LLVM CFI and Cross-Language LLVM CFI Support for Rust

Author: Ramon de C Valle

Digital Object Identifier
https://doi.org/10.47986/17/11

Editor's Comment

This paper is related to the author's talk at H2HC 2023 and was not reviewed by the H2HC Magazine team.

We're pleased to share that we've worked with the Rust community to add support for LLVM CFI and cross-language LLVM CFI (and LLVM KCFI and cross-language LLVM KCFI) to the Rust compiler as part of our work in the Rust Exploit Mitigations Project Group. This is the first implementation of cross-language, fine-grained, forward-edge control flow protection for mixed-language binaries that we know of.

As the industry continues to explore Rust adoption, cross-language attacks in mixed-language binaries (also known as "mixed binaries"), and critically the absence of support for forward-edge control flow protection in the Rust compiler, are a major security concern when gradually migrating from C and C++ to Rust, and when C or C++ and Rust-compiled code share the same virtual address space.

Introduction

With the increasing popularity of Rust both as a general purpose programming language and as a replacement for C and C++ because of its memory and thread safety guarantees, many companies and projects are adopting or migrating to Rust. One of the most common paths to migrate to Rust is to gradually replace C or C++ with Rust in a program written in C or C++.

Rust provides interoperability with foreign code written in C via Foreign Function Interface (FFI). However, foreign code does not provide the same memory and thread safety guarantees that Rust provides, and is susceptible to memory corruption and concurrency issues.[1] Therefore, it is generally accepted that linking foreign C- or C++-compiled code into a program written in Rust may degrade the security of the program.

[1] Modern C and C++ compilers provide exploit mitigations to increase the difficulty of exploiting vulnerabilities resulting from these issues. However, some of these exploit mitigations are not applied when linking foreign C- or C++-compiled code into a program written in Rust, mostly due to the absence of support for these exploit mitigations in the Rust compiler (see Table 1).

While it is also generally believed that replacing sensitive C or C++ with Rust in a program written in C or C++ improves the security of the program, Papaevripides and Athanasopoulos demonstrated that this is not always the case, and that linking foreign Rust-compiled code into a program written in C or C++ with modern exploit mitigations, such as control flow protection, may actually degrade the security of the program, mostly due to the absence of support for these exploit mitigations in the Rust compiler, mainly forward-edge control flow protection. (See Control flow protection.) This was later formalized as a new class of attacks (i.e., cross-language attacks).

The Rust compiler did not support forward-edge control flow protection when

- using Unsafe Rust.
- linking foreign C- or C++-compiled code into a program written in Rust.
- linking foreign Rust-compiled code into a program written in C or C++.

Table 1 summarizes interoperability-related risks when building programs for the Linux operating system on the AMD64 architecture and equivalent without support for forward-edge control flow protection in the Rust compiler.

	Without using Unsafe Rust	Using Unsafe Rust
Rust-compiled code only	▾ [I] Indirect branches in Rust-compiled code are not validated.[2]	▾ Unsafe Rust is susceptible to memory corruption and concurrency issues. ▾ Indirect branches in Rust-compiled code are not validated.
Linking foreign C- or C++-compiled code into a program written in Rust	▾ Foreign code is susceptible to memory corruption and concurrency issues. ▾ Indirect branches in Rust-compiled code are not validated.	▾ Foreign code is susceptible to memory corruption and concurrency issues. ▾ Unsafe Rust is susceptible to memory corruption and concurrency issues. ▾ Indirect branches in Rust-compiled code are not validated.
Linking foreign Rust -compiled code into a program written in C or C++[3]	▴ [II] Indirect branches in C- and C++-compiled code are validated. ▾ C and C++ are susceptible to memory corruption and concurrency issues. ▾ Indirect branches in Rust-compiled code are not validated.	▴ Indirect branches in C- and C++-compiled code are validated. ▾ C and C++ are susceptible to memory corruption and concurrency issues. ▾ Unsafe Rust is susceptible to memory corruption and concurrency issues. ▾ Indirect branches in Rust-compiled code are not validated.

[I] Downwards-pointing triangle (▾) precedes a negative risk indicator.
[II] Upwards-pointing triangle (▴) precedes a positive risk indicator.

Table 1: Summary of interoperability-related risks when building programs for the Linux operating system on the AMD64 architecture and equivalent without support for forward-edge control flow protection in the Rust compiler.

