CS 261 Srcibe

Scribe: Fred Zhang

April 21, 2021

1 Spectre Attack and Speculative Execution

Speculative Execution. We know that CPU and its cache are faster than memory. A read operation from memory can slow down CPU computation,; and CPU may have to wait for a memory fetch before further computation. The basic idea of *speculative execution* then is to boost the CPU performance by making use of this idle period. Specifically, it attempts to perform some computation that may or may not be necessary or even correct. For a concrete example, consider the code

```
if (x==1) {
    do abc
} else {
    do xyz
}
```

When `x` is uncached (i.e., in memory), processor faces a long waiting time. CPU can optimize its performance by guessing the execution path and proceed speculatively. Here, CPU may choose to speculatively do `abc`. Then when `x` is read from memory, CPU checks if the guess is correct, commits if so and reverts CPU state otherwise.

Spectre Attack. This leaves space for *spectre attack*. The idea is to exploit conditional branch misprediction to read arbitrary memory from another context, e.g., another process. To set up the notation, suppose `k` is a secret value in memory, and the attacker wants to learn `k`. Here is the attack on a high level.

(i) Trick the processor to fetch `k` from memory.

(ii) Use a arbitrary list $a[0], a[1], a[2], a[3], ...$

(iii) Put $a[k]$ in cache and all other $a[i]$ in memory.

(iv) Infer the value of `k` by measuring read time difference.

Example. Let's consider the following piece of code.

```
if (x < array1_size) {
    y = array2[array1[x] * 4096];
}
```

The code begins with a bounds check on `x`. This prevents the processor from reading sensitive memory outside of `array1`, so is it secure? What if the CPU is doing speculative execution? Consider an attack scenario where the code runs in a trusted context and the attacker controls `x`. Here's how the attack works.

(i) The first step is to trick processor to mispredict the value of `x < array1_size`. That is, even if this statement should be false, we trick the CPU to predict that it is true so that the second line gets executed. For example, previous operations received values of `x` that were valid, leading the branch predictor to assume the if() condition will likely be true. Now, processor will execute line 2 speculatively.

(ii) The second step is to trick CPU to read secret value `k`. We'll pick the value of `x` to be N, which is a large number such that `array1[N]` stores the secret value `k`. Now `k` is illegally fetched by the processor. However, at this moment we don't know its value.

(iii) At the third step, note that the processor now needs to read `array2[k * 4096]`, so its value is requested from the memory. Now, this specific element from `array2` is in cache. Once the fetch is completed, the CPU realizes that the if() statement is false and reverts CPU state. Observe, however, that the cache state is not reverted.

(iv) At the final step, the attacker notes that only `array2[k * 4096]` is cached among all elements of form `array2[i * 4096]`. Hence, he simply reads `array2[i * 4096]` for all `i` and times each read time. If, say, he finds reading `array2[10 * 4096]` is noticeably faster than other (i.e., cached), then he learns that `k == 10`.

Discussion. We know that it is getting harder to boost CPU performance further from low-level physics, industry focus on other techniques. This includes speculative execution. On the flip side, this brings new security risks.

Another point is that speculative execution is that it breaks the contract between software and hardware. In fact, there is no way for software engineers to turn off speculative execution.

2 Cryptograhic Voting

Public Voting. The setup is that there are two candidates A and B and n voters that decide the voting outcome. One protocol is for every voter to announce their vote in some *public* space, and all votes are tallied afterwards. The concern here is voter privacy. Without privacy, there can be a risk of voting coercion and vote buying. This was actually kind of an issue in 19th century America before secret ballot was introduced.

Secret Voting. Hence, we want secret voting. Let's consider using cryptography. Suppose we have some trusted party that's going to run our vote for us. They sample a public key and secret key pair for some encryption scheme. They send everyone the public-key and everyone encrypts their vote under this public key. Finally, the trusted party decrypt everyone's vote, tally it up, and then just post the results to everybody. Now that we achieved privacy, the additional security properties we want are the following.

(i) public verifiable tally: we want everyone to be able to verify that in fact, the tally was computed correctly based on these encrypted ballots;

(ii) individual vote verification: everyone can verify that the encryption of their voting is the correct encryption.

(iii) distributed trust: we can run this protocol while ensuring that no single person (trusted party) can decrypts all the ballots.

We will focus on the first two properties today, and we mention that there is a technique called *threshold encryption* that achieves distributed trust efficiently.

Public Verifiable Tally. Consider a setup where we have six voters in our election. And they each release a ciphertext $ct_1, ct_2, \cdots, ct_6$ of their encrypted votes. We want to allow this trusted party to demonstrate what these ballots correspond to without actually revealing which vote corresponds to which person.

