# Demonstration and Defense of GoFetch Attack

Zebang Fei
Southern University of Science and Technology
Shenzhen, Guangdong, China
12110608@mail.sustech.edu.cn

Ben Chen
Southern University of Science and Technology
Shenzhen, Guangdong, China
chenb2022@mail.sustech.edu.cn

Jiarun Zhu
Southern University of Science and Technology
Shenzhen, Guangdong, China
12210212@mail.sustech.edu.cn

Zhongwen Chen
SiChuan University
Chendu, Sichuan, China
2022141480020@stu.scu.edu.cn

Qingyang Zhang
Xi'an Jiao Tong University
Xi'an, Shaanxi, China
2203314931@xjtu.edu.cn

## ABSTRACT

Prefetcher has been widely applied in the caches of modern processors to boost the cache efficiency. However, its existence violates the constant-time programming paradigm that resists the Cache-based Side-Channel Attacks (SCA), making processors exposed to SCAs once more. To be worse, reseachers have found that Apple's M-series processors adapt a more vulnerable prefetcher that would dereference/prefetch the data pointed by cache content, which magnifies the flaw. GoFetch attack is thereby proposed to trigger that vulnerability. With the aim to give a comprehensive explanation about this brand-new attack, we develope a educational website with animation and reproduce the attack to clearly address it. We also adapt similar detection and prevention from cache SCAs to mitigate it, which is verified by our experiment. We believe that this paper will provide programmers with an distinct idea of principles and prevention of GoFetch attack, to avoid such issue in software.

## KEYWORDS

Side-channel attack, prefetcher-based side channel, Visualization of cache side-channel, Side-channel detection and prevention

## 1 INTRODUCTION

Side-channel attacks are a type of security exploit that leverage physical characteristics of a system, rather than exploiting algorithmic vulnerabilities, to extract sensitive information. Recent studies have demonstrated the evolving complexity and efficacy of side-channel attacks. Jin et al. presented a new side-channel attack on Intel CPUs that relies on timing transient execution to extract sensitive information [7]. Similarly, Kogler et al. introduced the Collide+Power attack, which measures CPU power consumption to deduce cryptographic keys [9]. Moreover, Pinto and Rodrigues showcased a side-channel attack on ARM TrustZone, highlighting the vulnerability of microcontrollers to these attacks [13].

GoFetch is a recently proposed side-channel attack method that was first introduced in 2023 [4]. This attack exploits the Data Memory-dependent Prefetcher (DMP) found in Apple's M-series CPUs. Unlike traditional cache-based SCAs, GoFetch targets at the behaviour of pre-dereference of pointer present in caches. By carefully crafting malicious eviction set and attacking with Prime&Probe, the attacker can infer sensitive information from the timing variations caused by the prefetching mechanism. This allows GoFetch to break constant-time cryptographic implementations.

In this paper, we will delve into the GoFetch attack, attempt to explain the principles of the GoFetch attack in a novel and comprehensive way via visualization website of the attack scheme. Based on the understanding, we try to reproduce, detect, defend against it. We managed to reproduce the chosen-input attack against the Constant-time Swap and Go's RSA encryption. To go beyond, we implement effective defense methods against the attack on software level. While the defense scheme suffers from performance loss known as "security tax", we measure the actual lost for each defense method, providing detail for trade-off. We also adapt side-channel detection techniques to detect attacks exploiting DMP, using dynamic detection of malicious inputs and priming in macOS.

## 2 BACKGROUND

### 2.1 Side Channel Attacks

In contrast to the mathematical models used in cryptanalysis, side-channel attacks target at the actual hardware systems implementing the cryptographic algorithms. These attacks exploit certain characteristics of specific hardware systems and carry out physical attacks on devices to steal secret from them. Compared with direct attack, SCAs are more powerful but less accurate.

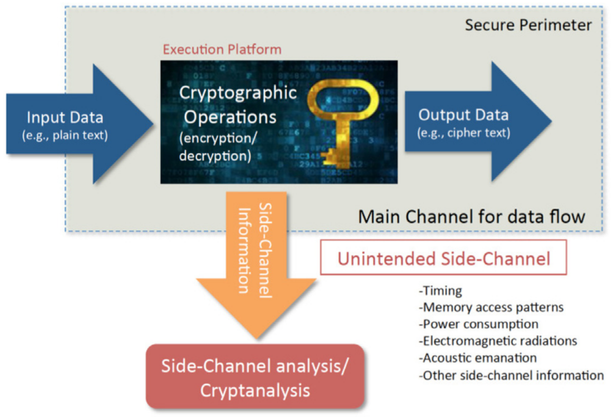


**Figure 1: Side-channel attack model**

As Figure 1 indicates[1], side-channel attacks (SCAs) exploit devices by the unintentional and largely unavoidable signals of it to steal secrets like private keys and sensitive information from victim machines. Although the victim might be a black or grey box for attacker, the attacker can utilize the information gathered at their best.

## 2.2 Cache

Cache is a smaller, faster memory component located closer to the CPU, designed to store copies of frequently accessed data from the main memory. Typically, a modern CPU has one isolated L1 cache for each core and shared L2 cache within a bundle of core. Note that to ensure memory consistency, content of L2 caches is inclusive of that of L1 caches. In other words, what is not present in L2 will never be present in L1. For specification of Apple's M1 processor, it uses 128 KB, 8-way associative L1 cache for each core and shared 12 MB, 12-way associative L2 cache for each 4 cores [16].

