CS261 Scribe

Zhihong Luo

April 21, 2021

1 Spectre Attacks: Exploiting Speculative Execution

Speculative Execution. Speculative execution is a technique that is widely adopted for mitigating the performance degradation due to the fact that CPU and its cache are much faster than memory. Specifically, a single memory read operation could significantly slow down CPU computation, if the CPU has to wait a large number of cycles for the memory fetch to complete in order to proceed. To mitigate this issue, instead of waiting for memory operations to finish, speculative execution allows the CPU to guess execution paths and proceed speculatively. Take the following snippet as an example:

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

When variable x resides in the memory, the CPU issues a memory read operation that could take a long time before it knows which branch to take. Instead of waiting, the CPU can achieve better performance by making a prediction of the execution path and proceed in advance. In this example, the CPU might predict the branch condition to be true, and execute "abc" speculatively. After that, when x is read from memory, the CPU checks if its prediction was correct, commits if so and reverts CPU state otherwise. Note that, in case of a misprediction, while CPU micro-state is reverted, the cache state is not reverted. This is the fundamental vulnerability that spectre attacks exploit.

Spectre Attacks. For conditional branch attack, the goal is to exploit conditional branch misprediction to read arbitrary memory from another context, e.g., another process. With k as a secret value in the memory, the following is the high-level logic of a conditional branch attack, through which the value k is leaked:

1. Trick processor to mispredict and fetch k from memory.
2. Prepare a list/array $a[0], a[1], a[2], a[3]$...
3. Access $a[k]$ so that it is put into the cache.
4. Iterate through the array and learn k by measuring read time difference.

Take the following code snippet as an example.

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

The code snippet begins with a bounds check on x, in order to prevent the process from being able to read outside of $array1$. This is sufficient if the CPU does not perform speculative executions. However, if the CPU speculatively executes code for performance reason, a conditional brunch attach could be initiated to read a secrete value k as long as the attacker controls x, in the following way:

- The fist step is to trick the CPU to mispredict the condition $x < array1\_size$ to be true by mistraining the predictor. By doing so, the CPU will speculatively execute the second line even if x is out of the bound of $array1$. One way to mistrain the predictor is to generate a large number of valid inputs prior to the out-of-bound x. This is likely to mislead the predictor into making an incorrect decision and executing the second line.

- The second step is to trick the CPU to perform an illegal read of the secret value k. In order to do so, the attacker sets x to a large value N, so that $array1[N]$ corresponds to the secret value k, which is an illegal read. However, we do not know the value of k yet, which leads to the next step.

- At the third step, the CPU accesses $array2[k * 4096]$, so its value is requested from the memory to the cache. Once the fetch is completed, the CPU realizes that there was a misprediction since the if condition is false, and reverts CPU state. However, the cache state is not reverted, so $array2[k * 4096]$ still resides in the cache. Note that the constant 4096 is selected so that other elements in $array2$ won't be prefetched into the cache due to spatial locality.

- Lastly, the attacker knows the value of k by inferring which element of $array2$ is cached. For this, the attacker iterates the $array2[i * 4096]$ for all possible i's and times each access. Since $array2[k * 4096]$ resides in the cache, accessing it is significantly faster than other elements that are in the memory. Therefore, with this timing side channel, the attacker could learn the value of k.

Discussion: It's getting harder to boost CPU performance further due to constraints from low-level physics. Because of that, techniques such as speculative execution have been proposed. However, although these techniques boost performance, they also bring security risks, as demonstrated with the spectre attack.

Moreover, the contract between hardware and software is broken. In fact, there is no way to turn off speculative execution for software developers

2 Cryptographic Voting

Voting Problem: The general setup of a voting problem is that there are two candidates A and B and n voters that collectively decide the voting outcome.

Public Voting: The most straightforward protocol is for every voter to announce their vote to the public, and all votes can be tallied by some party according to the published votes afterwards. However, despite the simplicity, this protocol is undesirable because it violates **vote privacy**, so that an attacker could trace a vote back to a particular voter. This not only is a violation of voter's privacy, but also allows vote coercion and vote selling/buying. Note that for this discussion, issues with voter registration, voter validation, preventing double voting are considered to be out-of-scope.

Vanilla Secret Voting: In this protocol, there is a trusted party that is going to run the votes for the voters. They sample a public key and secret key pair and send voters the public-key to use. Each voter then encrypts their vote under with this public key and sends the encrypted vote to the trusted party. Finally, the trusted party decrypts everyone's vote, tallies it up, and then publishes the voting results. While this naive approach provides vote privacy, there are additional security properties that one might desire:

- **public verifiable tally:** it should be publicly verifiable that the tally was computed correctly based on voters' ballots.

- **Individual vote verification:** each individual voter should be able to verify that the encrypted vote actually corresponds to the candidate they wanted to vote for.

- **distributed trust:** the protocol should be run in a distributed fashion so that no single party can decrypt all the ballots.

The remaining of this talk focuses on the first two properties. Without going into the detail, the last property could be efficiently achieved with a cryptographic primitive called threshold encryption. With threshold

encryption, each party only has a share of the secret key and they collaborate to decrypt ballots. No single party alone could break the vote privacy.

Publicly Verifiable Tally: Consider a setup where there are six voters in the election, each of them releases a ciphertext $ct_1, ct_2, ...ct_6$ of their encrypted votes. To achieve public verifiability, the trusted party should be able to demonstrate what candidates these ballots correspond to without actually revealing which vote corresponds to which person.

