Document number: P0928R1
Date: 2020-01-10
Reply To: Devin Jeanpierre (jeanpierreda@google.com), Geoff Romer (gromer@google.com), Chandler Carruth (chandlerc@google.com)
Audience: SG12, EWG

Mitigating Spectre v1 Attacks in C++

Background

Definition

Nearly all modern CPUs implement some form of *speculative execution*[1]: an optimization where some instructions are executed as a "guess", before it's known that they should actually be executed. When the CPU executes an instruction whose effect on the instruction pointer isn't immediately known (e.g. a branch on the result of a still-pending operation), the CPU may use some heuristic to predict the resulting instruction pointer (*branch prediction*), and then continue execution on that basis while buffering all outputs until the true result is known (whereupon the buffered outputs can be committed or discarded depending on whether the prediction was correct).

Crucially, although an incorrect speculative execution cannot perform I/O or write to main memory, its behavior can still be observed indirectly via a variety of side channels.

Spectre variant 1 occurs when an attacker can leak secret data via such a side channel, after training the branch predictor to predict incorrectly, bypassing safety protections. In particular, when:

1. Branch prediction for this branch is controllable by an attacker-controlled input, so that they can reliably train for and trigger incorrect predictions.
2. Speculative execution of these triggered incorrectly speculated branches can be made to later read secret data by an attacker.
3. This secret data can then be leaked via a side-channel attack on the CPU's state, bypassing rollback.

For additional background, see:

- CppCon 2019: Zola Bridges, Devin Jeanpierre "Spectre/C++" https://youtu.be/ehNkhmEg0bw
- CppCon 2018: Chandler Carruth "Spectre: Secrets, Side-Channels, Sandboxes, and Security" https://youtu.be/_f7O3IfIR2k
- Reading privileged memory with a side-channel https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

[1] Since speculative execution is a general technique, it is also implemented in compilers and application software. In this paper, we will exclusively be referring to the speculative execution implemented in CPUs, but we expect any speculative execution implemented in software to respect the security guarantees we propose.

- Spectre is here to stay: An analysis of side-channels and speculative execution https://arxiv.org/abs/1902.05178
- Runnable demonstrations of Spectre/Meltdown at https://github.com/google/SafeSide

Some shorter non-runnable examples follow.

Examples

Bounds check bypass

Consider the following application pseudo-code (see this post, from which the example was adapted, for a more in-depth explanation):

```
struct array {
 unsigned long length;
 unsigned char data[];
};
struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
unsigned long untrusted_offset_from_caller = ...;
if (untrusted_offset_from_caller < arr1->length) {
 unsigned char value = arr1->data[untrusted_offset_from_caller];
 unsigned long index2 = ((value&1)*0x100)+0x200;
 unsigned char value2 = arr2->data[index2];
}
```

On many modern CPUs, the code guarded by the bounds checks may be speculatively executed even when the bounds check fails. This results in a load at an arbitrary attacker-controlled offset (potentially containing secret data), and a subsequent load that depends on the result of the first. The attacker can then determine the bottom bit of `value` by using timing to determine which memory location was brought into cache by the second load.

Short-string optimization

As a more subtle example, consider the short-string optimization:

```
const char *string::get_pointer() const {
  if (is_long())
    return get_long_pointer();
  else
    return get_short_pointer();
}
const char &string::operator[](size_t i) const {
    const char *pointer = get_pointer();
  return pointer[i];
}
const char *string::data() const {
    return get_pointer();
}
```

In this case, an attacker may be able to train the branch predictor with a succession of short strings, and then supply a very long string, causing `operator[]` to speculatively load from `get_short_pointer()[i]`, where `i` may be far larger than the size of the short-string buffer. Notice that bounds-checking `i` will not help here: the problem is that a valid index into the long-string buffer is being speculatively used to index into the short-string buffer.

