RISC-V Control-flow integrity (Zicfi)

RISC-V Shadow-stack and Landing-pads Task Group

Version 0.1, 11/2022: This document is in development. Assume everything can change. See http://riscv.org/spec-state for details.

Table of Contents

Preamble

This document is in the Development state

Assume everything can change. This draft specification will change before being accepted as standard, so implementations made to this draft specification will likely not conform to the future standard.

Copyright and license information

Contributors

This RISC-V specification has been contributed to directly or indirectly by (in alphabetical order):

Andrew Waterman, Nick Kossifidis, George Christou, Vedvyas Shanbhogue

Chapter 1. Introduction

Control-flow Integrity (CFI) provides CPU instruction set architecture (ISA) capabilities to defend against Return-Oriented Programming (ROP) and Call/Jump-Oriented Programming (COP/JOP) style control-flow subversion attacks. This attack methodology uses code sequences in authorized modules with at least one instruction in the sequence being a control transfer instruction that depends on attacker-controlled data either in the return stack or in a register/memory for the target address. Attackers stitch these sequences together by diverting the control flow instruction (e.g., RET, CALL, JMP) from its original target address to a new target via modification in the data stack or in the register or memory used by these instructions.

This specification describes CFI security objectives, threat model, and various architectural design choices to ensure that the design meets the security objectives.

RV32/RV64 provide two types of control transfer instructions - unconditional jumps and conditional branches. Conditional branches encode an offset in the immediate field of the instruction and are thus direct branches that are not susceptible to control flow subversion.

Unconditional direct jumps using JAL transfer control to a target that is in a +/- 1 MiB range from the current PC. Unconditional indirect jumps using the JALR obtain their branch target by adding the sign extended 12-bit immediate encoded in the instruction to the rs1 register.

The RV32I/RV64I does not have a dedicated instruction for calling a procedure or returning from a procedure. A JAL or JALR may be used to perform either a procedure call or a return from a procedure. The RISC-V ABI however defines the convention that a JAL/JALR where rd (i.e. the link register) is x1 or x5 is a procedure call, and a JAL/JALR where rs1 is the conventional link register (i.e. x1 or x5) is a return from procedure. The architecture allows for using these hints and conventions to support return address prediction and the hints are specified in Table 2.1 of the Unprivileged ISA specifications [1].

The RISC-V control-flow integrity (CFI) extension (Zicfi) builds on these conventions and hints.

The term call is used to refer to a JAL or JALR instruction (and their compressed forms) with x1 or x5 as the rd. A call using JAL is termed a direct call and using JALR is termed an indirect call.

The term return is used to refer to a JAL or JALR instructions (and their corresponding compressed forms) with x1 or x5 as the rs1.

The term indirect jump is used to refer to an unconditional jump using JALR/C.JALR instruction where the rd i.e. the link register is not x1 or x5 (i.e. not an indirect call) and where the rs1 is not x1 or x5 (i.e. not a return).

The landing-pads are designed to provide integrity to control transfers performed using indirect call and indirect jump and this is referred to as forward-edge protection.

The shadow-stack is designed to provide integrity to control transfers performed using return instruction (where the return may be from a procedure invoked using an indirect call or a direct call) and this is referred to as backward-edge protection.

To enforce backward edge control flow integrity, the extension introduces a shadow stack. The

shadow stack is used to spill the link register if required by non-leaf functions. A shadow-stack-pointer (`ssp`) register is introduced in the architecture to hold the address of the top of the current active shadow stack. The shadow stack is protected from inadvertent corruptions and modifications as detailed later. The extension provides instructions to store and load the link register to the shadow stack. A function in a program compiled to use shadow stacks stores the link register to the data stack and a shadow copy of the link register to the shadow stack when the function is entered (the prologue). When the function needs to return (the epilogue), the function loads the link register from the data stack and the shadow copy of the link register from the shadow stack. The link register value from the data stack and the shadow link register value from the shadow stack are compared. A mismatch of the two values is indicative of a subversion of the return address control variable and causes an illegal-instruction trap.

Operating in shadow stack mode, i.e., where the call stack layout is preserved and the shadow stack is used to store a shadow copy of the link register, allowing preserving the ABI.

A program may alternatively operate in control stack mode where the link register is only stored on the shadow stack. Such programs break the ABI but benefit from avoiding the additional instructions to store and load the link register to the data stack and to compare the two before returning from a function. Control stack mode may also allow the program to have a smaller data stack as the space to save the link register is no longer needed.

To enforce forward edge control flow integrity, the extension introduces new landing pad instructions (`lpcll`) that enable software to indicate valid targets for indirect calls and jumps in a program. Compiler is expected to emit a `lpcll` as the first instruction of address-taken functions. Compiler is expected to emit a `lpcll` at an indirect jump target.

