Suporte a LLVM CFI e LLVM CFI Entre Linguagens para Rust

Autor: Ramon de C Valle

Digital Object Identifier

https://doi.org/10.47986/17/9

Nota do Editor

Esse artigo é relacionado à palestra do autor na conferência H2HC 2023 e não foi revisado pela equipe da H2HC Magazine.

Temos o prazer de compartilhar que trabalhamos com a comunidade Rust para adicionar suporte a LLVM CFI e LLVM CFI entre linguagens (e LLVM KCFI e LLVM KCFI entre linguagens) ao compilador Rust como parte do nosso trabalho no *Rust Exploit Mitigations Project Group*. Esta é a primeira implementação de *forward-edge control flow protection*, de fina granularidade, entre linguagens, para binários de linguagem mista que conhecemos.

À medida que a indústria continua a explorar a adoção de Rust, ataques entre linguagens em binários de linguagem mista (também conhecidos como "binários mistos"), e criticamente a ausência de suporte a *forward-edge control flow protection* no compilador Rust, são uma grande preocupação de segurança ao migrar gradualmente de C e C++ para Rust, e quando código compilado em C ou C++ e Rust compartilham o mesmo espaço de endereço virtual.

Introdução

Com a crescente popularidade de Rust como linguagem de programação de uso geral e como substituto de C e C++ devido às suas garantias de segurança de memória e de *thread*, muitas empresas e projetos estão adotando ou migrando para Rust. Um dos caminhos mais comuns para migrar para Rust é substituir gradualmente C ou C++ por Rust em um programa escrito em C ou C++.

Rust fornece interoperabilidade com código estrangeiro escrito em C via *Foreign Function Interface (FFI)*. No entanto, código estrangeiro não fornece as mesmas garantias de segurança de memória e de *thread* que Rust fornece, e é suscetível a corrupção de memória e problemas de simultaneidade.[1] Portanto, é geralmente aceito que introduzir código estrangeiro compilado em C ou C++ a um programa escrito em Rust pode degradar a segurança do programa.

[1]Compiladores C e C++ modernos fornecem mitigações de *exploit* para aumentar a dificuldade de explorar vulnerabilidades resultantes desses problemas. No entanto, algumas dessas mitigações de *exploit* não são aplicadas ao introduzir código estrangeiro compilado em C ou C++ a um programa escrito em Rust, principalmente devido à ausência de suporte a essas mitigações de *exploit* no compilador Rust (veja Tabela 1).

Embora também se acredite geralmente que substituir C ou C++ sensível por Rust em um programa escrito em C ou C++ melhora a segurança do programa, Papaevripides e Athanasopoulos demonstraram que nem sempre é esse o caso, e que introduzir código estrangeiro compilado em Rust a um programa escrito em C ou C++ com mitigações de *exploit* modernas, como *control flow protection*, pode na verdade degradar a segurança do programa, principalmente devido à ausência de suporte a essas mitigações de *exploit* no compilador Rust, principalmente *forward-edge control flow protection*. (Veja *Control flow protection*.) Isso foi posteriormente formalizado como uma nova classe de ataques (i.e., ataques entre linguagens).

O compilador Rust não suportava *forward-edge control flow protection* ao

- usar Unsafe Rust.
- introduzir código estrangeiro compilado em C ou C++ a um programa escrito em Rust.
- introduzir código estrangeiro compilado em Rust a um programa escrito em C ou C++.

A Tabela 1 resume os riscos relacionados à interoperabilidade ao compilar programas para o sistema operacional Linux na arquitetura AMD64 ou equivalente sem suporte a *forward-edge control flow protection* no compilador Rust.

