Last-Level Cache Side-Channel Attacks Are Feasible in the Modern Public Cloud

Zirui Neil Zhao
University of Illinois Urbana-Champaign, USA
ziruiz6@illinois.edu

Adam Morrison
Tel Aviv University, Israel
mad@cs.tau.ac.il

Christopher W. Fletcher
University of Illinois Urbana-Champaign, USA
cwfletch@illinois.edu

Josep Torrellas
University of Illinois Urbana-Champaign, USA
torrella@illinois.edu

Abstract

Last-level cache side-channel attacks have been mostly demonstrated in highly-controlled, quiescent local environments. Hence, it is unclear whether such attacks are feasible in a production cloud environment. In the cloud, side channels are flooded with noise from activities of other tenants and, in Function-as-a-Service (FaaS) workloads, the attacker has a very limited time window to mount the attack.

In this paper, we show that such attacks are feasible in practice, although they require new techniques. We present an end-to-end, cross-tenant attack on a vulnerable ECDSA implementation in the public FaaS Google Cloud Run environment. We introduce several new techniques to improve every step of the attack. First, to speed-up the generation of eviction sets, we introduce *L2-driven candidate address filtering* and a *Binary Search-based* algorithm for address pruning. Second, to monitor victim memory accesses with high time resolution, we introduce *Parallel Probing*. Finally, we leverage *power spectral density* from signal processing to easily identify the victim's target cache set in the frequency domain. Overall, using these mechanisms, we extract a median value of 81% of the secret ECDSA nonce bits from a victim container in 19 seconds on average.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Security and privacy** → **Hardware attacks and countermeasures**.

Keywords: Cloud computing, Last-level cache side-channel attack, Prime+Probe attack

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0385-0/24/04
https://doi.org/10.1145/3620665.3640403

ACM Reference Format:
Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2024. Last-Level Cache Side-Channel Attacks Are Feasible in the Modern Public Cloud. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3620665.3640403

1 Introduction

In modern public cloud environments, mutually-distrusting tenants share the underlying physical hardware resources. As a result, an attacker can monitor a victim tenant's secret-dependent usage of shared resources through various microarchitectural side channels and exfiltrate sensitive information [13–16, 24, 30, 32, 41, 42, 46, 48, 51, 56, 57, 60, 64, 65, 68, 69, 86, 92, 99, 102, 104, 106, 108].

A particularly dangerous class of attacks is Prime+Probe attacks on the last-level cache (LLC) [29, 39, 41, 51, 56, 71, 102]. This is because these attacks do not require the attacker to share a physical core or memory pages with the victim program. In such an attack, the attacker monitors the victim program's secret-dependent accesses to one or several LLC sets. We refer to these LLC sets as the *target LLC sets*.

Mounting LLC Prime+Probe attacks in the modern public cloud requires several steps [39, 41, 56, 75, 111], as listed in Table 1. First, the attacker *co-locates* their program with the target victim program on the same physical machine (Step 0) [75, 89, 100, 111]. Second, the attacker prepares LLC channels by constructing LLC eviction sets (Step 1) [41, 56]. An *Eviction Set* for a specific LLC set is a set of addresses that, once accessed, can evict any cache line mapped to that LLC set [41, 56]. Using an eviction set for an LLC set s, the attacker can monitor victim memory accesses to s.

In practice, the attacker does not generally know the target LLC sets. Hence, in Step 1, the attacker needs to build *hundreds to tens of thousands* of eviction sets, each corresponding to a potential target LLC set [41, 56, 63]. Then, the attacker scans through the potential target LLC sets and identifies the actual target LLC sets (Step 2). Finally, the attacker monitors the target LLC sets with Prime+Probe and exfiltrates the

Table 1. Steps of an LLC Prime+Probe attack in clouds.

Step	Description	Discussed in
Step 0. Co-location	Co-locate the attacker program on the same physical host as the target victim program	[111]
Step 1. Prepare LLC side channels	Construct numerous eviction sets, each corresponding to a potential target LLC set	Sections 4–5
Step 2. Identify target LLC sets	Scan LLC sets to identify those that the victim accesses in a secret-dependent manner	Sections 6–7
Step 3. Exfiltrate information	Monitor the target LLC sets and extract information	Sections 6–7

secret (Step 3). Since co-location can be achieved using techniques discussed in our prior work [111], this paper focuses on Steps 1–3, hereafter referred to as "attack steps."

1.1 Challenges of LLC Prime+Probe in Clouds

Despite the potency of LLC Prime+Probe attacks, executing them in a modern public cloud environment is challenging for several reasons. First, the modern cloud is *noisy*, as the hardware is shared by many tenants to attain high computation density [28, 87, 105]. In particular, the LLC is flooded with noise created by activities of other tenants. This noise not only interferes with eviction set construction (Step 1), but also poses challenges to identifying the target LLC sets (Step 2) and exfiltrating information (Step 3).

Second, the modern cloud is *dynamic*. With cloud computing paradigms like Function-as-a-Service (FaaS) [7, 8, 19], user workloads are typically short-lived on a host (e.g., they last only from a few minutes to tens of minutes [4, 9, 20, 35, 93]). As a result, the attacker has a *limited time window* to complete *all* the attack steps while co-locating with the victim. This challenge is exacerbated by the increased number of LLC sets in modern processors—which require preparing more eviction sets and monitoring more cache sets.

Third, the lack of huge pages in some containerized environments [19] and the wide adoption of non-inclusive LLCs increase the effort to execute LLC Prime+Probe attacks in clouds [102]. Thus, while İnci et al. [38, 39] conducted an LLC Prime+Probe attack on AWS EC2 in 2015, their techniques are incompatible with modern clouds, as they relied on huge pages, long-running attack steps, and inclusive LLCs.