Without support for forward-edge control flow protection in the Rust compiler, indirect branches in Rust-compiled code were not validated, allowing forward-edge control flow protection to be trivially bypassed as demonstrated by Papaevripides and Athanasopoulos. Therefore, the absence of support for forward-edge control flow protection in the Rust compiler was a major security concern when gradually migrating from C and C++ to Rust, and when C or C++ and Rust-compiled code share the same virtual address space.

[2] An attack that successfully allows a Rust-compiled code only program, without using Unsafe Rust, to have its control flow redirected as a result of a memory corruption or concurrency issue is yet to be demonstrated.
[3] Assuming forward-edge control flow protection is enabled.

Control flow protection

Control flow protection is an exploit mitigation that protects programs from having its control flow redirected. It is classified in two categories:

- Forward-edge control flow protection
- Backward-edge control flow protection

Forward-edge control flow protection

Forward-edge control flow protection protects programs from having their control flow redirected by performing checks to ensure that destinations of indirect branches are one of their valid destinations in the control flow graph. The comprehensiveness of these checks varies per implementation. This is also known as "forward-edge control flow integrity (CFI)".

Newer processors provide hardware assistance for forward-edge control flow protection, such as ARM Branch Target Identification (BTI), ARM Pointer Authentication, and Intel Indirect Branch Tracking (IBT) as part of Intel Control-flow Enforcement Technology (CET). However, ARM BTI- and Intel IBT-based implementations are less comprehensive than software-based implementations, such as LLVM ControlFlowIntegrity (CFI), and the commercially available grsecurity/PaX Reuse Attack Protector (RAP).

The less comprehensive the protection, the higher the likelihood it can be bypassed. For example, Microsoft Windows Control Flow Guard (CFG) only tests that the destination of an indirect branch is a valid function entry point, which is the equivalent of grouping all function pointers in a single group, and testing all destinations of indirect branches to be in this group. This is also known as "coarse-grained CFI".

This means that in an exploitation attempt, an attacker can redirect control flow to any function, and the larger the program is, the higher the likelihood an attacker can find a function they can benefit from (e.g., a small command-line program vs a browser).

Similar to the Microsoft Windows CFG implementation, this is unfortunately the implementation hardware assistance for forward-edge control flow protection (e.g., ARM BTI and Intel IBT) were initially designed based on, and as such, they provide equivalent protection with the addition of specialized instructions. Microsoft Windows eXtended Flow Guard (XFG), ARM Pointer Authentication-based forward-edge control flow protection, and Intel Fine Indirect Branch Tracking (FineIBT) aim to solve this by combining hardware assistance with software-based function pointer type testing similar to LLVM CFI. This is also known as "fine-grained CFI".

Backward-edge control flow protection

Backward-edge control flow protection protects programs from having their control flow redirected by performing checks to ensure that destinations of return branches are one of their valid sources (i.e.,

call sites) in the control flow graph. Backward-edge control flow protection is outside the scope of this article.

Details

There are several details in designing and implementing cross-language, fine-grained, forward-edge control flow protection using function pointer type testing between different languages. This section documents the major challenges in implementing it between Rust and C or C++, more specifically the Rust compiler and Clang.

Type metadata

LLVM uses type metadata to allow IR modules to aggregate pointers by their types. This type metadata is used by LLVM CFI to test whether a given pointer is associated with a type identifier (i.e., test type membership).

Clang uses the Itanium C++ ABI's virtual tables and RTTI `typeinfo` structure name as type metadata identifiers for function pointers.

For cross-language LLVM CFI support, a compatible encoding must be used. The compatible encoding chosen for cross-language LLVM CFI support is the Itanium C++ ABI mangling with vendor extended type qualifiers and types for Rust types that are not used across the FFI boundary (see Type metadata in the design document).

Encoding C integer types

Rust defines `char` as an Unicode scalar value, while C defines `char` as an integer type. Rust also defines explicitly-sized integer types (i.e., `i8`, `i16`, `i32`, ...), while C defines abstract integer types (i.e., `char`, `short`, `long`, ...), which actual sizes are implementation defined and may vary across different data models. This causes ambiguity if Rust integer types are used in `extern "C"` function types that represent C functions because the Itanium C++ ABI specifies encodings for C integer types (e.g., `char`, `short`, `long`, ...), not their defined representations (e.g., 8-bit signed integer, 16-bit signed integer, 32-bit signed integer, ...).

