Security Implications of Modern Microarchitecture

Nicholas Altemeier
June 5, 2024

Abstract

Over the past several years, the security research community has discovered and documented a significant number of exploits that are not software-caused, but exist directly within modern computer architectures. These are exploits that exist regardless of a victim's operating system or installed utilities, and are only truly correctable through replacement of the components at fault.

While these attacks vary in their nature and targeted hardware subsystems, one factor connecting them is the control of eviction sets in cache and execution of timing attacks on said cache. When paired with other vulnerabilities, malicious results can be achieved such as compromising modern encryption schemes to leak private keys, reading out-of-bounds memory, or flipping critical bits in memory via electromagnetic interference [8].

This paper will delve into the art of eviction set construction and timing attacks. Two prominent families of hardware exploits will be examined: Meltdown/Spectre, and GoFetch which built off work done by the Augury team. GoFetch in particular targets cryptographic schemes, and a hand-crafted approach is required for each one. We will primarily focus on RSA, along with a detailed explanation of how it is implemented on modern systems. Finally, we will address the broader implications surrounding these exploits and the current trends in microarchitecture.

Cache Hierarchy and Set Associativity

The Random Access Memory (RAM) in a computer can be broken down into two categories. The first is Dynamic RAM (DRAM), where each bit is stored as the presence or absence of charge in a capacitor, with a corresponding transistor to select it. This is a cheap design allowing for a massive density of bits but comes with drawbacks such as charge leakage requiring constant refreshes. In contrast, Static RAM (SRAM) uses a much more robust design in which each bit is a latching circuit consisting of six transistors. SRAM yields greater performance but is more expensive and uses more die space [1]. The compromise that modern computers have made is to use DRAM as the largest and main memory pool, whereas SRAM modules are used as high speed caches which store the values from main memory that have recently been or are likely to be accessed by the CPU.

This forms a sort of memory hierarchy. At the top are the registers which hold values within a CPU core, and just below that is a small Level 1 (L1) cache belonging to a single core. A larger Level 2 (L2) cache is shared between all cores on a processor, while the slower main memory and disk are their own separate modules attached to a computer's motherboard.

Traditionally, cache loads in data in fixed chunks called cache lines, where some substring of the line's address bits would directly map it to a location in the cache. Modern implementations instead divide each level of cache into sets, where each set can hold multiple lines with the same direct mapping [2]. When one line is loaded into a set, another line has to be evicted, which is done according to some variant of the Least Recently Used (LRU) principle. This pattern is referred to as being set-associative.

Attacking Cache

The profound idea about cache that sets the precedent for the rest of this paper is that it speeds up memory accesses to such an extent as to serve as a side-channel through which information can be revealed. If an attacker times a memory access, they will be able to deduce whether or not the corresponding data was previously in cache, because a load from main memory will take roughly two orders of magnitude longer [1, 12]. If an attacker fills in a region of cache and then a victim overwrites some portion of that data, timing analysis will reveal that it happened.

One way of achieving this is through Flush + Reload. Using the CLFLUSH instruction in x86 [3] or the MCR instruction in ARM [4], the attacker simply removes the victim's data from cache and later checks if it was loaded back in. This approach, however, requires that the attacker and victim share the same memory space. A similar tactic, Prime + Probe, is viable even when the attacker and victim occupy different memory spaces, or in high-level languages without direct access to flush instructions. This requires the attacker to find a virtual address which occupies the same location in cache as the victim's.

With set-associative cache, the attacker needs to find not just one line, but enough to fill an entire set such that when the victim performs its access, some of the attacker's data has to be removed. The collection of memory addresses invoked by the attacker to this end is called an eviction set, and is defined by the memory address that the victim loads. If the attacker iterates over the eviction set and observes longer than average latency, then they'll know that the victim made an access.

Timing Memory Accesses

In order to properly time anything, one first needs an accurate clock. In particular, a clock is needed of high enough accuracy to distinguish between a cache hit and miss. A computer's highest frequency timers may require special privileges to access, which an attacker is generally not assumed to have. To get around this, a counting thread can be used, in which a shared integer value is incremented in an endless loop. Since incrementing is an extremely fast operation, this method provides the necessary granularity for distinguishing cache hits from misses [18].

