FaSe: Fast Selective Flushing to Mitigate Contention-based Cache Timing Attacks

Tuo Li
University of New South Wales
Sydney, Australia
tuoli@unsw.edu.au

Sri Parameswaran
University of New South Wales
Sydney, Australia
sri.parameswaran@unsw.edu.au

ABSTRACT

Caches are widely used to improve performance in modern processors. By carefully evicting cache lines and identifying cache hit/miss time, contention-based cache timing channel attacks can be orchestrated to leak information from the victim process. Existing hardware countermeasures explored cache partitioning and randomization, are either costly, not applicable for the L1 data cache, or are vulnerable to sophisticated attacks. Countermeasures using cache flush exist but are slow since all cache lines have to be evacuated during a cache flush. In this paper, we propose for the first time a hardware/software flush-based countermeasure, called fast selective flushing (FaSe). By utilizing an ISA extension and cache modification, FaSe *selectively flushes cache lines* and provides a mitigation method with a similar effect to methods using naive flush. FaSe is implemented on RISC-V Rocket Chip and evaluated on Xilinx FPGA running user programs and the Linux OS. Our experiments show that FaSe reduces time overhead by 36% for user programs and 42% for the OS compared to the methods with naive flushing, with less than 1% hardware overhead. Our security test shows FaSe can mitigate target cache timing attacks.

1 INTRODUCTION

Cache timing side channel [1] has been a key component in recent lethal security attacks, such as Spectre/Meltdown, on contemporary commodity processors. Contention-based cache timing attack, e.g., PRIME+PROBE [2], is an important type of attack exploiting cache timing channels. In such an attack, an adversary utilizes carefully designed cache evictions, to learn the cache access pattern of the victim process.

Fig. 1 demonstrates an example PRIME+PROBE attack, which has three stages. In **Stage 1** (prime), the spy process places its own data (a to p) in all the cache lines (in light grey). In **Stage 2**, the spy process gives up the processor core and waits for the victim process to execute its task on this core. During this stage, any cache usage on a cache line from the victim will result in contention at the corresponding cache line, and hence, will have attacker's prime data at these cache lines replaced and moved out of cache to the next level of cache-memory hierarchy. For example, in Fig. 1, victim's
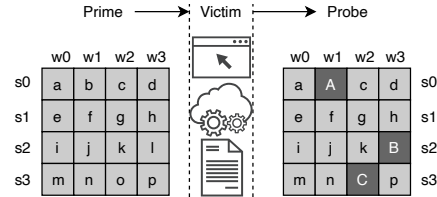
Figure 1: Abstract view of a PRIME+PROBE attack. The blocks in dark grey represent the cache lines utilized by victim. si: Cache Set i. wi: Cache Way i.

data A, B, and C, cause attacker's data b (at s0,w1), l (at s2,w3), and o (s3,w2) replaced. In **Stage 3** (probe), the spy process is switched back (simultaneously victim process is preempted from this core) to this core, and the spy proceeds to pinpoint the replaced cache lines, by loading or storing the prime data again. By measuring the time taken for accessing the prime data, the attacker can infer if this data access is a cache hit or miss, and hence, leak the information belonging to the victim process. The knowledge of such access patterns can leak critical information, e.g., secret key value in AES [3], even across sandbox in web browsers [1].

L1 data cache is a critical microarchitecture in modern processors, which must be protected from cache timing attacks [4]. Modifying cache architecture for partitioning [5] and randomized cache mapping [6] is costly for local private caches and is vulnerable to sophisticated exploits [7]. State-of-the-art software-based mitigation methods [8, 9, 4] mitigate timing attacks on private L1/L2 caches by performing *cache flush* upon preemption, process switch, and syscalls.[1] Cache flush is effective because it guarantees that the entire cache will be cleaned before the processor is switched from one process to another. After a cache flush, the hit/miss time difference can no longer be observed by the attacker's process, which enforces that the processes are isolated temporally. While cache flush is effective in protecting against cache timing attacks, it significantly increases the cache miss rates and cold cache effects, key factors affecting program performance [10]. Built upon cache flush, existing mitigation methods incur substantial program slowdown (more than 19% throughput overhead in Nginx in [9]). Moreover, cache flush is likely to incur greater performance overheads in future processors. In fact, the performance cores (named Firestorm) in the recently released Apple M1 chip, contain L1 data caches as large as 128 KB.[2]

In this paper, we propose a novel hardware/software method called fast selective flushing (FaSe), for countering contention-based cache timing attacks in L1 data cache. FaSe collectively leverages a customized cache microarchitecture and a specialized cache

---

[1]https://lwn.net/Articles/768418/

[2]https://en.wikipedia.org/wiki/Apple_M1

Flush Mechanism	Cost	Save
Naive	4	n/a
LLSF	2	0.5
CLSF ($|L_S| \geq 1$)	2	0.5
CLSF ($|L_S| = 0$)	0	1.0

