Software Defenses for Spectre

Understanding Spectre

The following code is vulnerable to Spectre variant 1.

```
int array1[ARRAY1_LEN];
int array2[ARRAY2_LEN];
foo_victim(int x /* untrusted parameter */)  {
  if (x >= 0 && x < ARRAY1_LEN)  {
    int val1 = array1[x];
    int val2 = array2[(val1 & 1) * N];
    // stuff
  }
  // more stuff
}
```

(1) Attack assumptions are as follows:

- The attacker and victim are in different protection domains.
- foo_victim is a function in the victim's protection domain.
- But foo_victim can be invoked by attacker with arbitrary values for parameter x.

(2) Expected behavior in a processor without speculation:

- array1[x] is only accessed if x < array1_len.
- Some information leakage through the cache.
    - The attacker can observe the access to address (array1+x)
    - And also observe similar access to array2
    - Attacker may learn some bits in the address of array1.
    - Harder to learn addr of array2, but possible depending on values in array1.
    - The attacker can learn (array1[x] & 1) for x >= 0 && x < ARRAY1_LEN.
        - This is the problematic leak.
- Information flow analysis reveals that:
    - array1 → cache   (information flows from array1 (addr of the array) to cache)
    - array2 → cache   (information flows from array2 (addr of the array) to cache)
    - array1[x] → cache   // this is the value, not the address.
        - Only for x >= 0 && x < ARRAY1_LEN // this path condition is important.
- TODO: the above idea, which attempts to capture the intuitive notion of what variables are observable through the cache needs to be formalized.

(3) With speculation, the picture is a little different.

- Everything the attacker could learn above is still possible.
- But it becomes possible to learn (array1[x] &1) for any x.
- Remark: one view might be that the path condition is ignored.

- Information flow analysis reveals that:
    - array1 → cache
    - array2 → cache
    - array1[x] → cache   // this is the value, not the address.
        - **For all x!   // this is *the* problematic leak in spectre.**

“Fixed” code for Spectre variant 1 is as follows.

```
int array1[ARRAY1_LEN];
int array2[ARRAY2_LEN];
foo_victim(int x /* untrusted parameter */)  {
  if (x >= 0 && x < ARRAY1_LEN)  {
    int val1 = array1[x];
    fence(); // ensure the above branch is resolved before loading.
    int val2 = array2[(val1 & 1) * N];
    // stuff
  }
  // more stuff
}
```

What really changed? The fence ensure that the path conditions reappear in the symbolic information flow analysis.

This leads to the following security property that should be satisfied by a spectre fix: the information flow analysis for a particular program should identical for processors with and without speculation. This captures the notion that speculation should not make a piece of code more insecure than before.

What I’d like is to develop a program analysis that can verify the above property. If the analysis finds violations, we will add enough fences so that it no longer violates the property and this new program will be “spectre safe” for all variants.

Suppose we have a function foo which is invoked with untrusted parameter X and returns the value R. During its execution, foo might have access to a confidential variable S.

Let’s use the term secure speculation to mean that foo does not leak any additional information about S when run on a processor with speculative execution. We’ll use the term indistinguishably secure speculation to mean secure speculation which also satisfies the property that the adversary cannot distinguish between whether the execution was speculative or not. Clearly indistinguishably secure speculation is a stronger property than just secure speculation.

Indistinguishably secure speculation (ISS) can be viewed as a 3-safety property.

```
forall t1, t2, t3 ::
      (forall i :: no_speculation(t1[i]) && no_speculation(t2[i])) ⇒
      (forall i :: t1[i].X == t2[i].X && t2[i].X == t3[i].X) ⇒
      (forall i :: t2[i].S == t3[i].S) ⇒
      (forall i :: obs(t1[i]) == obs(t2[i])) ⇒
            (forall i :: obs(t2[i]) == obs(t3[i]))
```

The property states:

- Traces t1 and t2 do not execute speculatively
- All three traces t1, t2 and t3 have the same adversary input
- Trace t2 and t3 have the same secret.
- And if we assume that the adversary is unable to distinguish between t1 and t2
- The adversary should also be unable to distinguish between t2 and t3.

In other words, a violation of this property would be an adversary input X which is unable to distinguish between the secret values of ‘S’ in t1 and t2 without speculative execution, but is able to do so on a processor with speculative execution.