	Sem usar Unsafe Rust	Usando Unsafe Rust
Código compilado somente em Rust	▾ [I] Chamadas indiretas em código compilado em Rust não são validadas.[2]	▾ Unsafe Rust é suscetível a corrupção de memória e problemas de simultaneidade. ▾ Chamadas indiretas em código compilado em Rust não são validadas.
Introduzindo código estrangeiro compilado em C ou C++ a um programa escrito em Rust	▾ Código estrangeiro é suscetível a corrupção de memória e problemas de simultaneidade. ▾ Chamadas indiretas em código compilado em Rust não são validadas.	▾ Código estrangeiro é suscetível a corrupção de memória e problemas de simultaneidade. ▾ Unsafe Rust é suscetível a corrupção de memória e problemas de simultaneidade. ▾ Chamadas indiretas em código compilado em Rust não são validadas.
Introduzindo código estrangeiro compilado em Rust a um programa escrito em C ou C++[3]	▴ [II] Chamadas indiretas em código compilado em C e C++ são validadas. ▾ C e C++ são suscetíveis a corrupção de memória e problemas de simultaneidade. ▾ Chamadas indiretas em código compilado em Rust não são validadas.	▴ Chamadas indiretas em código compilado em C e C++ são validadas. ▾ C e C++ são suscetíveis a corrupção de memória e problemas de simultaneidade. ▾ Unsafe Rust é suscetível a corrupção de memória e problemas de simultaneidade. ▾ Chamadas indiretas em código compilado em Rust não são validadas.

[I] Triângulo apontando para baixo (▾) precede um indicador de risco negativo.
[II] Triângulo apontando para cima (▴) precede um indicador de risco positivo.

Tabela 1: Resumo dos riscos relacionados à interoperabilidade ao compilar programas para o sistema operacional Linux na arquitetura AMD64 ou equivalente sem suporte a *forward-edge control flow protection* no compilador Rust.

Sem suporte a *forward-edge control flow protection* no compilador Rust, chamadas indiretas em código compilado em Rust não eram validadas, permitindo que a *forward-edge control flow protection* fosse contornada trivialmente conforme demonstrado por Papaevripides e Athanasopoulos. Portanto, a ausência de suporte a *forward-edge control flow protection* no compilador Rust foi uma grande preocupação de segurança ao migrar gradualmente de C e C++ para Rust, e quando código compilado em C ou C++ e Rust compartilham o mesmo espaço de endereço virtual.

[2] Um ataque que permite com sucesso um programa de código compilado somente em Rust, sem usar Unsafe Rust, ter seu fluxo de controle redirecionado como resultado de uma corrupção de memória ou problema de simultaneidade ainda não foi demonstrado.
[3] Assumindo que *forward-edge control flow protection* está habilitada.

Control flow protection

Control flow protection é uma mitigação de *exploit* que protege programas contra o redirecionamento de seu fluxo de controle. É classificado em duas categorias:

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

Forward-edge control flow protection

Forward-edge control flow protection protege programas contra o redirecionamento de seu fluxo de controle, realizando verificações para garantir que os destinos de chamadas indiretas sejam um de seus destinos válidos no gráfo de fluxo de controle. A abrangência dessas verificações varia de acordo com a implementação. Isso também é conhecido como *forward-edge control flow integrity (CFI)*.

Processadores mais novos fornecem assistência de hardware para *forward-edge control flow protection*, como ARM Branch Target Identification (BTI), ARM Pointer Authentication, e Intel Indirect Branch Tracking (IBT) como parte da Intel Control-flow Enforcement Technology (CET). No entanto, as implementações baseadas em ARM BTI e Intel IBT são menos abrangentes do que as implementações baseadas em software, como LLVM ControlFlowIntegrity (CFI), e o disponível comercialmente grsecurity/PaX Reuse Attack Protector (RAP).

Quanto menos abrangente for a proteção, maior é a probabilidade de ela ser contornada. Por exemplo, Microsoft Windows Control Flow Guard (CFG) testa apenas se o destino de uma chamada indireta é um ponto de entrada de função válido, o que equivale a agrupar todos os ponteiros de função em um único grupo, e testar se todos os destinos de chamadas indiretas estão neste grupo. Isso também é conhecido como *coarse-grained CFI*.

Isso significa que em uma tentativa de exploração, um atacante pode redirecionar o fluxo de controle para qualquer função e, quanto maior for o programa, maior será a probabilidade de um atacante encontrar uma função da qual possa se beneficiar (e.g., um pequeno programa de linha de comando versus um navegador).