Minimal Eviction Set Construction

Ideally, an attacker wants the smallest eviction set possible, who's size would equal the ways of associativity of the cache. Such a set is called a minimal eviction set. Given enough time, it's possible for an attacker to construct a minimal eviction set under practically any circumstances. All an attacker has to do is create an eviction set of any size, which is bound to occur if they include enough addresses in it. While knowing the details of architecture could allow for more sophisticated methods, these details are often kept obscure by manufacturers [5] and would serve little use in highly containerized environments [6]. However, the attacker simply needs to assemble an eviction set of arbitrary size, which is bound to happen if enough randomly chosen addresses are included. Any non-minimal set can easily be reduced to a minimal one, using either of the following two approaches.

The first approach is the intuitive one. The addresses within the large set are removed one by one, and an eviction test is conducted after each removal. If the test fails, then the address is placed back in the set and if the test succeeds, it is kept out. After iterating over the entire starting set, the only elements remaining must form a minimal eviction set. The time complexity of this process is $\Theta(s^2)$, where s is the size of the initial large set. This method is useful if the ways of associativity value isn't already known.

The second approach is based on work done on threshold group testing [6, 7]. Specifically, it involves the case where a set is inputted into some test with some upper threshold u. If the set has at least u positive elements, then the test passes and otherwise it fails. This exactly matches the scenario of finding an eviction set, with u being the ways of associativity. This value has to already be known but a much improved time complexity of $\Theta(s)$ can be achieved.

First, the large set is divided into $u + 1$ subsets. Suppose that the set had exactly u positive cases. It would then follow that at least one of the subsets created must have no positive cases. If we drop this subset, we obtain a smaller set with at least u positive cases. The same also has to hold true for any initial set with more than u positive cases. Repeating this process necessarily leads to a set with exactly u elements, all of which are positive. This is the minimal eviction set in our case.

This algorithm can be proven to have a time complexity of $\Theta(s)$. It is important to note that while any division of the set made is valid, the optimal choice is to make $u + 1$ subsets as close to equal in size as possible. This means that our set is shrinking at a roughly constant factor of $r = u / (u + 1)$. To shrink down to a size of u, it would take on the order of *log s* steps. Each of these steps requires iterating over the set, which gets smaller with each step. The total time complexity then resemble:

$$\sum_{i=0}^{T(s)-1} s \cdot r^{i}$$

where $T(s)$ is the number of steps required to produce a minimal set. Using the geometric series formula, this can be converted into a closed-form expression.

$$s \cdot \left(\frac{1 - r^{T(s)}}{1 - r}\right)$$

$T(s)$ has to continually grow with s, and so in the asymptotic case the complexity simplifies down to $\Theta(s)$.

Meltdown

Meltdown and Spectre are exploits which both target speculation in a processor. Both of these attacks rely on the CPU speculatively performing steps to leak data through a side-channel. Both allow the attacker and victim to share the same virtual memory space, and so Flush + Reload can be used for side-channel observation [10, 18]. Meltdown can primarily be attributed to a flaw inherent to Intel's architecture, whereas Spectre is more generalized, and can be executed on AMD and some RISC machines as well. Meltdown is particularly dangerous because it can be used to read arbitrary physical memory, including the kernel space. This would require the attacker to know their own virtual memory offset as well as the randomized offset assigned to the kernel or other privileged regions [9].

The specific blunder which makes meltdown possible on Intel systems is that permission checking of instructions does not occur until late in Intel's execution pipeline [12]. The check prevents the instruction from being committed, but the step of loading data into cache is allowed to progress unimpeded. An exception is eventually raised, causing the guilty process to crash if not handled. Before the exception is raised, the process has to perform a read on an out-of-bounds memory region, convert the read value into a user-space address, and load the corresponding memory into cache. The victim can then observe a low latency when accessing this address, confirming its correspondence to the secret data.