For example, the Rust compiler currently is unable to identify if an

```
extern "C" {
    fn func(arg: i64);
}
```

Listing 1: Example extern "C" function using Rust integer type.

represents a `void func(long arg)` or `void func(long long arg)` in an LP64 or equivalent data model.

For cross-language LLVM CFI support, the Rust compiler must be able to identify and correctly encode C types in `extern "C"` function types indirectly called across the FFI boundary when CFI is enabled.

For convenience, Rust provides some C-like type aliases for use when interoperating with foreign code written in C, and these C type aliases may be used for disambiguation. However, at the time types are encoded, all type aliases are already resolved to their respective `ty::Ty` type representations (i.e., their respective Rust aliased types), making it currently impossible to identify C type aliases use from their resolved types.

For example, the Rust compiler currently is also unable to identify that an

```
extern "C" {
    fn func(arg: c_long);
}
```

Listing 2: Example extern "C" function using C type alias.

used the `c_long` type alias and is not able to disambiguate between it and an `extern "C" fn func(arg: c_longlong)` in an LP64 or equivalent data model.

Consequently, the Rust compiler is unable to identify and correctly encode C types in `extern "C"` function types indirectly called across the FFI boundary when CFI is enabled:

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

// This definition has the type id "_ZTSFvlE".
void
hello_from_c(long arg)
{
    printf("Hello␣from␣C!\n");
}

// This definition has the type id "_ZTSFvPFvlElE"--this can be ignored for the
// purposes of this example.
void
indirect_call_from_c(void (*fn)(long), long arg)
{
    // This call site tests whether the destination pointer is a member of the
    // group derived from the same type id of the fn declaration, which has the
    // type id "_ZTSFvlE".
    //
    // Notice that since the test is at the call site and is generated by Clang,
    // the type id used in the test is encoded by Clang.
```

```
    fn(arg);
}
```

Listing 3: Example C library using C integer types and Clang encoding.

```
use std::ffi::c_long;

#[link(name = "foo")]
extern "C" {
    // This declaration would have the type id "_ZTSFvlE", but at the time types
    // are encoded, all type aliases are already resolved to their respective
    // Rust aliased types, so this is encoded either as "_ZTSFvu3i32E" or
    // "_ZTSFvu3i64E", depending to what type c_long type alias is resolved to,
    // which currently uses the u<length><type-name> vendor extended type
    // encoding for the Rust integer types--this is the problem demonstrated in
    // this example.
    fn hello_from_c(_: c_long);

    // This declaration would have the type id "_ZTSFvPFvlElE", but is encoded
    // either as "_ZTSFvPFvu3i32ES_E" (compressed) or "_ZTSFvPFvu3i64ES_E"
    // (compressed), similarly to the hello_from_c declaration above--this can
    // be ignored for the purposes of this example.
    fn indirect_call_from_c(f: unsafe extern "C" fn(c_long), arg: c_long);
}

// This definition would have the type id "_ZTSFvlE", but is encoded either as
// "_ZTSFvu3i32E" or "_ZTSFvu3i64E", similarly to the hello_from_c declaration
// above.
unsafe extern "C" fn hello_from_rust(_: c_long) {
    println!("Hello, world!");
}

// This definition would have the type id "_ZTSFvlE", but is encoded either as
// "_ZTSFvu3i32E" or "_ZTSFvu3i64E", similarly to the hello_from_c declaration
// above.
unsafe extern "C" fn hello_from_rust_again(_: c_long) {
    println!("Hello from Rust again!");
}

// This definition would also have the type id "_ZTSFvPFvlElE", but is encoded
// either as "_ZTSFvPFvu3i32ES_E" (compressed) or "_ZTSFvPFvu3i64ES_E"
// (compressed), similarly to the hello_from_c declaration above--this can be
// ignored for the purposes of this example.
fn indirect_call(f: unsafe extern "C" fn(c_long), arg: c_long) {
    // This indirect call site tests whether the destination pointer is a member
    // of the group derived from the same type id of the f declaration, which
    // would have the type id "_ZTSFvlE", but is encoded either as
    // "_ZTSFvu3i32E" or "_ZTSFvu3i64E", similarly to the hello_from_c
    // declaration above.
    //
    // Notice that since the test is at the call site and is generated by the
```