Speculative type confusion[2] and virtual dispatch

Finally, consider a cryptographic hash algorithm API consisting of an abstract base class with separate implementations for public and private keys:

```
struct KeyBase {
  virtual sha256 secure_hash() const = 0;
private:
  std::vector<std::byte> key_data;
};
struct PublicKey : KeyBase {
  sha256 secure_hash() const override {
    return fast_data_dependent_hash(key_data);
  }
};
struct PrivateKey : KeyBase {
  sha256 secure_hash() const override {
    return slow_data_invariant_hash(key_data);
  }
};
void do_crypto(const KeyBase &key) {
  auto h = key.secure_hash();
  // ...
}
```

In this example, `slow_data_invariant_hash()` is an implementation of the hash algorithm that is specially hardened against disclosure of the input data via side channel attacks (see N4534 for additional discussion of data-invariant algorithms), whereas `fast_data_dependent_hash()` is an implementation of the same algorithm that lacks such hardening, and as a consequence can be much faster.

[2] See also the Microsoft guidance on speculative execution and its discussion on speculative type confusion.

The virtual call to `secure_hash()` will often be lowered to some form of direct or indirect branch, and modern CPUs can speculatively predict the outcome of an indirect branch in the same way as a direct branch. Consequently, an attacker can cause the CPU to speculatively execute `fast_data_dependent_hash()` on private key data. As we've already noted, `fast_data_dependent_hash()` is known to leak its input via side channels, and many of those side channels remain exploitable under speculative execution, which can enable the attacker to obtain the private key.

Scope and goals

It should be possible for programmers to write software that is "secure", in some sense, on all conforming implementations. At the very least it should be possible for programmers to eliminate Spectre variant 1 vulnerabilities once they are discovered. However the only way offered by the C++ standard to prevent a Spectre variant 1 vulnerability is to ensure that untrusted inputs never exist in the same address space as secret data. For many applications this is not possible. We aim to make Spectre defenses possible for *all* applications.

An implementation can constrain the generated or executed machine code to ensure that incorrectly-speculated executions cannot observe the result of a load. This comes at a substantial performance cost, so we expect implementations to provide this as a separate build mode. (For example, MSVC, GCC, and Clang have all implemented such a mode.) This also means that some applications cannot use this functionality to defend against Spectre.

We propose the ability to work selectively turn off global hardening for performance-sensitive functions. C++ programmers will have to apply manual hardening when needed inside these unprotected functions, so we also propose a fine-grained mitigation API at the level of individual branches and variables. Since the motivating use cases are performance-sensitive, but have little else in common, we are prioritizing correctness, performance, and flexibility over ease of use. We see this facility as comparable to atomics (only more so), in that it provides the most efficient and flexible, but most engineering-intensive and lowest-level, means of addressing the problem portably.

All of the APIs introduced in this document take the form of attributes, since they don't affect the observable behavior of the abstract machine.

Mitigation via an attribute

Coarse grained mitigation

Clang has implemented an attribute to enable or disable global hardening on a per-function basis, when enabling it everywhere would be too costly to performance. We propose standardizing this attribute. It would be used in the following ways:

```
[[speculative_load_hardening(true)]] void foo {
  // No branches in this function will lead to a Spectre variant 1 vulnerability,
  // because mitigations are applied.
}
```

```
[[speculative_load_hardening(false)]] void foo {
  // Even if mitigations were applied to the entire program, they will not be
  // applied to this function, which promises that it does not introduce a
  // spectre variant 1 vulnerability.
}
```

The intent is that this attribute will be used to oppose the defaults, whatever those are. For example, mitigations may be turned off for a particular function if global mitigations were applied, but profiling revealed that this function was performance-critical, and careful auditing revealed it had no spectre variant 1 vulnerabilities. Or conversely, if the risk were narrowed to only a few risky functions, global mitigations could be turned off and turned back on only for those functions.

Regardless of whether mitigations are turned off or on globally, these would override that on a per-function basis, as performance/security needs warrant.

(Note: if misspeculation occurs in a `[[speculative_load_hardening(false)]]` function, execution might proceed into a `[[speculative_load_hardening(true)]]` function, which will not be aware that misspeculation is occurring.)

Implementation

To demonstrate that this proposal can actually be implemented, we describe here two approaches, one based on fencing, and another based on *Speculative Load Hardening* (SLH). We believe this can be implemented on all platforms where Spectre variant 1 is being actively studied, and we are working with hardware vendors to ensure that is the case. However, alternative approaches may also be followed: this is for illustrative purposes.