When the landing pad feature is active, the hart tracks an expected landing pad (`ELP`) state that is updated with the expected landing pad instruction on indirect calls and jumps to . An indirect call or jump updates the `ELP` to require a `lpcll` instruction at the target. If the instruction at the target is not `lpcll` then an illegal instruction exception is raised.

The landing pads may be labeled. With labeling enabled, the number of landing pads that can be reached from an indirect call or indirect jump site can be constrained using programming language based policies. A landing pad label register (`lplr`) is set up prior to initiating an indirect call or indirect jump with the expected landing pad label using an instruction to set the `lplr`. If the label of the landing pad does not match that in `lplr` then an illegal instruction exception is raised.

Up to 25-bit labels are supported by this extension.

In the simplest form the program may be built with a single label value to implement a coarse-grain version of forward-edge CFI. Such a program would significantly reduce the gadget space by constraining gadgets to be preceded by a landing pad instruction i.e., to the start of indirect callable functions.

A second form of label generation may generate a signature (e.g., a MAC) using the prototype of the functions. Such programs would further constrain the gadgets reachable from a call site to indirect callable functions that have the expected

prototype of functions called by that call site.

A third form of label generation may generate labels by analyzing the control-flow-graph (CFG) of the program and lead to even further constraining of the gadgets reachable. Such programs may further use the multi-label capability such that when a function is called from two or more call sites, then such common functions may be labeled as reachable from each of the call sites. For example, consider two call sites A and B. A invokes functions X and Y. B invokes functions Y and Z. With a single label scheme the functions X, Y, and Z would need to be assigned the same label such that both call site A and B can invoke the common function Y. This allows call site A to additionally call function Z and call site B to additionally call function X. However, if function Y was labeled with two labels - one corresponding to call site A and other to call site B then Y can be invoked by both but the X can only be invoked by call site A and Z only by call site B. To support multiple labels, the compiler may create a call site specific entrypoint to such shared functions with each entrypoint containing the call site specific landing pad instruction followed by a direct branch to the start of the function.

A portion of the label space may be dedicated to label landing pads that are only valid targets of an indirect jump (and not an indirect call).

Forward-edge and backward-edge CFI may be enabled for a program that executes in U-mode, S-mode, or M-mode by itself to enable a mix of CFI enabled applications, operating systems, and machine mode firmware to co-exist.The processor keeps track of the CFI enables and CFI state for each mode in the `mcfistatus` CSR. A subset of the fields in the `mcfistatus` CSR are accessible using the `scfistatus` CSR. A VS-mode's version of `scfistatus` is provided to track the CFI state for VS-mode and VU-mode.

To use Zicfi, the operating system has to be modified to enable Zicfi capabilities, including the context switching of additional CFI extension state. The set of programs installed in such an OS may however be a mix where some programs are compiled with Zicfi capabilities and others are not. Allowing the U-mode CFI be individually enabled from S-mode allows an operating system to keep CFI enabled when operating in S-mode but enable or disable it for U-mode depending on the program being executed in U-mode.

To support backward compatibility of the programs built with Zicfi support, the new instructions to operate on the shadow stack, the landing pad instructions, and the instructions to set the lplr are encoded using Zimops encodings. When Zicfi is not enabled for a program or the program is executing on a processor that does not support the Zicfi extension then the instructions introduced by the Zicfi extensions execute as defined by Zimops extension.

An OS distribution compiled with Zicfi extension typically also includes the system libraries (e.g., glibc, etc.) that are also compiled with the Zicfi extension. Such system libraries however may need to link dynamically to programs that are not compiled with the Zicfi extension. When such programs are executing, the OS may disable the Zicfi extension in U-mode. When these system libraries are invoked in U-mode by such programs, the Zicfi instructions in the libraries revert to their NOP

behavior. Without such encoding, the OS distribution may need to carry two versions of such libraries, one with Zicfi instructions and one without, and thus need significantly larger cost and complexity for supporting the Zicfi extension.

An OS distribution compiled with Zicfi extension may be installed on a machine that does not support Zicfi extensions. On such machines, as the Zicfi instructions are encoded as Zimops, they revert to their NOP behavior.

A program compiled with the Zicfi extension may be installed on an OS that is not compiled for the Zicfi extension or on a machine that does not support the Zicfi extension. The Zicfi instructions are encoded as Zimops revert back to their NOP behavior.

Chapter 2. Shadow Stack and Landing Pad CSRs

This chapter specifies the CSR state of the Zicfi extension.

2.1. Machine CFI status (`mcfistatus`)

The `mcfistatus` CSR is a machine-mode WARL read-write 32-bit register used to keep track of the processor's CFI state at M-mode, HS/S-mode and U-mode.