## 2.3 Cache-based Side Channel Attacks

Cache side-channel attacks leverage the differences in access times between cache hits (when data is present from the cache) and cache misses (when data is retrieved from the main memory). By measuring these access times, attackers can infer sensitive information about the data being processed [3, 11, 12]. Time-driven attacks measure the total execution time of cryptographic operations [3]. Trace-driven attacks monitor the processor's cache access patterns [11]. Figure 2 shows the attack path of Prime&Probe attack[8],

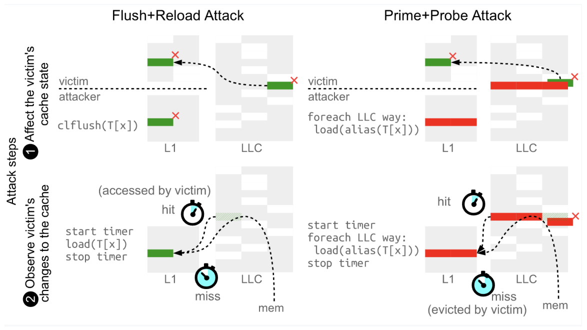


**Figure 2: Flush&Reload and Prime&Probe Attack**

which will used in GoFetch. Prime&Probe attacks involve the attacker filling the cache with their data (Prime) and measuring access times to infer cache usage by the victim process (Probe) [12]. From the statistical analysis of the timing difference in access, we can deduce the threshold that distinguish whether the target memory is accessed by victim.

## 2.4 Data Memory-dependent Prefetcher (DMP)

As of 2024, data prefetching has already been a common feature in modern CPUs, which fetches the data from memory before the actual access, by monitoring memory access patterns and predicting the possible next access. To boost more performance, the data memory-dependent prefetcher (DMP) takes the step further to search cache content for possible pointer values, and pre-fetch the target data of pointers under some criteria and restriction [14]. The detail found by Chen et al[4] will be explained at Section 3.1

## 3 UNDERSTANDING GOFETCH ATTACK

In GoFetch attack, attackers can exploit the behavior of DMP to infer the protected data patterns. The DMP in Apple's M1 processor architecture was demonstrated to be used as a memory side-channel in an attack published in early 2024 [6]. The DMP was subsequently discovered to be even more opportunistic than previously thought, and has now been demonstrated to be able to be used to effectively attack the constant-time cryptographic algorithms [4].

## 3.1 Apple's DMP Mechanism

Apple's DMP behaviour can be generalized as follows. DMP continuously monitors the processor's memory accesses and records the addresses and data being accessed. It analyzes these patterns and contents to identify potential pointers within the data. Once a pointer is identified, DMP attempts to predict future memory accesses by preloading the data located at these pointer addresses into the cache. This proactive data loading enhance the cache-hit rate. Furthermore, DMP's sophisticated filtering mechanisms, such as the history filter and do-not-scan hint, help in maintaining efficiency by avoiding redundant operations. However, the extreme proactivity makes it much simpler for attacker to maliciously trigger the DMP, raising the security concerns.

Through preliminary work by Chen et al[4], we can understand that the DMP used by Apple has the following characteristics, which are precisely utilized in subsequent attacks.

*Accessing Pattern.* DMP is an advanced prefetcher as mentioned, meaning that accessing an array of data will activate the pre-fetch of adjacent data. Beyond that, DMP will automatically dereference/fetch `data` pointed by `ptr` when accessing `ptr` with both implicit and explicit dereference of `ptr`. Experiment also suggests that DMP dereferences the adjacent `ptr`s with the same array and within the neighboring cache lines. These loose activation patterns imply a critical security issue.

*Activation.* To avoid redundant prefetches, DMP uses a history filter to record pointers that have already been dereferenced. If a pointer exists in the history filter, DMP will not dereference it again. Additionally, DMP uses a "do-not-scan" hint in complement of history filter. If a target data is brought and then evicted from caches with its `ptr` present in cache, DMP does not re-dereference `ptr` even if no related record in history filter, unless `ptr` is evicted from both caches. This mechanism plays significant role in GoFetch exploitation.

*Restriction.* DMP will only dereference pointers located within the same 4GB-aligned region. This means that DMP will only dereference pointers whose upper 32 bits of their addresses match those of the target address. The restriction should be considered essentially since it's the major reasons for dysfunction of eviction set.

We can utilized these characteristics of Apple's DMP in leaking sensitive information. Specifically, GoFetch leverages the DMP's content-driven prediction mechanism to deduce the victim program's behaviour. Based on the understanding of DMP's behaviour and by carefully crafting memory content and access patterns, an attacker can manipulate the DMP's activation on special condition (when the victim program is running a snippet or not), and then monitor that activity.

## 3.2 Threat Model

The GoFetch attack targets various cryptographic implementations. For example, although constant-time implementations aim to defend against side-channel attacks by eliminating timing differences, GoFetch circumvents these defenses by leveraging the characteristics of DMP. . The section provides an overview of exploitation using the characteristics of DMP.

