Control Flow Guard for LLVM

Andrew Paverd <andrew.paverd@microsoft.com>

This document describes the technical design of Control Flow Guard (CFG) for the LLVM compiler infrastructure, and the rationale for the various design choices.

Control Flow Guard shipped in the LLVM 10.0 release. This includes the dispatch mechanism for X86 (64-bit) targets, and the check mechanism for X86 (32-bit), ARM, and AArch64 targets. The Clang frontend supports the __declspec(guard(nocf)) modifier to elide CFG checks on specific functions.

Relevant commits

- d157a9b: Add Windows Control Flow Guard checks (/guard:cf)
- bdd88b7: Add support for __declspec(guard(nocf))

Control Flow Guard background

Control Flow Guard (CFG) is an exploit mitigation designed to ensure control-flow integrity for software running on Windows platforms. Specifically, in a CFG-enabled binary, runtime checks are added to ensure that the target address of every indirect branch corresponds to the address of a valid (i.e. address-taken) function. During compilation, the relative addresses of address-taken functions are emitted as metadata. When the image is loaded, the loader uses this list to generate a bitmap of addresses and marks the addresses that contain valid targets. On each indirect call, the target address is checked against this bitmap, and an error is raised if the target is not valid.

Pre-existing functionality

LLVM already included basic functionality (D42592 and D50513) for identifying address-taken functions and emitting the table of their addresses (i.e. the FID table), to facilitate linking with other CFG-enabled objects. This functionality remains unchanged.

Overview of changes

The guiding principle underpinning all of these changes has been to match the behaviour of MSVC as closely as possible.

The CFG checks on indirect function calls are primarily added by a new LLVM IR transform (the CFGuard library). This is added to the X86, ARM, and AArch64 backends before instruction selection. One new calling convention (CFGuard_Check) was required, with target-specific implementations added to the relevant backends.

The primary rationale for this design is that most of the CFG functionality is target-independent, but some target-specific options (e.g. register selection) are required. This design minimizes code duplication across different targets and allows new targets to be easily added in the future by defining the architecture-specific calling convention and adding the appropriate IR transform pass.

The following sections describe the individual components of this design:

CFGuard transform

This transform contains most of the functionality for adding CFG checks to indirect calls. It extends FunctionPass because it operates on functions individually. It can be instantiated in either check or dispatch mode, depending on the CFG mechanism required for the specific target, using createCFGuardCheckPass() or createCFGuardDispatchPass().

The doInitialization() method checks that the module has the cfguard flag, sets up the prototypes for the guard check and guard dispatch functions, and gets a reference to the relevant global symbol (__guard_check_icall_fptr or __guard_dispatch_icall_fptr). This is called once per module.

The runOnFunction() method iterates over the basic blocks to identify indirect call instructions. The CallBase superclass is used to identify Call, Invoke, and CallBr instructions. Depending on the mechanism selected, each call instruction is passed to the insertCFGuardCheck(CallBase *) or insertCFGuardDispatch(CallBase *) functions.

insertCFGuardCheck()

This function inserts a new call instruction immediately before the indirect call. The target of the new call instruction is the __guard_check_icall_fptr global symbol, with the address of the target function passed as the only argument. The new CFGuard_Check calling convention places this value in an architecture-specific register as follows:

X86_32: Target address passed in ECX; check function preserves ECX and floating point registers.
Aarch64: Target address passed in X15; check function preserves registers X0-X8 and Q0-Q7.
ARM: Target address passed in R0; check function preserves floating point registers.

The check function has no return value (an error will be raised if the target is not valid).

For example, the following LLVM IR:

```
%func_ptr = alloca i32 ()*, align 8
store i32 ()* @target_func, i32 ()** %func_ptr, align 8
%0 = load i32 ()*, i32 ()** %func_ptr, align 8
%1 = call i32 %0()
```

is transformed to:

```
%func_ptr = alloca i32 ()*, align 8
store i32 ()* @target_func, i32 ()** %func_ptr, align 8
%0 = load i32 ()*, i32 ()** %func_ptr, align 8
%1 = load void (i8*)*, void (i8*)** @__guard_check_icall_fptr
%2 = bitcast i32 ()* %0 to i8*
call cfguard_checkcc void %1(i8* %2)
%3 = call i32 %0()
```

For example, the following X86 assembly code:

```
movl  $_target_func, %eax
calll *%eax
```

is transformed to:

```
movl   $_target_func, %ecx
calll  *___guard_check_icall_fptr
calll  *%ecx
```

insertCFGuardDispatch()

This function replaces the indirect call instruction with a call to the guard dispatch mechanism, via the __guard_dispatch_icall_fptr global symbol. The __guard_dispatch_icall_fptr global symbol is loaded and set as the target of the new call instruction. The original indirect call target is added as a new type of operand bundle (cfguardtarget) to the new call.

The CFGuardTarget operand bundle is lowered to an ArgListEntry by the SelectionDagBuilder. Since the dispatch mechanism is currently only used on 64-bit X86 targets, the CC_X86_Win64_C calling convention has been modified to pass the CFGuardTarget parameter (if any) in register RAX. All other call arguments are passes as usual.