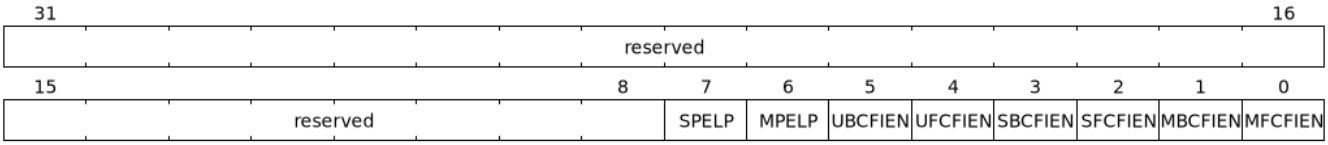

Figure 1. `mcfistatus` for RV32 and RV64

The `MFCFIEN`, `SFCFIEN`, and `UFCFIEN` are WARL fields that when set to 1 enable forward-edge CFI at M-mode, S-mode (if supported), and U-mode (if supported) respectively.

The `MBCFIEN`, `SBCFIEN`, and `UBCFIEN` are WARL fields that when set to 1 enable backward-edge CFI at M-mode, S-mode (if supported), and U-mode (if supported) respectively.

The `MPELP` and `SPELP` WARL fields are updated when a trap is taken into M-mode or S-mode respectively. When a trap is taken into privilege mode `x`, the `xPELP` fields are updated to indicate the type of landing pad that was expected at the privilege level at the time of taking the trap.

The `xPELP` fields are encoded as follows:

- 0 - `NO_LP_EXPECTED` - no landing pad instruction expected
- 1 - `LP_EXPECTED` - landing pad instruction expected

2.2. Supervisor and Virtual-supervisor CFI status (`scfistatus`/`vscfistatus`)

The `scfistatus` CSR is a supervisor-mode read-write 32-bit register which is a subset of the `mcfistatus` CSR and is used to keep track of the processor's CFI state at HS/S-mode and U-mode. Reading any implemented field, or writing any writable field, of `scfistatus` effects a read or write of the homonymous field of `mcfistatus`.

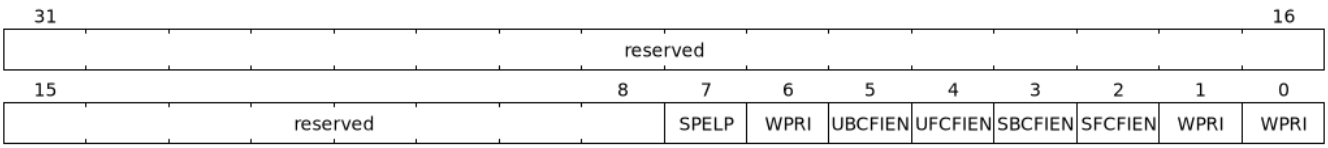

Figure 2. `scfistatus` and `vscfistatus` for RV32 and RV64

The `SFCFIEN`, and `UFCFIEN` are WARL fields that when set to 1 enable forward-edge CFI at S-mode, and U-mode respectively.

The `vscfistatus` CSR is a virtual-supervisor mode read-write 32-bit register which is VS-mode version of the scfistatus CSR and is used to keep track of the processor's CFI state at VS-mode. When V=1, `vscfistatus` substitutes for the usual `scfistatus`, so instructions that normally read or modify `scfistatus` actually access `vscfistatus` instead.

2.3. Landing pad label register (`lplr`)

The `lplr` CSR is a user-mode read-write (URW) 32-bit register that holds the label expected at the target of an indirect call or an indirect jump. The label is split into a 8-bit upper label (`UL`), 8-bit middle label (`ML`), and a 9-bit lower label (`LL`).

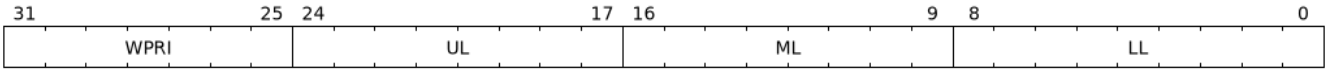

Figure 3. `lplr` for RV32 and RV64

2.4. Shadow stack pointer (`ssp`)

The `ssp` CSR is an unprivileged read-only (URW) CSR that reads and writes `XLEN` low order bits of the shadow stack pointer (`ssp`). There is no high CSR defined as the `ssp` is always as wide as the `XLEN` of the current privilege level.

Chapter 3. Backward-edge control-flow integrity

A shadow stack is a second stack used to push the link register if it needs to be spilled to make a new procedure call. Leaf functions do not need to spill the link register. A shadow stack is active for a privilege level `x` if `xBCFIEN` is 1. The shadow stack, similar to the regular stack, grows downwards, i.e. from higher addresses to lower addresses. Each entry on the shadow stack is `XLEN` wide and holds the link register value. The `ssp` points to the top of the shadow stack, i.e. address of the last element pushed on the shadow stack.

Compilers when generating code for a CFI enabled program must protect the link register, e.g. `x1` and/or `x5`, from arbitrary modification by not emitting unsafe code sequences.

Chapter 4. Backward-edge CFI instruction encoding

The backward-edge CFI extension introduces the following instructions for shadow stack operations. All instructions are encoded using the SYSTEM major opcode and using the `mop.r` and `mop.rr` instructions defined by the Zimops extension.

