Crypto Corner
Editors: Peter Gutmann, pgut001@cs.auckland.ac.nz
David Naccache, david.naccache@ens.fr
Charles C. Palmer, ccpalmer@us.ibm.com

# Side Channel Attacks on Cryptographic Software

Nate Lawson
*Root Labs*

When it comes to cryptographic software, side channels are an often-overlooked threat. A side channel is any observable side effect of computation that an attacker could measure and possibly influence. Crypto is especially vulnerable to side channel attacks because of its strict requirements for absolute secrecy. In the software world, side channel attacks have sometimes been dismissed as impractical. However, new system architecture features, such as larger cache sizes and multicore processors, have increased the prevalence of side channels and quality of measurement available to an attacker. Software developers must be aware of the potential for side channel attacks and plan appropriately.

## History of Side Channel Attacks

Side channels are a variant of the classic covert-channel problem. Covert channels involve two or more processes collaborating to communicate via a shared resource that they can both affect and measure. Attackers can exploit these channels to bypass operating system protections such as mandatory access control that are intended to keep the processes separate. For example, one process can allocate memory while the other measures the amount of free memory. Through repetition of this behavior, the first process can slowly communicate information to the second. The channel's signal-to-noise (S/N) ratio measures its quality. For example, memory allocations by unrelated processes might skew some measurements, so a particularly busy system might have a low S/N ratio. Error correction methods can assist with this case.

Whereas covert channels involve the problem of preventing cooperation, side channel attacks are a purely adversarial problem. Side channels emerge because computation occurs on a non-ideal system, composed of transistors, wires, power supplies, memory, and peripherals. Each component has characteristics that vary with the instructions and data being processed. When this variance is measurable by an attacker, a side channel is present.

Intelligence agencies have often relied on side channel attacks to monitor their foes. In one clever incident, the Soviet Union provided a large wooden seal to the American consulate in Moscow. The US ambassador proudly hung it in his office after it had been examined for covert transmitters. It appeared to be clean. Unbeknownst to the ambassador, the seal contained a carefully designed cavity that vibrated in response to sounds in the room. The spies transmitted a radio beam at the seal, measured the beam's modulation, and recreated the conversations in the room. This listening method went undetected for years.

More recently, side channel attacks have become a powerful threat to cryptography. One of the first papers on side channel attacks showed how to recover an RSA private key merely by timing how long it took to decrypt a message.[1] This was possible because RSA and other public-key cryptosystems work with large numbers (for example, 2,048 bits), whereas modern CPUs have a smaller word size. Crypto implementations compensate by using multiprecision arithmetic, representing large numbers by an array of words and using a loop to carry overflows from one word to the next.

To raise a multiprecision number to an exponent, systems such as RSA commonly use *square-and-multiply*. This optimization decomposes an exponentiation into a series of squarings ($x^2$) and conditional multiplies ($* x$), which occur if the bit in question is a one. This is similar to pencil-and-paper multiplication, in which trailing zeros mean that you shift the result one decimal place to the left while nonzero digits are multiplied and added to the result.

Because the multiply step is conditional, an attacker gains information about the total number of one bits with each decryption. By measuring the total time to perform a multiprecision exponentiation with different input messages, the attacker can eventually recover the entire private key or enough to brute-force the rest.

Timing attacks have continually improved, even being performed against an SSL (Secure Sockets Layer) implementation over a network.[2] New ways to filter jitter have improved the distinguishability to 200 ns over a LAN and 30 μs over the Internet.[3] Attacks have also exploited new side channels. Power consumption, RF and electromagnetic emissions, sound, vibration, and even heat give away information about secret computations. These attacks aren't merely the subject of research papers. Smartcards used for payment, transit, and satellite TV have been compromised by both active fault induction attacks ("glitching") and side channel attacks. Hackers used a timing attack against a secret key stored in the Xbox360 CPU to forge an authenticator and load their own code.[4]

Embedded-systems designers are no longer the only ones who must prevent side channel attacks. Previously, network-based timing attacks against SSL were the only side channel attack most software developers needed to consider. But today, virtualization and application-hosting services such as Amazon S3 have given attackers a more privileged vantage point of running code on the same system (possibly even at the same privilege level) as the target's code. Also, high-speed multicore CPUs with large caches and complicated instruction- and data-dependent behavior provide more possibilities for side channels and greater precision for measurements.

To illustrate side channel attacks against software cryptography, I analyze three recent attacks. Each is increasingly more powerful, to the point where the attacker can recover an entire RSA key by measuring the behavior of a single decryption operation.

## Keeping the Correct Answer Secret

The HMAC (Hash Message Authentication Code) hash construction is often used to authenticate messages. To compute a given message's HMAC, the sender hashes the message and a secret key twice using a cryptographic hash algorithm (for example, SHA-256). The result is attached to the message. The recipient then calculates the message's HMAC via the same process and compares the result to the value included with the message. If they match, the message wasn't tampered with after the sender calculated the HMAC.