As shown in Figure 1, a protocol is proposed to achieve this. Firstly, the trusted parties re-randomize each ciphertext with some injected noises. Then the trust parties permute the re-randomized votes. With re-randomization and permutation, there is no way to trace the final votes back to the original ciphertexts. Finally, the trusted parties can decrypt the re-randomized then permuted ballots in a publicly verifiable way.

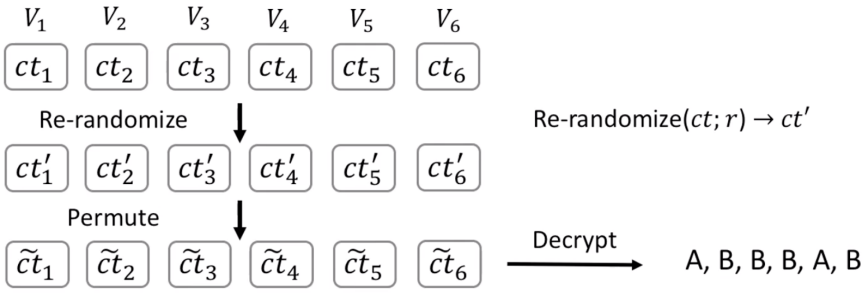


Figure 1: An publicly verifiable tally protocol.

With a publicly verifiable decryption scheme, what is left is to prove to the public that the re-randomization and permutation is done faithfully, which can be done by proving that $C = \{ct_i\}$ and $\tilde{C} = \{\tilde{ct}_i\}$ are equivalent; that is, they correspond to the same set of ballots, after being decrypted. One way to prove equivalence is as following:

1. Generate $C_1, ..., C_{100}$ by re-randomizing and permuting C.
2. Obtain 100 random bits $b_1, ..., b_{100}$ (e.g., by hashing $C_1, ..., C_{100}$).
3. For $i \in \{1, ..., 100\}$:
   - if $b_i = 0$, reveals re-randomization and permutation from C to C_i
   - if $b_i = 1$, reveals re-randomization and permutation from $\tilde{C}$ to C_i

If there are cheatings in the election process, there is an overwhelming probability that the trusted parties will fail this interactive test.

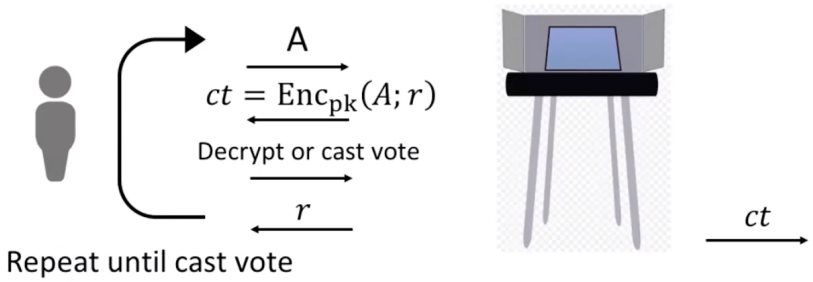
Individual Vote Verification: We would like a voter to be able to walk up to a machine, put in their ballot and out comes their encrypted vote ct, along with some extra information π, which we call a proof, that convinces them that that vote was correctly encrypted. The desired properties are the following:

- Non-verifying voters: voting is as simple as normal voting procedure.
- Honest voting voters: can use the proof π to verify the ciphertext ct.
- Dishonest voters: cannot prove to anyone else that ct encrypts A .

The last property is to prevent dishonest voters from selling their votes after proving who they vote for. The difficulty in designing such a protocol is due to the tension between the last two properties.

Figure 2 shows a rather simple protocol that achieves all three properties. The voter gives the vote, say, A to the machine. The machine returns ciphertext $ct = Enc_{pk}(A; r)$. Then the voter can choose between casting this vote or asking the machine to immediately decrypt it. If they choose to decrypt, the machine will return the randomness r used for encrypting the ballot. In this way, the voter can recompute the encryption

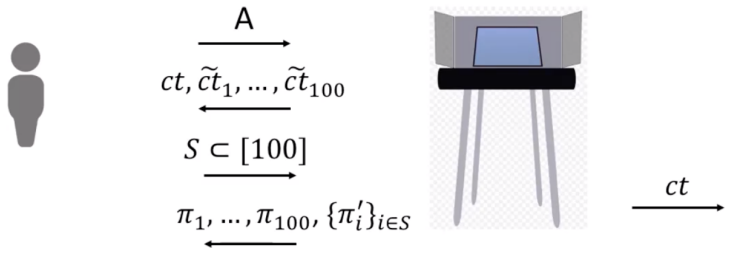
and verify that the ciphertext was an honest encryption of its vote A. The voter can choose to decrypt the vote as many times as they want until they are convinced that the machine is honest, and then they could choose to cast a vote and end the voting process. Note that there's no way to convincingly 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. For non-verifying voters, they could cast the vote at the first choice, leading to little overhead compared with normal voting procedures.



$$\pi = (ct_1, r_1, \ldots, ct_{m-1}, r_{m-1}), \qquad ct = ct_m$$

Figure 2: An iterative individual vote verification protocol.

This simple iterative proof protocol satisfies all three desired properties. However, it has a drawback of requiring a large number of message exchanges between the voter and the machine for an honest voter to be convinced. Figure 3 is an alternative iterative protocol proposed to achieve the desired properties with only two message exchanges.



For $i \in S, \pi_i$ is the randomness used to encrypt $\widetilde{ct}_i$
For $i \notin S, \pi_i$ is the randomness used to re-randomize $ct \to \widetilde{ct}_i$
For $i \in S, \pi'_i$ is an encryption of B and the randomness used to encrypt

Figure 3: Another iterative individual vote verification protocol that finishes with two message exchanges.