# — D1.1 State of the Art: Timing and Data Flow Side Channels

## Definition of relevant sources of side-channel leakage

| | |
|---|---|
| Version: | 1.01 |
| Publication date: | 16.06.2023 |
| Authors: | David Oswald (KAOS) |
| | Timo Kasper (KAOS) |
| | Sebastian Reiter (FZI) |
| | Julian Ganz (FZI) |
| | Anton Paule (FZI) |
| | Simon Wegener (AbsInt) |
| Contact: | David Oswald <info@kasper-oswald.de> |

GEFÖRDERT VOM

Bundesministerium
für Bildung
und Forschung

# – Table of contents

# 1 Introduction

Security gaps in systems of the Internet of Things (IoT) can cause considerable damage in both private and industrial applications, thus hindering broad market success to date. The considerable economic benefits of comprehensive networking, such as increases in convenience or efficiency due to the availability of networked data in edge devices, are negated by security gaps; the promised benefits can very quickly manifest themselves as damage.

One notable class of security vulnerabilities are so-called side-channel attacks. In these attacks, additional information, usually physical effects, are emitted in addition to the intended information flow.

> For side-channel attacks, insights are gained from observable physical effects during the processing of sensitive data. Observable effects are for example runtime behavior, power consumption, electromagnetic radiation and cache behavior.[1]

Side-channel attacks use a wide range of physical effects. As a result, side-channel attacks cover a wide range of attack vectors. These effects include, for example

- Cache behavior
- Computing or response times
- Electromagnetic effects
- Acoustic effects
- Power effects
- Radiation of heat

This report takes an in-depth look at the class of computing time and response time attacks.  Both attacks on individual IoT systems and on a network of IoT systems (system of systems) are considered. This includes, for example, the computing time of individual routines but also, for example, the delay between communication packets. In the context of this document, the focus is on timing side-channels, which cause a leak of information that, e.g., can be used in an attack. Generally, a timing side-channel can also be used for covert information exchange (covert timing channel) and for network flow watermarking. Timing side-channels are mostly used to attack confidential information such as keys, as it requires little information to be leaked in order to compromise the security architecture, thereby enabling the extraction of further information.

[1] https://www.bsi.bund.de/EN/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/Kryptografie/Seitenkanalresistenz/seitenkanalresistenz_node.html

# 2 Timing side-channel leakage sources

One approach to classify timing side-channels is the effective range of a side-channel. In [1], the side-channels are classified as remote timing side-channels and internal timing side-channels. This is particularly important for determining potential attacker models. However, it is also possible to make system-internal timing side-channels remotely accessible by chaining attacks (attack vector), e.g., by planting a trojan in advance.

In this report, potential causes of timing side-channels are examined. Therefore, the classification is made with regard to their cause. A distinction is made between data-flow-based, control-flow-based, and instruction- or microarchitecture-based side-channels. Because in literature side-channels are mostly presented and classified according to their manifested effect (cf. previous section), there are ambiguities when assigning them to causes. For example, a mainly control flow-based side-channel can additionally be influenced by microarchitecture effects.

## 2.1 Data flow

The flow of secret data to locations accessible to an adversary is a recurrent security problem. This leakage does not necessarily manifest through side-channels such as the execution time, but often more directly, e.g., by secret data being copied to an output buffer or similar.

**Example:** The infamous Heartbleed vulnerability in OpenSSL [2] resulted in memory contents of the webserver being leaked in TLS heartbeat packets. The issue was due to a missing bounds check in the respective library functions. Heartbleed allowed an adversary to leak a variety of secrets, including usernames and passwords, TLS private keys, and other data processed by a TLS server.

Beyond a buffer overflow or the lack of clearing a buffer before transmission, properties of programming languages can also lead to data flow leakage.