One subtlety with this process is that the value the recipient calculates must be kept secret. Consider what would happen if the result were revealed to the sender in the case of a mismatch, perhaps as part of an error message. An attacker could submit a message with an invalid HMAC field, observe the error message, and then resend the same message with the correct value attached. The recipient would accept this message as valid, even though the sender didn't create the authenticator. Although most systems probably don't reveal the correct HMAC value directly, a side channel attack can often produce the same effect.

I recently reviewed the open-source Google Keyczar cryptographic library for possible flaws.[5] This library provides useful high-level key management features, with separate implementations in Java, Python, and C++. Keyczar includes an HMAC implementation for authenticating messages.

The Python code compared the received and calculated byte values as follows:

```
return self.Sign(msg) ==
sig_bytes
```

The Java code was equivalent.

The underlying comparison operator for both high-level languages performs a byte-wise match of the two arrays. If any element didn't match, the comparison loop would terminate early. This would provide a timing side channel in which the attacker could iteratively fill in guesses for each byte of the HMAC field, resubmitting the same message each time. When the guess was correct, the comparison would take a little longer. Eventually, when the whole HMAC was correct, the recipient would accept the message.

An implementation of this attack over a TCP connection to localhost took about a thousand queries per byte of the secret key. This means that an attacker could find a 128-bit key in less than a few minutes. Because the array comparison operators in Java and Python aren't implemented natively, the timing difference for each loop iteration was relatively large. But even if there was more network jitter or the comparison loop was faster, the attacker could simply take more samples, apply an appropriate filter, and perform a statistical hypothesis test to determine which guess was correct.

The solution to this problem is to implement a comparison function that doesn't terminate early. Although this might sound easy at first, eliminating all conditional branches from a comparison loop is surprisingly difficult. Even with a correct algorithm, some underlying detail of the high-level language implementation (such as garbage collection) could still

**Power consumption, RF and electromagnetic emissions, sound, vibration, and even heat give away information about secret computations.**

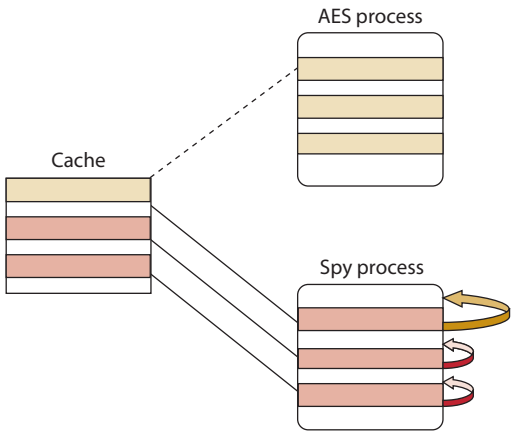


Figure 1. In a Prime+Probe attack, a spy process probes the cache by monitoring timing of accesses to its own memory. As the target process encrypts, it evicts portions of the attacker's memory from the cache, resulting in longer access times. The access times for the individual regions of the attacker's memory correspond to which tables the encryption process accessed, and thus the target's key.

leave a measurable timing difference. The standard C `memcmp()` function is unsafe as well because it also terminates early.

## Footprints in the Cache

Like the original RSA timing attack, the HMAC timing attack combines many measurements of the entire operation to find the target's secret. However, more powerful side channel attacks can give insight into an algorithm's intermediate working values, revealing the secret more quickly.

Modern systems employ a CPU cache to keep frequently accessed memory close to where it's needed. When data for a given address is in the cache, it's returned immediately. If not, it's fetched from memory into the cache, stalling the CPU for a few more cycles. A cache is often divided into blocks called *lines*.

Because a cache is smaller than the memory it shadows, the CPU must have a policy for filling and reusing its space. The most common implementation is a set associative cache, which maps multiple addresses to the same cache line on the basis of some fraction of the upper address bits. For example, the addresses 0x100, 0x200, and 0x800 would all use the same cache line if the cache had 256 lines of one byte each. "Set" refers to the number of possible destination cache lines per address (that is, *N*-way). The CPU evicts older entries when data is loaded into an already-filled cache line.

AES (Advanced Encryption Standard) is a standard block cipher. It encrypts and decrypts data with a secret key, using a combination of primitives such as `MixColumns`, `ShiftRows`, and `SubBytes` over many rounds (10 for a 128-bit key). A common optimization technique on 32-bit processors is to precompute a series of tables on the basis of the combination of these primitives. AES encryption then becomes a series of table lookups and XOR operations.

Because the index for these AES tables is the XOR of a plaintext byte and a key byte, the indices themselves must remain secret. However, a spy process running on the same system can observe the variable timing of the AES encryption due to cache behavior, narrowing down the possible values for the key.[6] Even if running a spy process isn't possible, a remote attacker can often trigger changes in the system cache state by interacting with other processes and timing those unrelated tasks' behavior.