(mop.r)	31 20	19 15	14 12	11 7	6 0
mnemonic	1.00..0111..	rs1	func3	rd	opcode
sspush	100000011100	src	100	0	1110011
ssppop	100000011100	0	100	dst	1110011
ssprr	100000011101	0	100	dst	1110011
	12	5	3	5	7

(mop.rr)	31 25	24 20	19 15	14 12	11 7	6 0
mnemonic	1.00..1	rs2	rs1	func3	rd	opcode
ssamoswap	1000001	src	addr	100	dst	1110011
sschkra	1000101	1	5	100	0	1110011
	7	5	5	3	5	7

If xBCFIEN is 0, the behavior of all backward-edge CFI instructions is as defined by the Zimops extension and the instructions just write 0 to [rd].

When programs using shadow stack instructions are installed on a machine that supports the CFI extensions but the operating system installed does not enable the CFI extensions, the program continues to function due to Zimops defined behavior of writing 0 to [rd] and not causing an illegal-instruction exception.

When programs using shadow stack instructions are installed on a machine that does not support the CFI extensions but support the Zimops extension, the program continues to function due to Zimops defined behavior of writing 0 to [rd] and not causing an illegal-instruction exception. On machines that do not support Zimops extension, the instructions cause an illegal-instruction exception. Installing programs that use the shadow stack instructions on such machines is not supported.

4.1. Push to and Pop from the shadow stack

A push operation is defined as decrement of the `ssp` by `XLEN` followed by a write of the link register at the new top of the shadow stack. A pop operation is defined as a `XLEN` wide read from the current top of the shadow stack followed by an increment of the `ssp`.

To push a link register on the shadow stack, the CFI extension provides `sspush` instruction. To pop a link register from the shadow stack, the CFI extension provides `sspop` instruction.

Programs operating in shadow stack mode store the return address to the data stack as well as the shadow stack in the function prologue. Such programs when returning from the function load the link register from the data stack and load a shadow copy of the link register from the shadow stack. The two values are then compared (using the sschkra instruction that is discussed later). If the values do not match it is indicative of a shadow stack violation and causes an illegal-instruction exception. The function prologue and epilog of a function using shadow stacks is as follows:

```
function_entry:
    sspush x1      # push link register x1 on shadow stack
     :
     :
    sspop x5       # pop from shadow stack into x5
    sschkra x1, x5 # compare link register x1 to shadow
                   # link register x5; faults if not same
    ret
```

When x1 is used by the ABI as the link register, the x5 may be used to load the shadow link register value from the shadow stack. Alternatively, if the ABI uses x5 as the link register, then the x1 register may be used as the shadow link register.

A leaf function i.e. a function that does not itself make function calls does not need to push the link register to the shadow stack or pop it from the shadow stack. The return value may be held in the link register itself for the duration of the function execution.

In the RVI psABI, the x1 register is designated as the link register and x5 register is designated as a temporary register. SInce the shadow link register is loaded at the tail of the function, prior to return, the x5 register being used as the shadow link register does not impose a burden on the compiler as the x5 register, being a temporary register that is not preserved across a call, is usually free for use at the tail of the function.

Programs operating in the control stack mode may store the return address only to the shadow stack in the function prologue. Such functions when returning from the function load the link register value from the shadow stack. Such programs may either use the x1 or x5 register, depending on their ABI, as the link register. The hardware return-address prediction stacks detect the use of x1/x5 as the rd and x1/x5 as the source for a JALR instruction to infer if the JALR is used for a call or a return semantic for the purposes of prediction. To support both options the CFI extension provides the x1 and x5 variants of the shadow stack load and store.

`sspop` performs a load identically to the existing `LOAD` instruction with the difference that the base is implicitly `ssp`, the width is implicitly `XLEN`. `sspush` performs a store identically to the existing `STORE` instruction with the difference that the base is implicitly `ssp`, the width is implicitly `XLEN`.

The `sspush` and `sspop` require the virtual address in `ssp` to have a shadow stack attribute (see

Shadow Stack Memory Protection).

If the virtual address in `ssp` is not `XLEN` aligned then the instructions cause a load or store/AMO address-misaligned exception.

Misaligned accesses to shadow stack are not required and enforcing alignment is more secure to detect errors in the program.

The operation of the `sspush` instructions is as follows:

Listing 1. `sspush` operation

```
If xBCFIEN != 0
    [ssp] = [ssp] - (XLEN/8)   # decrement ssp by XLEN/8
   *[ssp] = [src]              # Store src value to address in ssp
else
    [dst] = 0
endif`
```

The operation of the `sspop` instructions is as follows:

Listing 2. `sspop` operation

```
If xBCFIEN != 0
    dst   = *[ssp]             # Load dst from address in ssp
    [ssp] = [ssp] + (XLEN/8)   # increment ssp by XLEN/8
else
    [dst] = 0;
endif
```

Store to load forwarding is a common technique employed by high performance processor implementations. CFI implementations may restrict forwarding from a non-shadow-stack store to a `sspop` instruction. A non-shadow-stack store causes a fault if done to a page mapped as a shadow stack. However such determination may be delayed till the PTE has been examined and thus may be used to transiently forward the data from such stores to a `sspop`.

