Flush+Flush Cache Side Channel Attack

Tamir Skutelsky, Danielle Eida-Zakai

1. Introduction

1.1 Cache

The CPU cache is a microarchitectural element that reduces the memory access time of recently used data. It is shared across cores in modern processors.
On modern Intel processors, there are three cache levels. The L3 cache (also called last-level cache) is shared between all CPU cores. In addition, L3 cache is inclusive. That means that the data presented in L1&L2 is also present in the L3 cache.

1.2 Side-Channels

Side channels are channels that are created by unintended data leakage from a computing system to its environment. If the attacker can sample the Side-channel, then he can learn information about the system.
Side channels are common in modern processors due to their complexity. Potential attackers can collect sensitive information about the system and use it for his own benefits.
In our report, we deal with Cache side channel attacks. They are based on the attacker's ability to monitor cache accesses made (or not made) by the victim.

1.3 Weaknesses

The clflush instruction removes the cache line containing the element, given as a parameter, from all levels of the processor cache hierarchy (data and instruction). The execution time of the clflush instruction is shorter if the data is not cached and higher if the data is cached.
In addition, Operating systems uses shared memory to reduce the overall physical memory and TLB usage. Shared libraries are loaded into physical memory only once and shared by all programs using them. Another example of shared memory could be found on PCs, smartphones, clouds – based on the content of the page – The pages are scanned by content, and Identical pages are remapped to the same physical pages. Those kind of optimizations can create a shared memory between completely unrelated processes, possibly even running on different VMs. It brings up security and privacy concerns.

1.4 Detectability of existing cache attacks

Most cache attacks can lead to an increased number of cache hits or misses in the attacker process or in other processes. This kind of behavior makes it possible to detect abnormal behavior on the system. To prevent an attack, we need to detect the attacking process.
By definition, we call an attack stealthy if we cannot identify the attacking process. Several hardware-related events can be monitored to detect such an attack, such as cache references and cache misses. Detection of the Flush+Flush attack using those performance counters is not practical, since it doesn't load data into the cache and uses

minimal number of cache references, thus making Flush+Flush stealthy.

1.5 Background to previous attacks:

Flush+Reload:

Flush+Flush attacks exploits the same hardware and software properties as Flush+Reload, but in contrast to Flush+Reload, Flush+Flush does not make any memory accesses.
Flush+Reload is a side-channel attack, which targets the L3 cache.
This attack is based on the clflush instruction and using the shared cache.
Process 1 wants to know if process 2 used a specific line from their shared memory. Process 1 executes clflush to a line in the cache and waits – to let process 2 to use, or not to use this line. If process 2 uses this line – it will come back to the cache. Now process 1 will try to read this line – if it reads it fast, it means that this line exists in the cache, meaning process 2 used it. Otherwise, it will read it slow, meaning that process 2 didn’t use it.

Prime+Probe:

in this attack, process 1 wants to know if process 2 accessed to an address in process 1’s memory, or not.
Process 1 finds a group of lines from his memory, that fills the cache set that process 1 is interested in. the first step of the attack is prime - process 1 fills the cache set with his lines and waits, in order to let process 2 access it, or not. If process 2 accessed this cache set, it caused the removal of one of the lines of process 1 from the cache. Now process 1 will go on to the next step of the attack – probe. In this step, process 1 will access all its lines, and measures how much time it took to access it.
If all the accesses time were fast, it means that all its lines were already in the cache, meaning process 2 didn’t access those lines at all.
If at least one of the lines accesses time was slow – process 2 did access a line of process 1.

1.6 The article’s contribution:

The article presents the Flush+Flush attack, which unlike any other cache attack, performs no memory accesses. Also, the Flush+Flush does not trigger the prefetcher and thus is more applicable in more situations than other attacks.
The Flush+Flush attack is faster than any other cache attack, as it preforms no memory accesses. the attack causes no cache misses at all. For this reason, detection mechanisms based on performance counters to monitor cache activity fail. This fact makes it hard to detect.