As a result of the aforementioned challenges, cloud vendors believe that LLC Prime+Probe attacks are not a threat "in the wild." For instance, the security design whitepaper of Amazon's Elastic Compute Cloud (EC2) [6] explicitly rules out LLC Prime+Probe attacks as impractical [3].

1.2 This Paper

This paper refutes the belief that LLC Prime+Probe attacks are impractical in the noisy modern public cloud. We demonstrate an end-to-end, cross-tenant attack on cryptographic code (a vulnerable ECDSA implementation [62]) on Cloud Run [19], an FaaS platform from Google Cloud [18]. Every step of the attack requires new techniques to address the practical challenges posed by the cloud. While our demonstrated attack targets Google Cloud Run, the techniques that we develop are applicable to any modern Intel server with a non-inclusive LLC. Therefore, we believe that multi-tenant cloud products from other vendors, such as AWS [5] and Azure [10], may also be susceptible to our attack techniques.

This paper makes the following contributions:

① Existing Prime+Probe approaches fail in the cloud. We show that Steps 1–3 of Prime+Probe in Table 1 are made harder in the Cloud Run environment. In particular, we show that state-of-the-art eviction set construction algorithms, such as group testing [73, 81, 90] and Prime+Scope [71], have a low chance of successfully constructing eviction sets on Cloud Run. Due to the noise present, they take 10× to 24× as much time as when operating in a quiescent local setting. Since the attacker needs to construct eviction sets for up to tens of thousands of LLC sets within a limited time window, this low performance makes existing eviction set construction algorithms unsuitable for the public cloud.

② Effective construction of eviction sets in the cloud. To speed-up the generation of eviction sets in Step 1, we introduce: (1) a generic optimization named *L2-driven candidate address filtering* that is applicable to all eviction set construction algorithms, and (2) a new *Binary Search-based* eviction set construction algorithm. By combining these two techniques, it takes only 2.4 minutes on average to construct eviction sets for all the 57,344 LLC sets of an Intel Skylake-SP machine in the noisy Cloud Run environment, with a median success rate of 99.1%. In contrast, utilizing the well-optimized state-of-the-art eviction set construction algorithms, this process is expected to take at least 14.6 hours.

③ Techniques for victim monitoring and target set identification. We develop two novel techniques for Steps 2–3. The first one, called *Parallel Probing*, enables the monitoring of the victim's memory accesses with high time resolution and with a quick recovery from the noise created by other tenants' accesses. The technique probes a cache set with overlapped accesses, thus featuring a short probe latency and a simple high-performance prime pattern.

The second technique identifies the target LLC sets in a noise-resilient manner. This technique leverages *power spectral density* [96] from signal processing to detect the victim's periodic accesses to the target LLC set in the frequency domain. It enables the attacker to identify the target LLC set in 6.1 s, with an average success rate of 94.1%.

④ End-to-end attack in production. Using these techniques, we showcase an *end-to-end*, *cross-tenant* attack on a vulnerable ECDSA implementation [62] on Cloud Run. We successfully extract a median value of 81% of the secret ECDSA nonce bits from a victim container. The complete end-to-end attack, which includes Steps 1–3 from Table 1,

takes approximately 19 seconds on average after co-locating on the victim's host.

Availability. We open sourced our implementations at https://github.com/zzrcxb/LLCFeasible.

2 Background

2.1 Cache-Based Side-Channel Attacks

Since caches are shared between processes in different security domains, they provide an opportunity for an attacker to exfiltrate sensitive information about a victim process by observing their cache utilization. This constitutes a *cache-based side-channel attack*. Such attacks can be classified into *reuse-based attacks* and *contention-based attacks* [54].

Reused-based attacks rely on shared memory between attacker and victim, often the consequence of memory deduplication [2]. In such attacks, the attacker monitors whether shared data are brought to the cache due to victim accesses. Notable examples of such attacks include Flush+Reload [104], Flush+Flush [32], and Evict+Reload [33]. However, as memory deduplication across security domains is disabled in the cloud [3, 79], these attacks are inapplicable.

Contention-based attacks, such as Prime+Probe [64, 68], do not require shared memory between attacker and victim. In a Prime+Probe attack, the attacker monitors the victim's memory accesses to a specific cache set s. The attack begins with the attacker *priming* s by filling all its ways with attacker's cache lines. Subsequently, the attacker continuously *probes* these lines, measuring the latency of accessing them. If the victim accesses s, it evicts one of the attacker's cache lines, which the attacker can detect through increased probe latency. The attacker then re-primes s and repeats the probing process to continue monitoring.

Cloud vendors generally prevent processes of different tenants from sharing the same physical core at the same time [3, 50]. Therefore, the attacker has to perform a *cross-core* attack targeting the shared LLC. On modern processors, the LLC is split into multiple slices. Each physical address is hashed to one of the slices.

2.2 Eviction Sets

A key step in Prime+Probe is the construction of an *eviction set* [56, 90]. An eviction set for a specific cache set s is a set of addresses that, once accessed, evict any cache line mapped to s [56, 90]. Given a W-way cache, an eviction set needs to contain at least W addresses that are mapped to s. These addresses are referred to as *congruent addresses* [90]. An eviction set is *minimal* if it has only W congruent addresses.

2.2.1 Eviction Set Construction Algorithms. Building a minimal eviction set for a cache set s generally consists of two steps [56, 90]. The first step is to create a *candidate set* that contains a sufficiently large set of *candidate addresses*, of which at least W addresses are congruent in s. The second step is to prune the candidate set into a minimal set.

