The oddness of JALR

I believe the central question is as Krste asked:

`How is it more dangerous than any indirect jump?`

First, because we know it is different than other IA jumps and we fail to scrutinize it extensively.
The danger is that it is dismissed as an indirect jump, and "we already know it is potentially dangerous".
But because it is different, the controls, the best practices for a "normal" indirect jump cannot be assumed to apply.

For those of us that did not anticipate Meltdown, Spectre, Row Hammer [the list goes on], or independently discover and recognize the true risk of off-by-one dangers [and the list goes on]; it is hubris to believe we are Black Hat enough to know that a feature cannot be misused and misappropriated in nefarious ways.

RISCV is especially well designed.
It limits indirect jumps to
a) a single unprivileged instruction and
b) a limited and well defined set of privileged trap return instructions.
Only these two cases need to be scrutinized for risk.

Naively a student of the architecture might assume that low bit of code addresses are irrelevant to the eventual target.
If they read Vol II first, they would be so inclined:

> If an implementation allows IALIGN to be either 16 or 32 (by changing CSR misa, for example), then, whenever IALIGN=32, bit mepc[1] is masked on reads so that it appears to be 0. This masking occurs also for the implicit read by the MRET instruction. Though masked, mepc[1] remains writable when IALIGN=32.

All is well and good: return address can be tagged with the low bit(s) and the csr load will fix it.
A very nice design.

The same student moving on to jalr will note:

> The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the sign-extended 12-bit I-immediate to the register rs1, then setting the least-significant bit of the result to zero.

Student: Ah, but what of the next to least-significant? MRET cleared that via mepc load?

Well we get a trap, but the trap handler can emulate the jalr by loading the resultant address into mepc and MRET back. So ...., a IALIGN=32 only machine can justifiable implement a jarl that clears the low order bit and be compliant. No harm, no foul. And at least one implementation that I know of has done exactly that. What's not to like?

Well, mret [and company] are inconsistent with jalr by definition. One is IALIGN aware/sensitive, and the other is not. Do we care? Apparently not, we can justify the discrepancy in many ways: this

design has more flexibility for emulation of special cases, etc., etc. But the seed is sown, slight of hand. Other issues deflect from the real problem.

So does this explanatory text:

> Clearing the least-significant bit when calculating the JALR target address both simplifies the hardware slightly and allows the low bit of function pointers to be used to store auxiliary information. Although there is potentially a slight loss of error checking in this case,

Ouch. we have ignored the elephant in the room.

Wonderful that function pointers can contain 1 bit of auxiliary information, automatically ignored by jalr, "usually" and

> "in practice"

?
The concern raised is " a slight loss of error checking in this case", but lets minimize that:
> in practice jumps to an incorrect instruction address will usually quickly raise an exception.

This is outright dismissal of a potential security hole.

The informative text is all true, and reasonable justification were it not for "in practice" only addressing White Hat practitioners. Black Hats don't care if exceptions are raised while they infiltrate. They don't care that the opportune chance only occurs once in a billion. They care that there is a non-zero chance of it happening in a way that they can exploit.

The student is left with a false sense of security.
I can augment pointers with auxiliary information.
If the instruction clears the low bit, well then, I can alternately [this student thinks "also", and in "White Hat" practice rightly] augment the instruction itself with 1 bit of auxiliary info. "Neat!! I love this architecture".

So the germ of exposure, a neat trick to use sometime, is introduced into the students mind to be recollected later for exceptional situations. Tricks that the explanatory notes directed the reader to imaging. But not only in this readers mind but countless others that will use these "tricks of jalr" in unimagined ways.

The problem is these tricks are "neat" and individually they do not effect the target address. Jalr works identically with each of these tricks as it does without either. However, when both an odd offset trick and an odd pointer trick interact the end result is not the same. The branch result is 2 bytes beyond where it would be if neither or only one trick was in play.

There is a kind of cognitive dissonance present here, and it need not be present in everyone, but it could well be. If asked, one may agree that the sum of offset and pointer is used with the low bit cleared. But a substantial number are likely, if they know of the tricks, to believe these tricks are safe. They are, but ... they are not in combination.