Semelhante à implementação do Microsoft Windows CFG, esta é infelizmente a implementação que assistência de hardware para *forward-edge control flow protection* (e.g., ARM BTI e Intel IBT) foi inicialmente projetada com base e, como tal, fornece proteção equivalente com a adição de instruções especializadas. Microsoft Windows eXtended Flow Guard (XFG), ARM Pointer Authentication-based *forward-edge control flow protection* e Intel Fine Indirect Branch Tracking (FineIBT) visam resolver isso combinando assistência de hardware com testes de tipo de ponteiro de função baseados em software semelhantes ao LLVM CFI. Isso também é conhecido como *fine-grained CFI*.

Backward-edge control flow protection

Backward-edge control flow protection protege programas contra o redirecionamento de seu fluxo de controle, realizando verificações para garantir que os destinos de retornos de chamada sejam uma de suas origens válidas (i.e., locais de chamada) no gráfo de fluxo de controle. *Backward-edge control flow protection* está fora do escopo deste artigo.

Detalhes

Há vários detalhes no *design* e implementação de *forward-edge control flow protection*, de fina granularidade, entre linguagens usando testes de tipo de ponteiro de função entre diferentes linguagens. Esta seção documenta os principais desafios na implementação entre Rust e C ou C++, mais especificamente o compilador Rust e Clang.

Metadados de tipo

O LLVM usa metadados de tipo para permitir que os módulos IR agreguem ponteiros por seus tipos. Esses metadados de tipo são usados pelo LLVM CFI para testar se um determinado ponteiro está associado a um identificador de tipo (i.e., testar associação de tipo).

O Clang usa o nome da estrutura `typeinfo` da *virtual tables and RTTI* do Itanium C++ ABI como identificadores de metadados de tipo para ponteiros de função.

Para suporte a LLVM CFI entre linguagens, uma codificação compatível deve ser usada. A codificação compatível escolhida para suporte LLVM CFI entre linguagens é o *Itanium C++ ABI mangling* com *vendor extended type qualifiers and types* para tipos Rust que não são usados através da borda FFI (veja *Type metadata* no documento de *design*).

Codificando tipos inteiros C

Rust define `char` como um valor escalar Unicode, enquanto C define `char` como um tipo inteiro. Rust também define tipos inteiros de tamanho explícito (i.e., `i8`, `i16`, `i32`, ...), enquanto C define tipos inteiros abstratos (i.e., `char`, `short`, `long`, ...), cujos tamanhos reais são definidos pela implementação e podem variar entre diferentes modelos de dados. Isso causa ambiguidade se tipos inteiros Rust forem usados em tipos de função `extern "C"` que representam funções C porque o *Itanium C++ ABI* especifica codificações para tipos inteiros C (e.g., `char`, `short`, `long`, ...), e não suas representações definidas (e.g., inteiro com sinal de 8 bits, inteiro com sinal de 16 bits, inteiro com sinal de 32 bits, ...).

Por exemplo, o compilador Rust atualmente não consegue identificar se uma

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

Listagem 1: Exemplo de função extern "C" usando um tipo inteiro Rust.

representa uma `void func(long arg)` ou `void func(long long arg)` em um modelo de dados LP64 ou equivalente.

Para suporte a LLVM CFI entre linguagens, o compilador Rust deve ser capaz de identificar e codificar corretamente tipos C em tipos de função `extern "C"` chamadas indiretamente através da borda FFI quando CFI está habilitado.

Por conveniência, Rust fornece alguns *aliases* de tipos C para uso ao interoperar com código estrangeiro escrito em C, e esses *aliases* de tipos C podem ser usados para desambiguação. No entanto, no momento em que os tipos são codificados, todos os *aliases* de tipo já estão resolvidos para suas respectivas representações de tipo `ty::Ty` (i.e., seus respectivos tipos Rust), tornando atualmente impossível identificar o uso de *aliases* de tipos C a partir de seus tipos resolvidos.

Por exemplo, o compilador Rust atualmente também não consegue identificar que uma

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

Listagem 2: Exemplo de função extern "C" usando um alias de tipo C.

usou o *alias* de tipo `c_long` e não é capaz de desambiguar entre ela e uma `extern "C" fn func(arg: c_longlong)` em um modelo de dados LP64 ou equivalente.

