Mastik: A Micro-Architectural Side-Channel Toolkit

Yuval Yarom

The University of Adelaide and Data61, CSIRO
Adelaide, Australia
yval@cs.adelaide.edu.au

1 Introduction

Micro-architectural side-channel attacks exploit contention on internal components of the processor to leak information between processes. While in theory such attacks are straightforward, practical implementations tend to be finicky and require significant understanding of poorly documented processor features and other domain-specific arcane knowledge. Consequently, there is a barrier to entry into work on micro-architectural side-channel attacks, which hinders the development of the area and the analysis of the resilience of existing software against such attacks.

This document introduces *Mastik*, a toolkit for experimenting with micro-architectural side-channel attacks. Mastik aims to provide implementations of published attack and analysis techniques. At the time of writing this document, Mastik is at a very early stage of development. Version 0.02 codename "Aye Aye Cap'n" has been released.

The release includes implementation of six cache-based attacks on the Intel x86-64 architecture. These include the Prime+Probe attack on the L1 data cache [9, 10], Prime+Probe on the L1 instruction cache [1, 2], Prime+Probe on the Last Level Cache [7, 8], Flush+Reload [11], Flush+Flush [6] and a performance degradation attack [3].

In addition to the implementation of the attacks, Mastik provides several tools that facilitate the attacks. These include functionality for handling symbolic code references, such as loader symbol names or debug information, and functions that simplify some system features that are commonly used in side-channel attacks. New in version 0.02 is the utility `FR-trace`, which supports mounting the Flush+Reload attack from the command line.

We now proceed with a motivating example of the benefits of Mastik (Section 2), followed by a more detailed description of the interface (Section 3).

2 Mastik Examples

To demonstrate the power of Mastik we now show how to reproduce the Flush+Reload attack on GnuPG 1.4.13 [11].

GnuPG 1.4.13 uses the square-and-multiply algorithm [4] for perfomring the modular exponentiation step of the RSA decryption and signature. Yarom and Falkner [11] demonstrate that this implementation is vulnerable to the Flush+Reload side-channel attack. They use Flush+Reload to trace the victim's use of the multiply, square and modular reduction operations. From the traced operations, the attacker can recover the bits of the exponent, which correspond to the victim's private key.

While the core of the Flush+Reload attack is relatively straightforward, an implementation of the attack needs to repeatedly probe the memory at a fixed interval. It should further be able to re-synchronise with the victim when interrupted by the operating system. Additionally, the attacker needs to be able to convert source code locations to memory adresses. Mastik takes care of most of these operations. It hides the complexity and provides the user with a simple interface for mounting the attack.

Listing 1 shows the implementation of the attack. (A similar implementation is available in `demo/FR-gnupg-1.4.13.c` in the Mastik distribution.) Mastik uses the opaque handle type `fr_t` to abstract the attack. The attack handle is instantiated using a call to `fr_prepare()` (Line 9).

Lines 11 to 18 set up the memory locations the attack monitors. Like Yarom and Falkner [11] we set the attack to monitor locations within the code that computes the multiply, square and modular reduction operations. To specify these locations we use references to victim source lines (Line 5). The `sym_getsymboloffset()` uses the debugging information in the GnuPG binary to convert these references into offsets in the binary. We use the `map_offset()` function to map these offsets into the address space of the spy program and `fr_monitor()` sets the Flush+Reload attack to monitor these location.

Listing 1. Flush Reload attack on GnuPG 1.4.13

```
1  #define SAMPLES 100000
2  #define SLOT    2000
3  #define THRESHOLD 100
4
5  char *monitor[] = { "mpih-mul.c:85", "mpih-mul.c:271", "mpih-div.c:356" };
6  int nmonitor = sizeof(monitor)/ sizeof(monitor[0]);
7
8  int main(int ac, char **av) {
9    fr_t fr = fr_prepare ();
10
11   for (int i = 0; i < nmonitor; i++) {
12     uint64_t offset = sym_getsymboloffset(av[1], monitor[i]);
13     if (offset == ~0ULL) {
14       fprintf (stderr, "Cannot␣find␣%s␣in␣%s\n", monitor[i], av[1]);
15       exit (1);
16     }
17     fr_monitor(fr, map_offset(av[1], offset));
18   }
19
20   uint16_t *res = malloc(SAMPLES * nmonitor * sizeof(uint16_t));
21   bzero(res, SAMPLES * nmonitor * sizeof(uint16_t));
22   fr_probe(fr, res);
23
24   int l = fr_trace (fr, SAMPLES, res, SLOT, THRESHOLD, 500);
25   for (int i = 0; i < l; i++) {
26     for (int j = 0; j < nmonitor; j++)
27       printf ("%d␣", res[i * nmonitor + j]);
28     putchar('\n');
29   }
30 }
```

The attack itself is carried out in Line 24. The `fr_trace()` function waits for activity in any of the memory locations it monitors. It then colllects activity records at fixed intervals. Collection stops when a long enough period of inactivity is detected or when the function runs out of space to store the results. The activity records measure the time it takes to read data from the monitored locations. Short access times indicate that the locations cached and is therefore active.