Secure speculation (SS) can be viewed as 4-safety property.

```
forall t1, t2, t3, t4 ::
      (forall i :: no_speculation(t1[i]) && no_speculation(t2[i])) ⇒
      (forall i ::    t1[i].X == t2[i].X &&
                      t2[i].X == t3[i].X &&
                      t3[i].X == t4[i].X) ⇒
      (forall i :: t1[i].S == t3[i].S) ⇒
      (forall i :: t2[i].S == t4[i].S) ⇒
      (forall i :: obs(t1[i]) == obs(t2[i])) ⇒
            (forall i :: obs(t3[i]) == obs(t4[i]))
```

This property states:

- Traces t1 and t2 do not execute speculatively
- All four traces t1, t2, t3 and t4 have the same adversary input
- Trace t1 and t3 have the same secret, while t2 and t4 have the same secret
- And if we assume that the adversary is unable to distinguish between t1 and t2
- The adversary should also be unable to distinguish between t3 and t4

In this property, we are not asking the adversary to attempt to distinguish between t2 (a non-speculative processor) and t3 (a speculative processor). Instead the adversary has to distinguish between two identical speculative processors with different secrets, but the secrets are such that non-speculative processors are unable to tell the difference between them.

Note: This is not the same as this one: If there is Observational Determinism without Speculation, then there is OD with Speculation too. Because OD(t1,t2) ⇒ OD(t3,t4) does not permit us to compare t1,t2 with t3,t4 directly.
The SS formulation allows us to essentially state that speculation does not introduce NEW leaks beyond what was already there without speculation.

Comments

As stated above, this (SS) is a weaker property than indistinguishably secure speculation (ISS) but also captures a similar notion of no "new" information being leaked by the adversary.

It seems conceivable that a purely hardware defence to Spectre would need to satisfy ISS. For instance one could "undo" all changes to the cache when a misprediction is resolved and this would satisfy ISS for an adversary who can only observe cache hits/misses. Similarly, adding a fence after each branch that potentially accesses a secret would satisfy ISS, as we effectively end up eliminating speculation by doing this.

On the other hand, it would seem that a software defence to Spectre that does not end up serializing a lot of branches cannot provide this level of security, and probably can only satisfy SS.

Verifying the 4-safety property

Discussion Points

- When does the adversary get to observe the cache?
  - (O1) Is it when control returns to the adversary after "foo" finishes execution?
  - (O2) Or is the adversary executing in parallel and so can "see" each memory access?
- Suppose, the adversary is executing in parallel. Then the observation function could be the sequence of memory accesses made by the program.
  - If so, there cannot be control flow divergence between traces t1 and t2, because the observations would be different if the control diverged.
  - This means we can use a product program + taint construction to check the 2-safety property. This is described below.

Let the program be represented by the transition system M=(X, Init, R). We can derive a tainted transition system M^E=(X^E,$Init^E$,R^E) where x_i^E in X^E is a propositional variable such that $x_1^E \rightarrow t1.x_1 = t2.x_1$. Suppose x1' = op(x2, x3, x4). Then $x_1^{E'} = x_2^E \,\&\, x_3^E \,\&\, x_4^E$.

(This is basically the taint abstraction.) Suppose we have a bunch of constraints of the form above, then We can set obs^E = False and the remaining constraints will let us deduce what variables must be equal to each other to ensure the observations between traces t1 and t2 are the same. We can assert these equalities and then try to prove the observations are the same for t3/t4, the cases with speculation.

But the problem is that the above is not enough. Since $x_1^E \rightarrow t1.x_1 = t2.x_1$, it may be that there are cases when x_1^E is False but $t1.x_1 = t2.x_1$. Need to handle this case as well.

Control Flow Divergence in t3/t4

The other issue to discuss is whether t3 and t4 can have control flow divergence. Now they can't diverge for the "correct" path code, as it computes the same thing as t1 and t2, but they may misspeculate differently.

- One option is rule out this type of misspeculation -- this corresponds to the assumption that misspeculation is not influenced by secrets in t3/t4 -- it is purely adversary controlled, and the adversary makes the same tamper actions in both traces. This may make our life easier for verification -- and perhaps can be ensured using some architectural help.
- The other option is to allow misspeculation divergence between t3 and t4. This will complicate verification a little, but the details can be worked out.

Notes on Retpoline

1. Retpolines rely on fetch being redirected at decode when the attacker poisons the BTB to trick the processor into executing a non-existent branch.
2. Further, they rely on all predictions coming from the RAS, but we are back to using the BTB in certain weird scenarios on SkyLake. This thread has more details: https://lkml.org/lkml/2018/1/4/720
3. Seems like verification of retpoline will need a processor model which has:
    a. Fetch, decode and execute stages
    b. BTB, RAS, BPred
    c. Fetch redirection at decode *and* execute
    d. Calls and rets (can't use the RAS without these)

October 10, 2018 Meeting

- Need to work towards a publication

- Verification of 4-safety property on a simple model: Steps
    a. Convert speculative program into transition system and verify that this specific program satisfies 4-safety
    b. Focus on simple in-order pipelined processor and (i) find new Spectre-like attacks via 4-safety property, (ii) prove that after mitigations, the 4-safety property is satisfied
- Another idea: testing for the 4-safety property on Gem5 simulator
    a. Falsification rather than Verification
    b. Look into QED work
    c. Potentially use this in synthesis of high-level models in UCLID5