The mitigations never attempt to stop a potential attacker (i.e. an end user) from mistraining the branch predictor. This is impossible. Instead, the mitigations target the loads, and prevent the user from causing the secret to be loaded during incorrect speculative execution. In the case of lfence, this is by waiting until the load will not be speculative anymore. In the case of SLH, this is by changing pointers and offsets so that they do not point to a user-controlled value, and so the user cannot direct what is loaded.

LFence

Modern CPU architectures provide an instruction which effectively disables speculative execution, waiting for certainty before proceeding to the next instruction. On x86, this instruction is lfence. For example, if global mitigations are applied, then a snippet of code like the left might be transformed to add an lfence, as on the right:

```
if (i < N) {

  return arr[i];
}
else {
  return '\0';
}
```

```
if (i < N) {
  __mm_lfence();
  return arr[i];
}
else {
  return '\0';
}
```

By blocking speculative execution entirely, this lfence prevents the `arr[i]` from ever being executed during misspeculation, and so the user cannot influence what values are loaded in speculative execution.

Speculative Load Hardening

SLH ensures that during misspeculated execution, an operation that would load user-controlled data will instead load a trusted compile-time constant. It does this by tracking each branch, recording whether or not that branch was mispredicted, and using that record to mask the data.

To enable SLH to work, platforms need to provide a sufficient set of instructions for tracking and masking that are not subject to prediction. For example, the following operations must have deterministic behavior that are not themselves mispredicted somehow:

- bitwise `|=`

- conditional assignment (`CMOVcc` in x86), roughly equivalent to `var = cond ? true_value : var`. We can transcribe this as `__cmov(cond, true_value, &var)` in example code.

These are used to update a global predicate state (`__predicate_state`), which is all-ones if a misprediction has occurred, or all-zeroes if one has not. It is updated at each conditional using a cmov instruction. Every variable that contributes to the computation of an address is then `|`-masked against this. The effect is that an address can never be caused to point to a user-controlled secret via misprediction, because in the event of misprediction, no variables will be user-controlled at all.

For example, the snippet on the left might become, in example pseudocode, the snippet on the right, if SLH were applied.

```
if (i < N) {



  return arr[i];
}
else {
  return '\0';
}
```

```
const bool cond = i < N;
if (cond) {
  __cmov(!cond, all_ones_mask, &__predicate_state);
  arr |= __predicate_state;
  i |= __predicate_state;
  return arr[i];
}
else {
  return '\0';
}
```

Once SLH is applied, nothing anyone can do will affect the location of `arr[i]` during misspeculation: it is guaranteed to be all-ones.

For a more detailed explanation of SLH, see this talk given by Zola Bridges and Devin Jeanpierre.

Pros/Cons

SLH, due to its general nature, has downsides:

1. SLH requires knowledge of whether *any* condition was mispredicted (stored in `__predicate_state`) to ensure that misspeculation cannot possibly result in the use of attacker-controlled data. For example, this means that if the `__predicate_state` is not preserved for some reason, and in particular in the transition from unhardened to hardened functions, a fence must be applied so that we are guaranteed to be in a clean state.
2. The serial uses and updates of `__predicate_state` force a data dependency of every conditional on every previous conditional, preventing them from being speculatively

executed in parallel, or out of order, even when the security properties of one conditional do not depend on correct prediction of previous conditionals.

This leaves open an opportunity for a limited, fine-grained API for the cases where we know exactly which condition and which loads to harden, even in the face of otherwise unknown or misspeculated state.

Local mitigation for unprotected functions

We propose to implement mitigation in unprotected functions as a pair of parameterized attributes, which we'll introduce separately.