**Example:** In [3], it was noted that struct alignment in the C programming language causes padding bytes to be included. Per the language specification, these padding bytes are undefined, i.e., there is no requirement for the compiler to initialize padding to a fixed value. Thus, previous (stack) memory contents can leak when the whole struct (including padding) is revealed, e.g., by writing to a file or the network.

Such issues can also manifest in other subtle forms. For example, secret material might be processed in some form and then used in other parts of a protocol. While this might be benign, e.g., when a secret key is used as part of a secure cryptographic protocol, serious issues arise in other cases:

**Example:** In a widely used locking system, an intermediate value from a cryptographic construction was used as the random nonce in the next protocol execution [4]. This enabled a trivial mathematical attack that, given a few such nonces, allowed the creation of a “master key”.

Finally, data flow issues that can be in some cases detected and mitigated at program code level can also occur due to issues in the CPU’s microarchitecture.

**Example:** The ÆPIC Leak bug [5] in Intel CPUs is based on unaligned reads from a special memory page (the xAPIC MMIO page). In this case, stale data from the cache hierarchy could architecturally manifest in respective reads. Mitigation in the context of SGX enclaves required—until a microcode update was released—to not perform unaligned reads outside the enclave’s own memory range [6].

## 2.2 Control flow

Different control flows in the software usually lead to different observable time behavior. This timing behavior creates a side-channel that provides the attacker with information about the program flow. If this program flow is also dependent on confidential information, a direct conclusion can be drawn about this information.

### 2.2.1 Unbalanced branching dependent on a secret key

**Example:** Kocher et al. demonstrate in [7] an attack factorizing RSA keys and determine Diffie-Hellman exponents by computing the execution times of private keys.

```
S[0] = 1
for (k = 0; k <= w-1; k++)
{
 if(y[k] = 1)
 {
   R[k] = S[k] * y mod n
 } else
 {
   R[k] = S[k];
 }
 S[k+1] = R[k]^2 mod n
}
return R[w-1]
```

Figure 1: Implementation example for RSA encryption from Kocher et al.

Kocher et al. make use of the operation `S[k] * y mod n` having a longer execution time on average than the operation `R[k] = S[k]`, see Figure 1. This creates an unbalanced control flow that depends on the secret key. Comparable side-channels for DES where shown in [8]. Two DES implementations are analyzed against timing attacks and the authors have obtained the Hamming weight of the key used in both implementations. More interestingly, they highlighted the timing variance caused during encryption.

**Example:** RSA uses different optimizations to achieve a shorter runtime. The Chinese Reminder Theorem (CRT) enables faster calculation of the expression `c^d mod N`. Here, the exponent `d` and the modulo operator `N` are decomposed into two factors (`N=pq`). The same expression is calculated separately for the two decompositions. It is then combined using CRT to calculate the original expression. This can increase the decryption speed by a factor of four for RSA [9].

Brumley et al. address optimization using Montgomery arithmetic in [9] and show an unbalanced computation here as well. In Montgomery optimization, the input `x` is converted to Montgomery form `xR mod q`, which allows the posterior modular reductions to be implemented efficiently. In the final reconversion of the result, a check is made to see if the value is greater than the value `q`. If it is, `q` is subtracted once to stay within the range of values. It has been shown that the probability of this extra reduction increases as the input approaches either the `q` or `p` factors. This results in a timing side-channel, which allows conclusions to be drawn about the private key. An similar attack based on Montgomery arithmetic is presented in [10]. Using the same attack principle Brumley et al. present an further timing attack vulnerability in OpenSSL's ladder implementation for curves over binary fields [11].