Spectre

Spectre works similarly to Meltdown, but exploits branch prediction in such a way that the illegal operation occurs within a branch misprediction. Since modern branch predictors base their decisions on the outcomes of previous branches, it is possible to train them to predict certain outcomes, even when said outcome is unreachable in normal program flow. The operation within the misprediction cannot be so offensive, as to trigger a system interrupt such as a page fault, so accessing kernel space is ruled out [11]. Still, Spectre can be used to leak the memory contents of a concurrent process in user space, which the attacker wouldn't be able to access in a legitimate context. Additionally, Spectre has more flexibility as to what can be done speculatively. If the branch logic is sufficiently complex, this will cause the outcome of the branch to stall, allowing for more instructions to be speculatively executed.

Spectre can not only leak memory, but also the contents of privileged registers. This is done by testing some condition on a register and loading memory into cache if the result is true. Alternatively, if there is some non constant time operation that can be performed on a privileged register, then this can be done and the corresponding execution time can be measured to obtain insight into the register value.

The Data Memory Prefetcher

We now turn to GoFetch, which targets modern Apple silicon, particularly a component of architecture called the Data Memory-Dependent Prefetcher (DMP). Prefetchers themselves are not a novel idea. Conventional ones recognize memory patterns, such as accessing an array sequentially or in even strides [18], and preemptively load data into cache that is likely to be accessed in the future. This can significantly reduce the latency, but is only really effective if the data being accessed has an array-like structure.

This limitation excludes many of the structures and algorithms seen in modern programming, such as linked lists, trees, graphs, and object-oriented patterns in which objects are accessed through references. What all of these share in common is a process called a pointer chase, in which a single element contains its corresponding data along with one or more pointers containing the memory addresses of successive elements.

This is the particular pattern that Apple's implementation attempts to speed up. Data is prefetched not only according to access patterns but also the contents of memory being accessed. If a particular cache-loaded word resembles a pointer, then it is dereferenced and the corresponding data is also loaded into cache.

Specifics of Apple's Design

Until recently, this prefetcher variant only existed in theory, with the Augury paper in 2022 being the first identification of such a device in production hardware [19]. A full description of the DMP's behavior wasn't realized until 2024 with the release of GoFetch [18]. Due to Apple's secrecy, all information had to be reverse engineered. Through painstaking testing, they were able to establish the following rules:

1. The DMP scans 64-byte lines as they are loaded into L1 cache.
2. A word is defined as a pointer if the difference between its value and its memory address is less than 2^{32} or 4 GiB.
3. A record of the last 128 pointer accesses are kept so that the DMP does not redundantly activate on them.
4. When a line is removed from L1 cache but stays in L2, then it is tagged with a hint telling the DMP not to scan it if it's loaded back into L1.

GoFetch

The DMP can be used to inadvertently leak secrets during a cryptographic process through attacker-controlled inputs. This is done by embedding some address value, *ptr* into a ciphertext before starting the process. If this is done in such a way that *ptr* is only preserved if certain conditions are met, then the attacker can prompt a reload of *ptr* and observe whether the DMP activates to gain said insight. The attacker can confirm that the DMP has activated by timing an iteration over a *ptr* eviction set.

While Augury and previous theoretical work recognized that a DMP implementation could pose some security concerns, GoFetch escalated those concerns drastically by showing examples of modern cryptographic schemes being compromised through DMP-based attacks. These are schemes that have already had large amounts of effort put into making them impervious to side channel observation, with room left for many more schemes to be targeted. Specifically, the GoFetch paper was able to compromise four serious encryption schemes: RSA implemented in the Go SSL library, the Diffie-Hellman Key Exchange implemented in OpenSSL, Kyber key encapsulation, and Dilithium lattice-based encryption. The last two are notable candidates for post-quantum standards [24]. Additionally, the paper targets the conditional array swap which is used as an example of hardening. Since all of these require involved explanations, we will only cover the attacks on the conditional array swap and Go RSA, accompanied by explanations of how these schemes work in a modern, hardened context.

