrou Xia, Hong Hu, and Dinghao Wu. 2024. DEEPTYPE: Refining Indirect Call Targets with Strong Multi-layer Type Analysis. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 5877–5894. https://www.usenix.org/conference/usenixsecurity24/presentation/xia

[20] Dongrui Zeng and Gang Tan. 2018. From Debugging-Information Based Binary-Level Type Inference to CFG Generation. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy* (Tempe, AZ, USA) *(CODASPY '18)*. Association for Computing Machinery, New York, NY, USA, 366–376. doi:10.1145/3176258.3176309


Received 2026-03-18; accepted 2026-05-01