Document number: P0928R0
Date: 2018-02-09
Reply To: Geoff Romer (gromer@google.com), Chandler Carruth (chandlerc@google.com)
Audience: WG21 Evolution Working Group, Library Evolution Working Group

Mitigating Speculation Attacks in C++

Software systems routinely process untrusted input by checking various conditions on the input (such as that an input integer is within a specified bound), and then using the input to perform operations that are safe only if those conditions are satisfied (such as using the input integer as an array index). "Spectre variant 1" is a newly-discovered class of security vulnerability which permits an attacker to read arbitrary data by bypassing those checks using a combination of CPU speculation and a cache-timing side channel.

There are known techniques for blocking this attack in assembly code, but they cannot currently be expressed in standard C++. This attack relies entirely on hardware phenomena (namely CPU speculation and cache timing) that are not observable in the C++ abstract machine, and so any mitigation is susceptible to being optimized away under the as-if rule.

We must ensure that C++ programmers have the ability to block this attack. In this paper we informally describe a candidate API for doing so (along with some alternatives considered), but we do not have a solution to the problem of how to formally specify such an API in the standard. We intend this paper to encourage discussion on that topic.

Background: Spectre Variant 1 Attacks

As a basis for discussion, 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 most 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.

This paper is concerned with giving application programmers the ability to mitigate a vulnerability like this once they've identified it. We need to provide a facility that is as broadly applicable as possible (so that few or no use cases are left completely without mitigation options), and consequently we are prioritizing correctness, performance, and flexibility over ease of use. We hope and expect that most applications either will not have secrets sensitive enough to be worth attacking in this way, or will be able to employ other higher-level mitigations (e.g. offloading secrets to a separate process, compiling in a mode that conservatively inserts barriers everywhere to prevent the attack, etc). 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.

Mitigation

We believe that the mitigation should be presented as a library feature rather than a core language feature, because we hope it will relatively rarely be needed. For almost all programmers, the correct response to the introduction of this feature will be to ignore it, and libraries are much easier to ignore than language features.

Our recommended mitigation API is informally specified as follows:

```
// 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, the above code can be protected like so:

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

This change ensures that the untrusted value is never used as an offset for the load (even in speculative execution), except when it has passed the bounds check.

We believe this API can be implemented efficiently on all platforms where Spectre variant 1 is being actively studied, and we are working with hardware vendors to ensure that is the case. One point particularly worth noting is that in designing how this API lowers to assembly language, we assume that it will always be possible to implement it in terms of instructions that are not subject to value prediction, and so cannot suffer from misspeculated values (although they may be subject to misspeculated control flow due to branch prediction. For example, our current x86 lowering assumes that even if the CPU speculatively executes code inside the bounds check, it will not predict that the value of the condition bit is true when evaluating subsequent conditional moves. We think this assumption is justified, because value prediction is susceptible to timing side channels even in the absence of branching, so we believe that any architecture that purports to support security-sensitive applications will necessarily provide a means of generating code that does not permit value prediction.

The major specification challenge here is that the “Remarks” section above is, from the point of view of the C++ abstract machine, complete nonsense (and without that section, this function is a no-op). C++ lacks a notion of speculative execution, so there is no way to even talk about what happens if code on a non-taken branch is nonetheless executed. This is not a mere matter of standards-lawyering: the way we answer this question will determine what optimizations compilers can apply, and that will determine how users of this API reason about the safety of their code.

For example, consider the following code:

```
if (x < y) {
  if (protect_from_speculation(x < y, …)) {
    …
  }
}
```

Is the compiler permitted to transform that into the following?

```
if (x < y) {
  if (protect_from_speculation(true, …)) {
    …
  }
}
```

Can it then optimize the `protect_from_speculation` call away entirely, since it’s purely a no-op when the condition is true? How do we explain the answers of those questions to users, in a way that tells them what they have to do to secure their code? We are unlikely to be able to solve such problems through mere non-normative handwaving.

However, if we want to address this problem normatively, it's difficult to see how we could avoid being put in the daunting position of trying to guarantee that even an adversarial (but conforming) compiler cannot cause the application to leak secrets via timing information, assuming the user follows a particular set of programming practices. Such an approach seems tantamount to suspending the as-if rule (at least in certain contexts) and imposing local constraints on the mapping from C++ code to machine code.

Alternative APIs

We have also considered an 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 attacker-controlled 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. This API works only for bounds checks, not for general predicates, but it seems possible that we could identify a minimal set of predicates that cover all realistic cases.

However, we think this API will in practice be harder to use correctly than the one we propose, 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 `protect_from_speculation`.