2. Technical Details

2.1 Pre-Requirements

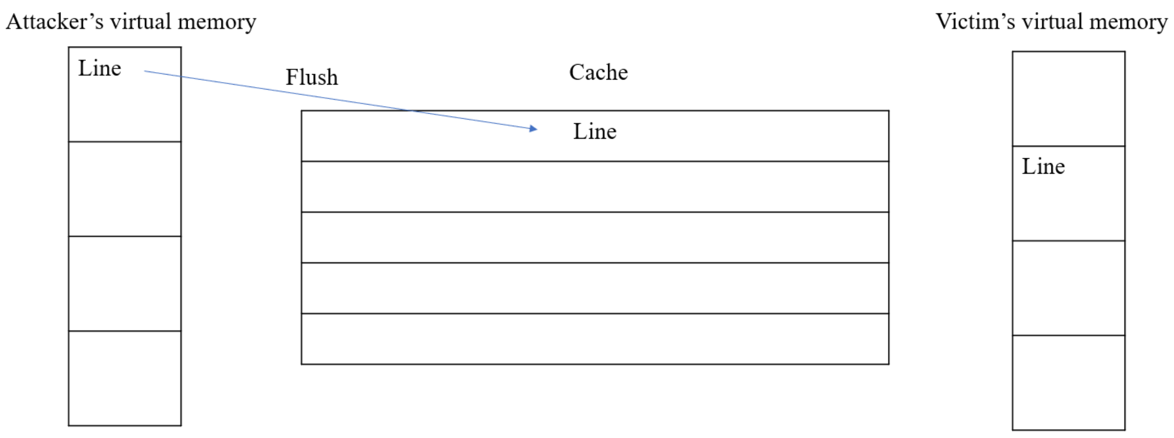
Flush+Flush cache attack is required to run on an Intel processor (x86), since the attack uses the clflush command. Second, our attack also takes advantage of Shared Memory, created by OS's optimizations, such as when the operating system loads Shared libraries into the physical memory only once. It creates a scenario where multiple programs can access the same physical memory pages mapped within their own virtual address space. Also, in order to the attack to be effective, the attacker must know the offset of a line of code within the executable file it is monitors.
Third, We also take advantage that the cache is inclusive – meaning that if a data is not presented in L3 cache – it is also not presented in L2 & L1 cache.

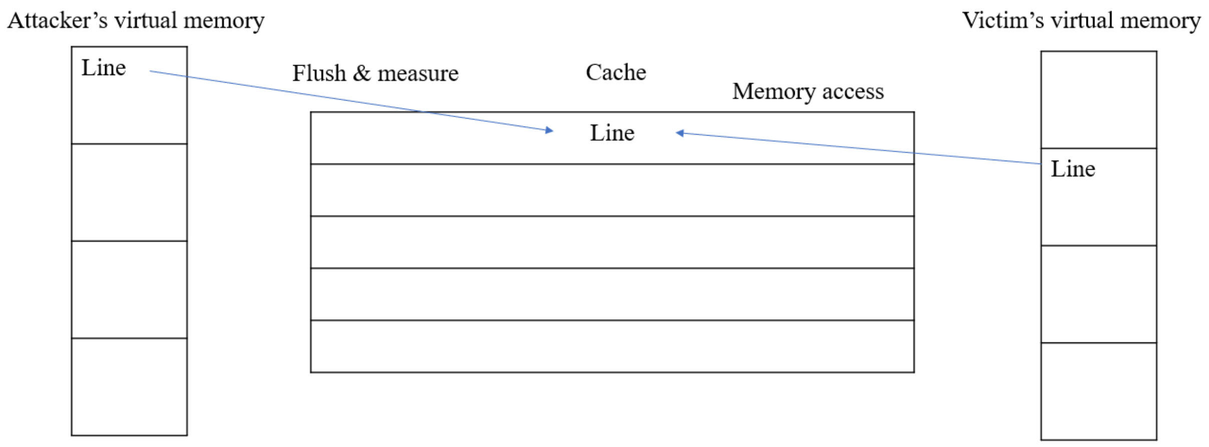
2.2 clflush Instruction

The main vulnerability that Flush+Flush uses is that the clflush instruction execution time is slower when we get a cache hit. The clflush instruction flushes the line that contains the memory address from all cache levels. The clflush instruction must be ordered by mfence.
We used the rdscp command to measure time (in clock-cycles).