Dag Arne Osvik and his colleagues have described two useful ways to induce variability and observe cache behavior: Evict+Time and Prime+Probe.[6]

Evict+Time works as follows:

1. Trigger an encryption in the target process.
2. Evict memory from chosen cache lines by accessing the appropriate addresses in the attacker's process.
3. Trigger and time another encryption of the same plaintext.

The first step ensures that all the AES lookup tables accessed by the given plaintext and key are cached. The second step forces the CPU to evict part of one AES table that the attacker is targeting, on the basis of a guess of the key byte. The final step tests the attacker's hypothesis. If a cache miss occurs and the AES encryption takes longer than other cases, the guess for the XOR of the plaintext and key bytes was correct and caused the CPU to reload the table from RAM after it had been evicted. If not, the guess was incorrect. The attacker repeats this process to narrow the possible key values.

Prime+Probe (see Figure 1) is more powerful. It's analogous to placing a film negative behind an object and measuring the outline cast by the object's shadow. Instead of timing the encryption process, which is subject to noise and jitter due to surrounding code in the target, the attacker repeatedly times accesses to its own memory while the target encrypts. Each time an encryption occurs, the CPU evicts one or more lines of the attacker's memory from the cache, causing timing variation. Because the cache eviction is local to the attacker, countermeasures such as randomizing or normalizing the total encryption time have no effect.

Such an attack isn't merely a timing attack. Although time is the method for probing cache behavior, this attack could use other methods to determine the cache state. For example, if an instruction provided the number of valid cache lines for the current task, it would directly provide the same information obtained from this timing side channel.

Intel and AMD (and previously, Via) introduced AES instructions to address this problem

and increase performance. Unfortunately, owing to the structure of AES, there appears to be no way to build a high-performance implementation on a general-purpose CPU while avoiding cache side channels.

## Which Way Did He Go?

A related but even more powerful attack uses the branch prediction cache's status as a side channel.[7] Instead of detecting memory accesses to the key data, this attack determines the code path the target process takes while executing the encryption code.

As I previously described, square-and-multiply has an optional multiplication step. If the attacker can detect when this branch is taken, he or she can determine which bits of the key are ones. (Other, more optimized routines such as sliding-window exponentiation have similar weaknesses.)

Because modern CPUs have a deep pipeline, they implement a *branch prediction unit*, which keeps track of the target address and whether the branch was taken in a cache called the *branch prediction target buffer* (BTB). As with the memory cache, an attacker can influence and measure the cache state by performing jumps and timing either the encryption process (as in Evict+Time) or its own execution speed (Prime+Probe).

One potential hurdle for branch prediction side channel attacks is disruption due to support code or other processes running. This adds noise to the measurements. However, Onur Aciiçmez and his colleagues discovered that this noise was highly periodic.[7] By taking several different measurements, they could select the one with the lowest noise and use it as the source for the key bits they were detecting. Unlike the cache attacks on AES, such an attack can derive enough key bits from a single trace that repeated analysis is unnecessary.

Side channel attacks were once esoteric, remaining the domain of special-purpose hardware. However, with the advent of cloud computing and virtualized servers, you can no longer assume that attackers are remote. Advanced statistical methods and modeling have given them precise measurements independent of jitter. Meanwhile, CPUs' increasing microarchitectural complexity has created more side channels to exploit. Any software developer who writes or deploys an application utilizing cryptography must be aware of this powerful class of attacks. ☐

### References


1. P. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," Cryptography Research, 1995; www.cryptography.com/resources/whitepapers/TimingAttacks.pdf.
2. D. Brumley and D. Boneh, "Remote Timing Attacks Are Practical," *Proc. 12th Conf. Usenix Security Symp.*, Usenix Assoc., 2003, p. 1.
3. S.A. Crosby, D.S. Wallach, and R.H. Riedi, "Opportunities and Limits of Remote Timing Attacks," *ACM Trans. Information and System Security*, vol. 12, no. 3, article 17; www.cs.rice.edu/~dwallach/pub/crosby-timing2009.pdf.
4. "Timing Attack Tested Successfully: Downgrade from Any Kernel without CPU-Key"; www.xbox-scene.com/xbox1data/sep/EElZluZypZpmixPJrS.php.
5. N. Lawson, "Timing Attack on Google Keyzar," blog, 28 May 2009; http://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library.
6. D.A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," *Topics in Cryptology—CT-RSA 2006*, LNCS 3860, Springer, 2006; pp. 1–20.
7. O. Aciiçmez, Ç.K. Koç, and J.-P. Seifert, "On the Power of Simple Branch Prediction Analysis," *Proc. 2nd ACM Symp. Information, Computer and Communications Security*, ACM Press, 2006, pp. 312–320.



***Nate Lawson*** *is the founder of Root Labs, a security consulting practice focusing on kernel, embedded-platform, and cryptography design and analysis. Contact him at nate@rootlabs.com.*