Here's one thing they can do. First, they can re-randomize each ciphertext. Then they permute the re-randomized votes randomly. If one only sees these re-randomized then permuted ballots, there is no way to trace them back to like which which original ciphertext each of them came from. Finally, the trusted party can decrypt the re-randomized then permuted ballots in a publicly verifiable way, which again is possible.

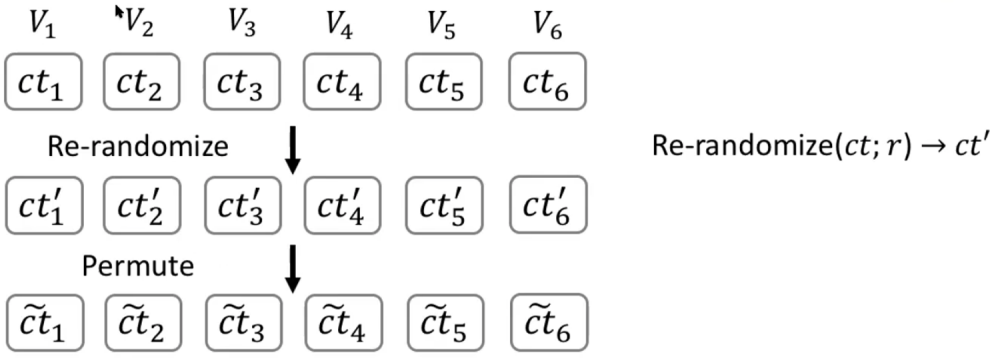

Figure 1: Protocol for achieving public verifiable tally.

However, we are not done here. We want to prove basically that $C = \{ct_i\}$ and $\widetilde{C} = \widetilde{ct_i}$ are equivalent; that is, they correspond to the same set of ballots, after being decrypted. The solution is the following:

(i) Generate $C_1, \cdots, C_{100}$ by re-randomizing and permuting C.

(ii) Obtain 100 random bits $b_1, \cdots, b_100$ (e.g., by hashing $C_1, \cdots, C_{100}$, known as Fiat-Shamir heuristic).

(iii) For $i \in \{1, \cdots, 100\}$,

- If $b_i = 0$, reveal re-randomization and permutation from C to C_i.
- If $b_i = 0$, reveal re-randomization and permutation from $\widetilde{C}$ to C_i.

If the people running the election lied (i.e., produced bogus $\widetilde{C}$), with overwhelming probability, they will fail the test.

Individual Voter Verification. We want a voter to be able to walk up to a machine, put in their ballot and out comes their encrypted vote ct, along with some extra information π, which we call a proof, that convinces them that that vote was correctly encrypted. The desired properties are the following.

- Non-verifying voters: voting is as simple as possible for folks who trust the machine and don't want to verify.
- Honest verifying voters: can use the proof π to verify the ciphertext ct.
- Dishonest voters: cannot prove to anyone else that ct encrypts A (or B).

These properties at first glance might be a little hard to obtain because there's some tension between these last two apparently.

Here is a pretty simple way to do it. The voter gives the vote, say, A to the machine. The machine returns ciphertext $ct = \text{Enc}_{pk}(A; r)$. Then the voter has a choice: either cast this vote or place immediately decrypt it. If they choose to decrypt, the booth will return the randomness r they used to encrypt the ballot. So now this voter can recompute the encryption and say that was an honest encryption of A. The voter

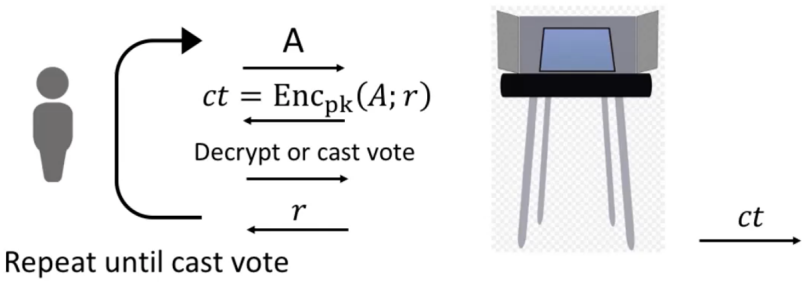

Figure 2: Protocol for individual voter verification.

can just do this as many times as they want until they cast a vote and be convinced the machine is honest. Also, note that there's no way to convincingly prove to anybody else what this final ciphertext is because the voter is not given the randomness that was used to encrypt the final ciphertext.