Conditional Array Swap

In this setup, two arrays of equal length are given as input, along with a secret value which will either be 0 or 1. The function has a simple goal: if the secret value is 1, swap the contents of the arrays and if the value is 0 don't. The intuitive approach with a conditional statement allows the function to return much quicker if the arrays aren't swapped, and so is easy to break. Ideally we want a pattern such that the same steps are taken in the same order regardless of the outcome.

```
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;
    }
}
```

The above pattern [18] cleverly works around this restraint. First *mask* is constructed, such that *secret* being 0 produces a string of zeros (through integer overflow) and 1 produces a string of ones. *delta* is then initialized to either have a value of 0 or *a*[*i*] ^ *b*[*i*]. Based on the properties of bitwise xor, we can deduce that both *a*[*i*] and *b*[*i*] will have their values swapped if *secret* is 1 and will be reassigned the same values otherwise. Notice how the above code is made up of nothing but fixed expressions of *a*[*i*], *b*[*i*], and secret so there is no inherent leakage of information.

GoFetch vs. Array Swap

For this setup, we assume that the attacker can specify the contents of arrays *a* and *b* but can't observe their outcome. The attacker has obtained eviction sets for two addresses: $a + i$ and *ptr*. The attacker sets the values for *a* and b such that *b*[*i*] equals *ptr*. This must be done on the same thread that the victim will use so that *b*[*i*] is loaded into the core-specific L1 cache and has the do-not-scan hint applied to it. Additionally, *ptr* must no longer be in the DMP's history table. Before starting the swap, the attacker iterates over the eviction set for *ptr*.

Within L1 cache, the value *a*[*i*] is part of a 64-byte cache line which will also contain some later value $a[i + k]$. Using a parallel thread, the attacker needs to invalidate this line between the victim accessing $a[i]$ and $a[i + k]$. If this is achieved, then the modified value of $a[i]$ is reloaded into L1 cache. If *secret* has a value of 1, $a[i]$ and $b[i]$ will have swapped and $a[i]$ will have the value of *ptr*. This will then be scanned and dereferenced by the DMP, leaking the value of *secret*.

Note on Modular Notation

The following two sections will constitute a deep dive into the theory behind the Go RSA implementation, expressed in the language of modular arithmetic. In this system, it is important to distinguish between two different types of equality. The first refers to absolute equality or, within the context of a program, assignment. This is expressed by the = symbol. In such statements, *mod n* is an explicit step of modular reduction, equivalent in most programming languages to the % operator. Whereas the symbol ≡ is used to express congruence under (*mod n*), i.e. the two sides fall into the same residue class. (*mod n*) is the context of the congruence, not a part of the calculation, and so the parentheses distinguish it as such.

Explanation of RSA

Rivest–Shamir–Adleman (RSA) is one of the oldest and most prominent implementations of public key encryption. Both the encryption and decryption steps are simple exponentiations:

$$c = m^e \bmod N$$
$$m = c^d \bmod N$$

where m is a plaintext, c is a ciphertext, e is the public key, d is the private key, and N is the product of two large primes, p and q. The scheme works because d is set up to be the modular inverse of e (i.e. the product of e and d is equivalent to 1) under $(mod\ \varphi(N))$. φ in this context refers to Euler's Totient Function, a function which counts the number of integers below N that are coprime to N. Using this fact, we can prove that the decryption step works:

$$c^d \equiv (m^e)^d \equiv m^{e \cdot d} \equiv m^{1 + k\,\varphi(N)}$$
$$\equiv m \cdot (m^{\varphi(N)})^k \equiv m\ (mod\ N)$$

The final step is an invocation of Euler's Theorem, which states that for any two coprime numbers, a and n:

$$a^{\varphi(n)} \equiv 1\ (mod\ n)$$

Note that the plaintext m will only be recoverable if its initial value was less than the smaller of p and q. To get around this, the plaintext is broken into smaller blocks and then padded according to some agreed upon padding scheme.