(a) Naive flush   **(b) Line level selective flush**   **(c) Cache level selective flush**   **(d) Flush cost estimation**
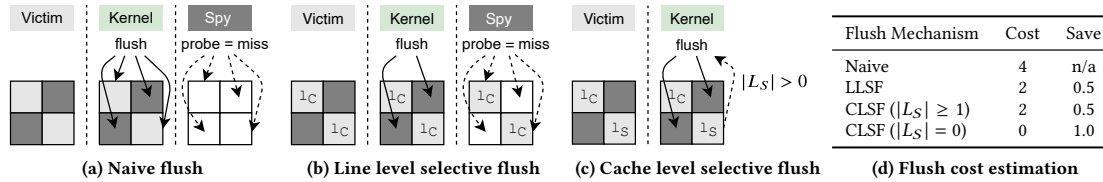
Figure 2: Brief illustration of FaSe's key idea.

flush instruction to create two new flush mechanisms, which significantly reduce the flush-based mitigation time cost. First, *line level selective flush (LLSF)* mechanism is proposed, which allows flushing of a subset of the cache lines, instead of all cache lines in the cache, while guaranteeing that the cache hit/miss time difference cannot be observed by the attacker. Second, *cache level selective flush (CLSF)* mechanism is proposed, which enables the cache flush to be guided (by user using a simple programming interface) and to perform "strategically" when necessary (when user-defined critical data is in cache), rather than always.

FaSe's hardware design extends the base processor architecture to support a specialized new cache flush instruction (scflush), as well as extends the L1 data cache with minimum additional state bits and control logic to perform both the LLSF and CLSF mechanisms. The software support for FaSe only takes a few lines of assembly code to: 1) integrate the specialized flush instruction in the software stack; and, 2) instrument program source code to define the critical data access. FaSe is implemented and evaluated on a RISC-V ISA processor, called the Rocket Core with Rocket Chip system-on-chip (SoC), on a Xilinx ZYNQ Ultrascale+ FPGA. The microbenchmark and OS evaluation results show that FaSe reduces 36% (user programs) and 42% (OS context switch latency) time overhead for flush-based mitigation on average. FPGA synthesis result shows that the hardware overhead of FaSe is negligible (less than 1% in tile with FPU excluded). The contribution of this research is summarized as follows.

- To the best of our knowledge, FaSe is the first selective flush-based mitigation for contention-based cache channel.
- A RISC-V processor with FaSe hardware and software modifications are implemented and validated on FPGA running user programs and Linux operating system.
- A contention-based cache timing attack evaluation, using Prime+Probe, is performed on the FaSe processor.

2 RELATED WORK

Hardware-based mitigation methods [5, 11] modify cache architecture by partitioning cache lines among processes, to counter cache timing channels. These partitioning-based methods result in significant cache under-utilization, and hence, are not practical for local private caches, which are small and time-shared. Other hardware-based methods [12, 6, 13] explore using randomized memory-to-cache mapping in cache hardware. These randomization-based caches typically incur substantial hardware modifications, such as adding a mapping table (1/8 to 1x of cache size) [12, 6], index table [13], and cryptography circuitry [6, 13]. In addition, software modifications are required in OS, including grouping tasks [6], extending page table [13], and additional kernel/user communications for passing unique process identification (PID) [13]. Randomized caches are prone to be breached if the attacker is provided with

sufficient trials [7]. **Software-based** spatial partitioning [14, 15, 16] are used to mitigate timing attacks on shared caches, such as Last-level caches (LLC). For private L1/L2 caches, due to the high cost of cache partitioning [4], software-based methods [8, 9, 4] are proposed to flush caches, when the processor core is about to run another user's thread or kernel thread. These methods typically require considerable software modification of the OS kernel (e.g., 1.4K LoC in [8]). **Compound flush instructions** are used in [17, 18, 19] to flush multiple on-core microarchitectural states, including L1 caches, TLB, branch prediction unit. This flush is a superset of naive cache flush, which flushes every cache line. These flush instructions are built on RISC-V processor variants, whose cache architectures and coherence protocols are different from the processor (Rocket Chip) used in this paper. For example, in [18], L1 data cache uses write-through policy, which does not require write-back during cache flush. Rocket Chip has a write-back L1 cache, which is more popular in modern processors. Compared to prior arts, FaSe is the first selective flush-based method to mitigate cache timing attacks. Such selective flushing reduces overheads significantly.

3 FASE SELECTIVE FLUSH

Threat Model. In this paper, contention-based, on-core L1 data cache timing attacks are targeted, which have been investigated in prior arts [6, 8, 9, 4]. The adversary is assumed to have user rights and mounts the spy process running on the same processor core. In this paper, Prime+Probe is used as the reference contention-based cache timing attack. Other levels of cache hierarchy are out of scope and assumed to be protected from cache timing attacks using existing methods discussed in Section 2.