A common operation performed on stacks is to unwind them to support constructs like setjmp/longjmp, C++ exception handling, etc. A program that uses shadow stacks must unwind the shadow stack in addition to the stack used to store data. The unwind function must verify that it does not accidentally unwind past the bounds of the shadow stack. Shadow stacks are expected to be bounded on each end using guard pages i.e. pages that do not have a shadow stack attribute. To detect if the unwind occurs past the bounds of the shadow stack the unwind may be done in maximal increments of 4 KiB and testing for the ssp to be still pointing to a shadow stack page or has unwound into the guard page. The following examples illustrate use of backward-edge CFI instructions to unwind a shadow stack.

```
setjmp() {
    :
    :
    // read and save top of stack pointer to jmp_buf
    asm("ssprr %0" : "=r"(cur_ssp):);
    jmp_buf->saved_ssp = cur_ssp;
    :
    :
}
longjmp() {
    :
    // Read current shadow stack pointer and
    // compute number of call frames to unwind
    asm("ssprr %0" : "=r"(cur_ssp):);
    // Skip the unwind if backward-edge CFI not active
    asm("beqz %0, 1f" : "=r"(cur_ssp):);
    num_unwind = jmp_buf->saved_ssp - cur_ssp;
    // Unwind the frames in a loop
    while ( num_unwind > 0 ) {
        step = ( num_unwind >= 4096 ) ? 4096 : num_unwind;
        cur_ssp += step;
        num_unwind -= step;
        // write the ssp register with unwound value
        asm("csrw %0, $ssp_csr_num" : "=r"(cur_ssp):);
        // Test if unwound past the shadow stack bounds
        asm("sspush x5");
        asm("sspop  x5");
    }
1f:
    :
}
```

4.2. Read ssp into a register

The ssprr instruction is provided to move the contents of ssp to the destination register.

The operation of the ssprr instructions is as follows:

Listing 3. ssprr operation

```
If xBCFIEN != 0
    [dst] = [ssp]
else
    [dst] = 0;
endif
```

The property of Zimops writing 0 to the rd when the extension using Zimops is not present or not active may be used by such functions to skip over unwind actions

by dynamically detecting if the backward-edge CFI extension is active.

An example sequence such as the following may be used:

```
    ssprr t0                  # mv ssp to t0
    beqz bcfi_not_active      # zero is not a valid shadow stack
                              # pointer by convention
    # Shadow stacks active
    :
    :
bcfi_not_active:
```

4.3. Verifying return address

Programs operating with a shadow stack push the return address onto the data stack as well as the shadow stack in the function prologue. Such programs when returning from the function pop the link register from the data stack and pop a shadow copy of the link register from the shadow stack. The two values are then compared. If the values do not match it is indicative of a corruption of the return address variable and the program causes an illegal instruction exception.

When x1 is used by the ABI as the link register, the x5 may be used to hold the shadow link register value from the shadow stack. Alternatively, if the ABI uses x5 as the link register, then the x1 register may be used as the shadow link register.

A `sschkra` instruction is provided to perform the comparison. The `sschkra` instruction causes an illegal instruction exception if the both of following conditions are satisfied:

- x1 is not equal to x5
- `xBCFIEN` is 1 i.e., backward-edge CFI is enabled for the program

4.4. Atomic Swap from a shadow stack location

The CFI extension defines an `ssamoswap` instruction to atomically swap the `XLEN` bits of src register with `XLEN` bits on the shadow stack at address in `addr` and store the value from address in `src` into register `dst`.

The `ssamoswap` is always sequentially consistent and cannot be reordered with earlier or later memory operations from the same hart.

The `ssamoswap` requires the virtual address in `ssp` to have a shadow stack attribute (see Shadow Stack Memory Protection).

If the virtual address is not `XLEN` aligned then the instructions cause a store/AMO address-misaligned exception.

The operation of the `ssamoswap` instructions is as follows:

Listing 4. `ssamoswap` operation

```
If xBCFIEN != 0
    Perform the following atomically with sequential consistency
        [dst]  = *[addr]
       *[addr] = [src]
else
    [dst] = 0;
endif
```

Stack switching is a common operation in user programs as well as supervisor programs. When a stack switch is performed the stack pointer of the currently active stack is saved into a context data structure and the new stack is made active by loading a new stack pointer from a context data structure.

When shadow stacks are enabled for a program, the program needs to additionally switch the shadow stack pointer. The pointer to the top of the deactivated shadow stack if held in a context data structure may be susceptible to memory corruption vulnerabilities. To protect the pointer value the program may then store it at the top of the shadow stack itself and thus create a checkpoint.