If the dispatch mechanism becomes supported on other targets, the relevant calling conventions can be similarly modified to pass the CFGuardTarget parameter in the appropriate register.

At runtime, the dispatch function checks that the target address is in the FID table and, if so, tail-calls the target.

For example, the following LLVM IR:

```
%func_ptr = alloca i32 ()*, align 8
store i32 ()* @target_func, i32 ()** %func_ptr, align 8
%0 = load i32 ()*, i32 ()** %func_ptr, align 8
%1 = call i32 %0()
```

is transformed to:

```
%func_ptr = alloca i32 ()*, align 8
store i32 ()* @target_func, i32 ()** %func_ptr, align 8
%0 = load i32 ()*, i32 ()** %func_ptr, align 8
%1 = load i32 ()*, i32 ()** @__guard_dispatch_icall_fptr
%2 = call i32 %1() [ “cfguardtarget”(i32 ()* %0) ]
```

The following X86_64 assembly code:

```
leaq   target_func(%rip), %rax
callq  *%rax
```

is transformed to:

```
leaq   target_func(%rip), %rax
callq  *__guard_dispatch_icall_fptr(%rip)
```

Relevant files:

- llvm\lib\Transforms\CFGuard\CFGuard.cpp
- llvm\include\llvm\Transforms\CFGuard.h

CFGuardLongJmp Pass

When linking with /guard:cf, link.exe switches to using a CFG-enabled version of longjmp. It therefore expects to find a list of valid longjmp targets in the .gljmp section of each object file. This pass iterates over all instructions in a MachineFunction to identify calls to setjmp. For each such call, it inserts a new MCSymbol and adds this to the list of valid longjmp targets stored by each MachineFunction.

Finally, code in the WinCFGuard AsmPrinterHandler writes the list of valid target symbols to the .gljmp section.

Relevant files:

- llvm\lib\CodeGen\CFGuardLongjmp.cpp
- llvm\lib\CodeGen\AsmPrinter\WinCFGuard.cpp

Support for __declspec(guard(nocf))

Some uses cases (e.g. bug 44096) require the ability to omit CFG checks on specific functions. This is achieved using the __declspec(guard(nocf)) modifier. This modifier has been added to Clang and is implemented as a new function attribute ("guard_nocf") in LLVM. CFG checks will not be added to functions with the "guard_nocf" attribute.

Relevant files:

- clang/include/clang/Basic/Attr.td
- clang/lib/Sema/SemaDeclAttr.cpp

Clang and clang-cl integration

CFG is integrated into Clang using the -cfguard CC1 option. Alternatively, the -cfguard-no-checks CC1 option can be used to emit the FID table without inserting checks.

It is also integrated into the clang-cl compatibility layer, using the standard /guard:cf, /guard:cf-, and /guard:cf,nochecks compiler flags.

Relevant files:

- clang/include/clang/Driver/CC1Options.td
- clang/lib/CodeGen/CodeGenModule.cpp
- clang/lib/Driver/ToolChains/Clang.cpp
- clang/lib/Driver/ToolChains/MSVC.cpp

Tests

The following tests are added for each supported architecture:

- Test that CFG checks are not added to functions with nocf_checks attribute
- Test that CFG checks are correctly added at -O0
- Test that CFG checks are correctly added in optimized code (common case)
- Test that CFG checks are correctly added on invoke instructions
- Test that Control Flow Guard preserves floating point arguments
- Test that Control Flow Guard checks are correctly added for tail calls
- Test that CFG checks are not added in modules with the cfguard-no-checks flag
- Test that longjmp targets are assigned labels and that these appear in .gljmp sections
- Test that Control Flow Guard checks are correctly added for x86_64 vector calls
- Test that Control Flow Guard checks are correctly added for x86 vector calls
- Test that the __declspec(guard(nocf)) modifier correctly adds the “guard_nocf” attribute
- Test that the __declspec(guard(nocf)) modifier can be places on either the function declaration or definition
- Test that the __declspec(guard(nocf)) modifier is correctly preserved when inlining
- Test that the __declspec(guard(nocf)) modifier correctly adds the “guard_nocf” attribute on override functions (C++ only)

Relevant files:

- llvm\test\CodeGen\X86\cfguard-checks.ll
- llvm\test\CodeGen\X86\cfguard-module-flag.ll
- llvm\test\CodeGen\X86\cfguard-x86-64-vectorcall.ll
- llvm\test\CodeGen\X86\cfguard-x86-vectorcall.ll
- llvm\test\CodeGen\AArch64\cfguard-checks.ll
- llvm\test\CodeGen\AArch64\cfguard-module-flag.ll
- llvm\test\CodeGen\ARM\cfguard-checks.ll
- llvm\test\CodeGen\ARM\cfguard-module-flag.ll
- llvm\test\CodeGen\WinCFGuard\cfguard.ll
- clang\test\CodeGen\guard_nocf.c
- clang\test\CodeGenCXX\guard_nocf.cpp