The first is `[[harden_misspeculation(arg...)]]`, which takes a list of variables as arguments, and can be applied to any branching control flow construct. It specifies that if that branch is mispredicted, the implementation must ensure that any subsequent accesses to the listed variables return values that are unrelated to the true values. (For example, via fencing or an SLH-like technique.)

```
if (untrusted_offset < arr.size())
[[harden_misspeculation(untrusted_offset)]]
{
  auto value = arr[untrusted_offset];
  ...
}

...
switch (untrusted_value) {
  [[harden_misspeculation(another_variable)]] case FOO:
    Foo(another_variable);
    ...
}
// etc.
```

This attribute can be placed on a label, or any block statement[3], as long as that statement is the first statement executed for `if`, `else`, `case:`, `for`, `while`, `do`, or `default:` (there will in some cases be multiple equivalent ways to apply the attribute). The actual behavior will apply to the immediately preceding branch. It can also be applied to a function definition, in which case the

[3] The location of these attributes may be surprising. Unfortunately, there is no other place that allows separate attributes for the `if` and the `else` blocks. And because C++ attributes cannot currently be applied at the expression level, conditional expressions like `a?b:c` do not receive any direct support, but can in general be converted to a lambda with an `if` statement.

hardened branch is the indirect function call (if any) that reached that function (e.g. virtual dispatch).

The second attribute is for the inverse case: `[[harden_misspeculation_else(arg...)]]`. This can be applied to the same statements, but instead hardens in the case where the branch is *not* taken. For example, if applied to a for loop, it hardens when the loop ends due to the loop conditional failing.

```
for (; i >= arr.size(); --i) [[harden_misspeculation_else(i)]] {}
// hardening applies when the loop ends due to the condition of the loop failing
auto value = arr[i];
```

These attributes both have the same effect as the global hardening, except that the *only* loads affected are (all) loads of the listed variables, and the *only* misprediction that is protected against is that of the branch that is associated with the attribute.

If a load of one of the affected variables is executed, and a corresponding branch was mispredicted, the load evaluates to an independent value that is not related to or affected by the true value of the variable. For example, it could always evaluate to zero. Pointers and references subsequently made to hardened variables are also hardened in the same way.

Note that since speculative execution doesn't affect observable program behavior, the implementation is given complete freedom to decide how to secure uses of the variable. We only require that it is not possible to obtain the unhardened value after the annotated condition was mispredicted in a speculative execution. For example, in the absence of backwards jumps, it may be sufficient to only change the evaluation of references to the variable that occur lexically after the condition. In the presence of backwards jumps, it may be necessary to harden all uses of the variable throughout its scope.

Note also that the arguments should not themselves be secret data: once the secret is loaded, it is immediately available to potential side channels. Later zeroing it out or erasing it does not remove the risk in the intervening period. It is safer to instead harden the inputs to misspeculated execution, so that subsequent behavior doesn't even perform the access of secret data.

Exceptions

We do not propose an attribute to mitigate Spectre variant 1 at the level of specific `try`/`catch` statements.

Exceptional control flow is implicit, and where the conditionals occur is left implementation defined. For example, it may be that an exception bubbles up to the call stack, and a `try`/`catch` is equivalent to a local `if`. But the more popular form is for exception handlers to be added as a list of blocks that may be executed as part of a separate exception runtime, which is not local to the exception handling function (e.g. here). It is impractical to change how spectre hardening works in a remote call. Instead, if fine grained hardening is desired, exceptions should be avoided.

Details

What follows are more detailed description of the behavior for different branching constructs, and a sketch of how they could be implemented.

`if` statements

```
int v1, v2;
...;
if (cond) [[harden_misspeculation(v1)]] {
  // Even under speculative execution, either
  // cond is true, or v1 evaluates to some hardened value.
  Foo(v1);
} else [[harden_misspeculation(v2)]] {
  // Even under speculative execution, either
  // cond is false, or v2 evaluates to some hardened value.
  Foo(v2);
}
```

This is very similar to the example given for SLH above. However, instead of using `__predicate_state`, it creates a new predicate state for this conditional in particular:

```
int v1, v2;
...
intptr_t __cond_predicate_state = 0;
if (cond) {
  __cmov(!cond, all_ones_mask, &__cond_predicate_state);
  v1 |= __cond_predicate_state;
  Foo(v1);
} else {
  __cmov(cond, all_ones_mask, &__cond_predicate_state);
  v2 |= __cond_predicate_state;
  Foo(v2);
}
```

This means that there is no data dependency between this and any other branching construct, restricting performance impact from this hardening to just this branch, while isolating it from the performance impact of hardening of other branches. (Note: this also means that if this condition is mispredicted, SLH would not be aware of it, as it doesn't update the predicate state.)