An example sequence to store and restore the shadow stack pointer is as follows:

```
# The a0 register holds the pointer to top of new shadow
# to switch to. The current ssp is first pushed on the current
# shadow stack and the ssp is restored from new shadow stack
save_shadow_stack_pointer:
    ssprr  x5                   # read ssp and push value onto
    sspush x5                   # shadow stack. The [ssp] now
                                # holds ssp+8. Save away x5
                                # into a context structure to
                                # restore later.
restore_shadow_stack_pointer:
    ssamoswap t0, a0, x0        # t0=*[ssp] and *[ssp]=0
    addi      t0, t0, (XLEN/8)  # t0+XLEN/8 must match to a0
    bnez      t0, a0, crash     # if not crash program
    csrw      ssp, t0           # setup new ssp
```

Further the program may enforce an invariant that a shadow stack can be active only on one hart by using the ssamoswap when performing the restore from the checkpoint such that the checkpointed data is zeroed as part of the restore sequence and multiple hart attempt to restore the checkpointed data only one of them succeeds.

4.5. Shadow Stack Memory Protection

To protect shadow stack memory the memory is associated with a new attribute - Shadow Stack (SS)

in the page tables and in the PMP configuration registers. When a virtual-memory system is enabled, the SS attribute is obtained from the page tables alone. When a virtual-memory system is disabled or not active, the SS attribute is obtained from the PMP configuration registers. The rules enforced by the SS attribute are specified in this section.

4.5.1. When Virtual-Memory system is enabled

The shadow stack memory is protected using page table attributes such that it cannot be written by instructions other than sspush and amossswap. The shadow stack can be read using all instructions that load from memory.

The encoding R=0, W=1, and X=0, is defined as the shadow stack attribute when UBCFIEN or SBCFIEN is 1. When UBCFIEN and SBCFIEN are 0, this encoding continues to be reserved. The following faults occur:

1. If the accessed page has SS attribute set
    a. Stores other than `sspush`, and `ssamoswap` cause write/AMO page faults.
    b. Instructions fetch causes an instruction page fault
2. if the accessed page does not have an SS attribute.
    a. `ssamoswap` causes store/AMO page fault
    b. `sspush` cause store/AMO page fault
    c. `sspop` cause load page fault

The `U` and `SUM` bit enforcement is performed normally. Svpbmt extension and Svnapot extensions are supported for shadow stack mappings.

When a virtual-memory system is enabled, the SS attribute from a matching PMP entry is ignored.

Operating systems should protect against writeable non-shadow-stack alias virtual-addresses mappings being created to the shadow stack physical memory.

The G-stage address translation and protections are not affected by the shadow stack extension. When G-stage page tables are active, the `ssamoswap`, and `sspop` instructions require the G-stage page table mapping the accessed memory to have read permission and the `ssamoswap` and `sspush` instructions require write permission. The SS encoding in the G-stage PTE remains reserved.

A future extension may define shadow stack encoding the G-stage page table to support use cases such as a hypervisor enforcing shadow stack protections for virtual-supervisor.

All loads are allowed to read the shadow stack. The shadow stack only holds a copy of the link register as saved on the regular stack. The ability to read the shadow stack is useful for debug, performance profiling, and other use cases.

4.5.2. When Virtual-Memory system is not enabled

When operating in M-mode, the virtual-memory system is not enabled. When operating in S-mode or U-mode (including when V=1), if the `satp` has `MODE` field set to `Bare`, then the virtual-memory system is not enabled.

A `pmpsscfg` CSR is a M-mode MXLEN wide register that is used to associate a PMP entry with shadow stack attributes. The fields of `pmpsscfg` register are WARL and bit `i` corresponds to the PMP entry `i`. When the backward-edge CFI extension is not supported the register is read-only zero. When bit `i` is set in `pmpsscfg`, the R/W/X permission bits in the corresponding `pmp<i>cfg` must be set to `010b`.

A memory access by `sspush`, `sspop`, and `ssamoswap` instructions to PMP `i` succeeds only if bit `i` of `pmpsscfg` is 1 and the R=0, W=1, and X=0 in the corresponding `pmp<i>cfg`.

A store to memory by instructions other than `sspush` and `ssamoswap` does not succeed if bit `i` of `pmpsscfg` is 1 and the R=0, W=1, and X=0 in the corresponding `pmp<i>cfg`.

Execute access that matches PMP entry `i` does not succeed if bit `i` of `pmpsscfg` is 1 and the R=0, W=1, and X=0 in the corresponding `pmp<i>cfg`.

Chapter 5. Forward-edge control-flow integrity

The forward CFI extension introduces landing pad instructions that enable software to indicate valid targets for indirect calls and indirect jumps in a program.

A landing pad (`lpcul`) instruction is defined as the instruction that must be placed at the program locations that can be valid targets of indirect jumps or calls.