Before terminating, the program outputs the results. A segment of the program output is shown in Figure 1. As the figure shows, there are clear areas of activity for each of the monitored locations. From these, the adversary can reconstruct the sequence of operations the victim executes and infer the exponent.

As we can see, Mastik provides an easy-to-use interface to the attack. It hides most of the attack implementation details, exposing only those parameters that are needed for targetting a specific victim.

We now proceed to describe some of the patterns Mastik employs when abstracting over the various attacks.

3 API Design

One of the challenges of designing an API for side-channel attacks is striking a balance between three conflicting aims. We would like the intertface to be simple and uniform. At the same time, we would like to expose the unique strengths

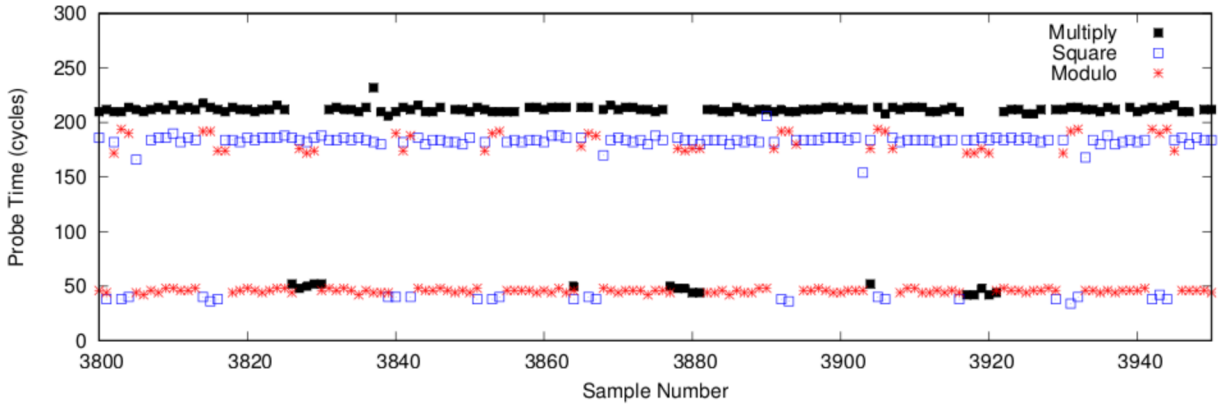

Fig. 1. Results of the Flush+Reload attack on GnuPG 1.4.13. Values below 100 cycles indicate that the monitored line was active during the sample.

of each of the attacks. In addition we want the implementation to be as optimised as possible so as to minimise the attack footprint.

Mastik achieves this balance by presenting similar interfaces for all of the attacks, without providing shared implementations of the underlying operations. Thus, attack interfaces share the same look-and-feel, but there are attack-specific variations in the interface and the types it uses for similar purposes across different attacks do not share common supertypes.

Attack	Descriptor	Prefix
L1-D Prime+Probe [9, 10]	`l1pp_t`	`l1_`
L1-I Prime+Probe [1, 2]	`l1ipp_t`	`l1i_`
LLC Prime+Probe [7, 8]	`l3pp_t`	`l3_`
Flush+Reload [11]	`fr_t`	`fr_`
Flush+Flush [6]	`ff_t`	`ff_`
Performance degradation [3]	`pda_t`	`pda_`

Table 1. Implemented Attacks

Every attack in Mastik is encapsulated using an attack *descriptor*, which is an opaque pointer to a structure describing information about the attack, and a set of functions that manage the structure and implement the attack. Table 1 summarises the implemented attacks and specify their descriptor type and the prefix used for the attack functions. In the future we expect some attacks to share descriptors. For example, the Evict+Reload [5] could use the LLC Prime+Probe descriptor (`l3pp_t`). For each attack Mastik provides three types of functions: descriptor management, attack set-up and attacks.

Descriptor Management The initialisation function `XX_prepare()` (where `XX` is the attack prefix) initialises a descriptor. For some descriptors types it may take an argument that provides parameters for the initialisation routine. Currently, only the LLC Prime+Probe takes an argument. Passing `NULL` chooses the default behaviour.

The `XX_release()` function releases the descriptor and all of the resources allocated by it.

Attack Set Up Each attack defines an attack space that consists of a set of *points* that the attack can operate on. For most of the attacks, the operation is to *monitor* activity at the point.