2.3 Flush+Flush cache Attack

Our attack consists of Three stages: In the first stage, The attack opens the file that the attacker wants to monitor, and map the file into the attacker's address space, and to get the memory address of the file within the attacker's virtual memory. This is the only memory access that the attack performs. Afterwards, the attacker flushes the line, to remove it from the cache.
Then, the attacker repeatedly flushes the specified cache line, while measuring the execution time of clflush command using rdscp. If the time measured is higher than the threshold, it means that the line is touched. Otherwise, it has not been touched. We can repeat this process until we finish monitoring the victim behavior and output the result.





2.4 Usage

Flush+Flush cache attack, similarly to Flush+Reload and Prime+Probe, can be effective in a certain cache attacks, such as Covert Channel, Side-Channel attack on user input and Side-Channel attack on AES with T-tables.
In our report we choose to implement the Side-Channel attack on user input, using Flush+Flush.

3. Implementation

3.1 GTK Library

The GTK Library is an open-source cross-platform widget toolkit for creating graphical user interface. GTK include GDK, which responsible for input device handling, that will be used later to execute our Side-Channel attack.

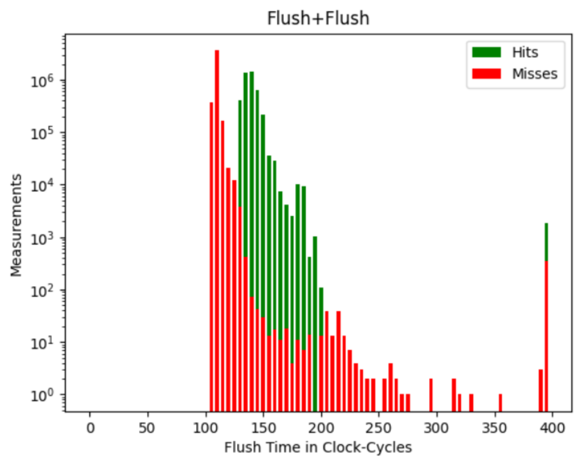
3.2 Flush +Flush cache attack on user input

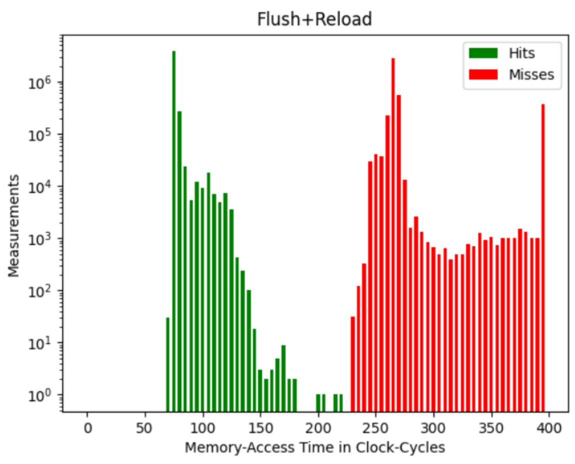
In our project, we choose to implement the Flush+Flush cache attack on user input by capturing the keystrokes timings and memory accesses. We repeatedly flush an address in the GTK library associated with certain key, to find out whether this key was pressed or not.

3.3 Challenges

3.3.1 Calibration

During our implementation, we encountered some challenges. The first one is calibrating our cache attack correctly. It is a vital stage, in order to get a correct measurements and minimize background noise.
In our project, we measured the hit duration and miss duration in Cycles, and received the following result for our Flush+Flush cache attack, and for comparison, we also measured the Hit/Miss maccess time of the Flush+Reload cache attack:





As we can see, the difference between Hit/Miss times in the Flush+Flush attack is less significant. This fact might make us monitor cache hits as cache misses and vise-verse, thus will result in a less accurate attack. However, as we mentioned before, Flush+Reload have a greater probability of the attacker process being detected, due to excess memory accesses.

3.3.2 Background noise

Every background process on the OS, such as internet connection and mouse movement might cause background noise. In order to minimize it, we add sleep before running the spy-process in order to eliminate the mouse-movement related cache hits, and try to close as much processes ad possible – such as internet connection and other applications on the background, and keep only the necessary ones, such as gedit (notepad) and the terminal.
One additional solution, is to monitor only when the spy process reaches the “stable” stage. By doing that, we eliminate the background noise created by starting the spy process.