*3.2.1 Prerequisite.* The capabilities of attacker are assumed as

a) **Observation of Memory Access Patterns**: The attacker should share the L2 cache with victim. Concurrent high-load application will bring noise to the observation which must be avoided.
b) **Chosen Input**: The attacker can inject arbitrary data into the victim to trigger the DMP to prefetch, thereby leaking sensitive data. Otherwise attacker should possess sufficient information.
c) **Limited System Access**: The attacker does not need full control over the system, restricted privilege as a normal user program would be enough for the attack.

*3.2.2 Attack Scheme.* Below shows the attack routine

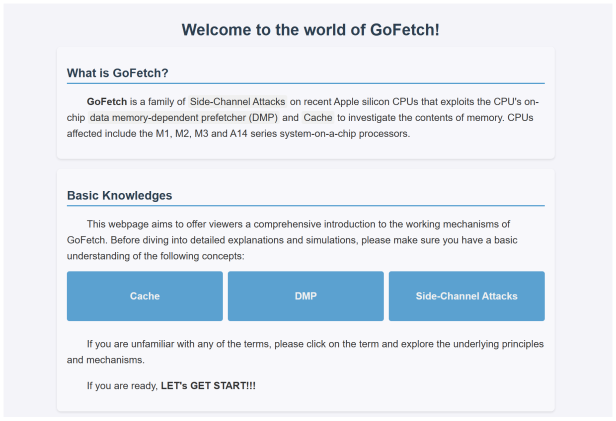
i. **Timing Source**: As the unprivileged attacker has no access to the high-resolution counter provided by OS, it has to prepare the timer as an independent thread to gain high precision of record. The timer thread acts normally as a user thread.
ii. **Generate Evictset**: To perform a standard Prime&Probe attack, the attack will have to generate a compound eviction set that evicts both victim array (memory that leaks secret) and `ptr` (address of probe array).
iii. **Inject Inductive Patterns**: The attacker injects specific data access patterns to make the DMP start prefetching data from the target memory region.
iv. **Evicting Target**: Attacker will keep evicting the target array and `ptr` with evictset generated in ii. The objective is to ensure that the DMP only activates under specific situation, which is explained in detail at Appendix A.
v. **Capture Leaked Data**: By monitoring the access time of probe array `ptr`, attacker can identify the activation of DMP so that it can capture the side channel signals to deduce the secret.
vi. **Analysis**: Attacker will finally analyze the leakage data and recover the secret through statistics, i.e., set a threshold by the latency with peak frequency, shown in Section 6.2

## 4 VISUALIZATION

With sufficient understanding of the GoFetch attack, we develop an animated website for education purpose where the audience receives a quick understanding of the attack by interacting with the pages. In this section, the design principles and expected effects of our website are comprehensively explained.

## 4.1 Design Principles

The website is mainly intended for the public, rather than the professional. The webpage starts with a introduction showing up. As in Figure 3, the audience will at once read the digest of GoFetch attack. If a term confuses the audience, he can naturally click the term before the brief explanation box pops up. In case that the audience ignores the clickable term, the webpage highlights three terms with large blocks in background knowledge section.



Welcome to the world of GoFetch!

What is GoFetch?

**GoFetch** is a family of Side-Channel Attacks on recent Apple silicon CPUs that exploits the CPU's on-chip data memory-dependent prefetcher (DMP) and Cache to investigate the contents of memory. CPUs affected include the M1, M2, M3 and A14 series system-on-a-chip processors.

Basic Knowledges

This webpage aims to offer viewers a comprehensive introduction to the working mechanisms of GoFetch. Before diving into detailed explanations and simulations, please make sure you have a basic understanding of the following concepts:

If you are unfamiliar with any of the terms, please click on the term and explore the underlying principles and mechanisms.

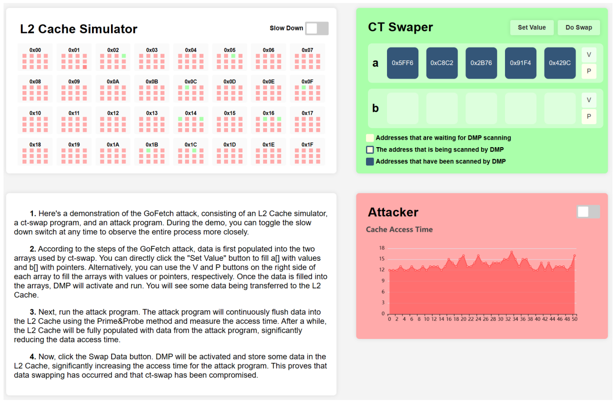
If you are ready, **LET's GET START!!!**

**Figure 3: Homepage of GoFetch introduction website**

The website provides a detailed exploration of these three terms. Next, we put a explanation of GoFetch attack on constant-time swap in simple language, after which the audience will enter the interactive part of our website.

## 4.2 Animation and Interaction

Understanding the text might be tricky for some of the audience. Therefore, we build three interactive webpages with animation for demo of Cache, Side-Channel and GoFetch.



Addresses that are waiting for DMP scanning
The address that is being scanned by DMP
Addresses that have been scanned by DMP

**1.** Here's a demonstration of the GoFetch attack, consisting of an L2 Cache simulator, a ct-swap program, and an attack program. During the demo, you can toggle the slow down switch at any time to observe the entire process more closely.