Nominal flush-based mitigation system model. A system with flush-based mitigation executes cache flush in between user processes at OS kernel [4] or enclave exit [9] (i.e., flush point). Fig. 2 briefly illustrates existing and our proposals in comparison. As shown in Fig. 2a, in a flush-based system, the CPU events of one CPU core can be viewed as interleaved cache flush events and compute/service events. Naive cache flush mechanism (Fig. 2a) *flushes all cache lines of the local private data caches*. Such a naive mechanism can either be implemented in software (e.g., run dc_cisw instruction in for loop in ARMv8 ISA) or hardware (e.g., [17, 18]), essentially in a for-loop, which traverses, invalidates, and cleans all cache lines in the cache. After a flush event, all the following memory accesses will be cache-miss and take a much longer time.

Our proposal 1: not every cache line requires a flush. In contrast to the naive flushing, the first method proposed here (called *Line Level Selective Flush* or **LLSF**), only flushes the cache lines that were not updated in the current time slice. The intuition behind this method is that the updated cache lines will not result in a hit with the spy process, and thereby reducing the number of lines being flushed (this significantly reduces the flushing time and the

time overhead associated with flushing). Like naive flushing, this method mitigates cache timing attacks and covert channels. As shown in Fig. 2b, at one flush point, let the process executed before this flush point be noted as current process τ_{cur} (victim in Fig. 2b), while the process executed after this flush point be noted as next process τ_{nxt} (spy in Fig. 2b). Let's denote the number of total cache lines as a set L and the number of dirty cache lines as L_D ($L_D \subseteq L$). There is a subset of cache lines L_C ($L_C \subseteq L_D$), which have been placed into the cache by τ_{cur}. Flushing the complementary of these cache lines, noted as $L_X = L_D \setminus L_C$, is sufficient to let τ_{nxt} observe cache miss at every cache line $l \in L$. Therefore, τ_{cur} and τ_{nxt} are temporally isolated while fewer cache lines need to be flushed. A decrease in flushed cache lines leads to the reduction of flush cost and alleviates the cold cache penalty (elaborated later).

Our proposal 2: not every flush point needs a cache flush. In the second proposed method, *cache level selective flush* (**CLSF**) the program section that requires protection is instrumented (for example with pragmas), and only if that part of the program brings data into the cache, then the cache is flushed. The intuition behind the second method is that there are many parts of the program which do not require temporal isolation for countering cache timing attacks, and experience shows a program executes in multiple OS time slices, and only some of these deal with protected data, such as an encryption key. This method mitigates the cache timing attacks, but will not fully counter cache covert channels, if the covert channel is implemented using the program parts that are not instrumented. Fig. 2c shows the brief idea of CLSF. Let us assume a program only has limited parts (e.g., T-table look-up in AES [20]), which are security-critical. We define the data accessed in security-critical parts of a program as security-critical data. At one flush point, if τ_{cur} (victim in Fig. 2b) has not accessed any security-critical data in any cache lines (noted as L_S), temporal isolation against τ_{nxt} at this point is unnecessary. Therefore, this cache flush can be nullified to trade flush cost with acceptable security degradation.

Flush cost-saving analysis. The time cost of one cache flush event at time t_i is mainly determined by the number of dirty cache lines L_D ($L_D \subseteq L$) and the number of total cache lines L. This relation can be expressed as $P_i = \alpha \cdot |L_D^i| + \beta \cdot |L|$ where the first term represents the time cost for invalidating and cleaning dirty cache lines, and the second represents the time taken to visit every cache line like a for-loop at flush. α stands for the time penalty of invalidating and cleaning one dirty cache line, which is mainly the time taken for maintaining data coherency, such as write-back. β denotes the time taken for traversing one cache line. α is order-of-magnitude larger than β. Flush cost is proportional to $|L_D|$. Moreover, cache flush induces *cold-cache penalty*, equivalent to the "start-up effects" [10]. For a period of time, the total time cost of a flush-based system is $\sum_i P_i + P_{CC}, i \in \{1, 2, \dots, N\}$ where N is the number of total flush events and P_{CC} is the lump sum of time penalties from cold cache effects. Fig. 2d shows a brief quantitative comparison (based on flushed cache lines) between FaSe and naive flush-based mitigation, corresponding to the scenarios in Fig. 2b and Fig. 2c. The flush cost is estimated as cache lines flushed. With LLSF, flush cost can be substantially reduced, if there are sizable L_C cache lines that can avoid flushing. In this example, two out of four cache lines are saved from flushing, which leads to proportional amount of time cost saving. With CLSF, if the condition meets, i.e., $L_S = \emptyset$, the flush
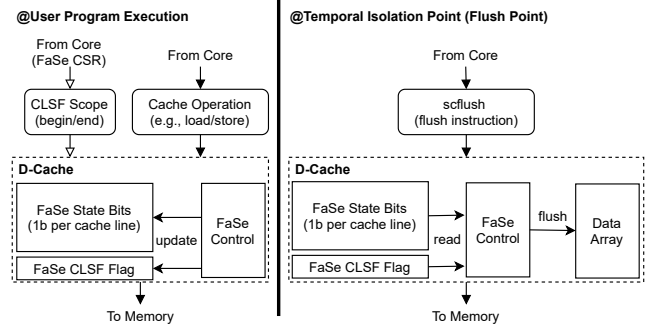
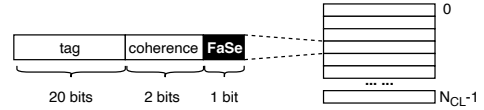