Modern RSA

While RSA in principle operates according to simple mathematical steps, the practical reality of performing it on a computer differs greatly from the ideal. The implementation of interest for this paper works on blocks with lengths of 2048 bits, or 32 words on a modern processor. With such a block size, simple arithmetic operations such as multiplication or division correspond to many machine instructions having to be performed on each of these words, which poses a barrier to execution speed. The biggest offender in this regard is modular reduction, requiring division by values which themselves are compositions of many words. The Go implementation adopts two strategies to lessen the severity of the problem, those being based on the Chinese Remainder Theorem (CRT) and Montgomery Multiplication.

Firstly, the large exponentiation and modulus of N can be converted into two separate calculations using *mod p* and *mod q* respectively [22]. This revolves around the central idea of the

CRT, that given a set of modular congruences of the form

$$x \equiv a_i \ (mod\ n_i)$$

in which all values of n_i are mutually coprime with each other, a unique mapping exists between the set and a larger congruence of the form

$$x \equiv A \ (mod\ \Pi\ n_i)$$

The CRT is notable in the context of RSA because one of its implications is that the decryption step can be uniquely broken down into two smaller exponentiations:

$$m_1 = c^{d_p}\ mod\ p$$

$$m_2 = c^{q}\ mod\ q$$

Where d_p and d_q are reductions of the private key d modulo (p - 1) and (q - 1) respectively. m can be recovered from the partial solutions m_1 and m_2. The savings from performing smaller exponentiations more than makes up for the cost of performing two of them.

Secondly, the number of necessary reductions using a difficult modulus such as p or q can be minimized through Montgomery Multiplication [21]. This process converts such reductions into ones using a different modulus, R, of our choosing. In choosing a value for R given the initial modulus n, there are only two requirements: R is greater than n and R is coprime to n. If R is chosen to be a power of 2, then the reduction step can be implemented through fast bit manipulation patterns.

First, some value a is converted into its Montgomery form, which is described by the equation

$$a_M = (a \cdot R)\ mod\ N$$

From this, we can observe the result of multiplying two variables in Montgomery form:

$$a_M \cdot b_M \equiv (a \cdot b)_M \cdot R\ (mod\ N)$$

Thus in order to retrieve the Montgomery form of $a \cdot b$, we must remove the extra factor of R and reduce modulo N. There is a way of doing this such that neither multiplication by the inverse of R nor full modular reduction are necessary. This is done via an algorithm known as REDC. The algorithm works on an inputted value x strictly less than $R \cdot N$, which will be important for the final step. Before starting, we precompute a value N' such that

$$N \cdot N' \equiv -1\ (mod\ R)$$

Now for each pass of the algorithm, we assign variables u and v as follows:

$$u = (x \cdot N')\ mod\ R$$

$$v = x + u \cdot N$$

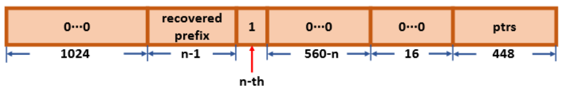
Because v is calculated by adding a multiple of N to x, we can say that it is congruent to x under ($mod\ N$). However, the math works out such that the result is also congruent to 0 under ($mod\ R$), meaning it's divisible by R. Thus we can remove the factor of R without multiplying by its inverse. With R as a power of 2, this division is a much faster operation. Finally, we can infer that the value of $v\ /\ R$ must be less than $2N$ because of the restraint applied to x, and so the final reduction step can be replaced with a conditional subtraction.

Running REDC($a_M \cdot b_M$) gives the result $(a \cdot b)_M$ without performing a modulus step. In fact, any number of multiplications can be strung together, with a modulus only being required to convert a result out of Montgomery form. Additionally, Montgomery multiplication blinds all operands in the face of side-channels. Long arithmetic functions tend to have widely varying execution times for different inputs, and it's possible for an input to be reverse engineered through measuring these variances [14, 15]. Conversion into Montgomery form makes it so that the original input values cannot be reconstructed from timing data.