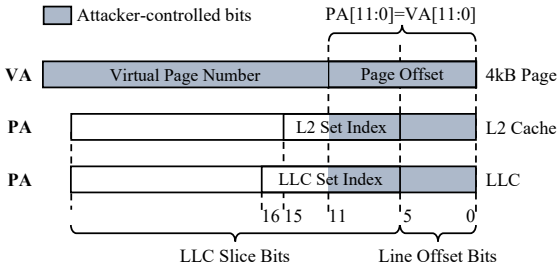


Figure 1. Mapping addresses to Skylake-SP's L2 and LLC.

① **Candidate set construction.** When a program accesses a virtual address (VA), the address is translated to a physical address (PA) during the access. Part of the PA is used to determine to which cache set the PA maps. For example, Figure 1 illustrates the address mapping of Intel Skylake-SP's L2 and LLC. The L2 uses PA bits 15–6 as the set index bits to map a PA to an L2 set. The LLC uses PA bits 16–6 as the set index bits. All the PA bits except for the low-order 6 bits are used to map a PA to an LLC slice [58]. The low-order 6 bits of the PA and VA are shared and are the line offset bits. The low-order 12 bits of the PA and VA are also shared and are the page offset bits. This is because the standard page size is 4 kB.

An unprivileged attacker can control only the page offset of a PA. They lack control and knowledge of the higher-order PA bits. As a result, the attacker has only partial control and knowledge of the set index bits of the L2 and LLC, as well as of the slice index bits of the LLC. Therefore, for a given attacker-controlled VA, there are a number of possible L2 or LLC sets to which it may map. We refer to this number as the *cache uncertainty*, denoted by U.

In general, the set index bits are directly used as the set number. Therefore, the L2 cache's uncertainty is $U_{L2} = 2^{n_{uc}}$, where n_{uc} is the number of uncontrollable L2 set index bits. For the sliced LLC, its uncertainty depends on the slice hash function. On modern processors, LLC slice bits usually map to individual slices via a complex, non-linear hash function [58, 102]. As a result, partial control over the slice index bits is not enough to reduce the number of possible slices that a VA might hash to. Hence, the LLC's uncertainty is $U_{LLC} = 2^{n_{uc}} \times n_{slices}$, where n_{uc} is the number of uncontrollable LLC set index bits and n_{slices} is the number of LLC slices. In the Skylake-SP's address mapping shown in Figure 1, there are 5 uncontrollable LLC set index bits and 4 uncontrollable L2 set index bits. Hence, a 28-slice Skylake-SP has an LLC uncertainty of $U_{LLC} = 2^5 \times 28 = 896$ sets and an L2 uncertainty of $U_{L2} = 2^4 = 16$ sets.

When constructing a candidate set for a target cache set s at page offset o, the set needs to contain a large number of addresses with page offset o due to cache uncertainty. Intuitively, the cache uncertainty U indicates how *unlikely* a candidate address maps to s. Therefore, the greater the value of U, the larger the candidate set needs to be [81, 90].

② **Pruning the candidate set into a minimal eviction set.** Given a candidate set, there are several algorithms [56, 70, 71, 73, 90, 101] to build a minimal eviction set with W congruent addresses. We briefly describe two state-of-the-art algorithms [71, 73, 90]. To simplify the discussion, we assume that there is an address T_a that is mapped to cache set s and accessible by the attacker. Consequently, the attacker can determine if a set of addresses forms an eviction set for s by testing whether they evict T_a after being accessed.

Algorithm 1: Group testing [73, 90]. Group testing splits the candidate set into G groups of approximately the same size. A common choice of G is $G = W + 1$ [73, 90]. After the split, the algorithm withholds one group from the candidate set and tests whether the remaining addresses can still evict T_a. This process involves first loading T_a into the cache, traversing the remaining addresses, and timing an access to T_a to check if it remains cached. If T_a is evicted, the withheld group is discarded and the candidate set is reduced; otherwise, the withheld group is added back to the candidate set and the algorithm withholds a different group. Overall, with $G = W + 1$, group testing has a complexity of $O(W^2N)$ memory accesses [90], where W is the associativity of the target cache and N is the candidate set size.

Algorithm 2: Prime+Scope [71]. Prime+Scope first loads T_a. Then, it *sequentially* accesses each candidate address from the list. After each candidate address is accessed, the algorithm checks whether T_a is still cached. If it is not, that indicates that the last accessed candidate address is congruent, and it is added to the eviction set. This search for congruent addresses is repeated until W different congruent addresses are identified, which form a minimal eviction set for s.

2.2.2 Number of Eviction Sets.

In practice, a victim often accesses only a few target cache sets in a secret-dependent manner. An unprivileged attacker, however, generally has limited or no information about the locations of these target cache sets. Consequently, in Step 1 of Table 1, the attacker needs to build eviction sets for all possible cache sets that might be the targets. Subsequently, in Step 2, the attacker uses Prime+Probe to monitor each of these possible cache sets to identify the actual target cache sets.

The number of eviction sets that the attacker needs to build and monitor depends on how much information about the target cache sets the attacker has. If the attacker knows the page offset of a target cache set, they only need to build eviction sets for cache sets corresponding to that page offset and monitor such sets [56, 63]. We refer to this scenario as PageOffset. Conversely, if the attacker has no information about the target sets, they must construct eviction sets for all cache sets in the system and monitor them [56, 63]. We refer to this scenario as WholeSys. Considering the standard 4 kB page size and 64 B cache line size, the attacker in the WholeSys scenario needs to build and monitor 64× as many eviction sets as in the PageOffset scenario. For a 28-slice Skylake-SP CPU, the attacker needs to build $U_{LLC} = 896$ eviction sets for the LLC sets at a give page offset and $U_{LLC} \times 64 = 57{,}344$ eviction sets for all LLC sets in the system.