```
    // Rust compiler, the type id used in the test is encoded by the Rust
    // compiler.
    unsafe { f(arg) }
}

// This definition has the type id "_ZTSFvvE"—this can be ignored for the
// purposes of this example.
fn main() {
    // This demonstrates an indirect call within Rust—only code using the same
    // encoding for hello_from_rust and the test at the indirect call site at
    // indirect_call (i.e., "_ZTSFvu3i32E" or "_ZTSFvu3i64E").
    indirect_call(hello_from_rust, 5);

    // This demonstrates an indirect call across the FFI boundary with the Rust
    // compiler and Clang using different encodings for hello_from_c and the
    // test at the indirect call site at indirect_call (i.e., "_ZTSFvu3i32E" or
    // "_ZTSFvu3i64E" vs "_ZTSFvlE").
    //
    // When using rustc LTO (i.e., —Clto), this works because the type id used
    // is from the Rust—declared hello_from_c, which is encoded by the Rust
    // compiler (i.e., "_ZTSFvu3i32E" or "_ZTSFvu3i64E").
    //
    // When using (proper) LTO (i.e., —Clinker—plugin—lto), this does not work
    // because the type id used is from the C—defined hello_from_c, which is
    // encoded by Clang (i.e., "_ZTSFvlE").
    indirect_call(hello_from_c, 5);

    // This demonstrates an indirect call to a function passed as a callback
    // across the FFI boundary with the Rust compiler and Clang using different
    // encodings for the hello_from_rust_again and the test at the indirect call
    // site at indirect_call_from_c (i.e., "_ZTSFvu3i32E" or "_ZTSFvu3i64E" vs
    // "_ZTSFvlE").
    //
    // When Rust functions are passed as callbacks across the FFI boundary to be
    // called back from C code, the tests are also at the call site but
    // generated by Clang instead, so the type ids used in the tests are encoded
    // by Clang, which do not match the type ids of declarations encoded by the
    // Rust compiler (e.g., hello_from_rust_again). (The same happens the other
    // way around for C functions passed as callbacks across the FFI boundary to
    // be called back from Rust code.)
    unsafe {
        indirect_call_from_c(hello_from_rust_again, 5);
    }
}
```

Listing 4: Example Rust program using Rust integer types and the Rust compiler encoding.

Whenever there is an indirect call across the FFI boundary or an indirect call to a function passed as a callback across the FFI boundary, the Rust compiler and Clang use different encodings for C integer types for function definitions and declarations, and at indirect call sites when CFI is enabled (see Figs. 3–4).

The integer normalization option

To solve the encoding C integer types problem, we added an integer normalization option to Clang (i.e., `-fsanitize-cfi-icall-experimental-normalize-integers`). This option enables normalizing integer types as vendor extended types for cross-language LLVM CFI (and cross-language LLVM KCFI) support with other languages that can't represent and encode C integer types.

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

// This definition has the type id "_ZTSFvlE", but will be encoded either as
// "_ZTSFvu3i32E" or "_ZTSFvu3i64E", depending on the data model, if the integer
// normalization option is enabled, which uses the u<length><type-name> vendor
// extended type encoding for the C integer types.
void
hello_from_c(long arg)
{
    printf("Hello from C!\n");
}

// This definition has the type id "_ZTSFvPFvlElE", but will be encoded either
// as "_ZTSFvPFvu3i32ES_E" (compressed) or "_ZTSFvPFvu3i64ES_E" (compressed),
// depending on the data model, if the integer normalization option is
// enabled--this can be ignored for the purposes of this example.
void
indirect_call_from_c(void (*fn)(long), long arg)
{
    // This call site tests whether the destination pointer is a member of the
    // group derived from the same type id of the fn declaration, which has the
    // type id "_ZTSFvlE", but will be encoded either as "_ZTSFvu3i32E" or
    // "_ZTSFvu3i64E", depending on the data model, if the integer normalization
    // option is enabled.
    fn(arg);
}
```

Listing 5: Example C library using C integer types and Clang encoding with the integer normalization option enabled.