I don't think I need say more. This should suffice. The danger has already been exposed in the authoritative text even if it has been obfuscated and dismissed by the explanatory text.

But note: this was not intentional misdirect; no malice intended; no purpose to confuse or cause dissonance; but rather, justification for an approach that was expected to be hardware friendly. And

I had my part in contributing to the obfuscation. It was in part due to my attention to this feature, examining its peculiarities [and suggesting alternate behaviour] that the phrase "slight loss of error checking in this case" was added. per, law of Unintended Consequences. Although this did not address the essence of my objections, I deferred to the experts. So, when I had opportunity during ratification to publicize my concerns, I did not. The justifications I contrived for myself were many and I regret that I didn't risk the ridicule/disregard. I did not protest when the opportunity was ripe.

However, whereas I believe I have sufficiently described the setup for the conflation of these two neat ideas to give unexpected results, I will put my Black Hat on and show mechanism that I could use to weaponize this feature.

BLACK HAT ON:

Note, I don't have to gain elevated privileged execution directly.
It is sufficient in Linux and many other contexts just to hijack a thread. If it has root authority all the better. If it is a system thread, better still.

As a black hat I don't have to make an exposure, but only to leverage one that is present but naturally so rarely occurring that it is not noticed. Row Hammer is one such example.

Off-by-one opportunities are certainly not uncommon, and code to check for such in data pointers is becoming "standard" and standardized. Much less so for code addresses. In part because this aspect of an IA is not particularly portable, it is private and particular to the IA.
For example, a table of function pointers would [usually] function perfectly in RISCV, even if the auxiliary bit were inadvertently 1 for any of the entries. For a complex generations for individual entries an off-by-plus-one would escape notice, but could potentially be weaponized by me.

So, opportunistic odd code pointers are inevitable, people make mistakes.
And not just in function tables, any non-trivial aupci/lui plus addi/jalrI based calculation could be misformulated and normally not detected.

But how to entice odd offsets in jalr?

There will inevitably opportunities. As one of 10s of thousands of black hats, we are looking for such and we will find them, or make them.

Consider this scenario:

In the near future a test-for-low-bit-set-and-branch instruction will be ratified for RISCV [or a pervasive adhoc feature].
As a low overhead returnable signal, the low bit of the return address will be used as "auxiliary data".

Community response:
A "neat trick" optimization, Yeah!!!
But only -Zilbsb [low bit set branch] machines can use this optimization, boo!!!
But LTO code replacement is available, [andi ra,ra,1; beq/bne] Yeah!!
But it is destructive, [the common code WE call wants to return], boo!!
But we can use this code sequence [ror ra,ra,1;beq/bne] Yeah!
But it requires -Zbror, boo!
Come on now you ingrates ... some murmurring .. OK, yeah
But it is not efficient, neither in call code space or speed; boohoo, boohoo, boohoo.

So I, as a "Black Hat", present an even neater LTO plug in.
Fully backwards compatible for all systems.

I reverse the polarity of the signal in callee:
Initially as an xor ra,ra,1 immediately before "affected" jalr x0,ra,1,,
BUT nefariously with a facility to target specific sentinel setting locations
AND even more nefariously to invert all jalr low bits in a range (defaulting to the load module).
Then on the caller side I change the test-for-low-bit-set-and-branch instruction to a C.bne/C.neq.
When the sentinel value is present [now a zero bit] the code branches to the special service routine.
[the one that returns via jalr x0,ra,2].

Super Neat , Yeah!!
Uses "only" RVI base instructions, Yeah!!!
Smaller caller code, Yeah!!!!
Equal or smaller callee code, Yeah!!!!!
Runs faster, Yeah!!!!!!

What's not to like!!!!!!!!

I've succeeded, by this LTO plugin, in weaponizing potentially all jalr calls, even those not actually participating in this feature, 'cause like who-cares, its benign, right?
This half of the exploit is now in widespread use, people will trip up, and We will be there.

The end. [like a bedtime Grimm's Fairy Tale, "sweet!", sweet dreams]