Alternatively, and again like SLH in general, an implementation may wish to instead lower to a fencing instruction, which stops speculative execution and waits until real execution catches up. This is probably less efficient, but works even if the implementation has not kept track of the global predicate state.

```
int v1, v2;
...
if (cond) {
  _mm_lfence();
  Foo(v1);
} else {
  _mm_lfence();
  Foo(v2);
}
```

In either case, neither `v1` nor `v2` evaluate to their "true" value under a mispredicted branch, and so they cannot be used to reliably compute a value to leak.

`switch` statements

```
int v1, v2;
...
switch (value) {
  [[harden_misspeculation(v1)]] case 1:
    Foo(v1);
    [[fallthrough]];
  [[harden_misspeculation(v2)]] default:
    Foo(v1, v2);
}
```

There is an ambiguity here. If the CPU mispredicts and speculatively executes `default:` , then `v2` should be hardened. But if the CPU mispredicts and speculatively executes `case 1`, and then falls through to `default:`, what happens to `v2`?

We suggest that hardening should occur after the label is hit, even if the misprediction was for reaching a different label.

If `value != 1`, but `case 1` is speculatively executed:

- `Foo(v1)` will receive an unrelated value for `v1`.

- `Foo(v1, v2)` will receive unrelated values for **both** `v1` and `v2`.

If `value == 1`, but `default:` is speculatively executed, instead of `case 1`:

- `Foo(v1, v2)` will receive an unrelated value for `v2`.

(As a motivating example, one might consider a switch with multiple cases that execute the same code. `switch(n) { case 1: case 2: case 3: ...}`. Should each case receive an identical `[[harden_misspeculation(...)]]`?)

So this could be lowered to something like the following:

```
intptr_t __switch_predicate_state = 0;
switch (value) {
  case 1:
    __cmov(!(value == 1), all_ones_mask, &__switch_predicate_state);
    goto actual_case_1;
  default:
    __cmov(value == 1, all_ones_mask, __switch_predicate_state);
    goto actual_default;
}

actual_case_1:
  v1 |= __switch_predicate_state;
  Foo(v1);
actual_default:
  v2 |= __switch_predicate_state;
  Foo(v1, v2);
}
```

Exhaustive `switch`

If a variable needs to be hardened when none of the cases are hit, this may require adding a default case:

```
switch (value) {
  case ONE: ...; break;
  case TWO: ...; break;
}

// If we need to harden a variable for the case that value != 1 and value != 2,
// there is nowhere to put it, so we need to add
// [[harden_misspeculation(variable)]] default: break;
```

This presents a problem: implementations can be configured so that switches on an `enum` will be checked for exhaustiveness. For each enumerator, there must be a matching `case` or `default`. If the program deliberately omitted the default so that the implementation would check that the cases and the enum variants were kept in sync, that workflow is now broken. We do not propose adding any mechanism to preserve the behavior of the diagnostic when a `default` case is added for Spectre hardening.

(It would help if implementations or C++ itself provided an attribute to specify whether `default` is considered sufficient to satisfy exhaustiveness checks. This would also simplify existing code, which sometimes contorts itself to avoid writing `default`.)

`for` / `while` statements

```
for (int i = 0; i < N; ++i)
  [[harden_misspeculation(v1),
    harden_misspeculation_else(v2)]] {
  // Even under speculative execution, either
  // i < N, or v1 evaluates to some hardened value.
}
// Even under speculative execution, either
// i >= N, or v2 evaluates to some hardened value.
```

```
for (auto a : b)
  [[harden_misspeculation(v1),
    harden_misspeculation_else(v2)]] {
  // Even under speculative execution, either
  // the iterator is not yet out of range, or v1 evaluates to some hardened value.
}
// Even under speculative execution, either
// the iterator is equal to b.end(), or v2 evaluates to some hardened value.
```

```
while (cond)
  [[harden_misspeculation(v1),
    harden_misspeculation_else(v2)]] {
  // Even under speculative execution, either
  // cond is true, or v1 evaluates to some hardened value.
}
// Even under speculative execution, either
// cond is false, or v2 evaluates to some hardened value.
```

Every loop has a condition which decides if the loop should run again (the branch target specified by `harden_misspeculation`), or if execution should continue after the loop (the branch target specified by `harden_misspeculation_else`). For instance, the last example is equivalent to the following, and can be implemented as such:

```
for (;;) {
  if (cond) [[harden_misspeculation(v1)]] {
    /* loop body */
  } else [[harden_misspeculation(v2)]] { break; }
}
```

The "else" condition applies when the loop exits normally, due to a failure of the condition, rather than due to an explicit `break` or `goto` inside of the loop body.