Specifically, integer types are encoded as their defined representations (e.g., 8-bit signed integer, 16-bit signed integer, 32-bit signed integer, ...) for compatibility with languages that define explicitly-sized integer types (e.g., `i8`, `i16`, `i32`, ..., in Rust) (see Fig. 5).

This makes cross-language LLVM CFI (and LLVM KCFI) work without changes, with minimal loss of granularity.[4]

[4] E.g., https://github.com/rust-lang/rfcs/pull/3296#issuecomment-1432190581 ~1% in the Linux kernel.

The cfi_encoding attribute

To provide flexibility for the user, we also provide a cfi_encoding attribute. The `cfi_encoding` attribute allows the user to define the CFI encoding for user-defined types.

```
#![feature(cfi_encoding, extern_types)]
#[cfi_encoding = "3Foo"]
pub struct Type1(i32);
extern {
    #[cfi_encoding = "3Bar"]
    type Type2;
}
```

Listing 6: Example user-defined types using the cfi_encoding attribute.

It allows the user to use different names for types that otherwise would be required to have the same name as used in externally defined C functions (see Fig. 6).

The cfi_types crate

Alternatively, to also solve the encoding C integer types problem, we provide the `cfi_types` crate. This crate provides a new set of C types as user-defined types using the `cfi_encoding` attribute and `repr(transparent)` to be used for cross-language LLVM CFI support.

```
use cfi_types::c_long;

#[link(name = "foo")]
extern "C" {
    // This declaration has the type id "_ZTSFvlE" because it uses the CFI types
    // for cross-language LLVM CFI support. The cfi_types crate provides a new
    // set of C types as user-defined types using the cfi_encoding attribute and
    // repr(transparent) to be used for cross-language LLVM CFI support. This
    // new set of C types allows the Rust compiler to identify and correctly
    // encode C types in extern "C" function types indirectly called across the
    // FFI boundary when CFI is enabled.
    fn hello_from_c(_: c_long);

    // This declaration has the type id "_ZTSFvPFvlElE" because it uses the CFI
    // types for cross-language LLVM CFI support--this can be ignored for the
    // purposes of this example.
    fn indirect_call_from_c(f: unsafe extern "C" fn(c_long), arg: c_long);
}

// This definition has the type id "_ZTSFvlE" because it uses the CFI types for
// cross-language LLVM CFI support, similarly to the hello_from_c declaration
```

```
// above.
unsafe extern "C" fn hello_from_rust(_: c_long) {
    println!("Hello, world!");
}

// This definition has the type id "_ZTSFvlE" because it uses the CFI types for
// cross-language LLVM CFI support, similarly to the hello_from_c declaration
// above.
unsafe extern "C" fn hello_from_rust_again(_: c_long) {
    println!("Hello from Rust again!");
}

// This definition also has the type id "_ZTSFvPFvlElE" because it uses the CFI
// types for cross-language LLVM CFI support, similarly to the hello_from_c
// declaration above--this can be ignored for the purposes of this example.
fn indirect_call(f: unsafe extern "C" fn(c_long), arg: c_long) {
    // This indirect call site tests whether the destination pointer is a member
    // of the group derived from the same type id of the f declaration, which
    // has the type id "_ZTSFvlE" because it uses the CFI types for
    // cross-language LLVM CFI support, similarly to the hello_from_c
    // declaration above.
    unsafe { f(arg) }
}

// This definition has the type id "_ZTSFvvE"--this can be ignored for the
// purposes of this example.
fn main() {
    // This demonstrates an indirect call within Rust-only code using the same
    // encoding for hello_from_rust and the test at the indirect call site at
    // indirect_call (i.e., "_ZTSFvlE").
    indirect_call(hello_from_rust, c_long(5));

    // This demonstrates an indirect call across the FFI boundary with the Rust
    // compiler and Clang using the same encoding for hello_from_c and the test
    // at the indirect call site at indirect_call (i.e., "_ZTSFvlE").
    indirect_call(hello_from_c, c_long(5));

    // This demonstrates an indirect call to a function passed as a callback
    // across the FFI boundary with the Rust compiler and Clang the same
    // encoding for the hello_from_rust_again and the test at the indirect call
    // site at indirect_call_from_c (i.e., "_ZTSFvlE").
    unsafe {
        indirect_call_from_c(hello_from_rust_again, c_long(5));
    }
}
```

Listing 7: Example Rust program using Rust integer types and the Rust compiler encoding with the cfi_types crate types.

