AMD CPU Timing and Power Side-Channel Guidance

Publication Number: 71030 **Revision: 1.10**
Date: February 2026

Contents

List of Tables

Revision History

Date	Revision	Summary
February 2026	1.10	• Initial Release

Chapter 1: Introduction

Side-channel attacks target the differences in execution timing, power consumption, or other physical properties such as electromagnetic emanations as a form of information leakage when a security-critical operation such as a cryptographic computation is happening on an electronic device. The two most popular and powerful side channels in relation to CPUs are execution timing and power consumption.

1.1 Acronyms and Terms

Table 1.1: Acroynms and Terms

Acronym/Term	Definition
AES	Advanced Encryption Standard
AOCL	AMD Optimizing CPU Libraries
BN	BigNum
CPU	Central Processing Unit
DVFS	Dynamic Voltage and Frequency Scaling
EM	Electromagnetic
ECC	Elliptic Curve Cryptography
HWMON	Hardware Monitoring Subsystem
MSR	Model Specific Register
OS	Operating System
POSIX	Portable Operating System Interface
RAPL	Running Average Power Limit
RDTSC	Read Time-Stamp Counter
RSA	Rivest-Shamir-Adleman public-key cryptosystem
SMT	Simultaneous Multi-Threading
SSL	Secure Sockets Layer

1.2 Related Documents

Table 1.2: Related Documents

Number	Title
[1]	OWASP Secure Coding Practices-Quick Reference Guide
[2]	AOCL-Cryptography

Table 1.2: Related Documents (continued)

Number	Title
[3]	Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks
[4]	Software Techniques for Managing Speculation on AMD Processors - Rev 5.09.23
[5]	Docs » Linux Hardware Monitoring » Kernel driver amd_energy
[6]	PLATYPUS: Software-based Power Side-Channel Attacks on x86
[7]	Dude, is my code constant time?/Git: oreparaz/dudect
[8]	Automated pipeline for Constant Time analysis of NIST candidates

Chapter 2: Side-channel Mitigations

2.1 Secret-dependent Access

Secret value-dependent data accesses in the context of side-channel attacks refers to memory access patterns within a program that are directly influenced by the value of a secret piece of data. They allow an attacker to observe these access patterns through timing information observation and potentially infer information about the secret by analyzing how the memory accesses change based on the secret value. This essentially causes the program's memory access behavior to become a "side channel" that leaks information about the secret. Hence, such secret dependent access should be avoided in all security critical applications/implementations. The *Array Index Based on a Secret Value Used to Access Memory* code example in section A.2 provides a secure coding practice to avoid data flow (memory access) based side-channel attacks.

Code programmers should be careful not to cause any secret-dependent access. These accesses are either based on control flow decisions or data flow (memory address-based) decisions. It is important to avoid writing code that will use secret-dependent control flow which could leak, via timing and read-write pattern observation, the secret based on the execution path taken. Using secure coding practices can help avoid any inadvertent timing side-channel leakages through the control flow. Some generic examples of secure coding practices to avoid these secret-dependent operations are presented in Appendix A.

2.2 Memory Read and Write Patterns

The pattern of memory reads and writes should remain consistent regardless of the secret data involved, effectively hiding any information leakage through timing or cache usage variations. This can be achieved through one or more of the following, depending on the use-case:

- Constant-time operation
- Avoiding conditional branching based on secrets
- Masking sensitive data
- Using secure libraries
- Flushing the cache

The guidance provided here is for non-speculative timing side-channels[1]. Secret dependent access patterns can be observed in a number of ways, including cache attacks, prefetchers, and Translation Lookaside Buffers (TLBs) [3]. Previously published AMD guidance regarding Spectre type attacks [4] should be followed as well, as the previous guidance remains applicable to mitigate these vulnerabilities.

Note: [1] Non-speculative side channels refer to side-channels which are not impacted by the processor's speculative execution mechanisms, such as the program's execution time.

2.2.1 Employ Constant-time Operation

Cryptographic operations should be implemented using constant-time algorithms, where the execution time remains the same regardless of the secret data being processed. An example of a constant time comparison function is shown in the second code example in section A.1. Another specific example of a timing side-channel mitigation is the constant-time *BN_mod_exp_mont_consttime* function used in the RSA implementation (rsa.cc) in the AMD Optimizing CPU Libraries (AOCL) Crypto library [2]. Secure coding practices should be used, such as not using *memcmp()* to compare confidential data (e.g., cryptographic secrets), because the CPU time required for the comparison depends on the contents of the addresses compared. This function is subject to timing-based side-channel attacks. In such cases, a function that performs comparisons in constant time, depending only on the quantity of bytes compared, is required. Some operating systems provide such a constant time function (e.g., NetBSD's *consttime_memequal()*), but no such function is specified in POSIX. On Linux, implementing such a custom constant time function is needed. Montgomery Powering Ladder[2] in RSA and Elliptic Curve Cryptography (ECC) should be used for constant time code implementations.