To enforce that the target of an indirect call or indirect jump must be a valid landing pad instruction, the hart maintains an expected landing pad (`ELP`) state to determine the landing pad instruction that is required at the target of a JALR instruction. The ELP state can be one of:

- 0 - NO_LP_EXPECTED
- 1 - LP_EXPECTED

A JALR with rd is x1/x5 is an indirect call; it updates ELP to `LP_EXPECTED`. A JALR with rd != x1/x5 and where rs1 is not x1/x5 is an indirect jump; it updates the `ELP` to `LP_EXPECTED`.

When `ELP` is set to `LP_EXPECTED` and the next instruction in the instruction stream is not `lpcll`, then an illegal instruction exception is raised. If the next instruction in the instruction stream is `lpcll` then the `ELP` updates to `NO_LP_EXPECTED`.

Tracking of `ELP` and requiring valid landing pad instructions at the target of indirect call and jump enables a processor implementation to significantly reduce or to prevent speculation to non-landing-pad instructions. Constraining speculation using this technique greatly reduces the gadget space and increases the difficulty of using techniques such as branch-target-injection, also known as Spectre variant 2, that use speculative execution to leak data through side channels.

The forward-edge CFI extension also supports labeling of landing pads. By itself a landing pad allows, for example, an indirect call to land on any `lpcll` in the program. This significantly reduces the number of valid targets for an indirect call. Labeling of the landing pads enables software to achieve greater precision in pairing up indirect call sites or indirect jump sites with valid targets. To support labeled landing pads, the indirect call/jump sites establish an expected landing pad label in the landing pad label register (`lplr`). If the target of the indirect call/jump is a valid landing pad instruction, the expected label established in the `lplr` is matched with the target's label. If a mismatch is detected then the label check instruction causes an illegal instruction exception.

Each landing pad may be labeled with a label that may be upto 25-bits wide. The `lplr` has three subfields - a 9-bit lower label (`LL`), a 8-bit middle label (`ML`), and an 8-bit upper label (`UL`).

The forward CFI extension provides a `lpsll` instruction to establish the expected `LL` in the `lplr`, a `lpsml` instruction to establish the `ML`, and a `lpsul` instruction to establish the `UL` in the `lplr`.

The `lpcll` instructions embed a 9-bit immediate field. The instruction compares this value to the `LL` and on a mismatch causes an illegal-instruction exception.

For label widths up to 17-bit a companion instruction `lpcml` is provided. The `lpcml` embeds a 8-bit immediate value that is compared to the `ML` and on a mismatch causes an illegal-instruction exception.

For label widths greater than 17-bit a second companion instruction `lpcul` is provided. The `lpcul` embeds a 8-bit immediate value that is compared to the `UL` and on a mismatch causes an illegal-instruction exception.

5.1. Forward-edge CFI Instruction encoding

The forward-edge CFI extension introduces the following instructions for landing pad operations. All instructions are encoded using the SYSTEM major opcode and using the `mop.rr` instructions defined by the Zimops extension.

(mop.rr)	31 25	24	23	22 15	14 12	11 7	6 0
mnemonic	1.00..1	t	t	tttssss	func3	rd	opcode
lpsll	1000001	0		LLPL	100	0	1110011
lpcll	1000001	1		LLPL	100	0	1110011
lpsml	1000011	0	0	MLPL	100	0	1110011
lpcml	1000011	0	1	MLPL	100	0	1110011
lpsul	1000101	1	0	ULPL	100	0	1110011
lpcul	1000101	1	1	ULPL	100	0	1110011
	7	1	1	8	3	5	7

5.2. Landing pad instruction

`lpcll` is the valid landing pad instructions at target of indirect jumps and indirect calls. When a forward CFI is enabled, the instructions cause an illegal instruction exception if they are not placed at a 4-byte aligned `pc`. The `lpcll` has the lower landing pad label embedded in the `LLPL` field. `lpcll` causes an illegal instruction exception if the `LLPL` field in the instruction does not match the `lplr.LL` field.

When the instructions cause an illegal-instruction exception, the `ELP` does not change.

The operation of the `lpcll` instruction is as follows:

Listing 5. `lpcll` operation

```
If xFCFIEN != 0
    // If PC not 4-byte aligned then illegal-instruction
    if pc[1:0] != 0
        Cause illegal-instruction exception
    // If lower landing pad label not matched -> illegal-instruction
    else if (inst.LLPL != lplr.LL)
        Cause illegal-instruction exception
    else
        ELP = NO_LP_EXPECTED
```

```
else
    [rd] = 0;
endif
```

5.3. Label matching instructions

The `lpcml` instruction matches the 8-bit wide middle label in its `MLPL` field with the `lplr.ML` field and causes an illegal instruction exception on a mismatch. The `lpcml` is not a valid target for an indirect call or jump.

The `lpcul` instruction matches the 8-bit wide upper label in its `ULPL` field with the `lplr.UL` field and causes an illegal instruction exception on a mismatch. The `lpcul` is not a valid target for an indirect call or jump.