This new set of C types allows the Rust compiler to identify and correctly encode C types in `extern "C"` function types indirectly called across the FFI boundary when CFI is enabled (see Fig 7).

Results

LLVM CFI support in the Rust compiler provides forward-edge control flow protection for both Rust-compiled code only and for C or C++ and Rust-compiled code mixed-language binaries, also known as "mixed binaries" (i.e., for when C or C++ and Rust-compiled code share the same virtual address space) by aggregating function pointers in groups identified by their return and parameter types.

LLVM CFI can be enabled with `-Zsanitizer=cfi` and requires LTO (i.e., `-Clinker-plugin-lto` or `-Clto`). Cross-language LLVM CFI can be enabled with `-Zsanitizer=cfi`, requires the `-Zsanitizer-cfi-normalize-integers` option to be used with the Clang `-fsanitize-cfi-icall-experimental-normalize-integers` option for cross-language LLVM CFI support, and proper (i.e., non-rustc) LTO (i.e., `-Clinker-plugin-lto`).

It is recommended to rebuild the standard library with CFI enabled by using the Cargo build-std feature (i.e., `-Zbuild-std`) when enabling CFI.

Example 1: Redirecting control flow using an indirect branch/call to an invalid destination

```
#![feature(naked_functions)]

use std::arch::asm;
use std::mem;

fn add_one(x: i32) -> i32 {
    x + 1
}

#[naked]
pub extern "C" fn add_two(x: i32) {
    // x + 2 preceded by a landing pad/nop block
    unsafe {
        asm!(
            "
␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣nop
␣␣␣␣␣␣␣␣␣␣␣␣lea␣eax,␣[rdi+2]
␣␣␣␣␣␣␣␣␣␣␣␣ret
␣␣␣␣␣␣␣␣",
            options(noreturn)
        );
```

```
    }
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The␣answer␣is:␣{}", answer);

    println!("With␣CFI␣enabled,␣you␣should␣not␣see␣the␣next␣answer");
    let f: fn(i32) -> i32 = unsafe {
        // Offset 0 is a valid branch/call destination (i.e., the function entry
        // point), but offsets 1–8 within the landing pad/nop block are invalid
        // branch/call destinations (i.e., within the body of the function).
        mem::transmute::<*const u8, fn(i32) -> i32>((add_two as *const u8).offset(5))
    };
    let next_answer = do_twice(f, 5);

    println!("The␣next␣answer␣is:␣{}", next_answer);
}
```

Listing 8: Redirecting control flow using an indirect branch/call to an invalid destination (i.e., within the body of the function).

```
$ cargo run --release
   Compiling rust-cfi-1 v0.1.0 (/home/rcvalle/rust-cfi-1)
    Finished release [optimized] target(s) in 0.43s
     Running `target/release/rust-cfi-1`
The answer is: 12
With CFI enabled, you should not see the next answer
The next answer is: 14
$
```

Listing 9: Build and execution of Fig. 8 with LLVM CFI disabled.

```
$ RUSTFLAGS="-Clinker-plugin-lto␣-Clinker=clang␣-Clink-arg=-fuse-ld=lld␣-Zsanitizer=cfi" cargo run -
    Zbuild-std -Zbuild-std-features --release --target x86_64-unknown-linux-gnu
   ...
   Compiling rust-cfi-1 v0.1.0 (/home/rcvalle/rust-cfi-1)
    Finished release [optimized] target(s) in 1m 08s
     Running `target/x86_64-unknown-linux-gnu/release/rust-cfi-1`
The answer is: 12
With CFI enabled, you should not see the next answer
Illegal instruction
$
```

Listing 10: Build and execution of Fig. 8 with LLVM CFI enabled.

When LLVM CFI is enabled, if there are any attempts to redirect control flow using an indirect branch/call to an invalid destination, the execution is terminated (see Fig. 10).