Figure 3: FaSe system overview.

Figure 4: FaSe cache's tag array.

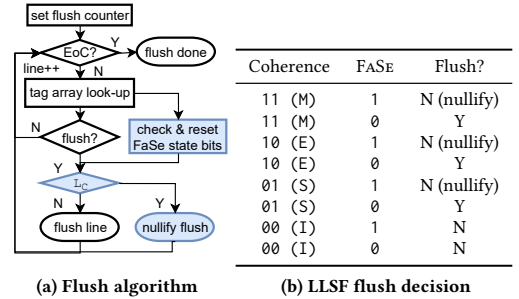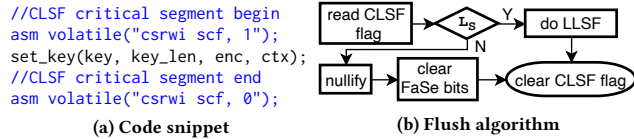(a) Flush algorithm   (b) LLSF flush decision

Figure 5: LLSF flush mechanism. EoC: end of cache lines.

can be nullified and leads to a complete save of flush cost at this flush point.

4 FASE DESIGN: THE CASE FOR RISC-V

Fig. 3 shows the overview of FaSe design, which realizes both LLSF and CLSF. FaSe design includes hardware modification to L1 D-cache, a cache flush instruction (scflush - which is an ISA extension), and a control/status register (CSR), denoted as csr.scf. FaSe works in two phases, *user program execution*, and *flush instruction execution*. FaSe extends L1-D cache with 1) FaSe state bits, 2) one CLSF flag status bit, and 3) FaSe control. Fig. 4 depicts FaSe cache modification with respect to the cache metadata (stored in tag array) in the RISC-V Rocket core processor. FaSe adds one FaSe state bit for each cache line. FaSe state bits are stored together with other cache metadata, such as tag bits and coherence bits, which are one-on-one mapped to cache lines. In a 4-way 64-set data cache, the total number of FaSe state bits is 1×256. Bit value "1" means this cache line has been accessed in the current process time slice (since the last cache flush).

LLSF control mechanism has two parts, corresponding to FaSe's two phases. During user program execution, when CPU core accesses the L1 cache, using instructions such as load or store, FaSe state bit is updated to "1". This update is executed simultaneously when coherence bits are updated for the corresponding cache line by the native cache-control hardware. Fig. 5 illustrates LLSF algorithm when flush instruction is executed. Fig. 5a shows a flowchart

```
//CLSF critical segment begin
asm volatile("csrwi scf, 1");
set_key(key, key_len, enc, ctx);
//CLSF critical segment end
asm volatile("csrwi scf, 0");
```


(a) Code snippet   **(b) Flush algorithm**

Figure 6: CLSF flush mechanism.

of the steps in cache flush. Fig. 5b depicts how LLSF decision is made based on coherence bits (assuming RISC-V's MESI-like coherence protocol) and FaSe state bit of one cache line. In Fig. 5b, Row 2, 4, and 6, are the cases where cache line flush is not necessary and hence avoided. As shown in Fig. 5a, when the flushing instruction is executed, LLSF control mechanism sets the flush counter (from 1 to number of cache lines) and performs the following steps for each cache line : ① read FaSe state bit and coherence bits of the current cache line from the tag array, using flush counter value as the tag array index; ② check whether the coherence state of this cache line determines that this cache line should be flushed, and whether FaSe state bit (shown in Fig. 5b indicates this line flush is unnecessary; ③ reset FaSe state bit of current cache line; ④ if FaSe control determines that this cache line should be flushed (based on Fig 5b), such as the situations in Row 3, 5, 7 in Fig. 5b, cache line flush takes place, otherwise, this cache line flush is nullified; and, ⑤ increment flush counter, and if the counter value reaches maximum value (meaning last cache line has been processed), cache flush is finished.