**Example:** The OpenSSL library provides two multiplication methods. The Karatsuba method, which realizes the multiplication of two large numbers into three small multiplications of small numbers, and the traditional multiplication. OpenSSL uses Karatsuba for numbers of similar word lengths as well as the normal multiplication if the two operands have different word lengths. Since Karatsuba is, on average, faster than normal multiplication.

```
int
memcmp (const void *str1, const void *str2, size_t count)
{
  register const unsigned char *s1 = (const unsigned char*)str1;
  register const unsigned char *s2 = (const unsigned char*)str2;

  while (count-- > 0)
    {
      if (*s1++ != *s2++)
          return s1[-1] < s2[-1] ? -1 : 1;
    }
  return 0;
}
```

Figure 2: memcmp() implementation

**Example**: The function memcmp()[2], which is often used in C, is an example of a fast-failing or early-exit function. Depending on the input data, different execution cycles are performed, see Figure 2. If the comparison is performed with secret information, the secret information can be tested via a sequential brute force attack. This type of timing side-channel often occurs in other comparison operations, e.g., string comparisons.

[2] https://github.com/gcc-mirror/gcc/blob/master/libiberty/memcmp.c (commit 50b009c5daef92bc60fc26fcc4c495e117667387)

## 2.3 Microarchitecture

Due to performance optimizations and other “shortcuts” taken in the microarchitecture of a processor, data-dependent execution timing still can be induced even if the program is free of timing variability due to control flow. In the following, we give examples of recent microarchitectural attacks. The focus is on mainly structural aspects of the microarchitecture. However, at least when observing the behavior of real hardware implementations, an even wider range of timing side-channels can be observed, e.g., caused by manufacturing deviations.

### 2.3.1 Cache-based side-channels

Data and instruction caches are indispensable for the performance of pipelined processors with high clock frequency, where the latency for DRAM accesses is several orders of magnitude longer than a single clock cycle. However, an adversary can deduce based on the latency of a memory access whether a specific location (more precisely “cache line”) was recently used or not by a victim process. Similarly, the cache state affects the overall runtime of a program and can thus lead to remotely observable timing leaks. A variety of attacks has been constructed based on this:

**Example:** Bernstein showed that the cache state influences the runtime of a T-table implementation of AES in OpenSSL [12]. The basic observation is that the lookup into the T-table depends on key and plaintext, allowing a remote adversary to reconstruct the full AES key through timing observation. Similar attacks with improved characteristics were presented by other authors, such as the differential cache-collision timing attack by Bogdanov et al. [13] or the work by Osvik, Tromer and Shamir[14], [15].

**Example:** Similar to the state of data caches, the instruction cache (I-cache) can result in side-channel leaks. The first such attack was discussed in [16]. A trace-driven timing attack against the RSA algorithm by observing the whole I-cache is proposed by Chen et al. [17]. Hidden Markov models can be used to improve I-cache techniques, as proposed in [18]. This is demonstrated by recovering keys by attacking OpenSSL’s DSA implementation.

**Example:** The FLUSH+RELOAD attack [19] is a generic method to infer memory access patterns at cache-line granularity by continuously flushing the cache and measuring the reload latency of a victim memory location. In the original paper, FLUSH+RELOAD was for example used to extract RSA keys from GnuPG by distinguishing squaring and multiplication function calls. In transient execution side-channels such as the Meltdown attack [20], FLUSH+RELOAD is used to leak data through changes to the cache state made during transient execution. Multiple variants of FLUSH+RELOAD have been proposed since, e.g., Flush+Flush [21].

**Example:** Apart from instruction and data caches, CPUs also contain other cache-like structures, in particular the Translation Lookaside Buffer (TLB), which caches the physical address for recently resolved virtual memory addresses. Gras et al. have shown that the TLB can lead to exploitable side-channel leakage and bypass countermeasures against other cache attacks [22].

### 2.3.2 Data-dependent instruction latency

Even if there are no secret dependencies in a program’s control flow, the underlying hardware implementation of certain CPU instructions can lead to data-dependent execution time.