Example 2: Redirecting control flow using an indirect branch/call to a function with a different number of parameters

```
use std::mem;

fn add_one(x: i32) −> i32 {
    x + 1
}

fn add_two(x: i32, _y: i32) −> i32 {
    x + 2
}

fn do_twice(f: fn(i32) −> i32, arg: i32) −> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The␣answer␣is:␣{}", answer);

    println!("With␣CFI␣enabled,␣you␣should␣not␣see␣the␣next␣answer");
    let f: fn(i32) −> i32 =
        unsafe { mem::transmute::<*const u8, fn(i32) −> i32>(add_two as *const u8) };
    let next_answer = do_twice(f, 5);

    println!("The␣next␣answer␣is:␣{}", next_answer);
}
```

Listing 11: Redirecting control flow using an indirect branch/call to a function with a different number of parameters than arguments intended/passed in the call/branch site.

```
$ cargo run −−release
   Compiling rust−cfi−2 v0.1.0 (/home/rcvalle/rust−cfi−2)
    Finished release [optimized] target(s) in 0.43s
     Running ‘target/release/rust−cfi−2‘
The answer is: 12
With CFI enabled, you should not see the next answer
The next answer is: 14
$
```

Listing 12: Build and execution of Fig. 11 with LLVM CFI disabled.

```
$ RUSTFLAGS="-Clinker-plugin-lto -Clinker=clang -Clink-arg=-fuse-ld=lld -Zsanitizer=cfi" cargo run -
    Zbuild-std -Zbuild-std-features --release --target x86_64-unknown-linux-gnu
   ...
   Compiling rust-cfi-2 v0.1.0 (/home/rcvalle/rust-cfi-2)
    Finished release [optimized] target(s) in 1m 08s
     Running `target/x86_64-unknown-linux-gnu/release/rust-cfi-2`
The answer is: 12
With CFI enabled, you should not see the next answer
Illegal instruction
$
```

Listing 13: Build and execution of Fig. 11 with LLVM CFI enabled.

When LLVM CFI is enabled, if there are any attempts to redirect control flow using an indirect branch/call to a function with a different number of parameters than arguments intended/passed in the call/branch site, the execution is also terminated (see Fig. 13).

Example 3: Redirecting control flow using an indirect branch/call to a function with different return and parameter types

```
use std::mem;

fn add_one(x: i32) -> i32 {
    x + 1
}

fn add_two(x: i64) -> i64 {
    x + 2
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {}", answer);

    println!("With CFI enabled, you should not see the next answer");
    let f: fn(i32) -> i32 =
        unsafe { mem::transmute::<*const u8, fn(i32) -> i32>(add_two as *const u8) };
    let next_answer = do_twice(f, 5);

    println!("The next answer is: {}", next_answer);
}
```

Listing 14: Redirecting control flow using an indirect branch/call to a function with different return and parameter types than the return type expected and arguments intended/passed at the call/branch site.

```
$ cargo run --release
    Compiling rust-cfi-3 v0.1.0 (/home/rcvalle/rust-cfi-3)
     Finished release [optimized] target(s) in 0.44s
      Running `target/release/rust-cfi-3`
The answer is: 12
With CFI enabled, you should not see the next answer
The next answer is: 14
$
```

Listing 15: Build and execution of Fig. 14 with LLVM CFI disabled.

```
$ RUSTFLAGS="-Clinker-plugin-lto -Clinker=clang -Clink-arg=-fuse-ld=lld -Zsanitizer=cfi" cargo run -
    Zbuild-std -Zbuild-std-features --release --target x86_64-unknown-linux-gnu
   ...
   Compiling rust-cfi-3 v0.1.0 (/home/rcvalle/rust-cfi-3)
    Finished release [optimized] target(s) in 1m 07s
     Running `target/x86_64-unknown-linux-gnu/release/rust-cfi-3`
The answer is: 12
With CFI enabled, you should not see the next answer
Illegal instruction
$
```

Listing 16: Build and execution of Fig. 14 with LLVM CFI enabled.

When LLVM CFI is enabled, if there are any attempts to redirect control flow using an indirect branch/call to a function with different return and parameter types than the return type expected and arguments intended/passed at the call/branch site, the execution is also terminated (see Fig. 16).