Consequentemente, o compilador Rust é incapaz de identificar e codificar corretamente tipos C em tipos de função `extern "C"` chamadas indiretamente através da borda FFI quando CFI está habilitado:

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

// Esta definição tem o id de tipo "_ZTSFvlE".
void
hello_from_c(long arg)
{
    printf("Hello␣from␣C!\n");
}

// Esta definição tem o id de tipo "_ZTSFvPFvlElE"--isso pode ser ignorado para
// os propósitos deste exemplo.
void
```

```
indirect_call_from_c(void (*fn)(long), long arg)
{
    // Este local de chamada testa se o ponteiro de destino éum membro do grupo
    // derivado do mesmo id de tipo da declaração de fn, que tem o id de tipo
    // "_ZTSFvlE".
    //
    // Observe que como o teste está no local da chamada e égerado pelo Clang,
    // o id de tipo usado no teste écodificado pelo Clang.
    fn(arg);
}
```

Listagem 3: Exemplo de biblioteca C usando tipos inteiros C e codificação do Clang.

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

#[link(name = "foo")]
extern "C" {
    // Esta declaração teria o id de tipo "_ZTSFvlE", mas no momento em que os
    // tipos são codificados, todos os aliases de tipo já estão resolvidos para
    // seus respectivos tipos Rust, então isso écodificado como "_ZTSFvu3i32E"
    // ou "_ZTSFvu3i64E", dependendo para qual tipo o alias de tipo c_long é
    // resolvido, que atualmente usa a codificação vendor extended type
    // u<length><type-name> para os tipos inteiros Rust—esse éo problema
    // demonstrado neste exemplo.
    fn hello_from_c(_: c_long);

    // Esta declaração teria o id de tipo "_ZTSFvPFvlElE", mas écodificado como
    // "_ZTSFvPFvu3i32ES_E" (comprimido) ou "_ZTSFvPFvu3i64ES_E" (comprimido),
    // da mesma forma que a declaração hello_from_c acima——isso pode pode ser
    // ignorado para os propósitos deste exemplo.
    fn indirect_call_from_c(f: unsafe extern "C" fn(c_long), arg: c_long);
}

// Esta definição teria o id de tipo "_ZTSFvlE", mas écodificado como
// "_ZTSFvu3i32E" ou "_ZTSFvu3i64E", da mesma forma que a declaração
// hello_from_c acima.
unsafe extern "C" fn hello_from_rust(_: c_long) {
    println!("Hello, world!");
}

// Esta definição teria o id de tipo "_ZTSFvlE", mas écodificado como
// "_ZTSFvu3i32E" ou "_ZTSFvu3i64E", da mesma forma que a declaração
// hello_from_c acima.
unsafe extern "C" fn hello_from_rust_again(_: c_long) {
    println!("Hello from Rust again!");
}

// Esta definição também teria o id de tipo "_ZTSFvPFvlElE", mas écodificado
// como "_ZTSFvPFvu3i32ES_E" (comprimido) ou "_ZTSFvPFvu3i64ES_E" (comprimido),
// da mesma forma que a declaração hello_from_c acima——isso pode ser ignorado
// para os propósitos deste exemplo.
```

```
fn indirect_call(f: unsafe extern "C" fn(c_long), arg: c_long) {
    // Este local de chamada indireta testa se o ponteiro de destino éum membro
    // do grupo derivado do mesmo id de tipo da declaração de f, que teria o id
    // de tipo "_ZTSFvlE", mas écodificado como como "_ZTSFvu3i32E" ou
    // "_ZTSFvu3i64E", da mesma forma que a declaração hello_from_c acima.
    //
    // Observe que como o teste está no local da chamada e égerado pelo
    // compilador Rust, o id de tipo usado no teste écodificado pelo compilador
    // Rust.
    unsafe { f(arg) }
}

// Esta definição tem o id de tipo "_ZTSFvvE"—isso pode ser ignorado para os
// propósitos deste exemplo.
fn main() {
    // Isto demonstra uma chamada indireta dentro do código somente em Rust
    // usando a mesma codificação para hello_from_rust e o teste no local de
    // chamada indireta em indirect_call (i.e., "_ZTSFvu3i32E" ou
    // "_ZTSFvu3i64E").
    indirect_call(hello_from_rust, 5);

    // Isto demonstra uma chamada indireta através da borda FFI com o compilador
    // Rust e Clang usando codificações diferentes para hello_from_c e o teste
    // no local de chamada indireta em indirect_call (i.e., "_ZTSFvu3i32E" ou
    // "_ZTSFvu3i64E" vs "_ZTSFvlE").
    //
    // Ao usar o LTO do rustc (i.e., −Clto), isto funciona porque o id de tipo
    // usado éda hello_from_c declarada em Rust, que écodificado pelo
    // compilador Rust (i.e., "_ZTSFvu3i32E" ou "_ZTSFvu3i64E").
    //
    // Ao usar LTO (adequado) (i.e., −clinker−plugin−lto), isto não funciona
    // porque o id de tipo usado éda hello_from_c definida em C, que é
    // codificado pelo Clang (i.e., "_ZTSFvlE").
    indirect_call(hello_from_c, 5);

    // Isto demonstra uma chamada indireta para uma função passada como uma
    // callback através da borda FFI com o compilador Rust e Clang usando
    // codificações diferentes para hello_from_rust_again e o teste no local de
    // chamada indireta em indirect_call_from_c (i.e., "_ZTSFvu3i32E" ou
    // "_ZTSFvu3i64E" vs "_ZTSFvlE").
    //
    // Quando funções Rust são passadas como callbacks através da borda FFI para
    // serem chamadas de volta do código C, os testes também estão nos locais de
    // chamadas, mas ao invés gerados por Clang, então os ids de tipo usados nos
    // testes são codificados pelo Clang, que não correspondem aos ids de tipo
    // de declarações codificados pelo compilador Rust (e.g.,
    // hello_from_rust_again). (O mesmo acontece ao contrário para as funções C
    // passadas como como callbacks através da borda FFI para serem chamadas de
    // volta do código Rust.)
    unsafe {
        indirect_call_from_c(hello_from_rust_again, 5);
    }
```

```
}
```