The nature of the points is attack-dependent. In the Flush+Reload attack, a point is an address in the virtual address space of the process running the attack. For the Prime+Probe attacks the points identify sets in the targeted cache. For

each attack Mastik provides several functions for managing the set of points which an attack which uses the descriptor would probe. The functions are:

`XX_monitor()` Adds a point to the set of points the descriptor monitors.
`XX_unmonitor()` Removes a point from the set of points the descriptor monitors.
`XX_monitorall()` Adds all possible points to the set of points the descriptor monitors. Only supported for the L1 attacks.
`XX_unmonitorall()` Removes all points from the set of points the descriptor monitors.
`XX_getmonitoredset()` Returns the set of points the descriptor monitors.
`XX_randomise()` Uses a non-secure pseudo-random generator to reorder the set of monitored points.

The L1 attack descriptors are initialised to monitor all of the points (cache sets) in a random order. Other descriptors are initialised to monitor no point.

Attack An attack round, implemented by `XX_probe()`, consists of probing each of the monitored points to determine activity. The typical result is a set of timing data, measuring the number of cycles to probe each point. The results match the order of probes returned by `XX_getmonitoredset()`. The meaning of the results differ between attacks. See the original publications for further information.

Attack Variations The Prime+Probe attacks often benefit from a bidirectional probing. For the L1 Data and LLC Prime+Probe attacks, Mastik provides the function `XX_bprobe()`, which preforms the probe at the reverse direction. For the LLC attack, it is often useful to count the number of cache misses rather than the total time to probe the cache set. The functions `l3_probecount()` and `l3_bprobecount()` performs this operation.

Repeated Attacks A single probe often provides too little information. The function `XX_repeatedprobe()` performs a sequence of probes, alternating between `XX_probe()` and `XX_bprobe()` if the latter is supported. The `slot` parameter to the function regulates the probes to perform one probe per `slot` cycles. If a slot is missed, the timing result for the points in that slot is set to 0. (For the probecount version the result is set to ~0.)

Traces The Flush+Reload and the Flush+Flush attacks support an extended version of repeated attack. This version, `XX_trace()`, waits for activity in the monitored cache lines. Collection of data starts when activty is sensed and stops when activity is no longer sensed or when the space for storing the results is exhausted.

Performance Degradation Attack Unlike the other attacks, the performance degradation attack does not monitor the victim. Instead, it *targets* cache lines that the victim uses frequently and evicts them from the cache. To reflect the different use, the functions that set the attack up use `target` instead of `monitor`. For example, the function `pda_target()` adds a targe to the target list of a performance degradation attack. The functions `pda_activate()` and `pda_deactivate()` start and stop the attack. `pda_activate()` spawns a subprocess that executes the attack. `pda_deactivate()` kills the subprocess.

Symbol Management Mastik provides three functions for converting symbols to file offsets. `sym_loadersymboloffset()` finds a symbol in the loader symbol table. `sym_debuglineoffset()` finds the machine code that corresponds to a specific source line. In Linux, these functions rely on `libbfd`, `libdwarf` and `libelf`. To use make sure that `libdwarf-devel` and `binutils-devel` (or `libdwarf-dev`, `binutils-dev` and `libelf-dev`) are installed.

`sym_getsymboloffset()` provides a generic interface for converting symbolic references to file offsets. It recognises four input formats: file offset, virtual addresse, loader symbol and line number. It can further recognises simple arithmetic operations that allow shifting the offset. For example, the input `"main+0x40"` refers to the location 64 bytes after the start of the function `main`.

Utility functions `map_offset()` maps a file to the virtual address of the process and returns a pointer to the data at a specified offset in the file. Only the page containing the offset is mapped. `unmap_offset()` removes the mapping.

`delayloop()` performs a busy-loop for a number of cycles. It can be used between probes, if `XX_repeatedprobe()` does not provide the desired functionality. Another use of `delayloop()` is to generate enough activity to avoid CPU frequency scaling. In our experience, `delayloop(3000000000U)` always achieves the desired result. Your mileage may vary.

Acknowledgements

Many thanks to Rusty Russell and to Joel Stanley for suggesting `FR-trace`.

Bibliography

[1] Onur Acıiçmez. Yet another microarchitectural attack: exploiting I-cache. In *ACM Computer Security Architecture Workshop (CSAW)*, Fairfax, VA, US, 2007.

[2] Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *Crytographers' track at the RSA Conference on Topics in Cryptology*, pages 256–273, San Francisco, CA, US, 2008.

[3] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Proceedings of the 32nd Annual Computer Security Applications Conference (ACSAC)*, Los Angeles, CA, US, December 2016.

[4] Daniel M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27(1):129–146, April 1998.

[5] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Security Symposium*, pages 897–912, Washington, DC, US, August 2015.

[6] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *Proceedings of the 13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, San Sebastián, ES, July 2016.

[7] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *IEEE Symposium on Security and Privacy*, San Jose, CA, US, May 2015.

[8] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, San Jose, CA, US, May 2015.

[9] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. http://www.cs.tau.ac.il/~tromer/papers/cache.pdf, November 2005.

[10] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, CA, 2005.

[11] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium*, pages 719–732, San Diego, CA, US, 2014.