Example 4: Redirecting control flow using an indirect branch/call to a function with different return and parameter types across the FFI boundary

```
int
do_twice(int (*fn)(int), int arg)
{
    return fn(arg) + fn(arg);
}
```

Listing 17: Example C library.

```
use std::mem;

#[link(name = "foo")]
```

```
extern "C" {
    fn do_twice(f: unsafe extern "C" fn(i32) -> i32, arg: i32) -> i32;
}

unsafe extern "C" fn add_one(x: i32) -> i32 {
    x + 1
}

unsafe extern "C" fn add_two(x: i64) -> i64 {
    x + 2
}

fn main() {
    let answer = unsafe { do_twice(add_one, 5) };

    println!("The answer is: {}", answer);

    println!("With CFI enabled, you should not see the next answer");
    let f: unsafe extern "C" fn(i32) -> i32 = unsafe {
        mem::transmute::<*const u8, unsafe extern "C" fn(i32) -> i32>(add_two as *const u8)
    };
    let next_answer = unsafe { do_twice(f, 5) };

    println!("The next answer is: {}", next_answer);
}
```

Listing 18: Redirecting control flow using an indirect branch/call to a function with different return and parameter types than the return type expected and arguments intended/passed in the call/branch site, across the FFI boundary.

```
$ make
mkdir -p target/release
clang -I. -Isrc -Wall -c src/foo.c -o target/release/libfoo.o
llvm-ar rcs target/release/libfoo.a target/release/libfoo.o
RUSTFLAGS="-L./target/release -Clinker=clang -Clink-arg=-fuse-ld=lld" cargo build --release
   Compiling rust-cfi-4 v0.1.0 (/home/rcvalle/rust-cfi-4)
    Finished release [optimized] target(s) in 0.49s
$ ./target/release/rust-cfi-4
The answer is: 12
With CFI enabled, you should not see the next answer
The next answer is: 14
$
```

Listing 19: Build and execution of Figs. 17–18 with LLVM CFI disabled.

```
$ make
mkdir -p target/release
clang -I. -Isrc -Wall -flto -fsanitize=cfi -fsanitize-cfi-icall-experimental-normalize-integers -fvisibility=
    hidden -c -emit-llvm src/foo.c -o target/release/libfoo.bc
llvm-ar rcs target/release/libfoo.a target/release/libfoo.bc
```

```
RUSTFLAGS="-L./target/release␣-Clinker-plugin-lto␣-Clinker=clang␣-Clink-arg=-fuse-ld=lld␣-Zsanitizer=
    cfi␣-Zsanitizer-cfi-normalize-integers" cargo build -Zbuild-std -Zbuild-std-features --release --
    target x86_64-unknown-linux-gnu
   ...
   Compiling rust-cfi-4 v0.1.0 (/home/rcvalle/rust-cfi-4)
    Finished release [optimized] target(s) in 1m 06s
$ ./target/x86_64-unknown-linux-gnu/release/rust-cfi-4
The answer is: 12
With CFI enabled, you should not see the next answer
Illegal instruction
$
```

Listing 20: Build and execution of Figs. 17–18 with LLVM CFI enabled.

When LLVM CFI is enabled, if there are any attempts to redirect control flow using an indirect branch/call to a function with different return and parameter types than the return type expected and arguments intended/passed in the call/branch site, even across the FFI boundary and for functions passed as a callback across the FFI boundary, the execution is also terminated (see Fig. 20).

Performance

Preliminary performance testing (i.e., one billion calls to a function containing an indirect branch without and with LLVM CFI enabled compared) using cargo bench indicates negligible performance impact (i.e., 0.01%).

Conclusion

LLVM CFI and cross-language LLVM CFI (and LLVM KCFI and cross-language LLVM KCFI) are available on nightly builds of the Rust compiler. As we work towards stabilizing these features (see our roadmap), we're starting to experiment with them in our products and services, and encourage you to try them as well and let us know if you have any issues (see the known issues).

Hopefully, this work also provides the foundation for future implementations of cross-language, fine-grained, combined hardware-assisted and software-based, forward-edge control flow protection, such as Microsoft Windows XFG, ARM Pointer Authentication-based forward-edge control flow protection, and Intel FineIBT.

Acknowledgments

Thanks to bjorn3 (Björn Roy Baron), compiler-errors (Michael Goulet), eddyb (Eduard-Mihai Burtescu), matthiaskrgr (Matthias Krüger), mmaurer (Matthew Maurer), nagisa (Simonas Kazlauskas), pcc (Peter Collingbourne), pnkfelix (Felix Klock), samitolvanen (Sami Tolvanen), tmiasko (Tomasz Miąsko), and the Rust community for all their help throughout this project.