GoFetch vs. RSA

Instead of deducing the private key d as would traditionally be done, GoFetch allows for one of the secret primes p or q to be leaked through selecting ciphertexts and observing DMP activations. Because RSA via the CRT involves working under *mod* p and *mod* q instead of *mod* N, there is a sense in which these secret values are more readily exposed to an attacker [x]. In this attack, the smaller of the two, which we'll assume is p. Because the block size is 2048 bits, we know that N must have an equal or smaller length. Since N is the product of values p and q where $p < q$, then we know that p can be no longer than 1024 bits. The step that is targeted is an initial reduction of the ciphertext c under *mod* p. We know that if c happens to be smaller than p, then this step leaves it unchanged. By selecting a value of c with one or more pointers embedded in it, we can evict the memory containing them after the step completes and then observe whether or not the DMP activates when that memory is recached. If we observe an activation, then that must mean the pointers were preserved through the reduction and thus $c < p$. This observation leads to a scheme in which an adversary uncovers the value of p bit by bit starting from the most significant one.

The number of pointers that the GoFetch team decided on was seven, in order to allow for a significant difference between results during the prime and probe analysis while accounting for noise from background activity [18]. These pointers occupy the 448 least significant bits of every chosen ciphertext. Additionally, the top 1024 bits are all set to 0.



Starting from the most significant bit of the lower half of the block, the position to be leaked is set to 1. All bits of p that have been found are matched to their corresponding positions in c, and all other bits excluding the pointers are set to 0. Upon observing a DMP activation, we can be reasonably certain that the secret p has a 0 in this position. The only way for a false positive to occur is if p matches our ciphertext exactly, except in the lowest 448 bits in which its value is less than that of the concatenated pointers. The upper bound on this probability would be $2^{n - 576}$ corresponding to the n-th bit being leaked. Based on the above formula, a consensus was reached that around 560 bits of p can be reliably recovered in this fashion.

The remaining 464 bits can be recovered using Coppersmith's attack [18, 27], which can be employed so long as the public key e is less than the square root of N (65537 is the default e value among Go and many other implementations), and so long as at least a quarter of either the most significant or least significant bits of p are already known.

Final Remarks

Meltdown/Spectre has been known about for several years now, and various mitigative efforts have been made against it. These have been fixes at the software/OS level, which need to be performed on a case-by-case basis. Legacy software and retired operating systems will never be protected against these attacks because there is no incentive to do so. For a more permanent hardware solution, it would be useful to revert edits to cache in the case of a faulty instruction or mis-speculation.

GoFetch is a much more recent discovery, so its fallout is still underway. The fact that it compromises hardened/blinded schemes does not leave much room for effective mitigations on the software end, aside from the frequent replacement of private keys. On certain Mac models, it's possible to disable the DMP entirely, but this comes with a major degradation in performance. It will likely be the case that future iterations of Apple silicon will allow selective disabling of the DMP when a sensitive process such as a decryption runs.

The general trend that can be observed from the above examples is that performance tends to have an inverse relationship with security. In an attempt to continue growth in the face of dwindling returns from Moore's Law, computer architects will likely pursue many more schemes to enhance performance, not just in the CPU, but also in the GPU as well as entirely new niche components, such as the emerging DPU and TPU architectures. What exploits may be possible in these exotic new components are yet to be seen. However, such exploits are bound to exist because capitalistic pressures ensure that growth in performance is continual. The most pertinent conclusion that I believe can be drawn from the above examples is that there will always be a need for security insights, and thus job stability for cybersecurity analysts will continue to be assured for the foreseeable future.

1Works Cited

[1] J. Ogawa. (1998). “DRAM Design Overview.” Stanford University. [Slide Deck.] Available: http://www.graphics.stanford.edu/courses/cs448a-01-fall/lectures/dram/dram.2up.pdf

[2] H. Perkins. (2011.) “Computer Organization and Assembly Programming.” University of Washington. [Slide Deck.] Available: https://courses.cs.washington.edu/courses/cse378/11wi/lectures/lec18.pdf

[3] F. Cloutier. (2024.) “x86 and amd64 instruction reference.” felixcloutier.com. [Online.] Available: https://www.felixcloutier.com/x86/