In addition to LLSF, FaSe CLSF further extends the CPU core with FaSe CSR register (denoted as csr.scf) and L1-D cache with one CLSF flag status bit. FaSe CSR is one-bit wide and programmable by the user. FaSe csr.scf allows users to mark the critical segment in the program code. When csr.scf's value is "1", the memory-related instructions during this time are considered critical and are protected. When csr.scf's value is "0", instructions executed are no longer considered critical. Fig. 6a depicts an example code snippet in AES program where CLSF critical segment is marked (assuming RISC-V ISA). In this code snippet, the user adds two additional lines of assembly code (inline assembly in C) to put the set_key function into CLSF critical segment. Before calling the set_key function, csr.scf is written with "1" by csrwi instruction (In RISC-V ISA, one CSR register can be written using CSR access instructions, such as csrwi and csrw instructions). After returning from the set_key function, csr.scf is reset to "0" by csrwi instruction. In L1 D-cache, CLSF flag status bit is added to indicate if any cache line has been used in the user-defined CLSF critical segment in current process time slice since last cache flush.

CLSF control mechanism has two parts with respect to user program execution and flush instruction execution. During user program execution, CLSF looks at the signal value from FaSe CSR csr.scf, when cache operation is issued from CPU core to L1 D-cache. If csr.scf's value is "0", the current cache operation is treated as non-critical. In this case, CLSF control does not do anything. If csr.scf's value is "1", the current cache operation is treated as critical. In this case, if any cache line's coherence state bits are updated due to this cache access, the CLSF flag status bit is asserted to "1". Interruptions and exceptions could occur during a CLSF critical segment. To handle such a situation, when the CPU switches from the current process to kernel space, csr.scf is saved as context and reset. When this process is switched back, csr.scf is also restored

as a part of the process context. This procedure incurs negligible additional code lines (less than 10 lines of assembly). As shown in Fig. 6b, when scflush is executed, CLSF control mechanism has the following steps: ① read and check the CLSF flag value; ② If CLSF flag value is "1", CLSF control nullifies this cache flush and clears FaSe state bits. If CLSF flag value is "0", like LLSF, CLSF can examine FaSe state bits and selectively flushes the cache lines, which are necessary; and, ③ At last, clear the CLSF flag.

5 EXPERIMENT AND RESULTS

FaSe is implemented on the RISC-V Rocket Core processor. Rocket Core is part of the Rocket Chip SoC generator [21]. In our experiment, we use the 64-bit generic RISC-V ISA, RV64GC. The L1 data cache is a 32-KB 8-way set-associative cache containing 64-byte cache blocks. We used an FPGA build[3] of Rocket Chip and ported this build to Xilinx Ultrascale+ ZCU102 FPGA board. The main memory is a 4-GB DDR4 SODIMM manufactured by Micron fitted on to the ZCU102. The FPGA synthesis tool used for synthesis is Vivado 2017.1. Our experiment mainly evaluates FaSe's LLSF, given CLSF requires user and domain knowledge to set the protection scope. To showcase the efficacy of FaSe's CLSF, we performed a case study on AES encryption. We compare our system against two reference systems in our experiments: 1) the original RV64GC Rocket Core, which is the **baseline** (without mitigation); and, 2) the RV64GC Rocket Core augmented with hardware-supported naive cache flush (cache-flush with hardware for-loop similar to [17, 18]),[4] which is denoted as the **naive** method or naive system.

In **Section 5.1**, to observe how much FaSe can reduce the OS overhead, we use context switch latency test, lat_ctx, in LMBench 3.0 [22] to evaluate overhead caused by FaSe and naive systems. We modified the Linux kernel 4.20 by adding a cache flush instruction in the context switch routine for deploying FaSe and the naive systems. The baseline processor runs the native Linux kernel without modification. Naive and FaSe systems run the modified Linux kernels. In this experiment, FaSe enables LLSF. lat_ctx test runs with varied process sizes (by default from 0 to 64 KB). When process size is 0 KB, the process does nothing except pass the token on to the next process. Non-zero process size means that the process does some work before passing on the token. lat_ctx test considers a range of process numbers (by default from 2 to 96) and has context switch latency measured for each process number. Context switch latency is the time needed to save the state of one process and restore the state of another process. In **Section 5.2**, we evaluated the user program performance using programs from MiBench benchmark suite [23]. We use RISC-V proxy kernel (riscv-pk), a basic application execution environment, which directly supports measuring cycle and instruction count. We modified riscv-pk to adopt the FaSe method. In each program, temporal isolation takes place at user/kernel switch, which is for syscalls and exception/interruption handling. In **Section 5.4**, for security evaluation, we created a PRIME+PROBE cache timing attack based on the codebase from [24]. We implemented the victim as a function in kernel space within riscv-pk. The attacker process prepares the attack and uses a special *syscall* to switch to the victim process. After the victim finishes

---

[3]https://github.com/ucb-bar/fpga-zynq
[4]Similar to CFLUSH.D.L1 https://github.com/chipsalliance/rocket-chip/pull/1712