2.2.3 Bulk Eviction Set Construction.

The process of constructing eviction sets for the PageOffset or WholeSys scenarios is based on the procedure to build a single eviction set. Because one can construct eviction sets for the WholeSys scenario by repeating the process for the PageOffset scenario at each possible page offset, we only explain the generation of eviction sets for the PageOffset scenario.

First, we build a candidate set containing addresses with the target page offset. The candidate set needs to contain enough congruent addresses for *any cache set* at that page offset. Second, we pick and remove one address from the candidate set and use it as the target address T_a. Third, we use either of the address pruning algorithms in Section 2.2.1 to build an eviction set for the cache set to which T_a maps. The constructed eviction set is removed from the candidate set and saved to a list L containing all the eviction sets that have been built so far. Fourth, we pick and remove another address A from the reduced candidate set. If A cannot be evicted by any eviction set in L, we use A as the target address T_a and proceed to the third step to construct a new eviction set; otherwise, we discard A and repeat the fourth step. We stop when either we run out of candidate addresses or enough eviction sets have been built.

2.3 Non-Inclusive LLC in Intel Server CPUs

Beginning with the Skylake-SP microarchitecture [84], Intel adopted a non-inclusive LLC design on their server platforms. Under this design, cache lines in private caches may or may not exist in the LLC. The Snoop Filter (SF) [84] tracks the ownership of cache lines present only in private caches, serving as a coherence directory for such cache lines. Similar to the LLC, the SF is shared among cores and sliced. The SF has the same number of sets, number of slices, and slice hash function as the LLC. Therefore, if two addresses map to the same LLC set, they also map to the same SF set.

The interactions among private caches, SF, and LLC are complex and undocumented. Based on prior work [102] and our reverse engineering, we provide a brief overview of these interactions, acknowledging that our descriptions may not be entirely accurate or exhaustive.

Lines that are in state Exclusive (E) or Modified (M) in one of the private caches are tracked by the SF; we call these lines *private*. Lines that are in state Shared (S) in at least one of the private caches are tracked by the LLC (and, therefore, are also cached in the LLC); we call these lines *shared*.

When an SF entry is evicted, the corresponding line in the private cache is also evicted. The evicted line may be inserted into the LLC depending on a reuse predictor [40, 82]. When a line cached in the LLC needs to transition to state E or M due to an access, it is removed from the LLC and an SF

Table 2. Parameters of the Skylake-SP cache hierarchy.

Structure	Parameters
L1	Data/Instruction: 32 kB, 8 ways, 64 sets, 64 B line
L2	1 MB, 16 ways, 1,024 sets, non-inclusive to L1
LLC Slice	1.375 MB, 11 ways, 2,048 sets, non-inclusive to L1/L2
SF Slice	12 ways, 2,048 sets
Num. Slices	Up to 28 slices. A 28-slice LLC and SF is the most common configuration in Cloud Run datacenters

entry is allocated to track it. When a private line transitions to state S, it is inserted into the LLC, and its SF entry is freed.

2.4 Function-as-a-Service (FaaS) Platform

Function-as-a-Service (FaaS) [7, 8, 19] is a popular cloud computing paradigm. The basic unit of execution is a function, which executes in an ephemeral, stateless container or micro virtual machine created and scheduled on demand in an event-driven manner. Applications are then composed of multiple functions that communicate with each other. Users upload their functions and the cloud provider supplies all the libraries, runtime environment, and system services needed to run them. The FaaS platform orchestrator automatically adjusts the number of container instances to match the function invocation demand. These instances often have a short lifetime [4, 9, 20]. This design allows the concurrent execution of many instances on a single physical host, improving hardware utilization.

In this paper, we use the FaaS Google Cloud Run platform [19]. In our experiments, we find that the CPU microarchitecture used in Cloud Run datacenters is dominated by Intel Skylake-SP and Cascade Lake-SP. Since these two microarchitectures have similar cache hierarchies, we focus our discussion on Skylake-SP. Table 2 lists the key parameters of Skylake-SP's cache hierarchy.

3 Threat Model

In this paper, we assume an attacker who aims to exfiltrate sensitive information from a victim containerized service running on a public FaaS platform such as Cloud Run [19] through LLC side channels. In our prior work [111], we have demonstrated how an attacker can co-locate with a target victim container on Cloud Run. If the victim runs container instances on multiple hosts, our techniques can co-locate attacker containers with a large portion of the victim instances. Therefore, we assume the co-location step is completed and focus on Steps 1–3 from Table 1.

We assume that the attacker is an unprivileged user of a FaaS platform. The attacker's interaction with the FaaS platform is limited to the standard interfaces that are available to all platform users. Using these interfaces, the attacker can deploy services that contain attacker-controlled binaries. Finally, we assume that the attacker can trigger the victim's execution by sending requests to the victim service, either directly or through interaction with a public web application that the victim service is part of.

Since cloud vendors typically prevent different users from simultaneously using the same physical core via Simultaneous Multithreading (SMT) [3, 50], the attacker must perform a *cross-core* cache attack. Similar to prior work [71, 102] that targets Intel Skylake-SP, we create eviction sets for the SF and monitor the SF for the victim's accesses. Note that an SF eviction set is also an LLC eviction set, as the SF and LLC share the same set mapping and the SF has more ways.

Lastly, we found that user containers on Cloud Run are unable to allocate huge pages. Therefore, we assume that the attacker can only rely on the standard 4 kB pages to construct eviction sets. This assumption is consistent with other restricted execution environments [31, 63, 78, 90] and places fewer requirements on the attacker's capability.

4 Existing Eviction Sets Fail in the Cloud

In this section, we show that existing algorithms to construct eviction sets fail in the cloud. This is because of the noise in the environment and the reduced time window available to construct the eviction sets. In the following, we first examine the resilience to environmental noise of a core primitive used by all address pruning algorithms (Section 4.1). Then, we evaluate the success rate and execution time of the two state-of-the-art address pruning algorithms on Cloud Run (Section 4.2), and investigate the reasons why they fall short in the cloud (Section 4.3).

4.1 TestEviction Primitive & Its Noise Susceptibility

All the address pruning algorithms require a primitive that tests whether a target cache line is evicted from the target cache after a set of candidate addresses are accessed [56, 73, 90, 102]. We refer to this generic primitive as *TestEviction*. Specifically, group testing uses *TestEviction* to prune away non-congruent addresses, while Prime+Scope employs it to identify congruent addresses.

Due to environmental noise, *TestEviction* can return *false-positive* results—i.e., the target cache line is evicted by accesses from other tenants and not by the accesses to the candidate addresses. When this occurs in group testing, the algorithm may discard a group of addresses with congruent addresses, falsely believing that the remaining addresses contain enough congruent addresses. Similarly, Prime+Scope can misidentify a non-congruent address as a congruent one, incorrectly including it in the eviction set. In both cases, the algorithms may fail to construct an eviction set.

In general, the longer the execution time of *TestEviction* is, the more susceptible it becomes to noise, due to the increased likelihood of the target cache set being accessed by other tenants during its execution. Thus, the execution time of *TestEviction* not only affects the end-to-end execution time of the algorithm, but also the algorithm's resilience to noise.

Prior work [90] that uses the group testing algorithm implements *TestEviction* with linked-list traversal [91]. As this

implementation serializes memory accesses to candidate addresses, we refer to this type of implementation as *sequential TestEviction*. Prime+Scope also uses sequential *TestEviction*, as it tests whether a target line is still cached after *each* access to a candidate address. Since sequential *TestEviction* does not exploit memory-level parallelism (MLP), it has a long execution time.

In our work, we find that overlapping accesses to candidate addresses to exploit MLP can significantly reduce the execution time of *TestEviction*. We refer to this implementation as *parallel TestEviction*. It is based on a pattern proposed by Gruss et al. [31], and our implementation can be found online [109]. However, as will be shown in Sections 4.2 and 4.3, even though parallel *TestEviction* is significantly faster than sequential *TestEviction*, it alone is not enough to overcome the noise in the cloud. In the rest of this paper, we use parallel *TestEviction* in all algorithms except for Prime+Scope, which is incompatible with parallel *TestEviction* due to its algorithm design.

4.2 Noise Resilience of Existing Algorithms

In this section, we implement both the group testing and Prime+Scope algorithms for Skylake-SP's SF. We then evaluate their success rates and execution times in a local environment with minimal noise, as well as in the Cloud Run environment, which features a significant level of environmental noise from other tenants.

Implementation. Following prior work that builds SF eviction sets [71, 102], we first construct a minimal LLC eviction set comprising 11 congruent addresses, and then expand it to an SF eviction set by finding one additional congruent address. To insert cache lines into the LLC, we use a helper thread running on a different physical core that repeats the accesses made by the main thread. These repeated accesses turn the state of the cache lines to *S*, and thus cause the lines to be stored in the LLC (Section 2.3). Similar techniques are used in prior work [71, 102]. Finally, as per Section 4.1, our group testing implementation uses parallel *TestEviction*, while our Prime+Scope implementation uses sequential *TestEviction*.

To ensure a fair comparison among algorithms, we re-implement group testing and Prime+Scope using the same data structures to store candidate sets and eviction sets, and the same primitives to test whether a set of addresses is an eviction set. We call these algorithms Gt and Ps, respectively. In addition, we also implement optimized versions of these algorithms for Skylake-SP. Details of these optimizations are presented in the extended version of this paper [112]. We call these optimized algorithms GtOp and PsOp.

Experiment setup. We evaluate these algorithms in both a cloud setup and a local setup.

Cloud setup. We deploy our attacker service to the *us-central1* data center, where we observe the largest Cloud Run cluster. Since our setup requires a concurrently running helper thread, each attacker instance requests 2 physical cores. In *us-central1*, the predominant CPU model used by Cloud Run is the Intel Xeon Platinum 8173M, which is a Skylake-SP processor with 28 LLC/SF slices.

During each experiment, we launch 300 attacker instances and retain only one per host. We then use each algorithm to build SF eviction sets for 50 random cache sets. To measure the effects of environmental noise fluctuations due to computation demand changes, we repeat our experiments for five days and at four different periods each day, namely, morning (9–11am), afternoon (3–5pm), evening (8–10pm), and early morning (3–5am). Altogether, we conducted 1,767 experiments on Cloud Run, totaling 88,350 eviction set constructions for each algorithm.

Local setup. Our local setup uses a Skylake-SP processor with the Intel Xeon Gold 6152, which has 22 LLC/SF slices. During the experiment, the system operates with minimal activity beyond the running attacker container instance. We employ each algorithm to construct 1,000 SF eviction sets.