**2.** According to the steps of the GoFetch attack, data is first populated into the two arrays used by ct-swap. You can directly click the "Set Value" button to fill a[] with values and b[] with pointers. Alternatively, you can use the V and P buttons on the right side of each array to fill the arrays with values or pointers, respectively. Once the data is filled into the arrays, DMP will activate and run. You will see some data being transferred to the L2 Cache.

**3.** Next, run the attack program. The attack program will continuously flush data into the L2 Cache using the Prime&Probe method and measure the access time. After a while, the L2 Cache will be fully populated with data from the attack program, significantly reducing the data access time.

**4.** Now, click the Swap Data button. DMP will be activated and store some data in the L2 Cache, significantly increasing the access time for the attack program. This proves that data swapping has occurred and that ct-swap has been compromised.

**Figure 4: Interactive demo of GoFetch attack on ct-swap**

Figure 4 shows the interface of GoFetch simulation where the audience can act like an attacker and victim at the same time. The items display the micro view of L2 cache, constant-time swap victim and GoFetch attacker. We highlight victim with green and attacker with red in their items and L2 cache. Once attacker initializes its attack, the audience could immediately be notified by the changes in cache and attacker's timing diagram. To simulate the attack, the audience will perform the victim's swap action, after which the changes in attacker's eviction begin to display.

Additionally, we also develop the simulation for the background knowledge about Cache and Side-Channel. For cache simulation, the audience could act as an user to fetch data, and then see how it's stored in cache. For side channel simulation, we borrow an interesting scenario that the theft breaks the lock by sound emitted in trial of unlock. The audience under guidance of slight difference in sound could guess the password effectively.

## 5 MITIGATION

In this section, we introduce the implementation of defense methods against GoFetch attack. In next section, we analyze the trade-off of these defenses, providing information for selection of them.

### 5.1 Disabling DMP

The intuitive idea is simple and straight: disable the flaw opponent, DMP. For now as the architectural flaw has not been completely patched, we can only disable DMP in trade for a better security when performing security-critical tasks, temporarily. This could be done by setting the DIT bit through syscall of macOS. A famous hacker Hector Martin found that Apple has already done so, by only allowing DMP work at user mode in M2/M3 processors [10]. Surely, disabling DMP will cost a slowdown in performance and raise in cache miss rate. We measured the "security tax" to find out how much the, whose results are compared in Section 6.2

### 5.2 Using Efficiency Core

As Augury reads [14], the DMP is only available on efficiency core (Icestorm). Thereby, it could be a short-term solution for latency-insensitive applications. macOS provides system interfaces for manual setting of scheduler, like binding program to specific core using `CPU_SET()`. However, no one knows if Apple will enable DMP on efficiency cores someday, and by then, nothing could be done to prevent GoFetch unless Apple had a better solution.

To give a more intuitive impression on how much slower it would cause, we also measure the time consumption on a typical algorithm. To magnify the slowdown caused by accessing cache/memory, we use sorting algorithm that takes a relatively high portion in accessing data, as the experiment sample.

### 5.3 Blinding

Blinding is a generalized name for the optimized solution which shared the same characteristics. It obfuscates the secrets in side-channel signals by adding noises to either the secret or the pointers (specially for DMP). The obfuscation mostly uses masking techniques like exclusive-or and symmetric cryptography. Note that we have to maintain the normal execution and ensure the correctness of program when blinding.

To fight against DMP-related attack, the idea is to disable the pointers. For instance, in constant-time swap, we can add a mask to the inputs to hide the pointer, as in Listing 1

Without any pointers present in arrays, the DMP never gets activation and attacker cannot effectively guess the value of `mask` either. Attacker fail to distinguish and DMP activation and the secret bit. Another binding for Constant-time Swap is that, as masking always produce `secret=0` for attacker since DMP never activates, we can also try to making `secret=1` by swapping a and b at the beginning and again at the end to blind attacker.

Similarly in RSA decryption, we can apply a mathematical mask rather than bitwise xor to ensure the correct decryption result.

```
void ct-swap(uint64_t secret, uint64_t *a, uint64_t *b,
        size_t len) {
    uint64_t tmp_a[len+1], tmp_b[len+1];
    uint64_t mask = (uint64_t) rand();
    for (size_t i = 0; i < len; i++) {
        tmp_a[i] = a[i] ^ mask; // masking
        tmp_b[i] = b[i] ^ mask;
    }

    // swap the temp arrays in constant time

    for (size_t i = 0; i < len; i++) {
        a[i] = tmp_a[i] ^ mask; // unmasking
        b[i] = tmp_b[i] ^ mask;
    }
}
```

**Listing 1: Masking Pointers in CT-Swap**

```
void rsa-decrypt(uint64_t ...) {
    uint64_t mask = (uint64_t) rand();
    uint64_t inv = pow(mask, -1, phi(N)); // masking
    cipher = pow(cipher, mask, N);

    // normal RSA decryption

    plain = pow(plain, inv, N); // unmasking
}
```

**Listing 2: Masking ciphertext using Math**

Listing 2 is barely a scratch, as CRT decryption acceleration can be adapted here for the masking and unmasking, too. However, it does double the time consumption of decryption, and possibly makes the CRT acceleration meaningless.