The operation of the `lpcml` instruction is as follows:

Listing 6. `lpcml` operation

```
If xFCFIEN != 0
    if (lplr.ML != inst.MLPL)
        cause illegal-instruction exception
else
    [dst] = 0;
endif
```

The operation of the `lpcul` instruction is as follows:

Listing 7. `lpcul` operation

```
If xFCFIEN != 0
    if (lplr.UL != inst.ULPL)
        cause illegal-instruction exception
else
    [dst] = 0;
endif
```

5.4. Setting up landing pad label register

Before performing an indirect call or indirect jump to a labeled landing pad, the `lplr` is loaded with the expected landing pad label - a constant determined at compilation time.

A `lpsll` instruction is provided to set the value of the lower label (`LL`) field of the `lplr`.

The operation of this instruction is as follows:

Listing 8. `lpsll` operation

```
If xFCFIEN == 1
```

```
   lplr.LL = inst.LLPL
   lplr.ML = lplr.UL = 0
else
   [rd] = 0;
endif
```

The following instruction sequence may be emitted at indirect call sites by the compiler to set up the landing pad label register when labels that are upto 9-bit wide are used:

```
   :
   # x10 is expected to have address of function bar()
   lpsll $0x1de   # setup lplr.LL with value 0x1de
   jalr %ra, %x10
   :
```

The following instruction sequence may be emitted at indirect call sites by the compiler to set up the landing pads at entrypoint of function bar():

```
bar:
    lpcll $0x1de    # Verifies that LPLR.LL matches 0x1de
```

A `lpsml` instruction is provided to set the value of the middle label (`ML`) field of the `lplr`. This instruction is used when labels wider than 9-bit are used.

The operation of this instruction is as follows:

Listing 9. `lpsml` operation

```
If xFCFIEN == 1
   lplr.ML = inst.MLPL
else
   [rd] = 0;
endif
```

The following instruction sequence may be emitted at indirect call sites by the compiler to set up the landing pad label register when labels that are upto 17-bit wide are used:

```
   :
   # x10 is expected to have address of function bar()
   lpsll $0x1de   # setup lplr.LL with value 0x1de
   lpsml $0x17   # setup lplr.ML with value 0x17
   jalr %ra, %x10
   :
```

The following instruction sequence may be emitted at indirect call sites by the compiler to set up the landing pads at entrypoint of function bar():

```
bar:
    lpcll $0x1de    # Verifies that LPLR.LL matches 0x1de
    lpcml $0x17     # Verifies that LPLR.ML matches 0x17
     :              # continue if landing pad checks succeed
```

A `lpsul` instruction is provided to set the value of upper label (`UL`) field `lplr`. This instruction is used when labels wider than 17-bit are used.

The operation of this instruction is as follows:

Listing 10. `lpsul` operation

```
If xFCFIEN == 1
   lplr.ML = inst.MLPL
else
   [rd] = 0;
endif
```

The following instruction sequence may be emitted at indirect call sites by the compiler to set up the landing pad label register when labels that are upto 25-bit wide are used:

```
  :
  # x10 is expected to have address of function bar()
  lpsll $0x1de   # setup lplr.LL with value 0x1de
  lpsml $0x17    # setup lplr.ML with value 0x17
  lpsul $0x13    # setup lplr.UL with value 0x13
  jalr %ra, %x10
  :
```

The following instruction sequence may be emitted at indirect call sites by the compiler to set up the landing pads at entrypoint of function bar():

```
bar:
    lpcll $0x1de    # Verifies that LPLR.LL matches 0x1de
    lpcml  $0x17    # Verifies that LPLR.ML matches 0x17
    lpcul  $0x13    # Verifies that LPLR.ML matches 0x13
     :              # continue if landing pad checks succeed
     :
```

5.5. Preserving expected landing pad state on traps

A trap may need to be delivered to the same or higher privilege level on completion of JALR but before the instruction at the target of JALR was decoded. To avoid losing previous ELP state, MPELP and SPELP bits are provided in the mcfistatus CSR for M-mode and HS/S-mode respectively. The SPELP bits can be accessed through the scfistatus CSR. To avoid losing ELP state on traps to VS-mode, SPELP bits are provided in vcfistatus (VS-modes version of scfistatus) to hold the ELP. When a trap is taken into VS-mode, the SPELP bits of vcfistatus CSR are updated with ELP. When V=1, scfistatus aliases to vcfistatus CSR. The xPELP fields in mcfistatus and vcfistatus are WARL fields. The trap handler should preserve the lplr CSR.

When a trap is taken into privilege mode x, the xELP bits are updated with current ELP and ELP is set to NO_LP_EXPECTED.

MRET or SRET instruction is used to return from a trap in M-mode or S-mode respectively. When executing an xRET instruction, the ELP is set to xPELP and xPELP is set to NO_LP_EXPECTED. The trap handler should put back the preserved lplr value before invoking SRET or MRET.