Note that all uses of `v2` that can come after the condition are considered suspect, and so `v2` remains hardened after the loop terminates.

`for` and `while` are the only branching constructs without an otherwise-available `else` case to annotate, and so the only constructs that require the use of `[[harden_misspeculation_else(...)]]`. (This attribute would be unnecessary if `for` and `while` loops had an explicit `else` syntax, as in Python.)

Function dispatch

```
[[harden_misspeculation(i)]]
void Function(int i) {
  // Even under speculative execution, one of the following is true:
  //   this was called directly, or
  //   this was executed from a correctly predicted indirect call, or
  //   i evaluates to some hardened value.
}
```

When a function has this attribute, the named variables are hardened if the function is entered due to a mispredicted indirect function call. So, for example, virtual function calls will either reach this function correctly, or else those variables will evaluate to some fixed value. (This can be implemented by comparing equality of vtable pointers.)

Other

Library functions also contain branches. For example, `std::copy` may have a speculative buffer overflow. If the implementation does not guarantee that std::copy is hardened against spectre (via secure defaults or a separate mode), and the user controls the bounds of the copy, then use of `std::copy` may be part of a Spectre vulnerability. In order to avoid this, one must write a new function instead, which does enable hardening and is not vulnerable to Spectre variant 1.

The implementation may choose to add branches or loads that were not present in the original program. For example, `int x[2] = {1, 2}; y = x[c]` may be implemented with a speculatable conditional, like `if (c) y = 2; else y = 1;`. This is permitted only so long as it does not change the behavior of subsequent code even in speculative execution. So, in a hardened function, the implementation must also harden this new control flow in the same way.

Things are more difficult in an unhardened function. Imagine the following subsequent snippet:

```
if (y == 2) [[harden_misspeculation(z) { … }
```

If a new conditional branch was introduced by the implementation as part of the computation of `y`, the programmer has no way of annotating that branch, as it does not exist in the source code as-written. Therefore, either the implementation is not allowed to introduce such a branch, or else it must include that branch as a potential source of mispredictions when deciding what to do with `z` in the body of this conditional.

Semantics

Fundamentally, C++ lacks a notion of speculative execution, so currently there is no way to talk about the behavior of code under incorrect speculation. We would like to constrain the behavior of the program under conditions that the standard unambiguously says cannot happen in the first place. The way we address this problem will determine what properties authors can rely on, and what optimizations implementations can apply.

The simplest example may be the easiest to address. Can an implementation consider these two snippets to be identical, and can it execute one instead of the other freely under "as-if"?

```
if (x < y) {
  if (x < y)
  [[harden_misspeculation(...)]] {
    …
  }
}
```

```
if (x < y) {
  if (true)
  [[harden_misspeculation(...)]] {
    …
  }
}
```

If so, can the compiler then remove the nested `if` entirely? We would like the answer to be no to all of these questions, but the only difference is in speculative execution. Speculative execution currently does not exist in C++, and is not considered "observable".

If C++ gains a model for speculative execution, it must deal with undefined behavior which is executed only speculatively: since speculative execution can bypass any conditional, it is virtually guaranteed to at some point do something C++ would consider UB. For example, it may be possible to train a conditional so that, with the right inputs, a `[[noreturn]]` function will return, even though it was impossible under normal execution.