### 5.4 Detection

As mentioned in previous sections, all methods that can defence GoFetch attack will reduce the performance. To improve, an intuitive idea is to detect the possible existence of attacker, after which the compromised computer could protect the secret in victim from being stolen. Researchers found that most of the cache SCA detection techniques are based on Hardware Performance Counter (HPC) [1].

The GoFetch attacker need to build eviction set to evict the data of victim out from the cache and record time with precision of nanoseconds[4]. The characteristic of GoFetch attacker shown in Section 3.2 inspires us with a clear instruction to locate it. Similar to techniques in previous work [1, 2, 5, 15, 16], we come up multiple methods to detect the attack on DMP. List below is the methods that we try to adapt in the detection. Notice that we use the constant-swap as victim example, which can be generalized with little modification.

*5.4.1 Static Detection of Binary File.* The first idea we come up with is to scan the binary file to find out whether it frequently invokes dangerous syscalls like `kpc_get_counter_count()` to use high-precision timer and/or request something in memory again and again in a short time period. However, both of them seems unreliable due to the limitation of attacker's access privilege.

It is trivial to find the frequent load/store instructions. The problem is, most of the harmless programs will also access memory frequently. For heavy load applications like video games, the memory usage can be far more frequent, even than the attacker needs. As a result, a mere detection of memory instructions can't help to distinguish attacker and folks.

*5.4.2 Dynamic Detection of Pointer Array.* Since it's impossible for us to detect GoFetch attack only by statistic detection, we turn to the dynamic detection to examine the inputs when the attacker behaves. One of the requirements to activate DMP is that the data present in cache lines looks like pointers. Under most situation, these pointer-like data need to be injected by GoFetch attacker. It seems possible to detect attacking through detect the parameter passing and search for arrays containing continuous pointer-like data.

This method seems better than above, but we met difficulties when trying to monitor the communication between progresses. To resolve this in interprocess communication, we create a isolated environment with the suspect process in it and attach a pipe monitor to suspect using system tools. For TCP communication using loopback, we can simply cover the listening port of vulnerable victim with proxy and expose the proxy port.

Another identified obstacle forbids us to distinguish the malicious process and normal process, since a normal process could've contained arrays of pointers. Therefore, the only way out is combining the detection of pointer array along with detection of malicious priming behaviour, which leads to the section.

*5.4.3 Dynamic Detection of Priming.* As the threat model suggests, GoFetch attack takes effect only under victim process is running. This means that we can choose to attach the detector on victim to detect possible attacker, instead of building a daemon that keeps monitoring all processes. Prime&Probe attack can be detected by measuring conflict miss rates, which caused a much longer time to read/write cache in the victim process.

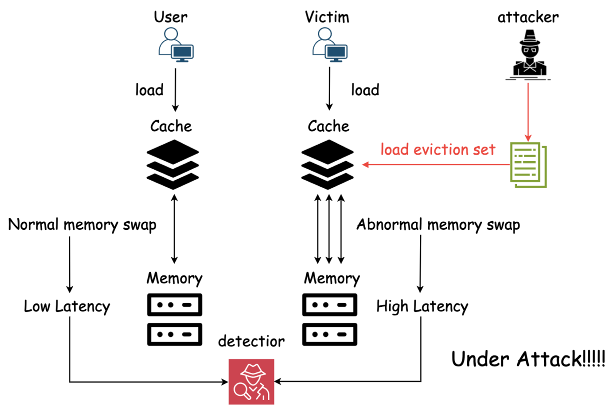


**Figure 5: GoFetch attack detection model**

In our model, when the progress begin, it record the time of access an array in cache without the eviction from attacker as the threshold. After that, before and when constant-time swap performs, detector measures access time of the array in charge. We shall note that for security concern, a fake swap might be done before actual swap. If no attacker is active, the chance of the array missing in cache is trivial, while it will almost immediately swapped out if there's an attacker evict it. So the timing difference again

For implementation, we integrate the detector process into the victim process as its own guard. Since HPC requires the kernel privilege, we use similar approach from the attacker and launch a independent thread of timer. Next, we keep monitoring the access time of victim's arrays and count the number of cache miss. We consider it abnormal for large amount of cache miss since constant-swap will firstly access the arrays to obtain `delta` in Appendix. Through fine-tune, we find a optimized threshold for the abnormal outcomes.

## 6 EVALUATION

In this section, we will present thoroughly the result of our experiments on reproducing the attack and effectively defending against it. Next, we're going to analyze the data to demonstrate the actual meaning of it in proof of the success in attack and defend.

### 6.1 Environment Setup

The experiment is done under the following environment

| | |
|---|---|
| SoC | Apple M1 8G Unified Memory |
| OS | macOS Sonoma 14.5 (23F79) |
| Model | MacBook Air (A2337) |

To ensure that DMP is genuinely available in the environment, we also reproduce the reverse engineering of DMP as described in Augury and GoFetch and the result matches that present in previous work[4, 14]. Meanwhile, we found a way to disable DMP[10] which distinctly shows the access-time difference between with and without DMP, supported by the real latency of 335 cycles and 585 cycles at average, respectively.