3.3.3 Associating addresses on memory with keystrokes

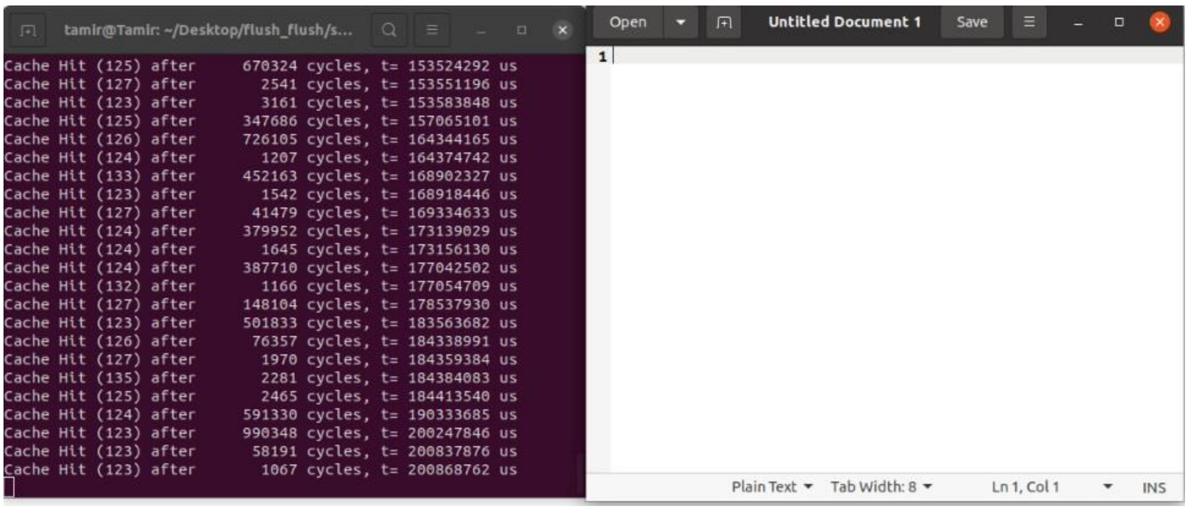
Our next challenge, after we identify a key was pressed, is to know the specific location in address space, that is affected by a certain key. By doing that, we could associate memory address with a key and approximate the key that was presses. Thus, we will try to create “map”-like array by profiling the key-presses events (connecting memory address to certain keys).

3.4 Choosing the right threshold - Calibration Stage

We must keep in mind that the measurements shown on section 3.3.1 can change between machines. So, when starting the cache attack, we perform the calibration stage – we take 4 million measurements of flush. Once when preforming a cache hit, and once when preforming a cache miss. The output of this stage will be a table of time (in clock-cycles), and the amount of cases. We choose our threshold to be after the maximum of the right column (Miss cases) and before the maximum of the left column (Hit cases) and change the MIN_HIT constant in the Profiling and Exploitation stages.