Table 1: Context switch latency (microseconds) across three systems, process sizes (SZ in kilobytes), process numbers (P·)

System	SZ	P2	P4	P8	P16	P24	P32	P64	P96
Baseline	0	40.5	40.3	69.6	86.4	101.3	102.3	108.3	109.5
	4	51.0	88.0	119.6	139.3	138.0	142.8	144.2	144.4
	8	65.5	131.5	155.0	164.5	168.3	172.8	175.7	174.3
	16	140.5	202.3	213.3	225.8	228.9	230.2	228.1	229.4
	32	197.5	222.8	232.1	251.2	247.2	251.1	248.6	247.5
	64	150.0	177.0	194.8	191.7	185.8	188.8	186.5	186.4
Naive	0	154.5	154.3	152.5	161.6	168.0	170.5	179.4	177.4
	4	182.5	180.0	178.0	189.8	193.4	198.3	202.2	200.7
	8	211.0	212.0	206.1	222.4	231.4	235.0	232.8	232.3
	16	256.0	253.8	257.3	270.5	276.3	287.1	282.8	280.9
	32	276.0	274.0	275.9	307.3	309.3	309.8	308.9	307.1
	64	183.0	205.0	235.4	234.7	228.7	229.2	227.4	229.2
FaSe	0	122.5	122.8	121.6	123.9	130.5	134.9	140.4	140.5
	4	161.0	160.3	155.0	163.7	167.9	172.4	175.4	172.5
	8	186.5	187.0	186.9	196.1	200.4	208.6	207.6	208.3
	16	212.5	218.0	216.1	234.2	240.1	241.4	242.0	241.7
	32	247.0	251.5	257.5	273.1	270.7	275.6	272.8	271.7
	64	171.0	195.3	223.8	214.7	210.3	212.2	211.6	210.2
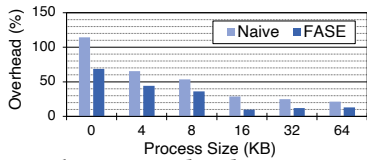

Figure 7: lat_ctx overhead vs. process sizes.

the task in kernel space, the attacker process switches back and probes the cache. In comparison to mounting Prime+Probe attack on OS, this setup leads to a faster Prime+Probe attack, since the code executed between victim and spy is much less.

5.1 Linux Kernel Results

Table 1 shows the context switch latency of the baseline, naive, and FaSe systems regarding different process sizes and numbers. Column 1 and Column 2 shows the system names and program sizes in KB. Columns 3 to 8 are the context switch latency in microseconds. Overall, smaller process sizes lead to lower context switch latency, since cache footprint is affected by the process size. The lowest context switch latency is witnessed when the baseline system is used with 0 KB process size, because it does not enforce flush of the cache and cache footprint is minimum. The highest context switch latency is found when the naive system is used with 32 KB process size, because full cache flush is enforced, and the cache footprint is maximum in this case. Process size of 64 KB is larger than cache size (32 KB) and shows smaller context switch time. This phenomenon is because the behavior of array accumulation in lat_ctx test and cache collision together lead to a "friendly" cache footprint at the context switch point.

Fig. 7 depicts the overhead of context switch latency as a function of process size. For each process size, the lat_ctx overhead is calculated from the geometric mean of the lat_ctx overhead among the process numbers. In general, when process size is equal to or smaller than cache size, the overhead decreases as process size increases. We also found process size affects context switch latency differently in FaSe and naive systems. For FaSe system, 16 KB process size results in the lowest overhead, followed by 32 KB and 64 KB. For the naive system, 64 KB process size has the lowest overhead, while 32 KB and 16 KB follow. This result is because FaSe's LLSF mechanism can save half the cache from flushing if the content in the cache is equally owned by two processes. FaSe
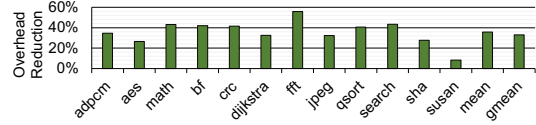
Figure 8: Execution time overhead reduction in percentage of FaSe on top of naive system across MiBench programs.

(a) Call graph & CLSF scopes   (b) Execution time versus CLSF schemes
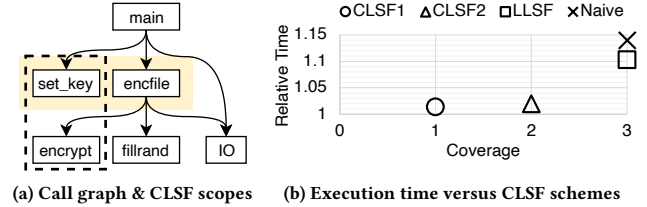
Figure 9: FaSe CLSF AES case study. Relative execution time is normalized on baseline AES. CLSF1: set_key and encfile in yellow background. CLSF2: functions in dashed box in (a).

shows the largest overhead reduction in percentage in comparison to the naive method when process size is 16 KB. On average, LLSF reduces context switch latency overhead of naive system by 42%. In comparison to the naive method, LLSF reduces 66% context switch latency overhead in the best case (when process size is 16 KB). In summary, Linux kernel results demonstrate that FaSe can significantly reduce mitigation overhead during OS execution.