Note: [2] See *The Montgomery Powering Ladder*, Marc Joye, Sung-Ming Yen, CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems, pages 291 - 302.

2.2.2 Avoid Conditional Branching Based on Secrets

If statements or loops where the condition depends directly on the secret should not be used, as this can lead to varying execution times depending on the secret value.

2.2.3 Mask Sensitive Data

When accessing memory locations based on a secret, the secret should be masked with a random value to obfuscate the access pattern. Masking sensitive data usually involves XORing with a random number to obfuscate the side-channel leakages. As an example, Boolean masking uses XOR operation of a random bit *r* with a variable to obtain a masked variable *am = a XOR r*. This removes the statistical dependency between any sensitive data and side-channel leakages. The masked variable *am* is again XORed with *m* at the end to remove the effect of masking to get the correct result. For instance, in AES an XOR with a random value r can be performed at an intermediate round state and then again XORed with *r* at the end after the last round to get the correct ciphertext.[3]

Note: [3] See *Secure and Efficient Masking of AES A Mission Impossible'*, Version 1.0, Elisabeth Oswald, Stefan Mangard and Norbert Pramstaller, Technical Report IAIK- TR 2003/11/1 (updated on 2004/06/04).

2.2.4 Use Secure Libraries

Well-vetted cryptographic libraries such as AOCL cryptographic library [2] that are designed to be side-channel resistant should be used, as they often incorporate optimized techniques to mitigate these vulnerabilities.

2.2.5 Flush Caches

Flushing the cache makes it difficult for attackers to infer memory access patterns or learn sensitive information by observing cache behavior. This can help mitigate such secret-dependent memory access This may be useful only when the safe memory access pattern cannot be guaranteed. Cache flushing is only effective if it is completed before the adversary is able to run. In CPUs with SMT, this is rarely the case, as the sibling thread can run while the other is doing a sensitive operation.

2.3 CPU Power Consumption Side-channel Mitigations

A power consumption-based side-channel attack on a CPU exploits the variations in power usage during processing to infer sensitive information like cryptographic keys. This type of attack can be mounted by observing the subtle changes in power consumption that occur depending on the data being processed, essentially revealing secrets through a physical characteristic of the chip rather than directly accessing the data itself.

Some of the timing side-channel countermeasures such as making cryptographic operations balanced and independent of key bits being processed (such as the use of Montgomery Powering Ladder in RSA and Elliptic Curve Cryptography (ECC) implementations) can remove the statistical dependence between the key bits being processed and the instantaneous power consumption, making power side-channels significantly less usable in extracting key information.

Specific defenses against physical power side-channels should also be considered. These side-channels are based on passively monitoring the power consumption on the appropriate voltage lines or electromagnetic (EM) emanations on the chip or wall socket power usage while a security sensitive operation such as a cryptographic operation is ongoing. Masking, randomization/blinding techniques (for instance, RSA base blinding in WolfSSL, and RSA blinding in OpenSSL) and data obfuscation can help reduce the correlation of power consumption and any secret data that is being processed. Randomization will make it more difficult for the attacker to reliably measure the time it takes to perform decryption or signature operations needed for effective power measurements and correlation to the secret.

CPU Performance counters and associated Model Specific Registers (MSRs) such as the Running Average Power Limit (RAPL) MSR [5] can be targeted by attackers to extract fine-grained power consumption information that can aid side-channel attacks. These can target critical assets and security sensitive operations such as cryptographic algorithm execution, for example as in the Platypus attack [6]. These power measurements are sometimes combined with timing related performance instructions such as RDTSC to extract fine-grain timing information. As a guideline to OS developers, access to performance

counters and power related MSRs should be restricted. The latest amd_energy driver on Linux restricts access to RAPL MSRs, requiring root or sudo privileges to access them. Increasing the time duration for these power measurements to make this information coarse-grained can act as defense-in-depth against side-channel attacks but does not prevent them.