3.5 Keypress effect on cache

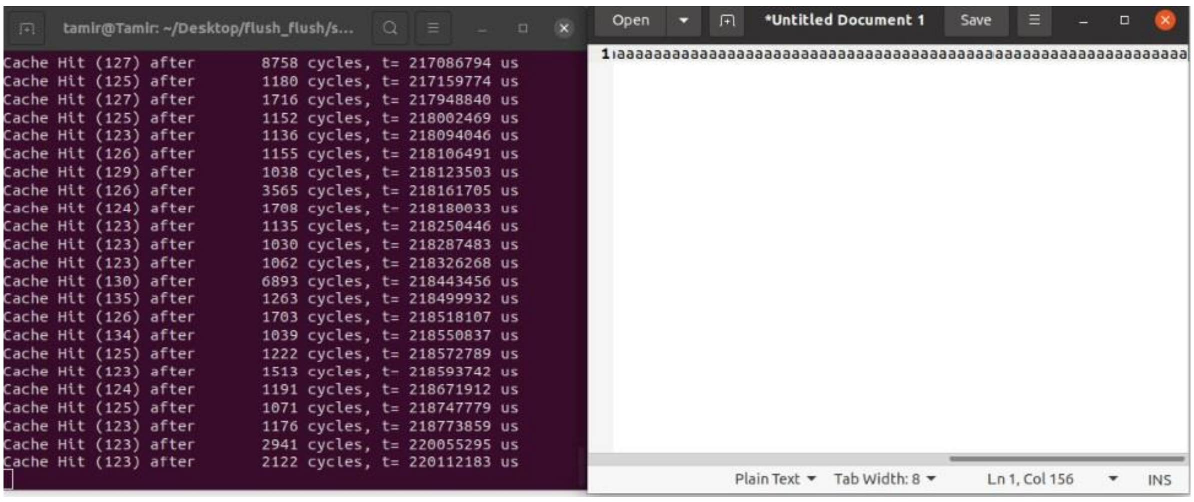
In order to verify that keystrokes indeed affect the cache, we tried to monitor the libgdk-3 library behavior when a key is pressed. We tried to test both cases – when a key is not pressed and when a key is pressed and see if there had been suspicious cache activity. Here is a screenshot of the spy process in "stable" stage, and without keystrokes:



```
Cache Hit (125) after     670324 cycles, t= 153524292 us
Cache Hit (127) after       2541 cycles, t= 153551196 us
Cache Hit (123) after       3161 cycles, t= 153583848 us
Cache Hit (125) after     347686 cycles, t= 157065101 us
Cache Hit (126) after     726105 cycles, t= 164344165 us
Cache Hit (124) after       1207 cycles, t= 164374742 us
Cache Hit (133) after     452163 cycles, t= 168902327 us
Cache Hit (123) after       1542 cycles, t= 168918446 us
Cache Hit (127) after      41479 cycles, t= 169334633 us
Cache Hit (124) after     379952 cycles, t= 173139029 us
Cache Hit (124) after       1645 cycles, t= 173156130 us
Cache Hit (124) after     387710 cycles, t= 177042502 us
Cache Hit (132) after       1166 cycles, t= 177054709 us
Cache Hit (127) after     148104 cycles, t= 178537930 us
Cache Hit (123) after     501833 cycles, t= 183563682 us
Cache Hit (126) after      76357 cycles, t= 184338991 us
Cache Hit (127) after       1970 cycles, t= 184359384 us
Cache Hit (135) after       2281 cycles, t= 184384083 us
Cache Hit (125) after       2465 cycles, t= 184413540 us
Cache Hit (124) after     591330 cycles, t= 190333685 us
Cache Hit (123) after     990348 cycles, t= 200247846 us
Cache Hit (123) after      58191 cycles, t= 200837876 us
Cache Hit (123) after       1067 cycles, t= 200868762 us
```

```
1
```

We can see here that there is a minimal number of cache hits, and there is a big timing difference between each cache hits (after X cycles means that X cycle has passed since the lase cache hit).
In comparison, those are the cache hits after we pressed a key repeatedly:



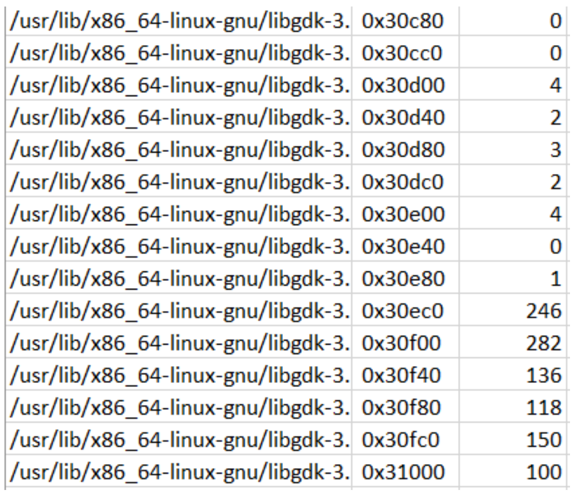
```
Cache Hit (127) after       8758 cycles, t= 217086794 us
Cache Hit (125) after       1180 cycles, t= 217159774 us
Cache Hit (127) after       1716 cycles, t= 217948840 us
Cache Hit (125) after       1152 cycles, t= 218002469 us
Cache Hit (123) after       1136 cycles, t= 218094046 us
Cache Hit (126) after       1155 cycles, t= 218106491 us
Cache Hit (129) after       1038 cycles, t= 218123503 us
Cache Hit (126) after       3565 cycles, t= 218161705 us
Cache Hit (124) after       1708 cycles, t= 218180033 us
Cache Hit (123) after       1135 cycles, t= 218250446 us
Cache Hit (123) after       1030 cycles, t= 218287483 us
Cache Hit (123) after       1062 cycles, t= 218326268 us
Cache Hit (130) after       6893 cycles, t= 218443456 us
Cache Hit (135) after       1263 cycles, t= 218499932 us
Cache Hit (126) after       1703 cycles, t= 218518107 us
Cache Hit (134) after       1039 cycles, t= 218550837 us
Cache Hit (125) after       1222 cycles, t= 218572789 us
Cache Hit (123) after       1513 cycles, t= 218593742 us
Cache Hit (124) after       1191 cycles, t= 218671912 us
Cache Hit (125) after       1071 cycles, t= 218747779 us
Cache Hit (123) after       1176 cycles, t= 218773859 us
Cache Hit (123) after       2941 cycles, t= 220055295 us
Cache Hit (123) after       2122 cycles, t= 220112183 us
```

```
1 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

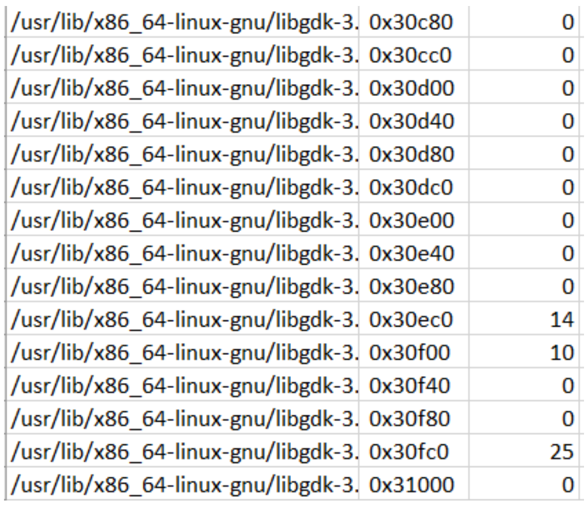
We can see here that the number between cache hits has increased dramatically. Thus, we can assume that keystrokes within the GTK library memory range indeed affects the cache.

3.6. Profiling a single keypress

One of the most important stages of our cache attack is to know which memory addresses the pressed key affects, in order to identify the keypresses correctly.
In order to do that, we monitored a range of memory addresses we know that the GTK-library is using, and check the number of cache hits in each address, while simultaneously pressing a single key. For comparison, we can see that some addresses are affected when hitting a key (a) and some not. However, when not hitting a key, the same addresses that are affected when pressing the key are not affected.



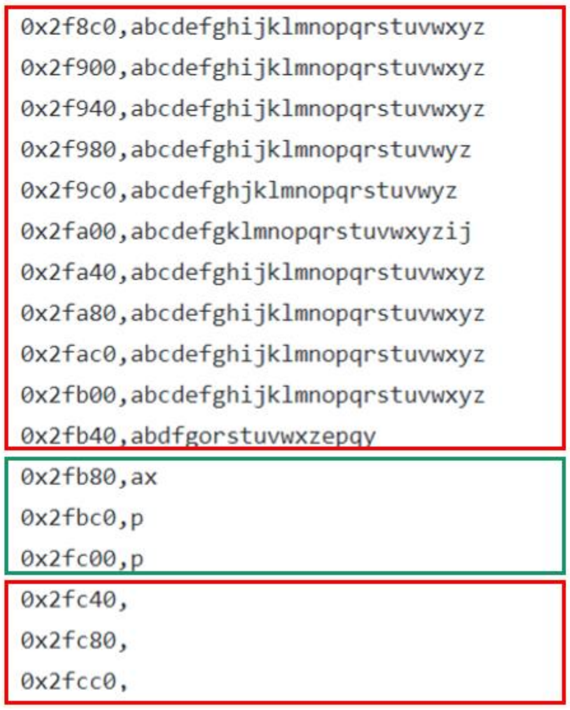
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30c80	0
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30cc0	0
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30d00	4
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30d40	2
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30d80	3
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30dc0	2
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30e00	4
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30e40	0
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30e80	1
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30ec0	246
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30f00	282
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30f40	136
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30f80	118
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30fc0	150
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x31000	100



/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30c80	0
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30cc0	0
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30d00	0
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30d40	0
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30d80	0
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30dc0	0
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30e00	0
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30e40	0
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30e80	0
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30ec0	14
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30f00	10
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30f40	0
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30f80	0
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x30fc0	25
/usr/lib/x86_64-linux-gnu/libgdk-3.	0x31000	0

3.7 Constructing a target – Profiling stage

To simulate user input in large scales, we could use the libxdo library. This library's purpose is to simulate keypresses. First, we need to open the gedit window (which uses the libgtk library). We could construct a program that create N keypresses with random keys in the range of a-z, with different timings using the sleep function. When running the program, we need to switch to the gedit window and let the program run on the background. If you are unable to install libxdo on your local machine, you will have to use the Non-Automated stage, meaning that you will have to generate keypresses and process the results yourself.
The Automated Profiling stage generates a list of memory addresses and the keys that are effecting this memory address. Keep in mind that some addresses are irrelevant, since no keys are affecting this memory address, and there are memory addresses that are affected by too many (We set the threshold to be 5) keys. So, our profiling stage filters those addresses at the end of the automated profiling stage.