Listagem 4: Exemplo de programa Rust usando tipos inteiros Rust e a codificação do compilador Rust.

Sempre que há uma chamada indireta através da borda FFI ou uma chamada indireta para uma função passada como uma *callback* através da borda FFI, o compilador Rust e Clang usam codificações diferentes para tipos inteiros C para definições e declarações de função, e em locais de chamada indireta quando CFI está habilitado (veja Figs. 3–4).

A opção de normalização de inteiros

Para resolver o problema de codificação de tipos inteiros C, adicionamos uma opção de normalização de inteiros ao Clang (i.e., `-fsanitize-cfi-icall-experimental-normalize-integers`). Esta opção habilita a normalização de tipos inteiros como *vendor extended types* para suporte a LLVM CFI (e LLVM KCFI) entre linguagens com outras linguagens que não podem representar e codificar tipos inteiros C.

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

// Esta definição tem o id de tipo "_ZTSFvlE", mas será codificado como
// "_ZTSFvu3i32E" ou "_ZTSFvu3i64E", dependendo do modelo de dados, se a opção
// de normalização de inteiros está habilitada, que usa a codificação vendor
// extended type u<length><type-name> para os tipos inteiros C.
void
hello_from_c(long arg)
{
    printf("Hello␣from␣C!\n");
}