In some implementations, the concrete behavior when returning from a `[[noreturn]]` function is to continue executing at some arbitrary other location inside the program. This could include a location past a hardened bounds check / array access -- there is no reason that it could not, via misprediction, jump after the hardening but before the array access, leaving some attacker-controlled value in a register or on the stack despite the hardening. The implementation can in principle create a brand new vulnerability out of any conditionally guarded UB, even if under normal execution it is impossible.

```
if (i < arr.size()) [[harden_misspeculation(i)]] {
  // In theory, the implementation may be able to generate a jump to here
  // from anywhere.
  leak(arr[i /* jump lands after the evaluation of i */]);
}
```

It's unlikely that any mitigation at the point of the array access would be sufficient, because the jump target can always be after that mitigation. Instead, the jump itself must be prevented. It seems as if we need to define or constrain the effects of undefined behavior when it is the result of incorrect speculative execution.

We would like it to be possible at least in theory to write a program that is secure in all implementations, but currently that is not so: even with the hardening semantics we've defined here, it appears that an implementation is free (in theory) to turn any undefined behavior into a Spectre vulnerability, even if that undefined behavior is only reachable speculatively. We might consider biting the bullet, and allowing the implementation to introduce vulnerabilities. Our goal then would be to make it possible to fix any vulnerability that arises by adding enough annotations and defensive constructs, even if those vulnerabilities were introduced by the implementation to begin with, and encourage implementations to try not to do things that would lead to vulnerabilities. This at least provides an option when faced with a live vulnerability, and is an improvement over the status quo. However, these vulnerabilities are difficult to detect, and it seems unreasonable to expect programmers to examine the generated machine code for weaknesses. Even if we do adopt this stance, it seems likely that any manual removal of an implementation-introduced vulnerability could instead be applied automatically.

In order to fully specify this feature, it may be necessary to add another layer of abstraction. When we speak about speculative execution, we are speaking about what actually executes on the physical machine, separately from our abstract model of C++, and it makes sense to us this way. It may be that we need to import this understanding directly into the standard via a second execution model beneath that of the abstract machine, that of the physical machine. Speculative execution can be defined at this lower level, and operations like return-from-noreturn or null pointer dereference can be treated as defined or unspecified behavior at this lower level. (Some forms of undefined behavior that we can't effectively model, like stack smashing, would remain undefined even here.) We could then place constraints on what physical machine operations are executed even within the bounds of undefined behavior in the abstract machine, while retaining the meaning of undefined behavior within the world of the abstract machine. This also hints at an answer to other abstraction-breaking security problems: for example, securely erased strings do not have a meaning in the abstract machine model, but have a clear meaning when we speak about the state of the actual machine that it runs on top of.

We would appreciate feedback on avenues of attack for the semantics, both at a high level and in detail when it comes to changing the execution model inside the standard. This feedback will be incorporated into a firmer proposal in a later revision.

Alternatives

Alternative solution: fine-grained SLH

Instead of adding a construct which ties specific branches to specific reads, one could imagine simply turning SLH off and on for particular variables and particular conditionals, retaining the global predicate state.

The benefit of retaining this predicate state is that conditionals can build on each other, and one may otherwise need to consider what is safe if a conditional arbitrarily far back in time was mispredicted. For example, in the case of the short-string optimization, guarding against a mispredicted bounds check does not prevent speculative buffer overflow with a mistrained short-string conditional. With a fine-grained API that lets one harden conditions totally independently, if the hardening only took place when the index was out of bounds, there could still be a Spectre vulnerability. However, with SLH and with fine-grained SLH, the entire predicate state is taken into account, automatically including the short-string optimization condition.

This is also its cost: if the fine grained API doesn't allow removing the data dependency on the predicate state, the program can't opt out of much of the cost of SLH. Performance critical code may wish to avoid this.

Alternative solution: protected control flow function

We have also considered a library API which protects branches, rather than an attribute:

```
// Requires: All `Ts` are integral or pointer types.
// Returns: `predicate`.
// Remarks: If `predicate` is false, then any speculative execution in
// which it is treated as true will also treat `zero_args...` as zero.
```

```
template <typename... Ts>
bool protect_from_speculation(bool predicate, Ts&... zero_args);
```

So for example, a speculative buffer overflow vulnerability can be mitigated like so:

```
if (protect_from_speculation(i < arr.size(), i)) {
  // either i < arr.size(), or i is some hardened value.
}
```

This is similar to `if (i < arr.size()) [[harden_misspeculation(i)]] { }` in the proposal above.