[4] “ARM940T Technical Reference Manual.” arm Developer. [PDF]. Available: https://documentation-service.arm.com/static/5e8e1e2d88295d1e18d369c7?token=

[5] G. Irazoqui. (2015.) “Systematic Reverse Engineering of Cache Slice Selection in Intel Processors.” IEEE. [PDF.] Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7302337

[6] P. Vila et al. (2018.) “Theory and Practice of Finding Eviction Sets.” IEEE. [PDF.] Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8835261

[7] P. Damaschke. (2006.) “Threshold Group Testing.” Chalmers University. [PDF.] Available: https://www.cse.chalmers.se/~ptr/thrgt.pdf

[8] O. Mutlu and J. S. Kim. (2019.) “RowHammer: A Retrospective.” IEEE. [PDF.] Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8708249

[9] Y. Jang et al. (2016.) “Breaking Kernel Address Space Layout Randomization with Intel TSX.” ACM. [PDF.] Available: https://dl.acm.org/doi/pdf/10.1145/2976749.2978321

[10] M. Lipp et al. (2018.) “Meltdown: Reading Kernel Memory from User Space.” meltdownattack.com. [PDF.] Available: https://meltdownattack.com/meltdown.pdf

[11] P. Kocher et al. (2018.) “Spectre: Exploiting Speculative Execution.” meltdownattack.com. [PDF.] Available: https://spectreattack.com/spectre.pdf

[12] N. Abu-Ghazaleh et al. (2019.) “How the Spectre and Meltdown Hacks Really Worked.” IEEE. [PDF.] Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8651934

[13] C. Matthews et al. (2004-2024). “Montgomery modular multiplication.” Wikipedia. [Online.] Available: https://en.wikipedia.org/wiki/Montgomery_modular_multiplication

[14] A. Boldt et al. (2001-2024). “RSA (cryptosystem).” Wikipedia. [Online.] Available: https://en.wikipedia.org/wiki/RSA_(cryptosystem)

[15] D. Brumley and D. Boneh. (2003). “Remote Timing Attacks are Practical.” Stanford University. [PDF.] Available: http://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf

[16] J. Keating. (2023). “How does RSA Cryptography work?” YouTube. [Video.] Available: https://www.youtube.com/watch?v=qph77bTKJTM

[17] C. Muratori. (2024). “The Apple M-Series GoFetch Attack.” YouTube. [Video.] Available: https://www.youtube.com/watch?v=uZEBkOrfUzM

[18] B. Chen et al. (2024.) “GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers.” gofetch.fail. [PDF.] Available: https://gofetch.fail/files/gofetch.pdf

[19] J. R. S. Vicarte et al. (2022.) “Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest.” prefetchers.info. [PDF.] Available: https://www.prefetchers.info/augury.pdf

[20] J. R. S. Vicarte et al. (2021.) “Opening Pandora’s Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data.” IEEE. [PDF.] Available: https://ieeexplore.ieee.org/abstract/document/9499823

[21] P. Montgomery. (1985.) “Modular Multiplication Without Trial Division.” AMS. [PDF.] https://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf

[22] R. L. Rivest. (1978.) “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.” archive.org. [PDF.] Available: https://web.archive.org/web/20230127011251/http://people.csail.mit.edu/rivest/Rsapaper.pdf

[23] D. Boneh. (1998.) “Twenty Years of Attacks on the RSA Cryptosystem.” Stanford University. [PDF]. Available: https://crypto.stanford.edu/~dabo/pubs/papers/RSA-survey.pdf

[24] G. Alagic et al. (2022.) “Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process.” NIST. [PDF.] Available: https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413.pdf

[25] P. Schwabe. (2020.) “Kyber.” pq-crystals.org. [Online.] Available: https://pq-crystals.org/kyber/

[26] P. Schwabe. (2021.) “Dilithium.” pq-crystals.org. [Online.] Available: https://pq-crystals.org/dilithium/

[27] D. Coppersmith. (1996.) “Finding a Small Root of a Univariate Modular Equation.” Springer. [PDF.] Available: https://
link.springer.com/content/pdf/10.1007/3-540-68339-9_14.pdf