// Esta definição tem o id de tipo "_ZTSFvPFvlElE", mas será codificado como
// "_ZTSFvPFvu3i32ES_E" (comprimido) ou "_ZTSFvPFvu3i64ES_E" (comprimido),
// dependendo do modelo de dados, se a opção de normalização de inteiros está
// habilitada--isso pode ser ignorado para os propósitos deste exemplo.
void
indirect_call_from_c(void (*fn)(long), long arg)
{
    // Este local de chamada testa se o ponteiro de destino éum membro do grupo
    // derivado do mesmo id de tipo da declaração de fn, que tem o id de tipo
    // "_ZTSFvlE", mas será codificado como "_ZTSFvu3i32E" ou "_ZTSFvu3i64E",
    // dependendo do modelo de dados, se a opção de normalização de inteiros
    // está habilitada.
    fn(arg);
}
```

Listagem 5: Exemplo de biblioteca C usando tipos inteiros C e codificação Clang com a opção de normalização de inteiros habilitada.

Especificamente, os tipos inteiros são codificados como suas representações definidas (e.g., inteiro com sinal de 8 bits, inteiro com sinal de 16 bits, inteiro com sinal de 32 bits, ...) para compatibilidade com linguagens que definem tipos inteiros de tamanho explícito (e.g., `i8`, `i16`, `i32`, ..., em Rust) (veja Fig. 5).

Isso faz com que LLVM CFI (e LLVM KCFI) entre linguagens funcione sem alterações, com perda mínima de granularidade.[4]

O atributo cfi_encoding

Para fornecer flexibilidade ao usuário, também fornecemos o atributo cfi_encoding. O atributo `cfi_encoding` permite ao usuário definir a codificação CFI para tipos definidos pelo usuário.

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

Listagem 6: Exemplo de tipos definidos pelo usuário usando o atributo cfi_encoding.

Ele permite ao usuário usar nomes diferentes para tipos que de outra forma seriam obrigados a ter o mesmo nome usado em funções C definidas externamente (veja Fig. 6).

A crate cfi_types

Alternativamente, para resolver também o problema de codificação de tipos inteiros C, fornecemos a *crate* `cfi_types`. Esta *crate* fornece um novo conjunto de tipos C como tipos definidos pelo usuário usando o atributo `cfi_encoding` e `repr(transparent)` para serem usados para suporte a LLVM CFI entre linguagens.

```
use cfi_types::c_long;

#[link(name = "foo")]
extern "C" {
    // Esta declaração tem o id de tipo "_ZTSFvlE" porque usa os tipos CFI para
    // suporte a LLVM CFI entre linguagens. A cfi_types crate fornece um novo
    // conjunto de tipos C como tipos definidos pelo usuário usando o atributo
    // cfi_encoding e repr(transparent) para serem usados para suporte a LLVM
```

[4] E.g., https://github.com/rust-lang/rfcs/pull/3296#issuecomment-1432190581 ~1% no kernel do Linux.

```
    // CFI entre linguagens. Este novo conjunto de tipos C permite que o
    // compilador Rust identifique e codifique corretamente os tipos C em tipos
    // de função extern "C" chamados indiretamente através da borda FFI quando
    // CFI está habilitado.
    fn hello_from_c(_: c_long);

    // Esta declaração tem o id de tipo "_ZTSFvPFvlElE" porque usa os tipos CFI
    // para suporte a LLVM CFI entre linguagens——isso pode ser ignorado para os
    // propósitos deste exemplo.
    fn indirect_call_from_c(f: unsafe extern "C" fn(c_long), arg: c_long);
}

// Esta definição tem o id de tipo "_ZTSFvlE" porque usa os tipos CFI para
// suporte a LLVM CFI entre linguagens, da mesma forma que a declaração
// hello_from_c acima.
unsafe extern "C" fn hello_from_rust(_: c_long) {
    println!("Hello, world!");
}

// Esta definição tem o id de tipo "_ZTSFvlE" porque usa os tipos CFI para
// suporte a LLVM CFI entre linguagens, da mesma forma que a declaração
// hello_from_c acima.
unsafe extern "C" fn hello_from_rust_again(_: c_long) {
    println!("Hello from Rust again!");
}

// Esta definição também tem o id de tipo "_ZTSFvPFvlElE" porque usa os tipos
// CFI para suporte a LLVM CFI entre linguagens, da mesma forma que a declaração
// hello_from_c acima——isso pode ser ignorado para os propósitos deste exemplo.
fn indirect_call(f: unsafe extern "C" fn(c_long), arg: c_long) {
    // Este local de chamada indireta testa se o ponteiro de destino éum membro
    // do grupo derivado do mesmo id de tipo da declaração de f, que tem o id de
    // tipo "_ZTSFvlE" porque usa os tipos CFI para suporte a LLVM CFI entre
    // linguagens, da mesma forma que a declaração hello_from_c acima.
    unsafe { f(arg) }
}