### 6.2 Latency Analysis in Attack

For the attack discussed in Section 3, we implement a attacker with techniques to identify victim address space, generate eviction set and monitor the latency in accessing the probe array. Before attacking, the process will ensure the functionality of eviction sets. We conducted 10 experiments for the two attacks with 8 of them success for constant-time swap, and the failure is discussed in Appendix A. During the experiment, we measured the actual latency in accessing the probe array `data`. The result is shown in Figure 6

The attack on constant-time swap and RSA-2048 has shown similar latency diagram. As we can see from Figure 6, the threshold between `secret=0` and `secret=1` lies around 700 cycles. For the key recovery in RSA-2048, the result analysis shows in Table

### 6.3 Performance Loss for Defense

To estimate the performance loss of hardware defenses in Section 5.1 and Section 5.2, we conducted 100 experiments to record the

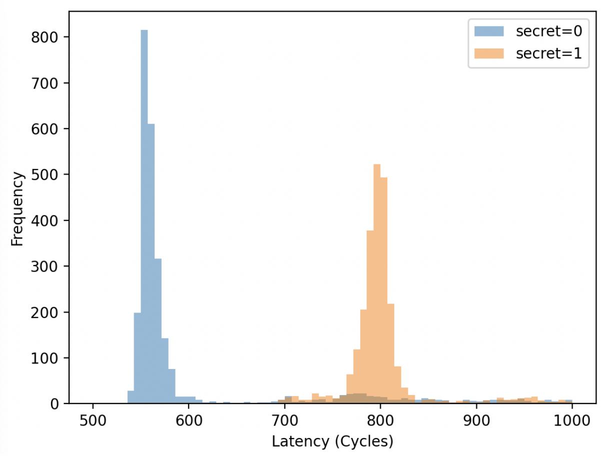


**Figure 6: Latency difference in Prime&Probe**

**Table 1: Success rate RSA key recovery of $p$**

| Key Recovery Rate | Success Rate | Avg. Running Time |
|---|---|---|
| 512 of 1024 bits | 99.8% | 53 min |
| 560 of 1024 bits | 90.7% | 72 min |

execution time with defense and without defense, with each experiment running the same algorithm for 100,000 times. The overall average result in shown in Table 2

| | DMP Enabled | DMP Disabled | Efficiency Core |
|---|---|---|---|
| Time | 81.9 us | 82.3 us | 378.9 us |
| Miss Rate | 0.09% | 0.83% | 2.17% |
| Slowdown | 0 | 1.46% | 362.64% |

**Table 2: Performance with DMP enabled/disabled, and efficiency core**

The slowdown of disabling DMP seems acceptable. With practice in designing processors, we consider the result normal, largely for the experience that improvement in cache hit rate eventually performs less satisfyingly than improvement in cache capacity or cache access latency. In contrast, the chips that cannot disable DMP will have to transfer to a slower core, resulting in a nearly 4 times slower overhead. As in the specification of experiment machine, the default clock rate is 2.064 GHz, compared with 3.2 GHz in performance core. The increase in cache miss rate also contributes to the slowdown. Nevertheless, all the mitigation is proven to be successful.

We also measure the cost for adding extra instructions to mask the pointers. For this experiment, due to the negligible operation, we repeat the codes for 100,000 times to magnify the execution time and acquire a high precision in record. For Constant-Swap, we test swapping an array of 1,000,000 elements; for RSA, we rewrote the program and used library in C language with the same implementation (mostly the RSA and math-related libraries).

| | CT-Swap | Blind CT-Swap | RSA | Blind RSA |
|---|---|---|---|---|
| Time | 1073 us | 3234 us | 159 us | 1063 us |
| Slowdown | 0 | 201.5% | 0 | 568.6% |
| Defense | fail | 100% | fail | 100% |

**Table 3: Performance loss of Blinding in CT-Swap and RSA**

Defense in software level could be much more lagging than hardware prevention since extra program code increases the time complexity by multiplying a constant value, as shown in Table 3.

## 6.4 Detection Accuracy

We conduct 10 experiment to examine our detector model in various situation with result shown below.

| | Attack | Normal | Attack (Noise) | Swap (Noise) |
|---|---|---|---|---|
| `secret=0` | Y | N | Y/N: 70%/30% | Y/N: 40%/60% |
| `secret=1` | Y | N | Y/N: 80%/20% | Y/N: 40%/60% |

**Table 4: Detection result under specific situation (Y/N for attacker detected or not**

Table 4 shows that our implementation effectively distinguish the attacker and user. However, detector is still under influence of the environment noise. We hope to improve our detection techniques with more consideration on attacker's profile and conduct more experiment to obtain higher success rate in the future.

## 7 CONCLUSION

In this paper, we first explained the principles of the recent GoFetch attack and introduce our education website for its demonstration. By carefully design the webpages on content, layout, animation and interaction, the website can provide a quick and comprehensive understanding of GoFetch attack. We also successfully reproduce the attack to confirm the side-channel leakage, and implement several mitigation methods along with detection of possible GoFetch attacks. For attack, we analyze the latency result to show the leakage. For prevention, we measure and compare the overall performance loss in different methods. For detection, we identify the difficulties in monitoring attacker's behaviour and implement a accurate GoFetch detector with a combination of detection.