5.2 Microbenchmark Results

Fig. 8 shows how much overhead due to flush-based mitigation in the naive method is reduced by using FaSe (LLSF alone). Overall, FaSe reduces the execution time overhead effectively by 36% on average (geometric mean is 33%). Since mibench programs mostly target embedded applications, some programs have few syscalls and/or a small cache footprint. Hence, the overhead in these programs can be very small, such as susan. For other programs, FaSe reduces the overhead significantly by around 30% to 56% from the naive method. FaSe's average time overhead is 7.6% (geometric mean is 3.4%).

CLSF AES case study. AES (also called rijndael), which is a security-critical application, from MiBench, is used to observe the time saving of FaSe's CLSF. Fig. 9 presents the schemes and results of this case study. Fig. 9a shows the call graph, which consists of the major functions in AES encryption. Based on this call graph, we created two choices of CLSF critical segments. First, CLSF1 denotes CLSF covering set_key function and the main while-loop in encfile function which calls encrypt. This scope is denoted as "1" on the X-axis in Fig. 9. Second, CLSF2 covers set_key function and encfile function. CLSF2 uses a larger scope as encfile function is the parent function of encrypt. This scope is denoted as "2" on the X-axis in Fig. 9. We compared CLSF1 and CLSF2 to LLSF and naive methods. The coverage of LLSF and naive methods is of the full program, denoted as "3" on the X-axis in Fig. 9. Fig. 9b compares the relative execution time of these schemes. The relative execution time is calculated by normalizing the baseline system. It can be seen that FaSe CLSF can reduce execution time greatly (to about 1% overhead) from around 1.1 (equivalent to 10% time overhead, LLSF) and 1.15 (equivalent to 15% time overhead, naive). Among the CLSF schemes, CLSF2 takes slightly more time than CLSF1, due to larger critical segment coverage.

Table 2: FPGA utilization across hierarchies.

Subject	Resource	Tile	Core	DCache	Frontend	Tile[-FPU]
Baseline	LUT	30953	5451	2820	4844	14855
	F/F	13916	2004	2596	4751	10125
FaSe	LUT	30997	5457	2877	4861	14912
	F/F	13991	2067	2607	4751	10198
Overhead	LUT	0.1%	0.1%	2.0%	0.4%	0.4%
	F/F	0.5%	3.1%	0.4%	0.0%	0.7%

(a) Baseline (b) Naive (c) FaSe
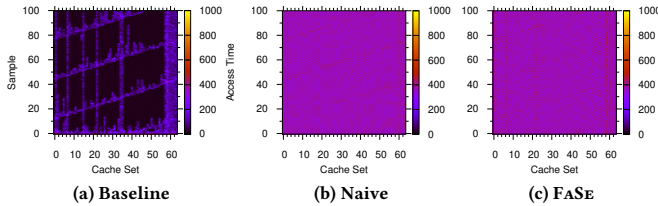
Figure 10: Prime+Probe attack results on the baseline system, naive mitigation, and FaSe mitigation.

5.3 Hardware Results

To understand the hardware cost of FaSe, FaSe processor is compared to the baseline processor. FaSe processor has both LLSF and CLSF implemented. Overall, the only overhead of FaSe in resource utilization is found in FPGA look-up tables (LUTs) and flip-flops (F/Fs). The block RAM utilization is unchanged, because FaSe only adds a few hundred bits in the tag array in the cache. The max clock speed (clock frequency) is unchanged as the baseline processor, since FaSe's hardware does not change the critical timing path.

Table 2 shows the FPGA resource utilization overhead of FaSe. Since the entire SoC includes many large uncore components, a direct comparison at the SoC level will show negligible differences. To make a finer comparison, we focus on the overhead at hierarchies at and beneath the tile. Note that a tile (RocketTile) is a processor core and its local private resources (caches, TLBs, etc.). In the tile, the resource utilization of the key components, core, L1 D-cache (DCache), and frontend (I-cache and other instruction fetch components) are compared. Because FPU is quite a large component in tile (and remains unchanged), a comparison is made of the tile resource utilization with FPU's utilization removed (written Tile [-FPU]). As shown in Table 2, FaSe increases 0.1% LUTs and 0.5% F/Fs. Without the FPU, this overhead becomes 0.4% LUTs and 0.7% F/Fs. FaSe increases LUTs in the cache by 2%. The 3% F/F overhead in the core is caused by cross-boundary optimization of Vivado synthesis. These additional F/Fs are mostly from the FaSe hardware in the cache.