Algorithms. For each SF eviction set, we allow each algorithm to make at most 10 construction attempts. If the algorithm fails these many times or it takes more than 1,000 ms to complete, we declare its failure. For group testing, which uses backtracking to recover from erroneous *TestEviction* results, we permit at most 20 backtracks per attempt.

In our experiments, we need to start by generating a set of candidate addresses for a given page offset. Empirically, we find that a set with $3UW$ candidate addresses is enough for Skylake-SP's LLC/SF, where U and W are the cache uncertainty (Section 2.2.1) and associativity, respectively.

Results. Table 3 shows the effectiveness of the state-of-the-art algorithms to construct an eviction set for SF in different environments: quiescent local, Cloud Run, and Cloud Run from 3am to 5am, which are typically considered "quiet hours". The metrics shown are the success rate, average execution time, standard deviation of execution time, and median execution time. The success rate is the probability of successfully constructing an SF eviction set. The execution time measures the real-world time that it takes to reduce a candidate set to an LLC eviction set and then extend it with one additional congruent address to form an SF eviction set.

We see that all algorithms achieve very high success rates and good performance in the quiescent local environment. However, on Cloud Run, where there is substantial environmental noise from other tenants, all algorithms suffer significant degradation in both success rate and performance. Moreover, we do not observe significant variations in success rate or execution time across different periods of a day, including the 3am to 5am quiet hours. We believe this could be due to certain server consolidation mechanisms that adjust the number of active hosts based on demand [11, 12, 49], leading to a relatively constant load level on active hosts throughout the day.

Table 3. Effectiveness of the state-of-the-art address pruning algorithms in different environments. The metrics shown are: success rate, average execution time, standard deviation of execution time, and median execution time.

Env.	Metrics	Gt	GtOp	Ps	PsOp
Quiescent Local	Succ. Rate	97.0%	98.8%	98.5%	98.2%
	Avg. Time	32.9 ms	21.1 ms	55.9 ms	54.9 ms
	Stddev Time	72 ms	35 ms	166 ms	156 ms
	Med. Time	18.5 ms	13.7 ms	23.8 ms	21.7 ms
Cloud Run	Succ. Rate	39.4%	56.0%	3.2%	6.9%
	Avg. Time	714 ms	512 ms	580 ms	572 ms
	Stddev Time	476 ms	457 ms	329 ms	331 ms
	Med. Time	1,015 ms	384 ms	504 ms	495 ms
Cloud Run (3-5am)	Succ. Rate	41.4%	57.2%	3.7%	7.6%
	Avg. Time	693 ms	499 ms	581 ms	576 ms
	Stddev Time	482 ms	456 ms	327 ms	332 ms
	Med. Time	1,009 ms	350 ms	509 ms	502 ms

Implications. As discussed in Section 2.2.2, an unprivileged attacker needs to construct eviction sets for all SF sets at a given page offset (PageOffset) or in the system (WholeSys). We estimate the time to construct many eviction sets as $n_{sets} \times t_{avg}/SR$, where n_{sets} is the number of eviction sets we need to build, t_{avg} is the average execution time of attempting to construct one eviction set, and SR is the success rate. Similar metrics are also used in prior work [81].

For the Skylake-SP processor that we are targeting, the attacker needs to build 896 and 57,344 SF eviction sets in the PageOffset and WholeSys scenarios respectively (Section 2.2.2). Hence, on Cloud Run, GtOp, the fastest and most noise-resilient of the evaluated algorithms, would take 13.7 minutes and 14.6 hours to construct eviction sets required in the PageOffset and WholeSys scenarios, respectively.

We performed two additional small-scale experiments to validate our estimation. In the first experiment, which is conducted on 95 hosts, GtOp attempts to construct the 896 eviction sets required in the PageOffset scenario. GtOp takes, on average, 9.9 minutes to complete the task, and it only succeeds in 37.3% of the sets. In the second experiment, which is conducted on 69 hosts, GtOp tries to construct the 57,344 eviction sets required in the WholeSys scenario. Due to the timeout constraint of Cloud Run [21], we can only run GtOp for one hour and thus report the number of eviction sets it constructs under the constraint. Our best outcome is constructing 3,741 eviction sets in one hour, with an average number of 1,074 sets in one hour. This means that building eviction sets for the system's 57,344 SF sets would take GtOp over $57{,}344/3{,}741 \approx 15$ hours even in the best case.

This performance is unsatisfactory for a practical attack on FaaS platforms for several reasons. First, on some popular FaaS platforms [7, 8], the attacker can only execute for 10 to 15 minutes before timeout [4, 9]. Even on a more permissive platform like Cloud Run, the maximum timeout is just one hour [21]. After a timeout, the attacker might *not* reconnect to the same instance [21], thus losing the attack progress. Second, container instances usually have a short lifetime before being terminated [35, 93]. Hence, the long eviction set construction time means that the co-located victim instance may get terminated before eviction sets are ready. Finally, as FaaS platforms charge customers by the CPU time, the long execution time can cause significant financial cost to the attacker. This is especially the case if the attacker is launching many attacker instances on different hosts to increase the chance of a successful attack.

4.3 Explaining the Results

Compared to a quiescent local environment, we find that the cloud environment has a drastically higher rate of LLC accesses made by other tenants, and that *TestEviction*'s execution is slower. These two factors contribute to why the state-of-the-art algorithms are ineffective in a cloud environment. Our conclusion is based on the following two experiments. To ensure meaningful comparisons, both experiments are conducted using the container instances of Section 4.2.

