Control Flow Guard for Rust

Andrew Paverd <andrew.paverd@microsoft.com>

This document describes the technical design of Control Flow Guard (CFG) for Rust.

Rust tracking issue: [Tracking issue for `-Z control_flow_guard` #68793]

Control Flow Guard background

[Control Flow Guard (CFG)] is an exploit mitigation designed to enforce control-flow integrity for software running on supported Windows platforms. Specifically, CFG uses runtime checks to validate the target address of every indirect call/jump before allowing the call to complete.

During compilation, the compiler identifies all indirect calls/jumps and adds CFG checks. It also emits metadata containing the relative addresses of all address-taken functions. At runtime, if the binary is run on a CFG-aware operating system[1], the loader uses the CFG metadata to generate a bitmap of the address space and marks those addresses that contain valid targets. On each indirect call, the inserted check determines whether the target address is marked in this bitmap. If the target is not valid, the process is terminated.

CFG is complementary to other exploit mitigations, such as Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP), which are available for Rust code linked with MSVC.

In terms of interoperability:

- Code compiled with CFG enabled can be linked with libraries and object files that are not compiled with CFG. In this case, a CFG-aware linker can identify address-taken functions in the non-CFG libraries.
- Libraries compiled with CFG can linked into non-CFG programs. In this case, the CFG runtime checks in the libraries are not used (i.e. the mitigation is completely disabled).

Control Flow Guard in Rust

The primary motivation for enabling CFG in Rust is to enhance security when linking against non-Rust code, especially C/C++ code. To achieve full CFG protection, all indirect calls (including any from Rust code) must have the appropriate CFG checks. The following table summarizes the potential for control flow integrity violations without and with CFG for Rust.

	Without Rust CFG	**With Rust CFG**
Rust program calling C/C++	No CFG protection for either Rust or non-Rust code. Potential for control flow integrity violations arising from either C/C++ code or unsafe Rust.	All indirect calls/jumps protected.
C/C++ program[2] calling Rust	Program can have CFG enabled, but indirect calls in the Rust code will be unprotected. Potential for control flow integrity violations if vulnerabilities in the C/C++ code allow corruption of control flow information in Rust.	All indirect calls/jumps protected.

[1] CFG is available in Windows operating systems from Windows 8.1 onwards.
[2] Assuming CFG is enabled for the C/C++ program.

CFG can also improve security for Rust code that uses the `unsafe` keyword. The following contrived example demonstrates a violation of control flow integrity[3] in Rust:

```
fn add_one(x: &mut i32) {
    *x += 1;
}

// Return a function pointer into the middle of the above function
fn get_add_one_fptr() -> fn(&mut i32) {
    return unsafe {std::mem::transmute::<*const u8, fn(&mut i32)>((add_one as
                    *const u8).offset(0x28))};
}

fn main() {
    // Calling the function by name behaves as expected
    let mut x = 1;
    add_one(&mut x);
    println!("Calling function by name: 1 + 1 = {}", x);

    // Calling via the function pointer leads to invalid control flow
    let mut x = 1;
    let func_ptr = get_add_one_fptr();
    func_ptr(&mut x);
    println!("Calling by function pointer: 1 + 1 = {}", x);
}
```

When compiled without CFG, the output of the above program[4] is:

```
Calling function by name: 1 + 1 = 2
Calling by function pointer: 1 + 1 = 1
```

When compiled with CFG enabled, the above program is terminated before the invalid call completes.

Using CFG in Rust

Enabling CFG does not require developers to make code changes. Developers can enable CFG using the `control_flow_guard=checks` compiler option.

For testing and debugging purposes, the `control_flow_guard=nochecks` option emits CFG metadata without adding runtime checks (this should not be used in production as it does not provide security enforcement). These flags are ignored on all non-Windows targets.

[3] Further examples of violating control flow integrity in unsafe Rust are available at: http://cs242.stanford.edu/assets/projects/2017/songyang.pdf

[4] Note: the 0x28 offset may need to be modified depending on the platform.

Implementation in rustc

All CFG functionality is implemented in LLVM[5]. LLVM 10.0 supports CFG for x86 (32 and 64 bit), ARM, and Aarch64 targets. The role of rustc is to parse the above compiler options and pass this information down to LLVM. Specifically, to enable CFG for an LLVM compilation module, rustc must add the `cfguard` LLVM module flag with a value of 1 (emit metadata but no checks) or 2 (emit metadata and checks). The compiler must also pass the `/guard:cf` option to the MSVC linker.

Relevant files:

- src/librustc_codegen_llvm/context.rs
- src/librustc_codegen_ssa/back/linker.rs

The following test cases have been added:

- Ensure the LLVM module flag is correctly set when checks are requested (src/test/codegen/cfguard_checks.rs)
- Ensure the LLVM module flag is not set when CFG is not requested (src/test/codegen/cfguard_disabled.rs)
- Ensure the LLVM module flag is correctly set when only CFG metadata is requested (src/test/codegen/cfguard_nochecks.rs)

Enabling CFG in libraries

To provide full protection, CFG should be enabled for the Rust standard library. If building the standard library from source, this can be done by setting `control-flow-guard = true` in the config.toml file.

This can also be achieved using cargo's `-Z build-std` functionality to recompile the standard library with the same configuration options as the main program. For example:

CMD

```
rustup toolchain install --force nightly
rustup component add rust-src
SET RUSTFLAGS=-Z control_flow_guard=checks
cargo +nightly build -Z build-std --target x86_64-pc-windows-msvc
```

Powershell

```
rustup toolchain install --force nightly
rustup component add rust-src
$Env:RUSTFLAGS = "-Z control_flow_guard=checks"
cargo +nightly build -Z build-std --target x86_64-pc-windows-msvc
```

Ideally, the distributed version of the standard library would be compiled with CFG enabled, so that it includes the necessary runtime checks and metadata. The runtime checks would only be used if the final program is compiled with CFG enabled.

[5] For full details of the LLVM implementation, please see: https://reviews.llvm.org/D65761

Overhead of Control Flow Guard

The CFG checks and metadata can potentially increase binary size and runtime overhead. The magnitude of any increase depends on the number and frequency of indirect calls.

Binary size: Enabling CFG for the Rust standard library increases binary size by approximately 0.14%.

Runtime overhead[6]: Enabling CFG in the SPEC CPU 2017 Integer Speed benchmark suite (compiled with Clang/LLVM) incurs approximate runtime overheads of up to 8%, as shown in the table below, with a geometric mean of 2.9%.

Benchmark[7]	Without CFG (seconds)	With CFG (seconds)	Overhead
600.perlbench_s	314	322	2.5%
602.gcc_s	538	546	1.5%
605.mcf_s	723	767	6.1%
620.omnetpp_s	486	521	7.2%
623.xalancbmk_s	225	243	8.0%
625.x264_s	186	193	3.8%
631.deepsjeng_s	326	323	-0.9%
641.leela_s	435	428	-1.6%
657.xz_s	487	488	0.2%
Geometric mean	**381.6**	**392.7**	**2.9%**

Use in other systems

OS support: CFG is enabled for the majority of components and libraries in the Windows operating system.

Compiler support: CFG support is available in Microsoft Visual Studio and was added to Clang/LLVM in version 10.0.

[6] Currently there are no standardized runtime benchmarks for Rust on Windows, so the SPEC CPU 2017 benchmarks are given as representative overhead estimates.

[7] Quoted times are the median of three runs. The 648.exchange2_s benchmark requires Fortran, so was not included.