**Example:** The latency of multiplication and division instructions on ARM Cortex-M3 CPUs depend on the operand values as explained in the technical reference manual: “UMULL, SMULL, UMLAL, and SMLAL instructions use early termination depending on the size of the source values. These are interruptible, that is, abandoned and restarted, with worst case latency of one cycle” and “Division operations use early termination to minimize the number of cycles required based on the number of leading ones and zeroes in the input operands” [23].

Similarly, Intel defines a “safe” subset of their instruction set for use in constant-time code:

**Example:** Intel state in [24] that “a new model specific register (MSR) control enables data operand independent timing for the listed data operand independent timing instructions” and further give a list of “instructions that have data-independent timing. These instructions can be used in conjunction with Data Operand Independent Timing Instruction Set Architecture Guidelines to help mitigate timing side-channels.”

### 2.3.3 Dynamic voltage and frequency scaling

Even when instruction timing is independent of the operands, the power consumption of the instruction is likely to leak information on the operand values. Recently, it was shown that this power consumption dependency can manifest in timing leakage due to dynamic voltage and frequency scaling (DVFS) used extensively on modern CPUs.

**Example:** The Hertzbleed attack [25] demonstrates that the frequency scaling on Intel and AMD x86 CPUs leaks information on the power consumption (and hence internal operands) of the processor through the (remotely observable) overall execution time. This is used to mount a remote attack against a constant-time implementation of the SIKE cryptosystem.

**Example:** Along similar lines as Hertzbleed, Liu et al. showed that the frequency throttling mechanism of Intel CPUs can be used to indirectly infer the power consumption of a victim program through throttling events [26]. This can be abused both by privileged (e.g., in the case of Intel SGX) and unprivileged adversaries and among others leads to attacks on the constant-time AES-NI instruction set extension.

### 2.3.4 Port and scheduler contention

Modern CPUs include a number of parallel execution units or ports, which, similar to caches, are shared resources in Simultaneous Multithreading (SMT) architectures. This results in attacks similar to cache-based side-channels:

**Example:** Cabrera Aldaya et al. show in [27] how contention on execution ports can be used to construct a high-resolution side-channel, allowing an adversary for example to leak an ECC private key from OpenSSL running on the same physical core in a different hyperthread. They note that this side-channel, independent of the memory subsystem, applies to a range of execution scenarios.

**Example**: Similar to execution ports, other components such as the scheduler queue can be vulnerable to contention-based side-channel attacks. In particular, Gast et al. show in [28] that the AMD Zen 2/3 and the Apple M1 microarchitecture feature separate scheduler queues per execution unit, enabling side-channel attacks on RSA implementations in mbedTLS.

### 2.3.5 Branch prediction and transient execution

Finally, even though largely out-of-scope for FreeSBee, transient execution (e.g., speculative execution, branch-prediction, and so on) can cause unexpected side-channel leakage. The first branch-prediction-based software side-channel attack was proposed by AciiçMez et al. [29].

The subsequent evolution of this field led to high-profile attacks such as Meltdown [20], in which kernel (and thus physical) memory is leaked through transient execution of forbidden memory reads. The related Spectre [30] issue is, in its simplest form, based on mis-training the branch predictor to transiently perform out-of-bounds memory accesses which are then made architecturally visible through the cache state. Other issues, such as Microarchitectural Data Sampling [31], similarly rely on issues in modern CPU design.

# 3 Timing side-channel leakage sources considered in FreeSBee

The key focus of the envisioned automatic mitigation of timing side-channels in the project FreeSBee is certainly the control flow-based side channels of Section 2.2. Here multiple attacks can be mapped to a similar code pattern that should be mitigated by the FreeSBee tooling. By reworking the source code in the middle-end of the envisioned tool chain, a wide set of the discussed attack surfaces can be mitigated. In addition to the initial reworking, it has to be assured, that following compiler transformations do not introduce the timing side-channel again.

The presented data-flow attacks in Section 2.1 consider mainly direct information leakage. Such attacks are not target of the project FreeSBee. In combination with the control flow attacks of Section 2.2 on the other hand, such unintended data flow can result in a variation of the execution time. Therefore, FreeSBee will consider a data flow analysis in the front-end to determine the manifestation of confidential information within the system.