## REFERENCES

[1] Ayaz Akram, Maria Mushtaq, Muhammad Khurram Bhatti, Vianney Lapotre, and Guy Gogniat. Meet the sherlock holmes' of side channel leakage: A survey of cache sca detection techniques. *IEEE Access*, 8:70836–70860, 2020.

[2] Mohammad-Mahdi Bazm, Thibaut Sautereau, Marc Lacoste, Mario Südholt, and Jean-Marc Menaud. Cache-Based Side-Channel Attacks Detection through Intel Cache Monitoring Technology and Hardware Performance Counters. In *FMEC 2018 - Third IEEE International Conference on Fog and Mobile Edge Computing*, pages 1–6, Barcelona, Spain, April 2018. IEEE.

[3] Daniel J Bernstein. Cache-timing attacks on aes. *University of Illinois at Chicago*, 152:1–5, 2005.

[4] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W. Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. Gofetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers. In *USENIX Security*, 2024.

[5] Hongyu Fang, Sai Santosh Dayapule, Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Prefetch-guard: Leveraging hardware prefetches to defend against cache timing channels. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 187–190, 2018.
[6] Dan Goodin. Unpatchable vulnerability in apple chip leaks secret encryption keys. *Ars Technica*, 2024.
[7] Yu Jin, Hans-Wolfgang Loidl, and Joe Smith. Timing the transient execution: A new side-channel attack on intel cpus. *arXiv*, 2023.
[8] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *DAC '16: Proceedings of the 53rd Annual Design Automation Conference*. ACM, 6 2016.
[9] Andreas Kogler, Stefan Mangard, and Michael Schwarz. Collide+power: A new side-channel attack. *SecurityWeek*, 2023.
[10] Hector Martin. Found the DMP disable chicken bit, 4 2024.
[11] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.
[12] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, volume 11, 2005.
[13] Sandro Pinto and Cristiano Rodrigues. New side-channel attack on arm trustzone. In *Proceedings of the Black Hat Asia Conference*, 2023.
[14] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. Augury: Using data memory-dependent prefetchers to leak data at rest. *IEEE Symposium on Security and Privacy (SP)*, pages 1491–1505, 2022.
[15] Younis A. Younis, Kashif Kifayat, and Abir Hussain. Preventing and detecting cache side-channel attacks in cloud computing. In *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing*, ICC '17, New York, NY, USA, 2017. Association for Computing Machinery.
[16] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W. Fletcher. Synchronization storage channels (S2C): Timer-less cache Side-Channel attacks on the apple m1 via hardware synchronization instructions. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1973–1990, Anaheim, CA, August 2023. USENIX Association.

## A EXPLOITATION

Section 3 analyzed the threat model and feasibility of GoFetch attack. To apply the attack in practice as a proof-of-concept, we conduct the experiment of two practical attacks. In this section, we firstly briefly introduce the background of the victim software and the techniques to leverage DMP's activation as leakage. Next, we will identify the difficulties in experiment.

### A.1 Attacking Constant Time Swap

Constant Time Programming (CTP) paradigm provides a effective practice to mitigate the Cache SCAs. The basic idea is that the execution time of program cannot be decided by the secret. For instance, Listing 3 shows a typical vulnerable swap function.

```
void vul-swap(uint64_t secret, uint64_t *a, uint64_t *b,
        size_t len) {
    uint64_t xored;
    if (secret) for (size_t i = 0; i < len; i++) {
        xored = a[i] ^ b[i];
        a[i] = a[i] ^ xored;
        b[i] = b[i] ^ xored;
    }
}
```

**Listing 3: Vulnerable Swap Code Snippet**

The code snippet above exposes two vulnerabilities related to SCA. The first is that for a large enough `len`, the execution time of `secret=1` can be clearly distinguished from `secret=0`, as the former is a non-zero number and the latter is nearly zero. While the other leakage is that if attacker performs a classical cache SCA, it can definitely observe the usage of `a` and `b` and thereby guess the secret with a probability close to 1. Listing 4 below shows the implementation of CTP.

```
void ct-swap(uint64_t secret, uint64_t *a, uint64_t *b,
        size_t len) {
    uint64_t delta;
    uint64_t mask = ~(secret-1);
    for (size_t i = 0; i < len; i++) {
        delta = (a[i] ^ b[i]) & mask;
        a[i] = a[i] ^ delta;
        b[i] = b[i] ^ delta;
    }
}
```

**Listing 4: Constant-time Swap Code Snippet**

The constant-time swap always accesses both `a` and `b` array, and iterates through the loop, which mostly eliminates the timing and cache difference. To attack on CTPs with DMP, we have an intuitive idea that to identify the swap action, the pointer must be dereferenced when being swapped. To achieve the assumption, we can carefully deploy the activation criteria and steps in Section 3.2.2. Figure 7 shows the difference in cache triggered by DMP on swap/no swap.