5.4 Security Results

Fig. 10 uses heat maps to depict the Prime+Probe (a representative contention-based cache timing attack) results of the three systems (the baseline system without protection in Fig. 10a, the naive mitigation system in Fig. 10b, and the proposed FaSe mitigation system 10c). FaSe LLSF and CLSF (covering victim function) showed equivalent effects in this experiment. In each figure, the X-axis is the cache set from 0 to 63. Y-axis is the sample number. One hundred samples are shown in these figures, which are sufficient to illustrate the afforded protection. The Z-axis (heat color) stands for the access time in clock cycles, taken for probing the cache sets. As a result, without protection, the baseline system's

victim cache access patterns can be seen (below 100 cycles shown in deep purple and black). As shown in Fig. 10b and 10c, on naive and FaSe systems, Prime+Probe observes misses in all cache sets (more than 100 clock cycles, similar colors across cache sets). This result shows that FaSe's flush mechanism is effective for countering contention-based on-core cache timing attacks.

6 CONCLUSION

In this paper, we have presented a novel flush-based method, FaSe, to mitigate contention-based cache timing attacks. FaSe leverages an ISA extension and modified cache microarchitecture to minimize flush cost of the mitigation. Our experiment shows that FaSe can mitigate contention-based timing attacks with negligible hardware cost while reducing time overhead by 36% for user programs and 42% for OS context switch, in comparison to naive method.

ACKNOWLEDGEMENT

REFERENCES

[1] Yossef Oren et al. 2015. The spy in the sandbox: practical cache attacks in javascript and their implications. In *CCS '15*.
[2] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of aes. In *CT-RSA'06*, 1–20.
[3] Daniel J Bernstein. 2005. Cache-timing attacks on AES. Technical report.
[4] Qian Ge et al. 2019. Time protection: the missing os abstraction. In *EuroSys '19*, 1:1–1:17.
[5] Leonid Domnitser et al. 2012. Non-monopolizable caches: low-complexity mitigation of cache side channel attacks. *TACO*, 8, 4, (January 2012).
[6] Moinuddin K. Qureshi. 2018. Ceaser: mitigating conflict-based cache attacks via encrypted-address and remapping. In *MICRO '18*, 775––787.
[7] Wei Song et al. 2021. Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it. In *S&P '21*, 955–969.
[8] Yinqian Zhang and Michael K. Reiter. 2013. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS '13*.
[9] Oleksii Oleksenko et al. 2018. Varys: protecting sgx enclaves from practical side-channel attacks. In *USENIX ATC '18*, 227–239.
[10] A. Agarwal, J. Hennessy, and M. Horowitz. 1989. An analytical cache model. *ACM Trans. Comput. Syst.*, 7, 2, (May 1989).
[11] Mengjia Yan et al. 2017. Secure hierarchy-aware cache replacement policy (sharp): defending against cache-based side channel atacks. In *ISCA '17*.
[12] Zhenghong Wang and Ruby B. Lee. 2008. A novel cache architecture with enhanced performance and security. In *MICRO '08*, 83–93.
[13] Mario Werner et al. 2019. Scattercache: thwarting cache attacks via cache set randomization. In *USENIX '19*. (August 2019), 675–692.
[14] Taesoo Kim et al. 2012. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *USENIX '12*, 189–204.
[15] Fangfei Liu et al. 2016. Catalyst: defeating last-level cache side channel attacks in cloud computing. In *HPCA '16*, 406–418.
[16] Xiaowan Dong et al. 2018. Shielding software from privileged side-channel attacks. In *USENIX '18*. (August 2018), 1441–1458.
[17] Thomas Bourgeat et al. 2019. Mi6: secure enclaves in a speculative out-of-order processor. In *MICRO '19*, 42–56.
[18] Nils Wistoff et al. 2020. Prevention of microarchitectural covert channels on an open-source 64-bit RISC-V core. *CoRR*, abs/2005.02193.
[19] Tuo Li et al. 2020. SIMF: single-instruction multiple-flush mechanism for processor temporal isolation. (2020). https://arxiv.org/abs/2011.10249.
[20] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient cache attacks on aes, and countermeasures. 23, 1.
[21] Krste Asanovic et al. 2016. The Rocket Chip Generator. Technical report UCB/EECS-2016-17. EECS Department, University of California, Berkeley.
[22] Larry McVoy and Carl Staelin. 1996. Lmbench: portable tools for performance analysis. In *ATEC '96*, 23–23.
[23] M. R. Guthaus et al. 2001. Mibench: a free, commercially representative embedded benchmark suite. In *WWC '01*, 3–14.
[24] Yuval Yarom. 2016. Mastik: a micro-architectural side-channel toolkit. (2016). https://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf.