A Distance-Based Side-Channel Attack in Non Uniform Cache and Possible Defenses

Farabi Mahmud
Texas A&M University
farabi@tamu.edu

Sungkeun Kim
Texas A&M University
ksungkeun84@tamu.edu

Harpreet Singh Chawla
Texas A&M University
harpreetsc@tamu.edu

Pritam Majumdar
Texas A&M University
pritam2309@tamu.edu

Jiayi Huang
University of California, Santa Barbara
jyhuang@ucsb.edu

Chia-Che Tsai
Texas A&M University
chiache@tamu.edu

EJ Kim
Texas A&M University
ejkim@tamu.edu

Abdullah Muzahid
Texas A&M University
Abdullah.Muzahid@tamu.edu

***Abstract*—For a distributed last level cache (LLC) in a large multicore chip, the access time to one LLC bank can significantly differ from that to another. The disparity in access time is due to the different physical distances to the target LLC slices. In this paper, we demonstrate the possibility of exploiting such a distance-based side channel, by timing a vulnerable version of AES decryption and extracting part of the secret keys. We introduce several techniques to overcome the challenges of the attack, including using multiple attack threads to ensure LLC hits of the vulnerable memory locations and to time part of the decryption function.**

We further propose CAMOUFLAGE, an efficient, architectural defense for the proposed distance-based side-channel attack. At runtime, when a potentially leaking memory instruction is executed by a victim function, CAMOUFLAGE uses a combination of jitter and bypass mechanisms to eliminate any LLC hit time difference due to the distance and thereby, prevent the attack. We evaluate two versions of CAMOUFLAGE - CAMOUFLAGE$_{JITTER}$ and CAMOUFLAGE$_{BYPASS}$using the Gem5 simulator with PARSEC and Rodinia benchmarks and show that they incur performance overheads of 14.14% or none over the baseline.

I. Introduction

Large-scale multicores are increasingly prevalent due to the shrinkage of process technology. Systems with 64 or more cores have become the backbone of cloud computing or high-performance computing. AMD, Intel or ARM has 64 [1], 72 [3] and 80 core [17] processors respectively in the market. To meet the demand of so many cores, large-scale multicores are equipped with a large last level cache (LLC). AMD Ryzen comes with 256 MB LLC whereas Intel Xeon Phi 7200 series has 36 MB LLC. Due to the physical and manufacturing limitations, such LLCs are distributed over multiple banks connected through a network-on-chip (NoC) to reduce access latency and improve core isolation. However, with a distributed LLC, a core may incur different latency when accessing banks of different physical distances. This type of architecture is referred to as a Non-Uniform Cache Access (NUCA) architecture. In this paper, we set out to investigate whether distance-induced non-uniform latency to NUCA caches can lead to security vulnerabilities and if so, how we can defend against those.

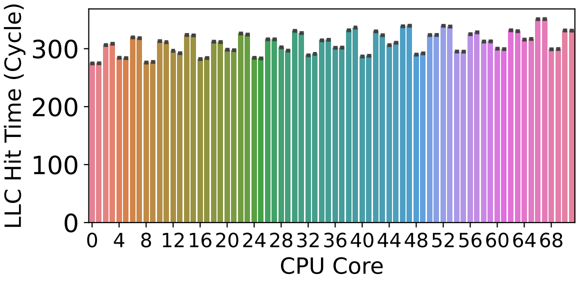

Fig. 1: LLC hit time measured in cycles when accessing the same physical address (`0x1000000000`) from different cores in the Intel Xeon Phi 7290 CPU. The latency numbers are averaged over 10,000 samples.

A. A Distance-Based NUCA Side-Channel Attack

Cache side-channel attacks [4], [10], [31], [43], [50], [52], [54], [59], [61] are proven to be a prominent threat to data security, especially during the past few years. The common form of cache side-channel attacks involves timing the cache access latency which depends on the state of the target cache lines. Take Flush+Reload [21], [22], [27], [62], [68], [69], [71] for example. The attacker tries to observe the target shared lines being accessed by the victim program, by detecting whether reloading the line incurs a cache hit or miss. Recent work investigates how network contention in NUCA architecture can be utilized to create timing differences and subsequently, a covert and side-channel [15], [63].

In this paper, we demonstrate a vastly different cache side-channel attack that relies on difference in physical distance among the LLC banks. Algorithm 1 shows the pseudo-code of a potential attack using this distance-based NUCA side-channel. In this example, an address is determined by the victim according to a specific bit of a secret. The address can be mapped to the LLC bank of the nearest core when the bit is 0, or the farthest core when the bit is 1. Therefore, by timing the access latency, an attacker can infer the secret bit. To further illustrate this scenario, Figure 1 shows access latency to the same address from different cores. We collected

Algorithm 1: Psuedocode of a victim function which is vulnerable to a side-channel based on the difference of LLC access time due to distance.

```
Input: BitMask
Data: Secret
if (Secret & BitMask) == 1 then
    $Addr = Addr_{Near}$ mapped to the LLC bank of the nearest core;
else
    $Addr = Addr_{Far}$ mapped to the LLC bank of the farthest core;
Load($Addr$);
```

these latency numbers from the Intel Xeon Phi 7290 CPU. This CPU model belongs to Intel's Knights Landings line and has a many-core architecture with at least 64 cores (the CPU we tested has 72 cores). The figure shows that the LLC hit latency for the same address has a range between 280–350 cycles, and this pattern is generally stable across cores. Ideally, if the attacker can measure the access latency in the victim code shown in Algorithm 1, he/she will be able to guess the secret bit by telling whether the addresses fall into a near or far LLC bank. We demonstrated the attack in Intel Xeon Phi 7290 using AES code. We addressed several challenges for the attack, namely (i) the overlapping of memory accesses to LLC, and (ii) the difficulty of timing only a portion of the decryption operation. Our proof-of-concept (POC) attack code is able to accurately extract the lower 4 bytes of any AES key with 4,000 decryption trials using a sequence of random plaintexts.

B. Mitigating a Distance-Based NUCA Side-Channel

To defend against a timing channel such as the distance-based NUCA channel, one simple strategy is to make the latency of all relevant operations constant so that the attacker cannot infer any access pattern from the time differences. Based on this observation, we propose two strategies– CAMOUFLAGE$_{\text{JITTER}}$ and CAMOUFLAGE$_{\text{BYPASS}}$. Collectively, we refer to the proposed strategies as CAMOUFLAGE.

For CAMOUFLAGE$_{\text{JITTER}}$, our approach is to make all operations as slow as the slowest operation in the system; that is, if we make the latency of accessing *any* LLC bank to be as long as the longest latency of accessing the farthest bank, the system will not exhibit any timing difference. To make any LLC bank latency equal to the longest latency (say, T_{worst}), CAMOUFLAGE$_{\text{JITTER}}$ adds some jitter with the original access latency (say, T_{orig}). An appropriate amount of jitter (i.e., $T_{worst} - T_{orig}$) is added right before the LLC bank sends the requested data back to the core. CAMOUFLAGE$_{\text{JITTER}}$ is simple and intuitive but hurts performance as it slows down LLC accesses significantly. To reduce the performance overhead of our defense, we must not make the constant latency of the system equal to T_{worst}. Therefore, we propose CAMOUFLAGE$_{\text{BYPASS}}$.

To design CAMOUFLAGE$_{\text{BYPASS}}$, we make the following observation - *instead of making every LLC access to have the latency T_{worst}, we can reduce the worst case latency to much lower latency and then, make every LLC access latency equal to that.* Luckily, there have been many studies on reducing packet latency in NoC designs. One class of techniques, called Router Bypass [14], [38], [39], [41], [44], [55], [67], [73], opens up an opportunity for speeding up the accesses to the farthest banks, thereby reducing the worst case latency. For instance, by applying the bypass technique to the furthest bank, we can reduce T_{worst} latency to a much lower latency T_{low}. CAMOUFLAGE$_{\text{BYPASS}}$ eliminates LLC access latency difference by making every latency equal to T_{low}. To achieve that, CAMOUFLAGE$_{\text{BYPASS}}$ works by adding jitter to accesses in nearby LLC banks (since access latency of nearby banks is lower than T_{low}) and a combination of bypass and jitter (if needed) to accesses in the distant LLC banks. Jitter and bypass mechanism is applied by modifying the NoC router hardware. This is the *first* work that uses router bypass mechanism for a security purpose.

C. Contributions

We make the following contributions:

- We demonstrated a *new* distance-based side-channel attack on NUCA on an Intel Knights Landing CPU against a vulnerable AES implementation. We addressed several challenges related to memory ordering and timing, and can accurately extract the lower 4 bytes of the AES key with only 4,000 decryption trials using a sequence of random plaintexts.
- We propose two defenses - CAMOUFLAGE$_{\text{JITTER}}$ and CAMOUFLAGE$_{\text{BYPASS}}$, to ensure constant-time LLC accesses with a combination of router bypass and jitter. *This is the first use of router bypass mechanisms for security.*
- We implemented our proposed defenses in the Gem5 [8] architectural simulator. We experimented with widely used PARSEC [7] and Rodinia benchmarks [13]. Our results indicated that on average, with the minimal NoC hardware modification, CAMOUFLAGE$_{\text{JITTER}}$ adds 14.14% runtime overhead to the execution time of a baseline vulnerable system while CAMOUFLAGE$_{\text{BYPASS}}$ does not cause any runtime degradation at all (instead improves performance by an average of 6.4%).

II. Background and Related Work

In this section, we explain the background of Non Uniform Cache Access (NUCA) Architecture, Network-on-Chip (NoC), router bypass, and cache side-channel attacks.

A. Non Uniform Cache Access Architecture

With the increasing demand for bridging the speed disparity between the CPU and the main memory, cache capacity keeps increasing to improve cache hit rate [1], [29], [48]. In addition, increasing bandwidth demand and limitation in number of ports result in physically separated banked last level caches (LLC) that are connected by a network-on-chip (NoC). Such a physical layout introduces the paradigm of non-uniform cache access (NUCA) architecture as cache accesses to different LLC banks from the same core can have non-uniform access latency [32]. Among several interconnection technologies, ring bus design in Intel Nehalem supports up to

8 core Xeon processors [36], which recently upgraded to mesh interconnect [26], [35] for Intel's Skylake-SP and Skylake-X processors that have 28 cores [47].

B. Network-on-Chip and Router Bypass

In a network-on-chip (NoC), Network Interfaces (NIs) connect the core/cache tile to the network that is formed by a set of routers in a certain topology, such as mesh. In general, a packet travels from the source tile's NI through the network routers to the destination tile's NI. A router consists of several pipeline stages, including routing computation, virtual channel allocation, switch arbitration, and switch traversal, followed by link traversal. Therefore, packets need to travel hop-by-hop basis while going through a complex router pipeline.

To reduce packet latency, router bypassing has been proposed in various forms in many NoC designs [38], [40]. Express virtual channel (EVC) is used to virtually bridge two nodes across multiple bypass routers so that a packet can traverse a bypass router in one cycle without virtual channel and switch arbitration, thereby effectively reducing router pipeline stages [40]. Following EVC, SMART [38] was proposed to enable a single-cycle data path from the source to the destination. So a packet can use SMART to physically bypass multiple routers in one cycle to further reduce latency. In this work, we leverage the bypass mechanism to selectively regulate packet latency for mitigating NUCA attacks. This is the *first* use of bypass mechanism for a security issue.

C. Cache Side-Channel Attacks

Side-channel attacks in hardware exploit observable changes to steal a secret. Hit/Miss or access latency of any memory structure such as TLB, Cache, and DRAM are examples of observable changes. The attacker of a side channel correlates the observable changes to the secret. There are side-channel attacks and countermeasures at different levels of the system. Cache side-channel attacks are one of the most common methods for attackers. Most cache side-channel attacks are categorized based on how they prepare the transmission and how they measure the changes in the cache.

Prime+Probe [4], [10], [31], [43], [50], [52], [54], [59], [61]: The attacker installs data to cache lines and observes whether those lines are evicted by the victim. The attacker can infer the victim's behavior by monitoring such evictions.

Flush+Reload [21], [22], [27], [62], [68], [69], [71]: Flush+Reload is opposite of Prime+Probe in that the attacker flushes the cache lines using the CLFLUSH instruction and checks if cache hit happens later. This attack requires that both the attacker and the victim share the memory to access the same cache line.

Evict+Reload [21]: Evict+Reload and Flush+Reload are similar as they evict cache lines during the preparation phase. The attacker installs many dummy data to evict the targeted cache lines and later checks if the dummy cache lines are evicted by the victim.

Evict+Time [26]: The attacker runs the victim programs multiple times with or without evicting some cache lines. By measuring the execution time difference, the attacker can determine if the victim uses the evicted cache lines.

Flush+Flush [20]: Flush+Flush attack exploits the execution time difference of CLFLUSH instruction depending on cache line states. For example, if the target cache line of CLFLUSH is shared by other caches, it will take more cycles to flush all of them than flushing the cache line owned by a single cache.

To mitigate above mentioned side-channel attacks, many approaches have been proposed. First, modifying the timing such as CPU cycles can affect the attacker's decoding of the transmitted data [18], [70]. Second, shared resources such as LLC can be isolated so that attackers cannot access the victim's resources [33], [34], [42], [64]. Finally, an Oblivious RAM (ORAM) design [45], [60] can eliminate the access patterns from a CPU or a program, and thus prevents a majority of side-channel attacks including the distance-based NUCA attacks. However, ORAM generally incurs significant overheads for shuffling the memory locations constantly after every access.

D. NoC-based Side-Channel Attacks

In MPSoCs, NoC has been leveraged for Prime+Probe cache side-channel attacks [57]. The attackers monitor the throughput changes to identify when the cryptography victim accesses the cache for key lookups. Then the attacker can start the probe phase for a successful attack. For mitigation, Gossip NoC was proposed to switch routing algorithms when abnormal network behavior is detected [57]. More recently, Paccagnella *et al.* developed several timing side-channel attacks on the CPU ring interconnect by exploiting NoC and cache contention [53]. A follow-up work investigates how network contention in 2D Mesh in a NUCA machine can lead to timing difference and subsequently, a covert and side-channel [15], [63]. However, there is no existing work that exploits differences in physical distances in a NUCA machine to create a side channel. As a countermeasure, prior works segregates or distributes network traffic [15], [65]. However, it fails to mitigate the non-uniformity in NUCA access time due to distance.

III. Threat Model and Assumptions

This work focuses on the attack and defense of a specific timing channel inside the NoC architecture. This timing channel is derived from the latency difference of LLC hits in a NUCA architecture due to differences in physical distances. Distinct from other side-channel works based on contention within the NUCA architecture [15], the distance-based attack requires the attackers to time at least part or the entirety of the victim functions known to access data in different cache banks in NUCA. This can be done either through timing the invocation of the victim function, likely inside the same context, or through timing the interaction with the victim function, such as sending or receiving messages through the network or inter-process communication (IPC), detecting modification of shared variables, or other side-channels. The attackers may or may not have access to an accurate timing function (e.g., `rdtsc`), and if not, they can use alternatives such as counting threads [58]. The proposed attack and defense techniques assume a trusted CPU and OS which are not inherently malicious. The CPU and the OS may be vulnerable to attacks, but no escalation to root (admin) privileges will be possible or necessary for conducting the attacks. The victim program/function can be triggered or invoked by the attackers and must interact with the attackers either remotely or locally. We assume that the attackers have at least remote access to the machine where the victim is running and can launch more than one thread on selected cores. The attackers' threads include one contention thread on the same core as the victim program, one preparation thread on a separate core, and one timing thread for timing the victim operations. In addition, the attacker has the access to the source code of the victim program, and has knowledge of the possible access patterns within the victim program (i.e., the conditions that cause access to near or far LLC banks), through either reverse-engineering the microarchitecture or profiling the victim program.

IV. A Distance-Based NUCA Side-Channel Attack

In this section, we describe the steps for realizing a distance-based NUCA LLC side-channel attack. Then, we demonstrate an attack example using the AES decryption function. We used the Intel Xeon Phi 7290 CPU.

A. Intel Xeon Phi 7290 LLC Organization

The CPU has a floorplan shown as Figure 2, where its 72 physical cores (or 288 physical threads with hyperthreading) are distributed across 38 tiles [25]. It is known that not all tiles have active physical cores on them, and the physical CPU IDs—the IDs which are typically obtained through ACPI and are recognized by OS—are arbitrarily assigned to tile in an order which tends to alternate between the four quadrants. The CPU employs a directory-based cache coherence mechanism using MESIF protocol [19] with a distributed directory system. Each tile includes a Caching/Home Agent (CHA) in charge of a portion of the directory. Each time a core requests a cache line due to an L1 miss, a corresponding CHA (distributed directory) is queried based on the line address. If the cache line is present in the LLC bank of a tile, the CHA will instruct the tile to forward the data to the requester. Thus, two sources of latency contribute to the difference in LLC hit times; one due to different distance to the CHA location, and the other due to different distance to the forwarding tile. Even if two cache lines reside in the same forwarding tile, their LLC hit times can differ if two different CHA handles the cache lines.

B. Identifying Far-Tile and Near-Tile Accesses

To perform the attack, we need to identify addresses that are mapped to CHA on a far tile or a near tile. To know such addresses for the victim program, one must know the core(s) which the victim program is assigned to, and which virtual addresses in the victim program are mapped to a CHA on a far tile or a near tile.

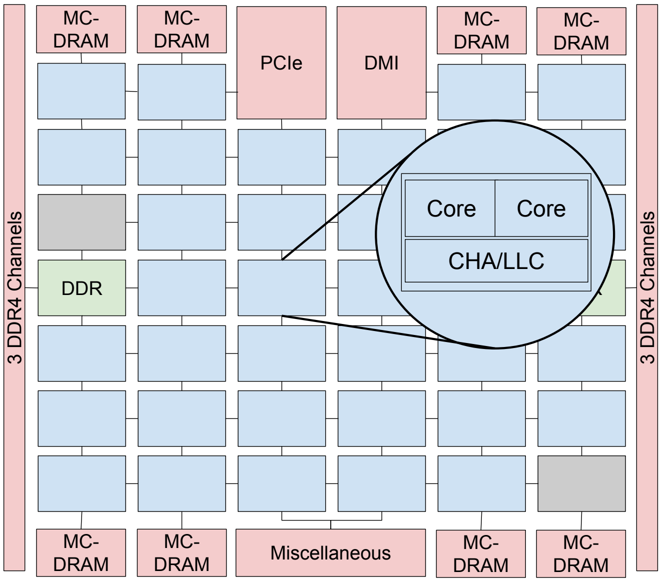

Fig. 2: Knights Landing Floorplan Block Diagram [25]. Blue rectangles denote active tiles. Grey rectangles denote tiles with disabled cores. One tile is zoomed to show that it contains two cores and a CHA/LLC

One possible method for identifying the far-tile and near-tile addresses is to reverse-engineer the mapping function of the physical addresses to the tiles where the CHA and the LLC slices of target lines reside. We argue that the reverse-engineering is not necessary for this attack. First of all, the reverse-engineering process can be extremely complex, given that multiple bits of a physical address can be involved and the number of tiles is not a power of two. Second, in order to know which *virtual addresses* belong to far tiles or near tiles, one must know the mapping between the virtual addresses and the physical addresses. The Linux system provides a system interface through `/proc/[pid]/pagemap` to show the page types and page mappings, but the physical page frame numbers of each process are only visible to a privileged user.

Instead of reverse-engineering the LLC mappings, we use a strategy we call *Profile and Remap*. The strategy contains two steps, and requires an attacker process to run on the same tile (not necessarily the same physical core or thread) with the victim program. The attacker process will first allocate a certain amount of virtual addresses and access them. The attacker process uses a helper thread that will access the addresses first to bring them to the LLC. The CPU tile of this LLC will act as the forwarding tile. When another thread of the attacker process accesses those addresses, LLC hits occur, and depending on the distance of CHA, different addresses will have different LLC hit times. Based on the LLC hit times, we can identify two sets of the virtual addresses in the attacker process, ones that are mapped to far tile's CHA, $\mathrm{VA}_{\mathrm{far}}$ and ones that are mapped to near tile's CHA, $\mathrm{VA}_{\mathrm{near}}$. Then, the attacker can force the OS to *remap* the physical pages backing these virtual addresses to the target virtual addresses in the victim program. Here, we adopted a technique called **Flip-Feng-Shui**, which is commonly used to control the physical memory layout of the victim process for Rowhammer attacks [56]. The technique requires unmapping a virtual page in the attack process right before the victim program accesses a virtual page for the first time so that the OS will reuse the physical page from the attack process because it was recently added to the free page list. Thus, the remapping can be done without any root privilege.

C. Ensuring an L1D Miss but an LLC Hit

To leak information through the distance-based NUCA side channel, the victim program must exhibit data-dependent access patterns across LLC tiles so that the attacker can time the victim program and infer the secret based on the difference in access time. Such an attack requires L1D misses but LLC hits on specific addresses. If accessing certain addresses in the victim program causes L1D hits, there will be no difference in the access time. On the other hand, if accessing the addresses causes both L1D and LLC misses, the CPU will send requests to the DRAM and the access latency will again be not dependent on the address. Therefore, it is crucial to keep the target addresses in the LLC but not in the local L1D cache of the core where the victim program is running.

Let us consider two different cores (say, core *i* and *j*) in two different tiles. Suppose the victim function runs on core *i*. The attacker's goal is to ensure that a cache line, say *L*, will cause an L1D miss but LLC hit when accessed from core *i*. There are two approaches to do that. *First,* the attacker can force core *i* to access *L* and a number of other cache lines that fall in the same set in L1D so that *L* eventually gets evicted from L1D of core *i* but still remains in the LLC. To use this approach, the attacker can run some code on core *i* before the victim function so that *L* remains in LLC but not in L1D of that core's tile. On top of that, the attacker requires knowledge about the L1D indexing function and replacement policy. *Second,* the attacker can execute a thread on some other core (such as core *j*) that accesses the cache line *L*. As a result, *L* will reside in the L1D cache and LLC bank associated with core *j*'s tile. When the victim function runs on core *i* and accesses *L*, it will have a miss in its own L1D cache but the line will be found in the LLC of core *j*'s tile. In other words, core *i* will have an LLC hit on *L*. Note that subsequent accesses from core *i* to the same cache line will cause L1D hits. In our paper, we follow this second approach for its simplicity.

D. Attack Example: AES in OpenSSL

The traditional AES implementation uses a number of transformation tables, known as T tables, to represent the computation and permutation of individual bytes during multiple rounds (9 rounds for AES-128, 11 rounds for AES-192, or 13 rounds for AES-256). So far, these T tables have been the targets of exploitation in many side-channel attacks to leak the AES secret keys [6], [9], [23], [28]. Take the AES implementation (`aes_core.c`) in OpenSSL 1.1.0f for example. We show a simplified version of the AES decryption code in Figure 3. Since AES is a block cipher, in each invocation, the `AES_decrypt` function will take a block of 128 bits as the input and decrypt it using a 128-bit, 192-bit, or 256-bit key. Note that `AES_decrypt` and `AES_encrypt` have very similar structures, except that they use two different sets of T tables, `Td0–Td4` and `Te0–Te4`, respectively, and that `AES_decrypt` has an extra round that uses only `Td4`. Generally, the last round of `AES_decrypt` has been targeted by cache side-channel attacks, such as FLUSH+RELOAD, since the attacker only needs to detect the change of cache states during the last round and can avoid any noise from prior rounds.

```c
static const u32 Td0[256] = ...;
static const u32 Td1[256] = ...;
static const u32 Td2[256] = ...;
static const u32 Td3[256] = ...;
static const u8 Td4[256] = {
  0x52U, 0x09U, 0x6aU, 0xd5U, 0x30U, 0x36U, 0xa5U, 0x38U,
  0xbfU, 0x40U, 0xa3U, 0x9eU, 0x81U, 0xf3U, 0xd7U, 0xfbU,
  ...
};

void AES_decrypt(u32 *in, u32 *out, u32 *rd_key) {
    u32 s0, s1, s2, s3, t0, t1, t2, t3;
    s0 = in[0] ^ rk[0];
    s1 = in[1] ^ rk[1];
    s2 = in[2] ^ rk[2];
    s3 = in[3] ^ rk[3];
    ...
    /* The last round */
    out[0] = ((u32)Td4[(t0 >> 24)        ] << 24) ^
             ((u32)Td4[(t3 >> 16) & 0xff] << 16) ^
             ((u32)Td4[(t2 >>  8) & 0xff] <<  8) ^
             ((u32)Td4[(t1      ) & 0xff])        ^
             rd_key[0];
    out[1] = ((u32)Td4[(t1 >> 24)        ] << 24) ^
             ((u32)Td4[(t0 >> 16) & 0xff] << 16) ^
             ((u32)Td4[(t3 >>  8) & 0xff] <<  8) ^
             ((u32)Td4[(t2      ) & 0xff])        ^
             rd_key[1];
    out[2] = ((u32)Td4[(t2 >> 24)        ] << 24) ^
             ((u32)Td4[(t1 >> 16) & 0xff] << 16) ^
             ((u32)Td4[(t0 >>  8) & 0xff] <<  8) ^
             ((u32)Td4[(t3      ) & 0xff])        ^
             rd_key[2];
    out[3] = ((u32)Td4[(t3 >> 24)        ] << 24) ^
             ((u32)Td4[(t2 >> 16) & 0xff] << 16) ^
             ((u32)Td4[(t1 >>  8) & 0xff] <<  8) ^
             ((u32)Td4[(t0      ) & 0xff])        ^
             rd_key[3];
}
```

Fig. 3: The vulnerable, fully unrolled (i.e., non-iterative) code for AES decryption in `aes_core.c` of OpenSSL 1.1.0f. The source code is simplified for brevity, and only shows the initial values of `Td4` and the last round of `AES_decrypt`.

Note that the attack on AES decryption requires that the plaintexts are either known or chosen by the attacker. Take FLUSH+RELOAD [22] for an example. The attacker first `clflush()`es all the elements from `Td4` and then waits for `AES_decrypt` in the victim program to access the elements of `Td4` and to bring the corresponding cache lines into the cache. Then, by timing the latency of all `Td4` elements, the attacker can tell which cache lines are recently brought into the cache and thus can guess the potential values of t0–t3 in the last round. Finally, by XOR'ing the known plaintext as the output of the last round and the potential values of the `Td4` elements accessed, the attacker can guess the potential lowest 32 bits of the decryption key.

The NUCA distance-based side-channel attack on AES is different from FLUSH+RELOAD and similar attacks since it cannot infer exactly which line is accessed by the function and brought into the cache. Instead, the attacker can only time the victim function, `AES_decrypt`, and use the latency to extract the bits inside the key. Specifically, this attack faces two major challenges: (1) **Overlapping of multiple cache loads**: An out-of-order CPU can issue multiple load instructions into the pipeline, and send multiple requests to the Load Store Queue (LSQ). Although the Total Store Ordering (TSO) model of most Intel CPUs forbids reordering of the load requests, requests can still be sent while the prior requests await responses. As a result, the latency of multiple load instructions without mutual dependency can overlap, and thus, the highest latency of individual loads will dominate the overall latency. (2) **Timing difficulty with multiple decryption rounds**: From the attacker's point of view, it is difficult to only time the last round of decryption where only elements of `Td4` and the lower 4 bytes of the key are accessed. This is because timing the entire `AES_decrypt` function will include the time of earlier rounds of decryption making it impossible to determine how much time it takes only to access `Td4` entries.

To overcome the challenges, we formulate the attack as follows: First, the attacker runs three threads–one thread running a loop on the same core as the AES program to bring `Td0`–`Td3` into L1D, while the other thread runs on another tile to keep the whole `Td4` inside LLC. The total size of `Td0`–`Td3` is 4KB, which can indeed be accommodated by the L1D cache of Intel Xeon Phi 7290, which is 32KB per core. Any future access to `Td0`–`Td3` entries does not cause any network traffic in the NoC and hence, `Td4` access times can be measured without any noise. Then, as the attacker, we do not time the end-to-end latency of all the decryption rounds. Instead we run a timer thread that will repeatedly check for any change in `out[0]` and `out[1]` (Figure 3). This is possible because the attacker provides a buffer as `out` parameter in the `AES_decrypt` function to collect the decrypted text. Therefore, the attacker code can check when `out[0]` has been modified. Then, it can start a timer (or alternatively, start a counting loop). The attacker stops the timer (or terminates the loop) when `out[1]` is modified. The time between these two modifications (to `out` entries) will be the time for executing the statements from Line 24 to 28 in Figure 3. In other words, this is the time for four accesses to `Td4` table. Let us denote this time as T_{24-28}. Due to the overlapping of memory loads between lines 24 and 28, the time T_{24-28} will be shorter if the cache lines accessed in `Td4` are all in near tiles or in L1D. On the other hand, if one or more accesses are to the cache lines in a far tile, the T_{24-28} will be higher. By observing a stream of plaintexts and measuring T_{24-28} as long as one or more accesses fall on the far tile, we can guess potential `Td4` elements accessed between Line 24 to 28 and retrieve the lower 4 bytes of the key by XOR'ing with the known plaintexts. In our proof-of-concept (POC) code, we take multiple samples of T_{24-28} from decrypting a specific 16-byte plaintext and use majority voting (using AdaBoost Classifier [11]) to determine if one or more accesses fall to a far tile. Figure 4 shows that with AdaBoost Classifier, we can determine accesses to a far tile with 100% accuracy by using 40 or more samples. If one or more accesses happen to a far tile, we determine the potential lower 4 bytes of the key by XOR'ing. We keep doing this using random plaintexts and eventually, extract the lower 4 bytes of the key using a simple majority for each byte. Figure 5 shows that our POC code can extract those bytes with 100% accuracy using only $\simeq$ 4000 decryption trials[1].

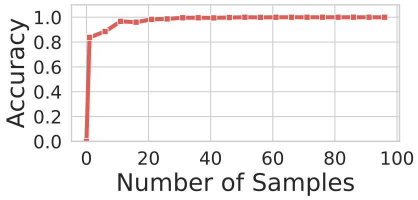

Fig. 4: Accuracy of determining if one or more accesses fall to the far tile. We reach 100% accuracy by taking majority voting of 40 samples or more.

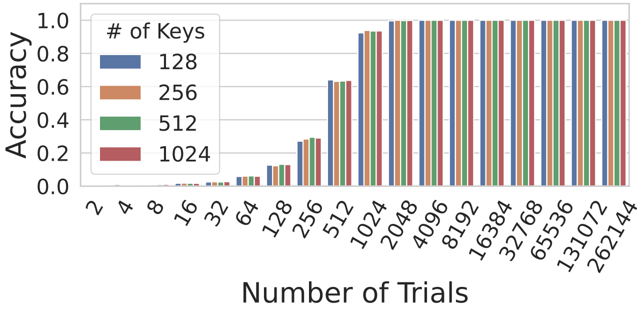

Fig. 5: Key extraction accuracy with repeated decryption trials. We can extract lower 4 bytes of any random key with 100% accuracy by using only $\simeq$ 4000 trials.

V. Mitigating Distance-Based NUCA Attacks

A. *Scope of* Camouflage

Distance-based NUCA attacks rely on LLC memory accesses that exhibit latency differences. These differences are caused by different LLC banks requiring network packets to traverse different distances in NoC. Therefore, our proposed defense mechanism, Camouflage, does not need to consider any access that causes an L1D hit because that access will not come down to LLC at all. Similarly, Camouflage does not need to consider any access that suffers from an LLC miss because the latency of such access is virtually unaffected by LLC bank distances in NoC (the latency will be dominated by the DRAM latency). In other words, Camouflage is

[1] Our POC code is here - https://anonymous.4open.science/r/ml-attack-nuca-DC02

applied to accesses that cause LLC hits. We refer to any load instruction that causes an LLC hit as a *Security Sensitive* instruction. We also refer to this as Secure LD.

Note that we do not consider any store instruction because of two reasons. *First,* from an attacker's point of view, a store instruction is harder to time (due to a write buffer) without the ability to use fence instructions. *Second,* an attacker cannot enforce LLC hits if the victim function contains store instructions. This is because when the attacker brings the cache lines corresponding to the store instructions to the LLC and the victim function later executes the store instructions, there will not be any LLC hit. Instead, each store instruction will upgrade the associated cache line's coherence state by generating invalidation requests followed by the actual write operation. Both of these issues make it difficult (if not impossible) to launch a NUCA attack using store instructions.

B. Overview of CAMOUFLAGE

Figure 6 shows the overview of CAMOUFLAGE. When a core executes a security-sensitive instruction, it sends a request to the CHA through the Network Interface (NI). CHA determines the destination tile of the request and sends a forwarding request. CAMOUFLAGE hardware at the NI of the destination tile checks the expected round trip latency with the target latency. CAMOUFLAGE chooses the target latency based on the worst case round trip latency from the source to the furthest LLC bank (in case of $\text{CAMOUFLAGE}_{\text{JITTER}}$) or the earliest round trip latency from the source to the farthest LLC bank using the bypass mechanism (in case of $\text{CAMOUFLAGE}_{\text{BYPASS}}$). With $\text{CAMOUFLAGE}_{\text{JITTER}}$ as the defense mechanism, the destination NI uses a normal routing channel to send back the response because the expected round trip latency will be less or equal to the worst-case round trip latency. The source NI adds the necessary jitter once it receives the response packet from the destination NI. When the jitter time has passed, the source NI sends the requested data back to the core. With $\text{CAMOUFLAGE}_{\text{BYPASS}}$ as the defense mechanism, the destination NI checks if the expected round trip latency is higher than the target latency due to the location of CHA. If so, the destination NI sends the response packet through a bypass channel and expedites the packet delivery. When the response is received, the source NI checks if it arrives earlier than the target latency. If so, additional delay (i.e., jitter) is added so that the target latency is met. On the other hand, if the request from the security-sensitive instruction is satisfied by the local LLC bank or if the request and response use the normal routing protocol and still arrive at the source NI earlier than the target latency, necessary jitter is added to meet the target latency. Thus, in both schemes, the CAMOUFLAGE hardware ensures constant latency for packets originating from the security-sensitive instructions, thereby preventing any distance-based NUCA attack.

C. Details of CAMOUFLAGE

The goal of CAMOUFLAGE is to ensure constant round trip latency of any LLC request from the source NI to any destination NI in the NoC. We explain the details of $\text{CAMOUFLAGE}_{\text{JITTER}}$ followed by $\text{CAMOUFLAGE}_{\text{BYPASS}}$.

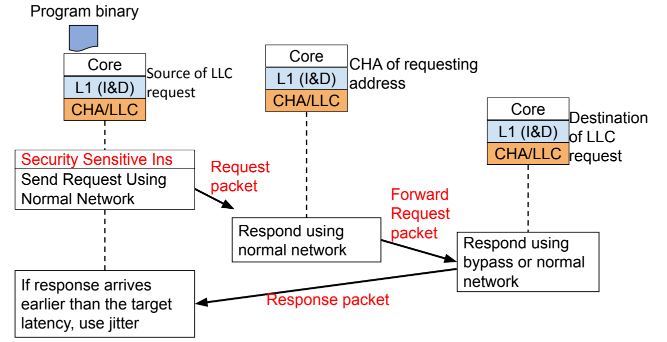
Program binary
Core
L1 (I&D)
CHA/LLC
Source of LLC request
Core
L1 (I&D)
CHA/LLC
CHA of requesting address
Core
L1 (I&D)
CHA/LLC
Destination of LLC request
Security Sensitive Ins
Send Request Using Normal Network
Request packet
Respond using normal network
Forward Request packet
Respond using bypass or normal network
If response arrives earlier than the target latency, use jitter
Response packet

Fig. 6: Overview of how CAMOUFLAGE works. CAMOUFLAGE hardware ensures fixed latency for LLC memory accesses using jitter and/or bypass techniques.

1) $\text{CAMOUFLAGE}_{\text{JITTER}}$*:* $\text{CAMOUFLAGE}_{\text{JITTER}}$ is the simplest and most intuitive approach to ensure constant LLC access latency. $\text{CAMOUFLAGE}_{\text{JITTER}}$ considers the worst case latency from the source NI to any destination NI as the target latency (T_{worst}) and delay (jitter) every response until that time. For example, in the case of a 2D mesh topology, if the source NI is at one corner, the furthest NI would be the one in the opposite corner. Therefore, the worst-case scenario happens when the distance between the source and CHA is the worst and so are the distances between CHA and destination and destination and source. Since the worst-case latency could vary depending on the network traffic, the target latency is set based on the worst-case latency in a highly congested network. As an example, in the case of a 2D mesh topology for the NoC, we simulate the topology using the Booksim2.0 [30] network simulator and determine the round trip latency between the furthest source and destination NIs. In the Booksim 2.0 simulator, we simulate the 8x8 mesh network and measured the network latency with varying injection rates. We keep increasing the network packet injection rate until the NoC gets saturated i.e., the network reaches a point where the round trip latency is abruptly increased. This is shown in Figure 7. We set T_{worst} to the round trip latency just before the saturation point (e.g., at injection rate 0.0875). At that injection rate, one-way network latency is < 50 cycles on average. From this network sweep simulation, we set the jitter threshold T_{worst}=100 cycles to safely hide any latency differences caused by a network that is not yet saturated. We did not choose any point at or above the saturation point because in that case, the network packets suffer prohibitively large round-trip latency and the latency becomes non-deterministic. Therefore, when the NoC gets saturated, it is not possible to launch a distance-based NUCA attack.

For any request packet originating from a security-sensitive instruction, the source NI sends the request packet through the network and receives the corresponding response packet. After receiving the response, the source NI compares the round trip latency of the request-response packets (T_{orig}) with the target latency, T_{worst} and determines the amount of jitter (i.e., $T_{worst} - T_{orig}$) that should be added to the response time. The response packets are buffered in a Jitter Queue in the NI (more on this in Section VI-B) to add the necessary jitter. Thus, CAMOUFLAGE$_{\text{JITTER}}$ makes the round trip latency of every LLC hit equal to T_{worst} and prevents any NUCA attack.

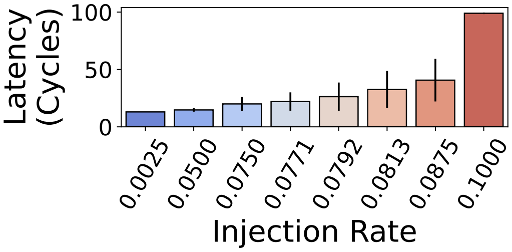

Fig. 7: Average Network Latency of the worst case scenario in 8×8 mesh topology network simulated using Booksim [30]. The network gets saturated at the injection rate of 0.01 so we picked the $T_{worst} = 100$ Cycles to cover Round Trip Latency for the worst-case scenario right before the network is saturated.

CAMOUFLAGE$_{\text{JITTER}}$ can be easily implemented in the existing hardware design with minimal changes as shown in Figure 8. Existing Stall Queues of the NIs can be used as Jitter Queues (JQ) with some extra logic such as multiplexers and an arbitrator (ARB) to make them work properly. Note that jitter is added to only those packets resulting in LLC hits. Therefore, when the destination bank determines that an access request is going to cause an LLC miss, the NI sends back a response packet by marking it as not requiring any jitter. This is done by setting a flag, called *Jitter Required* to *False*. When the source NI receives the response packet with the flag set to *False*, it does not add any jitter. Otherwise, the source NI calculates the necessary jitter amount and keeps the response packet in the Jitter Queue until the elapsed time equals to the jitter amount. Although CAMOUFLAGE$_{\text{JITTER}}$ is simple and intuitive, it incurs significant performance penalties because the packets suffer the worst-case latency. Moreover, as multicores become larger [1], [2], the worst-case latency could be in the range of hundreds of cycles. Therefore, we propose another defense, called CAMOUFLAGE$_{\text{BYPASS}}$.

2) CAMOUFLAGE$_{\text{BYPASS}}$: To ensure both security and performance, we propose CAMOUFLAGE$_{\text{BYPASS}}$ that utilizes both jitter and bypass techniques. The goal of CAMOUFLAGE$_{\text{BYPASS}}$ is to make the target round trip latency as low as needed to eliminate any performance degradation. Since the round trip latency has three parts namely, source to CHA, CHA to destination, and destination to source, we can expedite all of them or only a few of them. We choose to expedite the latency from destination to source because actual data packets traverse in that part. Besides, expediting only the destination to source part already eliminates any runtime overhead of our defense mechanism (Section VII-E). Following this intuition, we set a round trip target latency T_{low} as follows. T_{low} is the summation of (i) worst case latency from the source to CHA, (ii) the worst-case latency from the CHA to destination, and (iii) the lowest latency from the destination to source (even when destination and source are the furthest apart). For (i) and (ii), the worst case latency occurs when the tiles are the furthest apart and for (iii), the fastest traversal happens even in the case of the furthest destination and source when router bypass is enabled between them. Packets coming from far away destination NIs (to the source NI) will mostly use bypass techniques to stay with the target round trip latency, whereas those from the nearby NIs will use jitter.

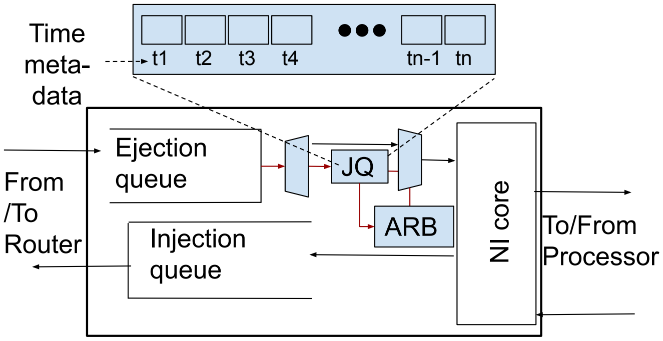

Fig. 8: Network Interface is extended with Jitter Queue (JQ) and arbitrator (ARB) for CAMOUFLAGE$_{\text{JITTER}}$.

Algorithm 2 shows the detailed steps that the source and destination NI uses for every packet that caused the security-sensitive instruction. Based on the expected round trip latency [24], [66], [72] from Src to Dest NI, CAMOUFLAGE$_{\text{BYPASS}}$ decides to either use bypass channel or regular network. If the expected latency is above T_{low} (Line 11), the bypass technique is used for the packet. After a response is received at the Src NI, CAMOUFLAGE$_{\text{BYPASS}}$ checks if the response arrives early (Line 17). If so, the response is jittered inside the Src NI until the target latency is met. Figure 9 shows a comparison of different policies.

To implement bypass, CAMOUFLAGE$_{\text{BYPASS}}$ does not need any new physical link. However, one flit size buffer is reserved for every router at the Input Buffer along the path to make sure that there is space to hold the bypassed flit. These can be extended to support multiple flits when there are multiple simultaneous NUCA timing attacks.

VI. IMPLEMENTATION DETAILS

In this section we will describe the design choices made to implement CAMOUFLAGE hardware. We first explain the changes made to a typical NI hardware followed by the details of how to implement jitter.

A. Changes in the Network Interface

We extend Network Interface (NI) since CAMOUFLAGE controls the timing of packet ejection from NI. It has been previously used to provide security features in malware detection [51] or network intrusion detection [46]. We extend NI capability to detect the potential packets to add jitter to or

Algorithm 2: How Camouflage$_{\text{BYPASS}}$ works at the NI to choose jitter or bypass for a packet from a security sensitive instruction.

```
Data: Packet P
if Is P from sensitive instructions then
    Src := Source NI of the sensitive instruction;
    Dest := Destination NI of the sensitive instruction;
    $T_{low}$ := Target round trip latency;
    $EL_P$ := Expected round trip latency of P from Src to Dest;
    if P is a request packet then
        Use normal routing protocol for P;
    else
        ▷ P is a response packet from Dest ◁
        if Current NI = Dest then
            if $EL_P > T_{low}$ then
                Use bypass for P from Dest to Src;
            else
                Use normal routing protocol for P;
        else if Current NI = Src then
            $AL_P$ := Actual round trip latency from Src to Dest;
            if $AL_P < T_{low}$ then
                Jitter P for $T_{low} - AL_P$;
else
    Use normal routing protocol for P;
```

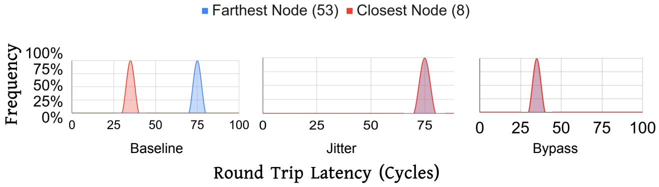

Fig. 9: Comparison of different strategies

select for bypass based on the expected latency following the Camouflage logic. When the NI receives a load request it sets the flags to decide on either of the two available networks (normal or bypass) or adding jitter (for Camouflage$_{\text{JITTER}}$).

The bypass network uses the same physical links as the normal network. We implemented VC separation [16] technique for the Bypass and Regular network. The NI uses a 1-bit flag called `bypass_flag` on each output port of the Router. When the NI sends any flit using the output port, it checks whether `bypass_flag` is enabled for that cycle. If the flag is disabled, then it cannot send the flit using the output port on that cycle. When Camouflage$_{\text{BYPASS}}$ wants to send some flit using the bypass path, the NI first checks and sets this flag. If the flag is already set, then the NI stalls the flit for that cycle and tries again on the subsequent cycles until it succeeds.

Since the best case scenario from the attacker's perspective is to use only two distinct cache banks which reside physically distant from one another, there should not be any contention between the bypass flits. However, we can easily resolve the collision by pipelining the bypass. When an NI sends out the flit to the network using either the bypass path or regular path, it sets the timestamp of the request of the flit in the header flit. If this is a flit of a request packet, then we use the flit creation time. If the flit is of a response packet, we save the timestamp of the corresponding request flit creation time. This timestamp is used to process the algorithm in the NI. Camouflage uses a comparator and MUX in NI to implement the defense technique. The block diagram in Figure 10 gives an example of bypass operation using the existing buffers.

B. Jitter Implementation

To implement the dynamic jitter we include a Jitter Queue (JQ) in each NI. This Jitter Queue functions as the buffer for holding and sending the flits that are associated with the Jitter cases. Every flit contains a Target Latency (TL) in the flit header. This is calculated based on the current configuration of the network. We need to deliver the flits within this Target Latency. If the Tail flit associated with a secure LD response arrives at the Source NI of the instruction, we compare the latency that has been spent inside the network. When the response for the Secure LD reaches the NI of the Source Router, we calculate the Round Trip Latency (RTL) of that request. To do that we use a comparator that compares this RTL with the TL. We can have two cases here.

1) **Case 1: RTL $\geq$ TL**, in this case, we do not have to add any jitter at all. We can forward this flit to the ejection queue of the NI if the ejection queue is available. Else we can add it to the stall queue accordingly.
2) **Case 2: RTL $<$ TL**, in this case, we add the packet P_i in the JQ of the destination NI with a jitter amount = TL− RTL. We set up a counter that wakes up this NI to consume the flit after this specific amount of clock cycle waiting.

VII. Evaluation

In this section we will discuss in detail the experiments and results to evaluate Camouflage.

A. Experimental Setup

We run several experiments to evaluate our proposed defenses. First, we established that it is possible to create a side-channel using the NoC access latency difference using NUCA architecture cache in a real machine like Intel Xeon Phi 7290 CPU [3]. Then we showed the impact of Camouflage$_{\text{BYPASS}}$ and Camouflage$_{\text{JITTER}}$ on the average packet network latency with varying packet injection rate using Garnet [37] simulator. We provided a proof of concept solution using the Camouflage$_{\text{JITTER}}$ and Camouflage$_{\text{BYPASS}}$ methods. The result of the proof of concept scenario is explained here in Figure 12. We used PARSEC [7] and Rodinia v3.0 benchmark [13] to evaluate the impact of our proposed defense mechanisms on a large multi core processor with Gem5 simulator. Finally, we performed some sensitivity studies to find out the optimal parameters for our design.

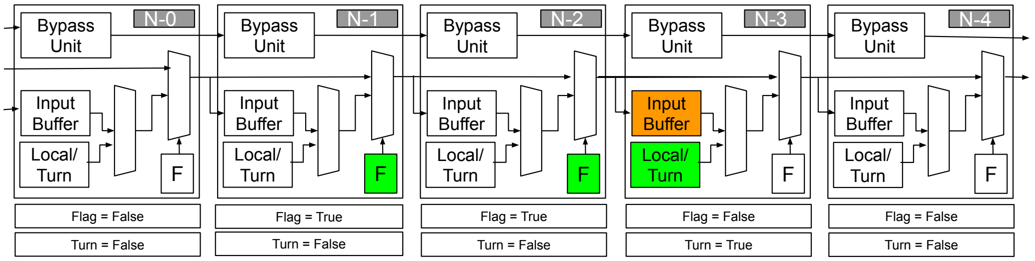

Fig. 10: Block diagram of bypass operations.

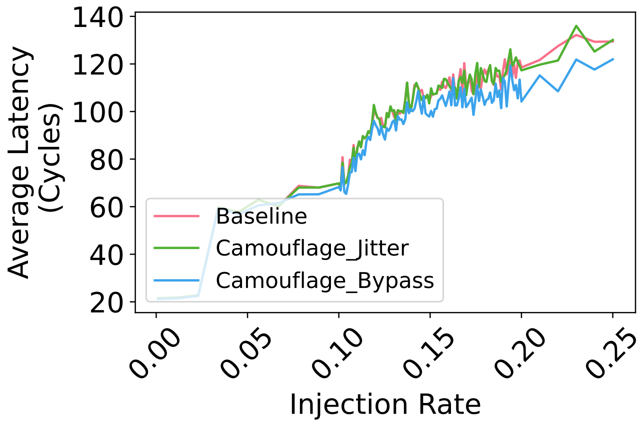

Fig. 11: Impact of our policy on varying injection rate.

B. Impact on Performance

We have used PARSEC and Rodinia benchmarks to evaluate the impact of different defense mechanisms on the performance of an application. We use any load request arriving to the LLC as a security-sensitive instruction. We applied our policies CAMOUFLAGE$_{\text{JITTER}}$ and CAMOUFLAGE$_{\text{BYPASS}}$ to see the impact on performance. We have used a 64-core machine with 32kB L1 cache, 2MB distributed Last Level Cache scenario. The results are shown in Figure 13. For Rodinia applications, we executed 1B instructions in the region of interest where the threads are spawned by the OpenMP library [12]. For PARSEC applications we execute 1B instruction or end of Region of Interest as marked inside the PARSEC applications.

We observe that most of the applications suffer from high performance overhead with the CAMOUFLAGE$_{\text{JITTER}}$ approach. We have 0.1%-57% performance overhead over baseline with a geomean of 14% overhead across all applications. For most of the applications, we observe that CAMOUFLAGE$_{\text{BYPASS}}$ can retain the performance benefits from the bypass network. In this case, we have performance improvement of 2%-36% over baseline without any bypass[2].

However, we have seen in some applications the performance benefits were not observed on using CAMOUFLAGE$_{\text{BYPASS}}$ over Baseline policy. For example, in the case of x264 we get 4% overhead using CAMOUFLAGE$_{\text{BYPASS}}$ policy. Since the terminating condition of the simulator was set to be any threads reaching 1B instructions, the number of executed instructions were not the same for all the applications. So, in this particular application, we observed more instructions are executed, hence more simulation time is required even though the terminating condition for the simulation remained the same. In other applications, we observed performance improvement using CAMOUFLAGE$_{\text{BYPASS}}$ which showed an average of 7.5% performance improvement over Baseline and 18% improvement over CAMOUFLAGE$_{\text{JITTER}}$

C. Impact on Average Packet Network Latency

We used the Garnet [37] network simulator to determine the average packet latency with varying injection rate. The results of this experiment can be seen in Figure 11.

From this experiment, we can conclude that our policy performs at least similarly to the baseline configuration. Using bypass paths to secure LDs does not adversely impact the overall average packet network latency. However, if we only add jitter we can see that at a higher injection rate, the network starts saturating earlier compared to the usage of CAMOUFLAGE.

D. Impact on Round Trip Latency of Secure LD with Varying Injection Rate

We can also verify the impact of varying injection rates on the Round Trip Latency of the Secure LD with the Garnet network simulator [5]. The following figure 12 shows the impact on Round Trip Latency of Secure LD instructions using different policies on varying injection rates.

Here we can see that both the CAMOUFLAGE$_{\text{JITTER}}$ and CAMOUFLAGE$_{\text{BYPASS}}$ make sure that there are no discernible differences between the round trip latencies of Secure LD instructions going to the closest and the farthest destination nodes, which is prevalent in the case of Baseline scenario 12(a). In the case of CAMOUFLAGE$_{\text{JITTER}}$ solution, we see that the average Round Trip Latency is much higher as evident in Figure 12(b) than the average Round Trip Latency of the CAMOUFLAGE$_{\text{BYPASS}}$ which is evident at Figure 12(c).

E. Impact of Varying Number of Destinations in Destination List

From the attackers' perspective this is best to have only one pair of destinations. By varying the number of destinations,

[2] Note that SMART [38] reported on average 26% performance improvement due to bypass network over the network without any bypass mechanism.

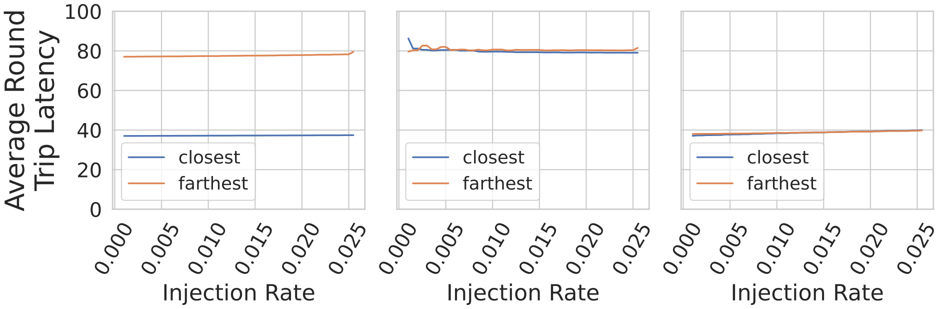

Fig. 12: Impact on Secure LD round trip latency on (a) Baseline (b) CAMOUFLAGE$_{\text{JITTER}}$ (c) CAMOUFLAGE$_{\text{BYPASS}}$ policies

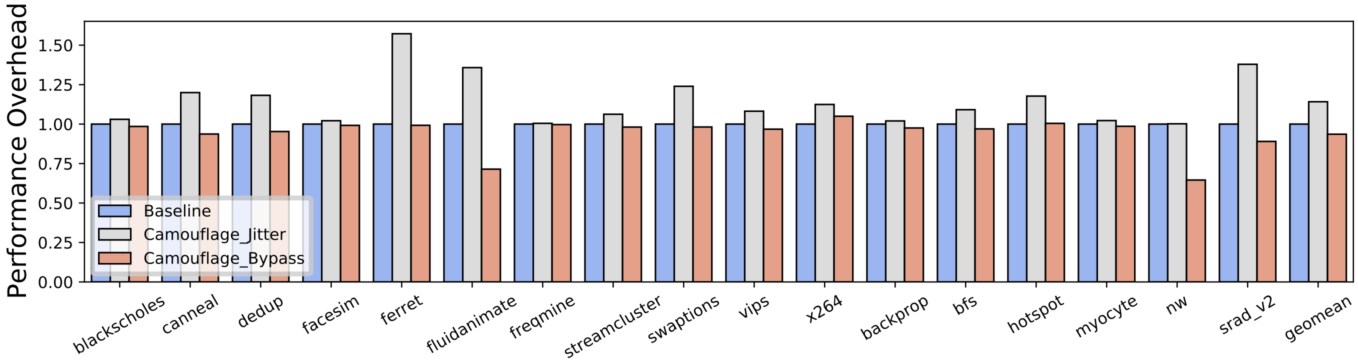

Fig. 13: Performance overhead of CAMOUFLAGE$_{\text{JITTER}}$ & CAMOUFLAGE$_{\text{BYPASS}}$ using PARSEC [7] & Rodinia [13] benchmarks

we can compare the impact of the performance using the Rodinia v3 benchmark. We can see that even if we consider the worst-case scenario, CAMOUFLAGE performs well compared to Naive Jitter All scenario as evident in Figure 13. In some benchmark applications like srad_v1 the improvement over Naive Jitter All scenario is more than 15%. This is due to the fact that Jitter All scenario adds extra waiting time for the loads which would have been serviced otherwise. On average, we can see around 2.5% improvement in the CAMOUFLAGE compared to Naive Jitter All baseline.

F. Impact of CAMOUFLAGE$_{\text{JITTER}}$ *and* CAMOUFLAGE$_{\text{BYPASS}}$ *on Secure Load Latency*

Different policies have different impacts on the secure load latency. The following is the secure load latency distribution for Facesim application from the PARSEC [7] benchmark. We can clearly see that both the CAMOUFLAGE$_{\text{BYPASS}}$ and CAMOUFLAGE$_{\text{JITTER}}$ achieve similar results from Figure 14. The latency distribution Secure Load Latency that hit at Last-Level cache of closest pair of nodes i.e. same nodes (0,0) and farthest pair of nodes (0,64) is reflected in the graph.

As we can see in Figure 14, the latency distribution using CAMOUFLAGE$_{\text{BYPASS}}$ overlaps for both the closest and farthest pair of nodes. Similarly, for CAMOUFLAGE$_{\text{JITTER}}$ we can achieve the same latency distribution for the closest and farthest pair of nodes, providing a security guarantee in both cases.

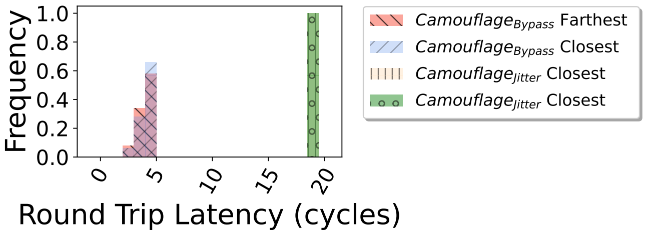

Fig. 14: Impact of Different Policies on Secure Load Latency

G. Hardware Overhead

We estimate the hardware budget using CACTI-7 [49] at 22nm. Camouflage introduces minimal hardware changes both in the network interfaces (NI) as well as in the routers. The changes in the NIs include Jitter queue (0.981 mm^2, 0.028 nJ/access) and 512 64-bit comparators. We also include an arbitration unit with 2 multiplexers whose overhead is not significant. In the case of the bypass channel [44] in the baseline, it incurs 2.34% ∼4.69% of overhead compared to the conventional NoC.

VIII. CONCLUSION

We explored a new distance-based side-channel in NUCA architecture. We could extract the lower 4 bytes of the AES key with only 4,000 decryption trials. CAMOUFLAGE, at runtime, uses a combination of jitter and bypass mechanisms to eliminate any timing difference of those memory accesses and thereby, preventing the attack with minimal overhead.

References

[1] Amd epyc 7742. https://www.amd.com/en/products/cpu/amd-epyc-7742.

[2] Ampere altra. https://amperecomputing.com/altra/.

[3] Intel xeon phi. https://ark.intel.com/content/www/us/en/ark/products/series/75557/intel-xeon-phi-processors.html.

[4] Onur Acıiçmez and Werner Schindler. A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl. In *Cryptographers' Track at the RSA Conference*, pages 256–273. Springer, 2008.

[5] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *2009 IEEE international symposium on performance analysis of systems and software*, pages 33–42. IEEE, 2009.

[6] Daniel J. Bernstein. Cache-timing attacks on aes. 2005.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[9] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. In *Proceedings of the 8th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'06, page 201–215, Berlin, Heidelberg, 2006. Springer-Verlag.

[10] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure:sgx cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[11] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[12] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[13] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[14] Chia-Hsin Owen Chen, Sunghyun Park, Tushar Krishna, Suvinay Subramanian, Anantha P Chandrakasan, and Li-Shiuan Peh. Smart: A single-cycle reconfigurable noc for soc applications. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 338–343. IEEE, 2013.

[15] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. Don't mesh around: Side-Channel attacks and mitigations on mesh interconnects. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2857–2874, Boston, MA, August 2022. USENIX Association.

[16] Jose Duato and Timothy Mark Pinkston. A general theory for deadlock-free adaptive routing using a mixed set of resources. *IEEE Transactions on Parallel and Distributed Systems*, 12(12):1219–1235, 2001.

[17] Andrei Frumusanu. The ampere altra review: 2x 80 cores arm server performance monster. https://www.anandtech.com/show/16315/the-ampere-altra-review.

[18] Michael Godfrey and Mohammad Zulkernine. A server-side solution to cache-based side-channel attacks in the cloud. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 163–170. IEEE, 2013.

[19] JR Goodman and HHJ Hum. Mesif: A two-hop cache coherency protocol for point-to-point interconnects (2009). *URL: https://www. cs. auckland. ac. nz/˜ goodman/TechnicalReports/MESIF-2009. pdf*, 2004.

[20] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[21] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, 2015.

[22] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

[23] Berk Gülmezoğlu, Mehmet Sinan undefinednci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. A faster and more realistic flush+reload attack on aes. In *Revised Selected Papers of the 6th International Workshop on Constructive Side-Channel Analysis and Secure Design - Volume 9064*, COSADE 2015, page 111–126, Berlin, Heidelberg, 2015. Springer-Verlag.

[24] Kris Heid, Haoyuan Ying, Christian Hochberger, and Klaus Hofmann. Latest: Latency estimation and high speed evaluation for wormhole switched networks-on-chip. In *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–7. IEEE, 2014.

[25] Marcos Horro, Mahmut T Kandemir, Louis-Noël Pouchet, Gabriel Rodríguez, and Juan Touriño. Effect of distributed directories in mesh interconnects. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[26] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[27] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-vm attack on aes. In *International Workshop on Recent Advances in Intrusion Detection*, pages 299–319. Springer, 2014.

[28] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-vm attack on aes. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, pages 299–319, Cham, 2014. Springer International Publishing.

[29] Aamer Jaleel, Matthew Mattina, and Bruce Jacob. Last level cache (llc) performance of data mining workloads on a cmp-a case study of parallel bioinformatics workloads. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 88–98. IEEE, 2006.

[30] Nan Jiang, Daniel U Becker, George Michelogiannakis, James Balfour, Brian Towles, David E Shaw, John Kim, and William J Dally. A detailed and flexible cycle-accurate network-on-chip simulator. In *2013 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 86–96. IEEE, 2013.

[31] Mehmet Kayaalp, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2016.

[32] Changkyu Kim, Doug Burger, and Stephen W Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 211–222, 2002.

[33] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, 2012.

[34] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987. IEEE, 2018.

[35] Steve Kommrusch, Marcos Horro, Louis-Noël Pouchet, Gabriel Rodríguez, and Juan Touriño. Optimizing coherence traffic in manycore processors using closed-form caching/home agent mappings. *IEEE Access*, 9:28930–28945, 2021.

[36] Sailesh Kottapalli and Jeff Baxter. Nahalem-ex cpu architecture. 2009.

[37] Tushar Krishna. A detailed on-chip network model inside a full-system simulator. *Monday, September*, 2017.

[38] Tushar Krishna, Chia-Hsin Owen Chen, Woo Cheol Kwon, and Li-Shiuan Peh. Breaking the on-chip latency barrier using smart. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 378–389. IEEE, 2013.

[39] Tushar Krishna, Chia-Hsin Owen Chen, Sunghyun Park, Woo-Cheol Kwon, Suvinay Subramanian, Anantha P Chandrakasan, and Li-Shiuan Peh. Single-cycle multihop asynchronous repeated traversal: A smart future for reconfigurable on-chip networks. *Computer*, 46(10):48–55, 2013.

[40] Amit Kumar, Li-Shiuan Peh, Partha Kundu, and Niraj K Jha. Express virtual channels: Towards the ideal interconnection fabric. *ACM SIGARCH Computer Architecture News*, 35(2):150–161, 2007.

[41] Hyoukjun Kwon and Tushar Krishna. Opensmart: Single-cycle multi-hop noc generator in bsv and chisel. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 195–204. IEEE, 2017.

[42] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*, pages 406–418. IEEE, 2016.

[43] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[44] Wenheng Ma, Xiyao Gao, Yudi Gao, and Ningmei Yu. A latency-optimized network-on-chip with rapid bypass channels. *Micromachines*, 12(6):621, 2021.

[45] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 311–324, New York, NY, USA, 2013. Association for Computing Machinery.

[46] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: an ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*, 2018.

[47] Hassan Mujtaba. Intel skylake-x and skylake-sp mesh architecture for xcc "extreme core count" cpus detailed – features higher efficiency, higher bandwidth and lower latency.

[48] Naveen Muralimanohar and Rajeev Balasubramonian. Interconnect design considerations for large nuca caches. *ACM SIGARCH Computer Architecture News*, 35(2):369–380, 2007.

[49] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40)*, pages 3–14, 2007.

[50] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on aes. In *International Workshop on Selected Areas in Cryptography*, pages 147–162. Springer, 2006.

[51] Carlos Novo and Ricardo Morla. Flow-based detection and proxy-based evasion of encrypted malware c2 traffic. In *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, pages 83–91, 2020.

[52] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[53] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[54] Colin Percival. Cache missing for fun and profit, 2005.

[55] Iván Pérez, Enrique Vallejo, and Ramón Beivide. Smart++ reducing cost and improving efficiency of multi-hop bypass in noc routers. In *Proceedings of the 13th IEEE/ACM International Symposium on Networks-on-Chip*, pages 1–8, 2019.

[56] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1–18, 2016.

[57] Cezar Reinbrecht, Altamiro Susin, Lilian Bossuet, Georg Sigl, and Johanna Sepúlveda. Side channel attack on noc-based mpsocs are practical: Noc prime+ probe attack. In *2016 29th Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6. IEEE, 2016.

[58] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *International Conference on Financial Cryptography and Data Security*, pages 247–267. Springer, 2017.

[59] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.

[60] Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. *J. ACM*, 65(4), April 2018.

[61] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[62] Joop Van de Pol, Nigel P Smart, and Yuval Yarom. Just a little bit more. In *Cryptographers' Track at the RSA Conference*, pages 3–21. Springer, 2015.

[63] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. Meshup: Stateless cache side-channel attack on cpu mesh. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1506–1524, 2022.

[64] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494–505, 2007.

[65] Hassan MG Wassel, Ying Gao, Jason K Oberg, Ted Huffmire, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip. *ACM SIGARCH Computer Architecture News*, 41(3):583–594, 2013.

[66] Xiyue Xiang, Saugata Ghose, Onur Mutlu, and Nian-Feng Tzeng. A model for application slowdown estimation in on-chip networks and its use for improving system fairness and performance. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 456–463. IEEE, 2016.

[67] Lei Yang, Weichen Liu, Peng Chen, Nan Guan, and Mengquan Li. Task mapping on smart noc: Contention matters, not the distance. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[68] Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack. *IACR Cryptol. ePrint Arch.*, 2014:140, 2014.

[69] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[70] Danfeng Zhang, Aslan Askarov, and Andrew C Myers. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 563–574, 2011.

[71] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003, 2014.

[72] Di Zhu, Lizhong Chen, Siyu Yue, Timothy M Pinkston, and Massoud Pedram. Balancing on-chip network latency in multi-application mapping for chip-multiprocessors. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 872–881. IEEE, 2014.

[73] Wen Zong and Qiang Xu. Doart: A low-power and low-latency network-on-chip. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 352–355. IEEE, 2016.