The presented timing side-channel caused by the microarchitecture will be considered in the back-end phase. Here it is important to note, that no rework of the microarchitecture will be considered. Therefore, only attacks that can be mitigated by software modifications are considered. This covers an important subset, such as cache-based side-channels or data dependent latencies, if corresponding runtime configuration/selection of the microarchitecture is possible.

# 4 References

[1] A. K. Biswas, D. Ghosal, and S. Nagaraja, “A Survey of Timing Channels and Countermeasures,” *ACM Comput. Surv.*, vol. 50, no. 1, pp. 1–39, Jan. 2018, doi: 10.1145/3023872.

[2] Synopsys, Inc., “The Heartbleed Bug,” 2020. https://heartbleed.com/

[3] “Padding the struct: How a compiler optimization can disclose stack memory,” *NCC Group Research Blog*, Oct. 30, 2019. https://research.nccgroup.com/2019/10/30/padding-the-struct-how-a-compiler-optimization-can-disclose-stack-memory/ (accessed Jun. 08, 2023).

[4] D. Strobel *et al.*, “Fuming Acid and Cryptanalysis: Handy Tools for Overcoming a Digital Locking and Access Control System,” in *Advances in Cryptology – CRYPTO 2013*, R. Canetti and J. A. Garay, Eds., in Lecture Notes in Computer Science, vol. 8042. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 147–164. doi: 10.1007/978-3-642-40041-4_9.

[5] “ÆPIC Leak.” https://aepicleak.com/ (accessed Jun. 08, 2023).

[6] “Stale Data Read from xAPIC / CVE-2022-21233 / INTEL-SA-00657,” *Intel*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/stale-data-read-from-xapic.html (accessed Jun. 08, 2023).

[7] P. C. Kocher, “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems,” in *Advances in Cryptology — CRYPTO ’96*, N. Koblitz, Ed., in Lecture Notes in Computer Science, vol. 1109. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113. doi: 10.1007/3-540-68697-5_9.

[8] A. Hevia and M. Kiwi, “Strength of two data encryption standard implementations under timing attacks,” *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 4, pp. 416–437, Nov. 1999, doi: 10.1145/330382.330390.

[9] D. Brumley and D. Boneh, “Remote Timing Attacks Are Practical,” in *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C.: USENIX Association, Aug. 2003. [Online]. Available: https://www.usenix.org/conference/12th-usenix-security-symposium/remote-timing-attacks-are-practical

[10] W. Schindler, “A timing attack against RSA with the chinese remainder theorem,” in *Cryptographic Hardware and Embedded Systems—CHES 2000: Second International Workshop Worcester, MA, USA, August 17–18, 2000 Proceedings 2*, Springer, 2000, pp. 109–124.

[11] B. B. Brumley and N. Tuveri, “Remote Timing Attacks Are Still Practical,” in *Computer Security – ESORICS 2011*, V. Atluri and C. Diaz, Eds., in Lecture Notes in Computer Science, vol. 6879. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 355–371. doi: 10.1007/978-3-642-23822-2_20.

[12] D. J. Bernstein, “Cache-timing attacks on AES,” 2005.

[13] A. Bogdanov, T. Eisenbarth, C. Paar, and M. Wienecke, “Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs,” in *Topics in Cryptology - CT-RSA 2010*, J. Pieprzyk, Ed., in Lecture Notes in Computer Science, vol. 5985. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 235–251. doi: 10.1007/978-3-642-11925-5_17.

[14] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: the case of AES,” in *Topics in Cryptology–CT-RSA 2006: The Cryptographers’ Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*, Springer, 2006, pp. 1–20.

