LLVM Control Flow Integrity (CFI) Support for Rust Design Doc

go/rust-cfi-design-doc
Authors: rcvalle@
Status: Current

This document describes the overall design of LLVM CFI support for the Rust compiler. This document does not describe LLVM SafeStack or LLVM ShadowCallStack (i.e., backward-edge control flow protection) support for the Rust compiler.

This document is for software engineers working with the Rust programming language that want information about the overall design of LLVM CFI support for the Rust compiler. This document assumes prior knowledge of the Rust programming language and its toolchain.

Objective

Add LLVM CFI support to the Rust compiler (i.e., rustc).

Background

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]–[3]

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[4] 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

[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 I).

these exploit mitigations in the Rust compiler, mainly forward-edge control flow protection. (See Control flow protection.)

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

- using Unsafe Rust.[1],[6]
- linking foreign C- or C++-compiled code into a program written in Rust.[1]–[3]
- linking foreign Rust-compiled code into a program written in C or C++.[4]

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

Table I
Summary of interoperability-related risks when building programs for the Linux operating system on the AMD64 architecture and equivalent without forward-edge control flow protection support 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	▲ Indirect branches in C- and C++-compiled code are validated. ▼ C and C++ are susceptible to memory corruption and concurrency issues.

[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.

	validated.	▼ Unsafe Rust is susceptible to memory corruption and concurrency issues. ▼ Indirect branches in Rust-compiled code are not validated.

[I] Red down-pointing triangle (▼) precedes a negative risk indicator.
[II] Green up-pointing triangle (▲) precedes a positive risk indicator.

Without forward-edge control flow protection support in the Rust compiler, indirect branches in Rust-compiled code are not validated, allowing forward-edge control flow protection to be trivially bypassed as demonstrated by Papaevripides and Athanasopoulos[4]. Therefore, the absence of support for forward-edge control flow protection in the Rust compiler is a major security concern[5] when migrating to Rust by gradually replacing C or C++ with Rust, and when C or C++ and Rust-compiled code share the same virtual address space.

Control flow protection

Control flow protection is an exploit mitigation that protects programs from having their 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)[7], and the commercially available grsecurity/PaX Reuse Attack Protector (RAP)[8].

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 document.

Detailed design

LLVM CFI support for the Rust compiler will be implemented similarly to how it is implemented for Clang. It will be implemented by adding support for emitting type metadata and checks to the Rust compiler code generation.

The Rust compiler provides support for multiple code generation backends.[9] The actual code generation is done by a third-party library, such as LLVM. The `rustc_codegen_ssa` crate contains backend-agnostic code and provides an abstraction layer and interface for backends to implement, such as implemented in the `rustc_codegen_llvm` crate.

MIR is the Rust compiler Mid-level Intermediate Representation. In MIR, there is no difference between method calls, overloaded operators, and function calls anymore. (Method calls and overloaded operators are lowered to the same kind of terminator that function calls are.)

The `rustc_codegen_ssa::mir::block` module lowers MIR blocks and their terminators to the selected backend intermediate representation (IR), such as LLVM IR. It is the module that does the code generation for function calls.

The `do_call` method of the `TerminatorCodegenHelper` type in the `rustc_codegen_ssa::mir::block` module does the code generation for function calls, including all indirect function calls. It uses the `call` and `invoke` methods of the selected backend `Builder`. The `call` and `invoke` methods of the selected backend `Builder` build the call in the selected backend IR, making these the preferred location for emitting type checks.

To emit type metadata and checks, the type membership abstraction will be added to the to backend-agnostic code for backends that support control flow protection using type membership (i.e., testing whether a given pointer is associated with a type identifier), such as LLVM, to implement.

LLVM CFI support for the Rust compiler work is broken in these steps:

- Create tracking issue.
- Add option to the Rust compiler driver and frontend.
- Add support for emitting type metadata and checks to the Rust compiler code generation. (Broken in detailed steps below.)
- Implement Rust-compiled code only LLVM CFI support by using Rust-specific type metadata identifiers.
- Define type metadata identifiers for cross-language LLVM CFI support.
- Change initial implementation to use the defined type metadata identifiers for cross-language LLVM CFI support.
- Create documentation and tests.

Add support for emitting type metadata and checks to the Rust compiler code generation work is broken in two categories:

- Add type membership abstraction to backend-agnostic code.
- Implement type membership abstraction for LLVM backend.

Add type membership abstraction to backend-agnostic code work is broken in these steps:

- Add support for computing a type identifier for a given `FnAbi` or `FnSig` to `rustc_symbol_mangling` crate. (See Type metadata.)
- Add `type_test` method declaration to backend-agnostic `IntrinsicCallMethods` trait declaration for backends to implement. The `type_test` method should test whether a given pointer is associated with a type identifier.
- Add `TypeMembershipMethods` supertrait to backend-agnostic `TypeMethods` subtrait for backends to implement. These methods should emit type metadata for functions.

Implement type membership abstraction for LLVM backend work is broken in these steps:

- Add `type_test` method implementation to LLVM backend `IntrinsicCallMethods` trait implementation using `llvm.type.test` intrinsic. The `LowerTypeTests` link-time optimization pass replaces calls to the `llvm.type.test` intrinsic with code to test type membership.
- Add `TypeMembershipMethods` supertrait implementation to LLVM backend `TypeMethods` subtrait implementation and any necessary FFI bindings/declarations to the `rustc_codegen_llvm::llvm` module.
- Add support for emitting type metadata for functions to `declare_fn` method of LLVM backend `CodegenCx` implementation.
- Add support for emitting type checks to `call` and `invoke` methods of LLVM backend `Builder` implementation.

Type metadata

LLVM uses type metadata to allow IR modules to aggregate pointers by their types.[10] 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[11] virtual tables and RTTI typeinfo structure name[12] as type metadata identifiers for function pointers. The typeinfo name encoding is a two-character code (i.e., “TS”) prefixed to the type encoding for the function.

For cross-language LLVM CFI support, a compatible encoding must be used by either

1. using Itanium C++ ABI mangling for encoding (which is currently used by Clang).

2. creating a new encoding for cross-language CFI and using it for Clang and the Rust compiler (and possibly other compilers).

And

- provide comprehensive protection for Rust-compiled only code if used as main encoding (and not require an alternative Rust-specific encoding for Rust-compiled only code).
- provide comprehensive protection for C- and C++-compiled code when linking foreign Rust-compiled code into a program written in C or C++.
- provide comprehensive protection across the FFI boundary when linking foreign Rust-compiled code into a program written in C or C++.

Providing comprehensive protection for Rust-compiled only code if used as main encoding

This item is satisfied by the encoding being able to comprehensively encode Rust types. Both using Itanium C++ ABI mangling for encoding (1) and creating a new encoding for cross-language CFI (2) may satisfy this item by providing support for (language or vendor) extended types, by defining a comprehensive encoding for Rust types using (language or vendor) extended types and implementing it in the Rust compiler.

Providing comprehensive protection for C- and C++-compiled code when linking foreign Rust-compiled code into a program written in C or C++

This item is satisfied by the encoding being able to comprehensively encode C and C++ types, and Clang being able to continue to use a comprehensive encoding for C- and C++-compiled code when linking foreign Rust-compiled code into a program written in C or C++.

Both using Itanium C++ ABI mangling for encoding (1) and creating a new encoding for cross-language CFI (2) may satisfy this item by providing support for (language or vendor) extended types. However, creating a new encoding for cross-language CFI (2) also requires defining a comprehensive encoding for C and C++ types using (language or vendor) extended types and implementing it in Clang, so it is able to continue to use a comprehensive encoding for C- and C++-compiled code when linking foreign Rust-compiled code into a program written in C or C++. This introduces as much complexity and work as redefining Itanium C++ ABI mangling and reimplementing it in Clang.

Additionally, creating a new encoding for cross-language CFI (2), depending on its requirements, may be required to use a generalized encoding across the FFI boundary. This may result in using a generalized encoding for all C- and C++-compiled code instead of across the FFI boundary only (because there is no indication to Clang of which functions are called across the FFI boundary), or may require changes to Clang to use the generalized encoding

across the FFI boundary only (which may also require new Clang extensions and changes to C and C++ code and libraries to indicate which functions are called across the FFI boundary).

Either using a generalized encoding for all C- and C++-compiled code or across the FFI boundary do not satisfy this or the following item, and will degrade the security of the program when linking foreign Rust-compiled code into a program written in C or C++ because the program previously used a more comprehensive encoding for all its compiled code.

Providing comprehensive protection across the FFI boundary when linking foreign Rust-compiled code into a program written in C or C++

This item is satisfied by being able to encode uses of Rust or C types across the FFI boundary by either

- changing the Rust compiler to be able to identify and encode uses of C types across the FFI boundary.
- changing Clang to be able to identify and encode uses of Rust types across the FFI boundary.

Both using Itanium C++ ABI mangling for encoding (1) and creating a new encoding for cross-language CFI (2) require changing either the Rust compiler or Clang to satisfy this item, and may also require changes to Rust or C and C++ code and libraries.

However, as described in the previous item, creating a new encoding for cross-language CFI (2), depending on its requirements, may be required to use a generalized encoding across the FFI boundary, and while using a generalized encoding across the FFI boundary does not require changing the Rust compiler or Clang to be able to identify and encode uses of Rust or C types across the FFI boundary, it does not satisfy this item either, and will degrade the security of the program when linking foreign Rust-compiled code into a program written in C or C++ because the program previously used a more comprehensive encoding for all its compiled code.

Using Itanium C++ ABI mangling for encoding (1) versus creating a new encoding for cross-language CFI (2)

Using Itanium C++ ABI mangling for encoding (1) provides cross-language LLVM CFI support with C- and C++-compiled code as is, provides more comprehensive protection by satisfying all previous items, does not require changes to Clang (either to implement a new encoding or to be able to identify and encode uses of Rust types across the FFI boundary), and does not require new Clang extensions and changes to C and C++ code and libraries (either to be able to identify and encode uses of Rust types across the FFI boundary or to indicate which functions are called across the FFI boundary).

While using Itanium C++ ABI mangling for encoding (1) requires defining a comprehensive encoding for Rust types using (language or vendor) extended types and implementing it in the Rust compiler, creating a new encoding for cross-language CFI (2) requires defining comprehensive encodings for both Rust and C and C++ types using (language or vendor) extended types, and implementing them in both the Rust compiler and Clang respectively. This introduces as much complexity and work as redefining Itanium C++ ABI mangling and reimplementing it in Clang.

Additionally, creating a new encoding for cross-language CFI (2), depending on its requirements, may provide less comprehensive protection by either using a generalized encoding for all C- and C++-compiled code or across the FFI boundary, not satisfying all previous items, and will degrade the security of the program when linking foreign Rust-compiled code into a program written in C or C++ because the program previously used a more comprehensive encoding for all its compiled code.

Table II summarizes the advantages and disadvantages of using Itanium C++ ABI mangling for encoding (1) versus creating a new encoding for cross-language CFI (2).

Table II
Summary of advantages and disadvantages of using Itanium C++ ABI mangling for encoding (1) versus creating a new encoding for cross-language CFI (2).

Using Itanium C++ ABI mangling for encoding	**Creating a new encoding for cross-language CFI**
▲[I] Provides cross-language LLVM CFI support with C- and C++-compiled code as is. ▲ Provides more comprehensive protection by satisfying all previous items. ▲ Does not require changes to Clang (either to implement a new encoding or to be able to identify and encode uses of Rust types across the FFI boundary). ▲ Does not require new Clang extensions and changes to C and C++ code and libraries (either to be able to identify and encode uses of Rust types across the FFI boundary or to indicate which functions are called across the FFI boundary). ▼[II] Requires changes to the Rust compiler (to be able to identify and encode uses of C and C++ types across the FFI boundary). ▼ Requires defining a comprehensive encoding for Rust types using (language or vendor) extended types and implementing it in the Rust compiler.	▼ Provides less comprehensive protection by either using a generalized encoding for all C- and C++-compiled code or across the FFI boundary.[III] ▼ Requires changes to Clang (either to implement the new encoding or to be able to identify and encode uses of Rust types across the FFI boundary). ▼ Requires new Clang extensions and changes to C and C++ code and libraries (either to be able to identify and encode uses of Rust types across the FFI boundary or to indicate which functions are called across the FFI boundary).[III] ▼ Requires defining a comprehensive encoding for Rust types using (language or vendor) extended types and implementing it in the Rust compiler. ▼ Requires defining a comprehensive encoding for C and C++ types using (language or vendor) extended types and implementing it in Clang. This introduces as much complexity and work as redefining Itanium C++ ABI mangling and reimplementing it in Clang.

[I] Green up-pointing triangle (▲) precedes a positive indicator.
[II] Red down-pointing triangle (▼) precedes a negative indicator.
[III] Mutually exclusive.

Defined type metadata identifiers (using Itanium C++ ABI mangling for encoding)

Table III defines type metadata identifiers for cross-language LLVM CFI support using Itanium C++ ABI mangling for encoding (1).

Table III
Type metadata identifiers for cross-language LLVM CFI support using Itanium C++ ABI mangling for encoding with vendor extended type qualifiers and types for Rust types that are not used across the FFI boundary.

Rust type	**Rust encoding**	**Itanium C++ ABI type**	**Itanium C++ ABI encoding**
()[4]	v	void	v
*mut core::ffi::c_void, *const core::ffi::c_void[5]	P[K]v	void *, const void *	P[K]v
bool	b	bool	b
core::ffi(::cfi)::c_char	c	char	c
core::ffi(::cfi)::c_schar	a	signed char	a
core::ffi(::cfi)::c_short	s	short	s
core::ffi(::cfi)::c_int	s	int	i
core::ffi(::cfi)::c_long	l	long	l
core::ffi(::cfi)::c_longlong	x	long long, __int64	x
core::ffi(::cfi)::c_ssize_t	i	long, long long, __int64	l, x
core::ffi(::cfi)::c_uchar	h	unsigned char	h
core::ffi(::cfi)::c_ushort	t	unsigned short	t
core::ffi(::cfi)::c_uint	s	unsigned int	i
core::ffi(::cfi)::c_ulong	m	unsigned long	m
core::ffi(::cfi)::c_ulonglong	y	unsigned long long, __int64	y

[4] () (i.e., unit type) is equivalent to void return type in C.
[5] *mut c_void is equivalent to void* in C and *const c_void is equivalent to const void* in C. Not the same as void return type in C, which is equivalent to () (i.e., unit type) in Rust.

core::ffi(::cfi)::c_size_t	j	unsigned long, unsigned long long, __int64	m, y
i8	u2i8	signed char	a
i16	u3i16	short	s
i32	u3i32	int, long	i, l
i64	u3i64	long, long long, __int64	l, x
i128	u4i128	__int128	n
isize	u5isize	long, long long, __int64	l, x
u8	u2u8	unsigned char	h
u16	u3u16	unsigned short	t
u32	u3u32	unsigned int, unsigned long	j, m
u64	u3u64	unsigned long, unsigned long long, __int64	m, y
u128	u4u128	unsigned __int128	o
usize	u5usize	unsigned long, unsigned long long, __int64	m, y
f32	f	float	f
f64	d	double	d
...[6]	z	ellipsis	z
char	u4char	n/a	
str	u3str	n/a	
never	u5never	n/a	
tuple	u5tupleI<element-type1 ..element-typeN>E	n/a	
array	A<array-length><element-type>	n/a	
slice	u5sliceI<element-type>E	n/a	

[6] Variadic parameters can only be specified with extern function types with the "C" calling convention.

struct, enum, union	<length><name>, where <name> is <unscoped-name>; <element-type1>; u<length><name>[I<element-type1..element-typeN>E], where <element-type> is <subst>	struct, enum, union	<length><name>
extern type	<length><name>, where <name> is <unscoped-name>	struct, enum, union	<length><name>
reference	[U3mut]u3refI<element-type>E	n/a	
raw pointer	P[K]<element-type>	pointer	P[K]<element-type>
function pointer, function item, closure, Fn trait object	PF<return-type><parameter-type1..parameter-typeN>E	function pointer	PF<return-type><parameter-type1..parameter-typeN>E
trait object	u3dynI<element-type1[..element-typeN]>E, where <element-type> is <predicate>	n/a	
lifetime/region	u6region[I[<region-disambiguator>]<region-index>E]	n/a	
const	L<element-type>[n][<element-value>]E	n/a	

To minimize the length of external names, the Itanium C++ ABI specifies a substitution encoding (i.e., compression)[14] to eliminate repetition of name components, which ends up also being applied to type metadata identifiers. Therefore, compression will also be implemented as specified in the Itanium C++ ABI Compression[14].

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`, ...), whose 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);
}
```
Fig. 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, when types are encoded, all type aliases are already resolved to their respective `ty::Ty` type representations[15] (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);
}
```
Fig. 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 when types are encoded.

To solve this, an RFC will be submitted proposing improving the existing set of C types in `core::ffi` or creating a new set of C types in `core::ffi::cfi` as user-defined types using `repr(transparent)` to be used in `extern "C"` function types indirectly called across the FFI boundary when cross-language LLVM CFI support is needed (see Table III). This improved or new set of C types will be used by 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.

The cfi_encoding attribute

To provide flexibility for the user, a `cfi_encoding` attribute will also be provided. The `cfi_encoding` attribute will allow 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;
}
```

Fig. 3. Example user-defined types using the cfi_encoding attribute.

For example, it will allow 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. 3).

Defined type metadata identifiers (creating a new encoding for cross-language CFI)

Creating a new encoding for cross-language CFI (2) was also explored with the Clang CFI team. This new encoding needed to be language agnostic and ideally compatible with any other language. It also needed to support extended types in case it was used as the main encoding to provide forward-edge control flow protection.

However, to satisfy these requirements, this new encoding neither distinguishes between certain types (e.g., `bool`, `char`, integers, and enums) nor discriminates between pointed element types.

This results in less comprehensive protection by either using a generalized encoding for all C- and C++-compiled code or across the FFI boundary, and will degrade the security of the program when linking foreign Rust-compiled code into a program written in C or C++ because the program previously used a more comprehensive encoding for all its compiled code.

This encoding may be provided as an alternative option for interoperating with foreign code written in languages other than C and C++ or that can not use Itanium C++ ABI mangling for encoding.

Table IV defines type metadata identifiers for cross-language LLVM CFI support creating a new encoding for cross-language CFI (2).

Table IV
Type metadata identifiers for cross-language LLVM CFI support creating a new encoding for cross-language CFI with extended types for Rust types that are not used across the FFI boundary.

Rust type	**Rust encoding**	**New encoding for cross-language CFI type**	**New encoding for cross-language CFI encoding**
()[7]	v	void	v
*mut core::ffi::c_void,	P	void *, const void *	P

[7] () (i.e., unit type) is equivalent to void return type in C.

*const core::ffi::c_void[8]			
bool	U8	n/a	
i8	I8	8-bit unsigned integer	I8
i16	I16	16-bit unsigned integer	I16
i32	I32	32-bit unsigned integer	I32
i64	I64	64-bit unsigned integer	I64
i128	I128	128-bit unsigned integer	I128
isize	IN, where N is pointer size	n/a	
u8	U8	Two's-complement 8-bit signed integer	U8
u16	U16	Two's-complement 16-bit signed integer	U16
u32	U32	Two's-complement 32-bit signed integer	U32
u64	U64	Two's-complement 64-bit signed integer	U64
u128	U128	Two's-complement 128-bit signed integer	U128
usize	UN, where N is pointer size	n/a	
f32	F32	IEEE 754 32-bit base-2 floating point	F32
f64	F64	IEEE 754 64-bit base-2 floating point	F64
...[9]	z	ellipsis	z
char	<lang-prefix>4char	n/a	
str	<lang-prefix>3str	n/a	
never	<lang-prefix>5never	n/a	
tuple	<lang-prefix>5tupleI<el	n/a	

[8] *mut c_void is equivalent to void* in C and *const c_void is equivalent to const void* in C. Not the same as void return type in C, which is equivalent to () (i.e., unit type) in Rust.
[9] Variadic parameters can only be specified with extern function types with the "C" calling convention.

	ement-type1..element-typeN>E		
array	A<array-length><element-type>	n/a	
slice	<lang-prefix>5sliceI<element-type>E	n/a	
struct, union	S<length><name>, where <name> is <unscoped-name>; <element-type1>; <lang-prefix><length><name>[I<element-type1..element-typeN>E], where <element-type> is <subst>	struct, union	S<length><name>
enum	IN, UN, where N is the enum layout size; <lang-prefix><length><name>[I<element-type1..element-typeN>E], where <element-type> is <subst>	enum	IN, UN, where N is the enum layout size
reference	<lang-prefix>[U3mut]u3refI<element-type>E	n/a	
raw pointer	P	pointer	P
function pointer, function item, closure, Fn trait object	P	function pointer	P
trait object	<lang-prefix>3dynI<element-type1[..element-typeN]>E, where <element-type> is <predicate>	n/a	
lifetime/region	<lang-prefix>u6region[I[<region-disambiguator>]<region-index>E]	n/a	
const	<lang-prefix>L<element-type>[n][<element-value>]E	n/a	

This encoding may be formally specified in a separate document.

Dependency considerations

LLVM CFI support depends on LLVM for code generation, the `LowerTypeTests` LLVM optimization pass, and Link Time Optimization (LTO). The Rust compiler already supports using LLVM for code generation, supports LTO, and the `LowerTypeTests` pass is already included in the default optimization pipelines for LTO.

Work estimates

LLVM CFI support for Rust work is broken in these steps:

- Create tracking issue (rust-lang/rust#89653)—*done.*
- Add option to the Rust compiler driver and frontend (rust-lang/rust#89652)—*done*.
- Add support for emitting type metadata and checks to the Rust compiler code generation (rust-lang/rust#89652)—*done*.
- Implement Rust-compiled code only LLVM CFI support by using Rust-specific type metadata identifiers (rust-lang/rust#89652, rust-lang/rust#95548)—*done*.
- Define type metadata identifiers for cross-language LLVM CFI support (See Type metadata)—*done*.
- Change initial implementation to use the defined type metadata identifiers for cross-language LLVM CFI support (rust-lang/rust#95548, rust-lang/rfcs#3296, https://reviews.llvm.org/D139395, rust-lang/rust#105452)—*done*.
- Create documentation and tests (rust-lang/rust#89652, rust-lang/rustc-dev-guide#1234, rust-lang/rust#95548, rust-lang/rust#105452)—*done*.

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`, and 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);
}
```

Fig. 4. 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
$
```

Fig. 5. Build and execution of Fig. 4 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
$
```

Fig. 6. Build and execution of Fig. 4 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. 6).

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);
}
```

Fig. 7. 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
$
```

Fig. 8. Build and execution of Fig. 7 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
$
```

Fig. 9. Build and execution of Fig. 7 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. 9).

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);
}
```

Fig. 10. 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
$
```

Fig. 11. Build and execution of Fig. 10 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
$
```

Fig. 12. Build and execution of Fig. 10 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. 12).

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);
}
```

Fig. 13. 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);
}
```

Fig. 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 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
$
```

Fig. 15. Build and execution of Figs. 13–14 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
$
```

Fig. 16. Build and execution of Figs. 13–14 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 in the call/branch site, even across the FFI boundary and for `extern "C"` function types indirectly called (i.e., callbacks/function pointers) across the FFI boundary, the execution is also terminated (see Fig. 16).

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.

References

1. Y. Song. “On Control Flow Hijacks of unsafe Rust.” GitHub. https://stanford-cs242.github.io/f17/assets/projects/2017/songyang.pdf.
2. A. Paverd. “Control Flow Guard for Rust.” GitHub. https://github.com/rust-lang/rust/files/4723836/Control.Flow.Guard.for.Rust.pdf.
3. A. Paverd. “Control Flow Guard for LLVM.” GitHub. https://github.com/rust-lang/rust/files/4723840/Control.Flow.Guard.for.LLVM.pdf.
4. M. Papaevripides and E. Athanasopoulos. “Exploiting Mixed Binaries.” ACM. https://dl.acm.org/doi/pdf/10.1145/3418898.
5. B. Spengler. “Open Source Security, Inc. Announces Funding of GCC Front-End for Rust.” Open Source Security. https://opensrcsec.com/open_source_security_announces_rust_gcc_funding
6. R. de C Valle. “Exploit Mitigations.” The rustc book. https://doc.rust-lang.org/nightly/rustc/exploit-mitigations.html.
7. “LLVM ControlFlowIntegrity (CFI).” Clang Documentation. https://clang.llvm.org/docs/ControlFlowIntegrity.html.
8. “Frequently Asked Questions About RAP.” Open Source Security. https://grsecurity.net/rap_faq.
9. “Backend Agnostic Codegen.” Guide to Rustc Development. https://rustc-dev-guide.rust-lang.org/backend/backend-agnostic.html.
10. “Type Metadata.” LLVM Documentation. https://llvm.org/docs/TypeMetadata.html.
11. “Itanium C++ ABI”. https://itanium-cxx-abi.github.io/cxx-abi/abi.html.
12. “Virtual Tables and RTTI”. Itanium C++ ABI. https://itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling-special-vtables.
13. “Type Encodings”. Itanium C++ ABI. https://itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling-type.
14. “Compression”. Itanium C++ ABI. https://itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling-compression.
15. “The ty module: representing types”. Guide to Rustc Development. https://rustc-dev-guide.rust-lang.org/ty.html.