```
0x2f8c0,abcdefghijklmnopqrstuvwxyz
0x2f900,abcdefghijklmnopqrstuvwxyz
0x2f940,abcdefghijklmnopqrstuvwxyz
0x2f980,abcdefghijklmnopqrstuvwyz
0x2f9c0,abcdefghjklmnopqrstuvwyz
0x2fa00,abcdefgklmnopqrstuvwxyzij
0x2fa40,abcdefghijklmnopqrstuvwxyz
0x2fa80,abcdefghijklmnopqrstuvwxyz
0x2fac0,abcdefghijklmnopqrstuvwxyz
0x2fb00,abcdefghijklmnopqrstuvwxyz
0x2fb40,abdfgorstuvwxzepqy
0x2fb80,ax
0x2fbc0,p
0x2fc00,p
0x2fc40,
0x2fc80,
0x2fcc0,
```

3.8 Full attack process

We can repeat the process described in section 3.5 for every key that we want to monitor using the libxdo tool and check the ratio of cache hits and misses while simulating the keypresses using the libxdo tool. If the ratio is higher than 70%, we associate the key with the memory address. Note that there may be addresses that are affected by every key press and thus we want to ignore that, and only deal with the most specific addresses for each keystroke.

We can monitor the cache hits/misses on those addresses. If we receive a cache hit on a certain address, we output the event (the keystroke) associated with this memory address. After we Identify a keypress, we output the timing and the key that was pressed.

This is the full pseudocode of the Attack:

```
For Every Key monitored k
    For Every Memory address m in the monitored file
         Check number of cache hits in m while pressing k
           If (Cache_Hits/Number_of_mesurments) > 0.7
           Associate k with m
```

3.9 Running the attack

Full Instructions and running examples are provided in the Examples folder, and in the “How to run” text file.

4. Summary

4.1 Conclusions

Although Flush+Flush cache attack is not hard to implement, as it relies mostly on the execution time of the Flush instruction, it is very hard to get accurate results by using this cache attack, because of the lower timing differences between cache hit and cache misses, as we saw in section 3.3.1, and the amount of background noise created by other processes that running on the OS.
We have to keep in mind that we tested this cache attack under “Lab conditions”, with only two processes running. In most of the tests we conducted, we managed to identify the key (or a group of keys, that contains the pressed key), but received a lot of false positives along the way, which made the attack not reliable.
On a normal “day-to-day” scenario, when a user is connected to the internet and have additional processes running on the background, We believe that this cache attack would not be practical.

4.2 Countermeasures

As we said in section 1.4, Flush+Flush is hard to detect. However, we can suggest a solution that will prevent this cache attack: making the clflush execution time constant. As the clflush command is not used in high frequency, it will not lead to significant decrease in performance.

4.3 Credits

Our implementation is based on Daniel Gruss’s implementation for the Flush+Reload cache attack on user input, found on:
https://github.com/IAIK/cache_template_attacks
We fully converted the cache attack to work with Flush&Flush, and added several features such as filtering irrelevant memory addresses (memory addresses which effected by zero keys, or memory addresses which effected by more than five keys). We also added comments to the code and running examples.