CPU frequency scaling attacks (such as those targeting DVFS functionality) are types of power-based side-channel attacks which work by converting power changes into the frequency domain in order to measure power via timing attacks[4]. Hence the guidance on power side-channel mitigations is also applicable to these types of attacks.

Note: [4] See *DVFS Frequently Leaks Secrets:* Hertzbleed Attacks Beyond SIKE, Cryptography, and CPU-Only Data, Yingchen Wang, Riccardo Paccagnella, Alan Wandke, Zhao Gang, Grant Garrett-Grossman, Christopher W. Fletcher, 44th IEEE Symposium on Security and Privacy, San Francisco, 22-25 May 2023.

2.4 General Side-channel Mitigations

Compiler optimizations can affect the implementation of an algorithm and potentially create side-channels. Software countermeasures introduced to protect against side-channels can be removed through these optimizations, leading to timing or power asymmetries in the generated machine code. The use of specific type qualifiers like “volatile” in C can be used to prevent certain compiler optimizations and thus help in securing software code. Compiler arguments such as -O0 can be used to prevent such optimizations, especially on critical portions of the software code.

Coverity and CodeQL scans may also be used to identify some secure coding issues that may lead to inadvertent side-channel leakages. Some open-source tools such as “dudect” [7] and Blatchley Timing Analysis Pipeline [8] can help detect timing side-channel vulnerable software code and hence help designers with writing secure code. These tools can be executed on security sensitive code to identify such timing issues and fix them before such code is released for production.

Appendix A: Secure Coding Examples

A.1 Timing Insecure Function

An example of a timing-insecure function would be when a for loop is exited as soon as even a single comparison fails, leading to timing information leakage from the comparison operation, as shown below:

```
bool insecure_compare(const unsigned char* a, const unsigned char* b, int len) {
    for (int i = 0; i < len; i++) {
        if (a[i] != b[i])
            return false;
    }
    return true;
  }
```

Use constant timing comparison functions that always perform the same operations regardless of the input to prevent attackers from inferring secret data by analyzing execution time variations.

```
bool constant_time_compare(const unsigned char* a, const unsigned char* b, int len) {
  uint8_t result;
  for (int i = 0; i < len; i++) {
     result |= a[i] ^ b[i]; // XOR each byte and combine with OR
   }
   return result == 0; // Return true if all bits are 0
 }
```

A.2 Array Index Based on a Secret Value Used to Access Memory

If an array index based on a secret value is used to access memory as shown in the example below, it leads to an insecure implementation though side-channel leakage. This allows an attacker to observe the memory access pattern and deduce information about the secret.

```
// Lookup table used for computation
uint8_t lookup_table[LOOKUP_TABLE_SIZE];
uint8_t insecure_leak(uint8_t index) {
  // Access based on secret value
  return lookup_table[secret[index]];
}
```

Use data-independent memory access, as shown below, to prevent attackers from inferring secret data by observing memory access patterns.

```
// Constant-time lookup function
uint8_t constant_time_lookup(uint8_t value) {
  uint8_t result = 0;
  for (int i = 0; i < LOOKUP_TABLE_SIZE; i++) {
    // if i == value, mask becomes 0xFF; otherwise, 0x00
    uint8_t mask = (i == value);
    mask = -mask; // Convert 1 to 0xFF, 0 to 0x00
    result |= lookup_table[i] & mask;
  }
  return result;
}
```

```
uint8_t secure_access(uint8_t index) {
  // Use constant-time lookup instead of direct indexing
  return constant_time_lookup(secret[index]);
}
```

A.3 Control Flow

An example of control flow for a secret-dependent operation might involve using a conditional statement to determine which code path is executed, as shown below:

```
int insecure_add(int x, int secret) {
  int num_bits = sizeof(secret) * 8;
  for(int i=num_bits-1; i>=0; i--) {
    int secret_bit = (secret >> i) & 1;
    if (secret_bit) {
        x += 5; // Addition only happens if secret bit is 1
    }
  }
  return x;
}
```

Use control flow to ensure that the execution flow of the program remains consistent regardless of the secret value to prevent attackers from inferring secret data by observing control flow variations. This can be achieved by avoiding secret data in conditional statements or loops that control the execution flow, as shown in the example below.

```
int secure_add(int x, int secret) {
  int num_bits = sizeof(secret) * 8;
  for (int i = num_bits - 1; i >= 0; i--) {
    int secret_bit = (secret >> i) & 1;
    // Always perform the operation, but mask the effect
    x += secret_bit * 5;
  }
  return x;
}
```

Appendix B: Notices

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED "AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

B.1 Trademarks

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.