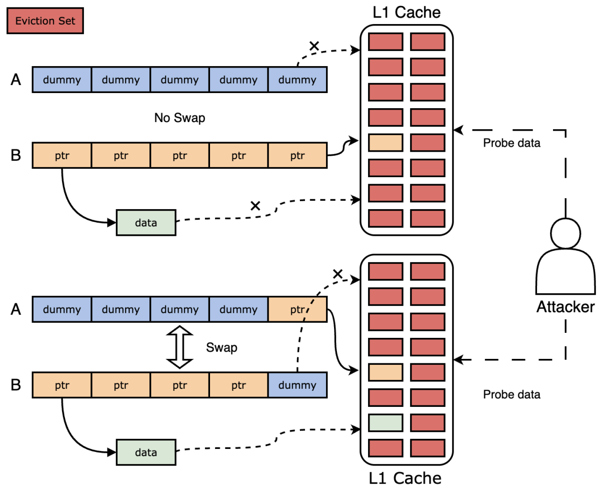


**Figure 7: CT-Swap Attack**

On input `a` with dummy values and `b` with pointers, the compound eviction set keeps evicting `a` and target data in caches. Restricted by the "do-not-scan hint", the `ptr` will never be dereferenced in cache if no swap is done (`secret=0`), and therefore, this indicates a L1 cache miss of target `data`. On the other hand, if a swap is done (`secret=1`), the attacker can observe a low latency in accessing `data`. Note that we can evict `a` and `data` from L1 simply by evicting them from L2, which is mentioned in Section 2. In short summary, the specific condition is that `ptr` only gets dereference by swapping from `b` to `a`.

### A.2 Attacking Go's RSA

The standard RSA encryption would have not been compromised, whose decryption method uses $c^d \bmod n$. However, since the power calculation is computation-consuming, in real-world implementation, the RSA decryption adapts Chinese Remainder Theorem (CRT) to accelerate the calculation. In Go's RSA-2048, in replacement of $c^d \bmod n$, it calculates $m_1 = (c \bmod p)^{D_p} \bmod p$ and $m_2 = (c \bmod q)^{D_q} \bmod q$, where $D_p = d \bmod (p-1)$ and $D_q = d \bmod (q-1)$. And finally $m = m_2 + h \times q$, where $h = (m_1 - m_2)(q^{-1} \bmod p) \bmod p$. Without lose of generosity, we assume that $p \geq q$ and we focus on the attack of $p$. The correctness of equation is not shown here. Note that $p, q$ are 1024 bits.

With this in hand, the vulnerable routine is that if the attacker construct a malicious ciphertext $c$ containing pointers since

- The first step in power is to calculate $t = c \bmod p$
- If $c < p$, then $t$ becomes $c$ and activates DMP.
- Otherwise, $t = c - k * p$ for some $k$ and $t$ has little chance to activate DMP, since $c$ is unlikely to contain pointer.

Next, the attacker can recover $p$ bit by bit starting from the most significant bit, with the input ciphertext for RSA decryption in Figure 8.

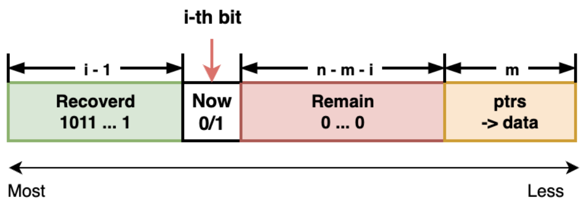


**Figure 8: Malicious input of ciphertext construction**

For the $i$-th bit of ciphertext $c$ with the most $i-1$ bits recovered, we put a random bit into it and feed $c$ to the decryption oracle (victim), and monitor whether the DMP activates. Since the most $i-1$ bits of $c$ are the same as $p$'s, the $i$-th bit immediately implies that $c < p$ or otherwise, with possibility of $1 - 1/(n-m-i) \approx 1$ for $i < n - m$, where $m$ is the total length of pointers. As the $i$ approximates, the probability of success keeps reducing, for which we stop at $i = n - m + 16$ and the $p$ is not fully recovered. Nevertheless, the remaining part will be uncomplicated to be brute forced, just to check $gcd(n, p) = 1$. Thus, the attacker has a non-negligible chance to succeed with the hint from DMP.

### A.3 Identifying the Difficulties

Similar to other cache SCAs, the biggest difficulty lies in the generation of standard eviction set and compound eviction set. As discussed in Section 3, the DMP restricts the search region to be within the same 4GB page, which makes it hard to find the `ptr` and `data` that locates in 4GB page. Meanwhile, for a simulation of real-world attack, the victim's virtual address is transparent to the attacker process. So the other problem would be to identify the victim's address of array `a`. Another possible failure could be that `a` is not sit in page of `data`.

To resolve the problems mentioned above, the GoFetch researchers found that macOS allocates a identical virtual base address to the user process, which can be recovered by a unprivileged process. Once the attacker knows the boundary of virtual addresses, it will be able to generate the eviction set and test the effect.

However, another common obstacle in side-channel attack is also critical, that the environment noise is hard to avoid. For the environment noise detection, the attacker will concurrently measure the access time of eviction set and compare it with target access time.

## B WHY RANDOMIZING ADDRESS NOT WORKING

Due to the allocation policy in macOS/Linux, the virtual address in randomized only during boot time, and macOS allocates stably the pages upon the same base address even with ASLR enabled.

The randomization of address is supposed to be done on page allocation that will affects the ancient implementation of continuous memory management, i.e., the first-fit, next-fit, best-fit and worse-fit. The research to implement the randomized memory allocation is yet done to out best knowledge, and we consider it to be future work of the related research on defense of cache SCAs or other exploitation.