Experiment 1: LLC set access frequency. In this experiment, we measure how frequently an LLC set is accessed by background activities, such as system processes and processes of other tenants. The reason why we focus on the access frequency of the LLC instead of the SF is because address pruning algorithms build eviction sets in the LLC and then expand them to SF eviction sets (Section 4.2).

During the experiment, we first construct an eviction set for a randomly chosen LLC set. Then, we detect background LLC accesses with Prime+Probe [64]. We record the timestamp of each LLC access. Each experiment trial collects the timestamps of 1,000 back-to-back LLC accesses. On Cloud Run, we perform 50 trials per host (88,350 trials in total). In the local environment, we carry out 1,000 trials.

Experiment 2: TestEviction execution duration. In this experiment, we measure the execution duration of both the parallel and sequential *TestEviction* when testing varying numbers of candidate addresses. We perform the measurement in both the Cloud Run and local environments. For each host and candidate set size, we measure *TestEviction*'s execution time for 100 times after 10 warm-ups.

Results. Figure 2 shows the cumulative distribution function (CDF) of the time between LLC accesses by background activity to a randomly chosen LLC set in both environments. On Cloud Run, the average LLC access rate is 11.5 accesses per millisecond per set. In the local environment with minimal noise, the average access rate is merely 0.29 accesses per millisecond per set. Figure 3 shows the execution time of the parallel and sequential *TestEviction*, for varying numbers of candidate addresses on Cloud Run. As the execution times of *TestEviction* in the local environment follow similar trends, we omit them in the plot. It can shown that, on average, the

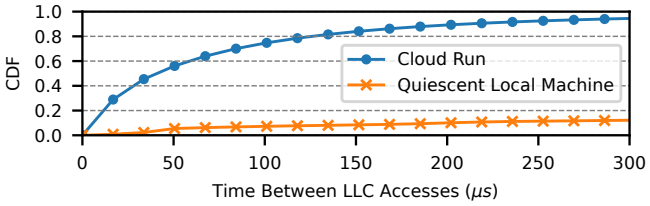


Figure 2. CDF of the time between accesses by background activity to a randomly chosen LLC set.

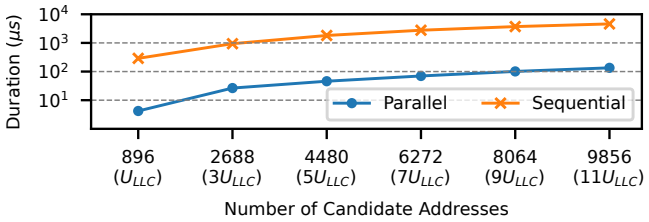


Figure 3. Different *TestEviction*'s execution times on Cloud Run under various number of candidate addresses.

execution times of the sequential and parallel *TestEviction* are 26.9% and 42.1% lower in the local environment compared to Cloud Run, respectively.

These results explain why existing address pruning algorithms show unsatisfactory effectiveness on Cloud Run. For Prime+Scope, when using sequential *TestEviction* to identify the first congruent address, it is expected to test $11U_{LLC}$ candidate addresses. This takes approximately 4.6 ms on average. However, during this time, the target LLC set is expected to experience 53.0 background LLC accesses. Consequently, Prime+Scope's *TestEviction* very likely reports erroneous results under this level of noise.

As for group testing, its parallel *TestEviction* executes an order of magnitude faster than the sequential *TestEviction*. For example, it takes only 134.8 µs to test $11U_{LLC}$ candidates. Given the background LLC access rate, the probability of the set *not* being accessed during the *TestEviction* execution is about 18.4%. This permits the parallel *TestEviction* a reasonable chance to complete without experiencing interference from background accesses. In combination with the backtracking mechanism [90], group testing has a substantially higher probability of success compared to Prime+Scope on Cloud Run. Still, both Gt and GtOp experience a large number of backtracks due to erroneous *TestEviction* results and are drastically slowed down on Cloud Run. For example, the optimized GtOp performs an average number of 32.2 backtracks per eviction set on Cloud Run, while it only needs 4.0 backtracks on average in the local environment.

5 Constructing Eviction Sets in the Cloud

Based on the insights from Section 4, we propose two techniques that enable *fast* (and therefore also noise-resilient), eviction set construction in the cloud: *L2-driven candidate address filtering* (Section 5.1) and a *Binary Search-based* algorithm for address pruning (Section 5.2).

5.1 L2-driven Candidate Address Filtering

To speed-up the eviction set construction, we propose to reduce the candidate set size with an algorithm-agnostic optimization that we call *candidate address filtering*. Our insight is that the L2 set index bits are typically a subset of the LLC/SF set index bits. For example, Skylake-SP uses PA bits 15-6 as the L2 set index and PA bits 16-6 as the LLC/SF set index (Figure 1). Hence, if addresses A and B are not congruent in the L2, then A and B have different PA bits 15-6 and, therefore, they must *not* be congruent in the LLC/SF.

Based on this insight, we introduce a new *candidate filtering step* after candidate set construction and before address pruning. Assume that we want to construct an eviction set for an LLC/SF set to which an attacker-accessible address T_a maps. To perform the candidate filtering, we first construct an L2 eviction set for T_a. Then, using the L2 eviction set, we test whether it can evict each address from the candidate set. If a candidate address A cannot be evicted by the L2 eviction set, then it implies that A and T_a are not congruent in either the L2 or the LLC/SF. Consequently, A is removed from the candidate set. After candidate filtering, the candidate set contains only addresses that are congruent with T_a in the L2. These filtered addresses are passed to the address pruning algorithm to find a minimal LLC/SF eviction set.