// Esta definição tem o id de tipo "_ZTSFvvE"——isso pode ser ignorado para os
// propósitos deste exemplo.
fn main() {
    // Isto demonstra uma chamada indireta dentro do código somente em Rust
    // usando a mesma codificação para hello_from_rust e o teste no local de
    // chamada indireta em indirect_call (i.e., "_ZTSFvlE").
    indirect_call(hello_from_rust, c_long(5));

    // Isto demonstra uma chamada indireta através da borda FFI com o compilador
    // Rust e Clang usando a mesma codificação para hello_from_c e o teste no
    // local de chamada indireta em indirect_call (i.e., "_ZTSFvlE").
    indirect_call(hello_from_c, c_long(5));

    // Isto demonstra uma chamada indireta para uma função passada como uma
    // callback através da borda FFI com o compilador Rust e Clang usando a
    // mesma codificação para hello_from_rust_again e o teste no local de
    // chamada indireta em indirect_call_from_c (i.e., "_ZTSFvlE").
```

```
    unsafe {
        indirect_call_from_c(hello_from_rust_again, c_long(5));
    }
}
```

Listagem 7: Exemplo de programa Rust usando tipos inteiros Rust e a codificação do compilador Rust com os tipos da crate cfi_types.

Este novo conjunto de tipos C permite que o compilador Rust identifique e codifique corretamente tipos C em tipos de função `extern "C"` chamadas indiretamente através da borda FFI quando CFI está habilitado (veja Fig 7).

Resultados

O suporte LLVM CFI no compilador Rust fornece *forward-edge control flow protection* para código compilado somente em Rust e para binários de linguagem mista de código compilado em C ou C++ e Rust, também conhecidos como "binários mistos" (i.e., para quando código compilado em C ou C++ e Rust compartilham o mesmo espaço de endereço virtual), agregando ponteiros de função em grupos identificados por seus tipos de retorno e parâmetro.

LLVM CFI pode ser habilitado com `-Zsanitizer=cfi` e requer LTO (i.e., `-Clinker-plugin-lto` ou `-Clto`). LLVM CFI entre linguagens pode ser habilitado com `-Zsanitizer=cfi`, requer que a opção `-Zsanitizer-cfi-normalize-integers` seja usada com a opção do Clang `-fsanitize-cfi-icall-experimental-normalize-integers` para suporte LLVM CFI entre linguagens, e LTO adequado (i.e., não-rustc) (i.e., `-Clinker-plugin-lto`).

É recomendado recompilar a biblioteca padrão com CFI habilitado usando o recurso Cargo build-std (i.e., `-Zbuild-std`) ao habilitar o CFI.

Exemplo 1: Redirecionando o fluxo de controle usando uma chamada indireta para um destino inválido

```
#![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 {
        // O Offset 0 éum destino de chamada válido (i.e., o ponto de entrada
        // da função), mas os offsets 1–8 dentro do bloco de NOPs são destinos
        // de chamada inválidos (i.e., dentro do corpo da função).
        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);
}
```

Listagem 8: Redirecionando o fluxo de controle usando uma chamada indireta para um destino inválido (i.e., dentro do corpo da função).

```
$ 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
$
```

Listagem 9: Compilação e execução da Fig. 8 com LLVM CFI desabilitado.

```
$ 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
$
```

Listagem 10: Compilação e execução da Fig. 8 com LLVM CFI habilitado.

Quando o LLVM CFI está habilitado, se houver alguma tentativa de redirecionar o fluxo de controle usando uma chamada indireta para um destino inválido, a execução é terminada (veja Fig. 10).

Exemplo 2: Redirecionando o fluxo de controle usando uma chamada indireta para uma função com um número diferente de parâmetros

```
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);
}
```

Listagem 11: Redirecionando o fluxo de controle usando uma chamada indireta para uma função com um número de parâmetros diferente dos argumentos passados no local de chamada.