`protect_from_speculation` can be used to protect bounds-check and short string optimization code, but still offers no help with type confusion. It is also much more difficult than an attribute to apply to branching constructs that aren't `if`/`else` (e.g. `switch`, `for`, etc.). Finally, while it uses function call syntax, it cannot have normal C++ semantics: constant propagation would render it useless. For example, consider the following:

```
const int i = …;
if (i < arr.size()) {
  …
  if (protect_from_speculation(i < arr.size(), i)) { … }
}
```

An implementation is allowed, under as-if, to treat it the same as the following:

```
const int i = …;
if (i < arr.size()) {
  …
  if (protect_from_speculation(true, i)) { … }
}
```

It is impossible to detect, using the argument `true`, whether or not the true branch is the result of branch misprediction. The semantics necessary to make this transformation invalid are a poor fit for a function.

Alternative solution: protected load function

We have also considered mitigation via a library API like the following:

```
// If lower <= loc < upper, returns the value of `*loc`. Otherwise,
// returns `fail_value`.
// Notes: Does not access `*loc` if `loc` is out of bounds, even under
// speculative execution.
template <typename T>
T speculation_safe_read(T* loc, T* lower, T* upper, int fail_value);
```

This ties the mitigation API to the load, rather than to the branch that guards it, which may simplify the lowering on some architectures, and it is at least superficially easier to explain, since the function has nontrivial normative semantics even if you disregard speculation.

We think this API will in practice be hard to use, because it gives the programmer no access to the outcome of the bounds check, and therefore in most cases they will need to perform the bounds check themselves, before or after the load. Consequently, programmers and compilers will be continually tempted to optimize the code by eliminating the actual or apparent duplication, in ways that may risk undoing the safety benefits of the function call.

We could solve that problem by making `lower <= loc < upper` be a precondition of the function (which also permits more efficient lowering in some cases), rather than a condition that controls its behavior, but then this function is no longer any easier to explain than the alternatives.

Moreover, we do not see any feasible way to extend this solution to cover non-bounds check versions of Spectre variant 1, such as the short-string optimization or type confusion. It is possible that we could identify a minimal set of predicates that cover all realistic cases, but we believe efforts are better spent on a more general solution.

Alternative solution: `__builtin_speculation_safe_value`

GCC and LLVM have implemented an intrinsic, `__builtin_speculation_safe_value(v)`, which returns `0` if the current execution is the result of a previous misprediction as tracked by SLH, and `v` otherwise.

This is a natural evolution of SLH: it reuses the same tracking work that SLH does, but allows the programmer to specify when the `cmov` (or equivalent) should take place, and when it should not. It lends itself to an atomic-style `speculation_safe<T>` wrapper.

However, `__builtin_speculation_safe_value` does not provide us with the ability to be selective about which conditionals are tracked, only which loads are hardened. We believe our proposal to be more efficient as a consequence of this, and therefore more broadly applicable.

Alternative solution: secure secret storage

A common intuition is to protect the secret itself. Done correctly and completely, this is the safest option, and requires no changes to C++. For example, it may be sufficient to move all secrets processing to another process. This paper is predicated on the infeasibility of doing this in the general case, as it may be too slow, or may restrict the way we process data.

Note: any data that is accessible to the CPU during speculative execution, even if it is stored separately from insecure data, can in theory be leaked.

Alternative solution: CPU changes

In principle, hardware could be changed so that speculative loads don't execute at all, but to date CPU vendors have not pursued this approach. Alternatively, the CPU could help enforce "secure strings" by blocking any speculative load for specially protected regions of memory, or blocking speculation based on that load. (For example, SAPM.)

Another option is to allow speculative loads but prevent CPU side channels. Unfortunately, the consensus of security researchers is that it is impossible to implement general-purpose computation in a way that excludes all side channels, and surprising side channels have already been discovered for Spectre (e.g. Zombieload)

Other options also exist; this is not meant to be a complete list of CPU changes that could mitigate Spectre.

The problem remains that, no matter what CPU advances occur in the future, software must be made secure on the hardware that we currently use.

Appendix

Revision History

P0928R0: initial proposal
P0928R1: R0's proposal (`protect_from_speculation`) was moved to "Alternative solution: protected control flow function", with a new primary proposal based on attributes.