[15] E. Tromer, D. A. Osvik, and A. Shamir, “Efficient Cache Attacks on AES, and Countermeasures,” *J Cryptol*, vol. 23, no. 1, pp. 37–71, Jan. 2010, doi: 10.1007/s00145-009-9049-y.

[16] O. Acıiçmez, W. Schindler, and Ç. K. Koç, “Cache Based Remote Timing Attack on the AES,” in *Topics in Cryptology – CT-RSA 2007*, M. Abe, Ed., in Lecture Notes in Computer Science, vol. 4377. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 271–286. doi: 10.1007/11967668_18.

[17] C. Chen, T. Wang, Y. Kou, X. Chen, and X. Li, “Improvement of trace-driven I-Cache timing attack on the RSA algorithm,” *Journal of Systems and Software*, vol. 86, no. 1, pp. 100–107, Jan. 2013, doi: 10.1016/j.jss.2012.07.020.

[18] O. Acıiçmez, B. B. Brumley, and P. Grabher, “New Results on Instruction Cache Attacks,” in *Cryptographic Hardware and Embedded Systems, CHES 2010*, S. Mangard and F.-X. Standaert, Eds., in Lecture Notes in Computer Science, vol. 6225. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 110–124. doi: 10.1007/978-3-642-15031-9_8.

[19] Y. Yarom and K. Falkner, “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack,” in *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[20] M. Lipp *et al.*, “Meltdown: Reading Kernel Memory from User Space,” presented at the 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 973–990. Accessed: Jun. 08, 2023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[21] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, “Flush+Flush: A Fast and Stealthy Cache Attack,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds., in Lecture Notes in Computer Science, vol. 9721. Cham: Springer International Publishing, 2016, pp. 279–299. doi: 10.1007/978-3-319-40667-1_14.

[22] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with {TLB} Attacks,” presented at the 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 955–972. Accessed: Jun. 08, 2023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[23] “Cortex-M3 Technical Reference Manual r2p0.” https://developer.arm.com/documentation/ddi0337/h/programmers-model/instruction-set-summary/cortex-m3-instructions (accessed Jun. 08, 2023).

[24] “Data Operand Independent Timing Instructions,” *Intel*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/data-operand-independent-timing-instructions.html (accessed Jun. 08, 2023).

[25] Y. Wang, R. Paccagnella, E. T. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner, “Hertzbleed: Turning Power {Side-Channel} Attacks Into Remote Timing Attacks on x86,” presented at the 31st USENIX Security Symposium (USENIX Security 22), 2022, pp. 679–697. Accessed: Jun. 08, 2023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen

[26] “Frequency Throttling Side-Channel Attack | Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security.” https://dl.acm.org/doi/abs/10.1145/3548606.3560682 (accessed Jun. 08, 2023).

[27] A. C. Aldaya, B. B. Brumley, S. Ul Hassan, C. Pereida Garcia, and N. Tuveri, “Port Contention for Fun and Profit,” in *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA: IEEE, May 2019, pp. 870–887. doi: 10.1109/SP.2019.00066.

[28] S. Gast *et al.*, “SQUIP: Exploiting the Scheduler Queue Contention Side Channel,” presented at the 2023 IEEE Symposium on Security and Privacy (SP), IEEE Computer Society, Oct. 2022, pp. 2256–2272. doi: 10.1109/SP46215.2023.00027.

[29] O. Aciicmez, C. K. Koc, and J.-P. Seifert, “On the Power of Simple Branch Prediction Analysis.” 2006. [Online]. Available: https://eprint.iacr.org/2006/351

[30] P. Kocher *et al.*, “Spectre Attacks: Exploiting Speculative Execution,” in *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA: IEEE, May 2019, pp. 1–19. doi: 10.1109/SP.2019.00002.

[31] “Intel Side Channel Vulnerabilities: MDS and TAA,” *Intel*. https://www.intel.com/content/www/us/en/architecture-and-technology/mds.html (accessed Jun. 08, 2023).