```
$ 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
$
```

Listagem 12: Compilação e execução da Fig. 11 com LLVM CFI desabilitado.

```
$ 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
$
```

Listagem 13: Compilação e execução da Fig. 11 com LLVM CFI habilitado.

Quando o LLVM CFI está habilitado, se houver alguma tentativa de redirecionar o fluxo de controle usando uma chamada indireta para uma função com um número de parâmetros diferente dos argumentos passados no local de chamada, a execução também é terminada (veja Fig. 13).

Exemplo 3: Redirecionando o fluxo de controle usando uma chamada indireta para uma função com diferentes tipos de retorno e parâmetro

```
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);
}
```

Listagem 14: Redirecionando o fluxo de controle usando uma chamada indireta para uma função com tipos de retorno e parâmetro diferentes do tipo de retorno esperado e argumentos passados no local de chamada.

```
$ 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
$
```

Listagem 15: Compilação e execução da Fig. 14 com LLVM CFI desabilitado.

```
$ 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
$
```

Listagem 16: Compilação e execução da Fig. 14 com LLVM CFI habilitado.

Quando o LLVM CFI está habilitado, se houver alguma tentativa de redirecionar o fluxo de controle usando uma chamada indireta para uma função com tipos de retorno e parâmetro diferentes do tipo de retorno

esperado e argumentos passados no local de chamada, a execução também é terminada (veja Fig. 16).

Exemplo 4: Redirecionando o fluxo de controle usando uma chamada indireta para uma função com diferentes tipos de retorno e parâmetro através da borda FFI

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

Listagem 17: Exemplo de biblioteca C.

```
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);
}
```

Listagem 18: Redirecionando o fluxo de controle usando uma chamada indireta para uma função com tipos de retorno e de parâmetro diferentes do tipo de retorno esperado e argumentos passados no local de chamada, através da borda FFI

```
$ 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
$
```

Listagem 19: Compilação e execução das Figs. 17–18 com LLVM CFI desabilitado.

```
$ 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
$
```

Listagem 20: Compilação e execução das Figs. 17–18 com LLVM CFI habilitado.

Quando o LLVM CFI está habilitado, se houver alguma tentativa de redirecionar o fluxo de controle usando uma chamada indireta para uma função com tipos de retorno e de parâmetro diferentes do tipo de retorno esperado e argumentos passados no local de chamada, mesmo através da borda FFI e para funções passada como uma *callback* através da borda FFI através da borda FFI, a execução também é terminada (veja Fig. 20).

Desempenho

Testes preliminares de desempenho (i.e., um bilhão de chamadas para uma função contendo uma chamada indireta sem e com LLVM CFI habilitado comparados) usando cargo bench indicam impacto insignificante no desempenho (i.e., 0,01%).

Conclusão

LLVM CFI e LLVM CFI entre linguagens (e LLVM KCFI e LLVM KCFI entre linguagens) estão disponíveis em *nightly builds* do compilador Rust. À medida que trabalhamos para estabilizar esses recursos (veja nosso *roadmap*), estamos começando a experimentá-los em nossos produtos e serviços, e incentivamos você a experimentar também e nos informar se tiver algum problema (veja os problemas conhecidos).

Esperamos que este trabalho também forneça a base para futuras implementações de *forward-edge control flow protection*, de fina granularidade, entre linguagens, combinada com suporte de hardware e baseada em software, como Microsoft Windows XFG, ARM Pointer Authentication-based forward-edge control flow protection, e Intel FineIBT.

Agradecimentos

Agradecimentos a 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), e a comunidade Rust por toda a ajuda neste projeto.

Ramon de C Valle - rcvalle@google.com

Ramon é Engenheiro de Segurança da Informação no Google, trabalhando com pesquisa de vulnerabilidades e desenvolvimento de mitigações. Também é um dos primeiros desenvolvedores e contribuidor de longa data do Metasploit, e líder do Exploit Mitigations Project Group do compilador Rust. (Veja https://rcvalle.com/about.)