As Skylake-SP has an L2 uncertainty of $U_{L2} = 16$, only about 1/16 of the candidate addresses are congruent with T_a in L2. Therefore, the size of the filtered candidate set is only about 1/16 of the original set size. On a common 28-slice Skylake-SP CPU, we expect to find one congruent address every $U_{LLC} = 896$ candidates in the candidate set before filtering. With candidate filtering, we now expect to find one congruent address every $896/16 = 56$ candidates.

Since the candidate set is universally used by different address pruning algorithms, including both group testing and Prime+Scope, our candidate filtering is a generic optimization. Moreover, in modern processors, the number of L2 sets is typically smaller than the number of LLC sets in one LLC slice. Hence, the property that the L2 set index bits are a subset of the LLC set index bits generally holds for other processors as well, Therefore, the candidate filtering optimization also applies to them. Lastly, the idea of candidate filtering can be applied to a more restricted environment where the attacker cannot even control the page offset bits [90]. In such an environment, the attacker can hierarchically construct L1 and L2 eviction sets to gradually filter candidates for the next lower cache level.

5.2 Using Binary Search for Address Pruning

To further speed-up eviction set construction in the cloud, we propose a new address pruning algorithm based on binary search. Our algorithm uses *parallel TestEviction*.

Algorithm design. Given a list of candidate addresses, we test whether the first n addresses can evict a target address T_a. For a W-way cache, increasing n from zero will result in

```
1  // T_a: target address
2  // addrs: an array of candidate addresses
3  // N: size of the addrs array (N >= W)
4  size_t LB, UB = N;
5  for (size_t i = 1; i <= W; i++) {
6     LB = i - 1;
7     while (UB - LB != 1) {
8        n = (LB + UB) / 2;
9        if (TestEviction(T_a, addrs, n))
10          UB = n; // T_a can be evicted
11       else
12          LB = n; // T_a cannot be evicted
13    }
14    size_t tau_i = UB;
15    swap(addrs[i], addrs[tau_i]);
16 } // addrs[1]~addrs[W] form an eviction set
```

Figure 4. Pseudo code of our proposed algorithm. All array indexes start from 1. Parallel $TestEviction(T_a, addrs, n)$ returns a boolean value that indicates whether the first n candidate addresses from array *addrs* can evict the target T_a.

a negative test outcome until the first n addresses include W congruent addresses. We define the *tipping point*, denoted by τ, as the smallest n for which the first n addresses evict T_a. Therefore, τ is the index of the W-th congruent address in the list, assuming that the indexation begins from 1. For a given n, if the first n addresses evict T_a, it means that $n \geq \tau$; otherwise, $n < \tau$. Our main idea is to use binary search to efficiently determine τ and thus identify one congruent address. Then, we exclude the congruent address from any future search, and repeat the binary search process until W different congruent addresses are found.

Figure 4 shows the pseudo code of the algorithm. It takes as inputs a target address T_a, an array of addresses *addrs* representing the candidate set, and the array size N. The array *addrs* should contain at least W congruent addresses, and thus $N \geq W$. The algorithm iteratively finds W congruent addresses by finding the tipping point at each iteration (Lines 5–16 in Figure 4). Within each iteration, the algorithm tests in a loop if the first $n = \lfloor (LB + UB)/2 \rfloor$ addresses from *addrs* can evict T_a (Line 9). The variables LB and UB are then updated in a manner that LB always represents the largest n such that the first n addresses *cannot* evict T_a and UB always represents the smallest n such that the first n addresses *can* evict T_a. Therefore, when $UB = LB + 1$, UB is the tipping point of iteration i, denoted by τ_i. Consequently, the τ_i-th address of the array is a congruent address. The algorithm then swaps the just-found congruent address with the i-th address in *addrs* and proceeds to the next iteration (Line 15).

Before the binary search in the next iteration starts, LB is reset to $i - 1$ (Line 6), as the first $i - 1$ addresses are the congruent addresses found in previous iterations and are thus excluded from the search. In contrast, UB needs *not* to be reset to N, as the first UB addresses always contain W congruent addresses due to the swapping. Finally, after W iterations, the first W addresses in *addrs* form a minimal eviction set for T_a (Line 16).

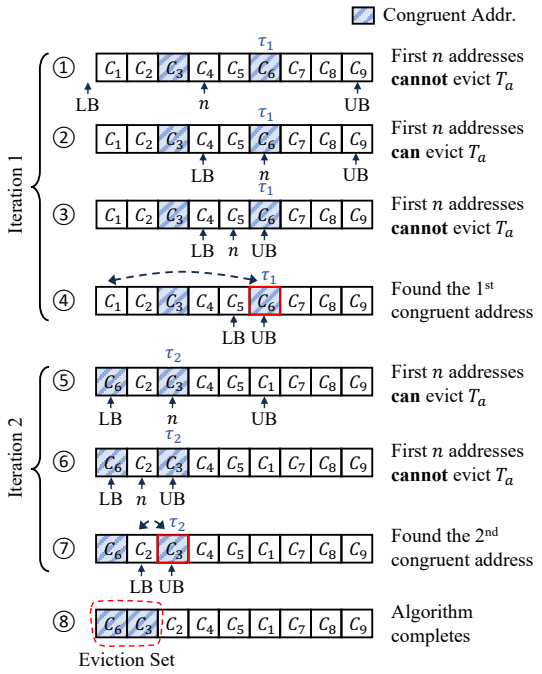


Figure 5. Illustration of our proposed binary search-based algorithm (assuming $W = 2$). Blocks with shaded pattern represent congruent candidate addresses.

Example. Figure 5 demonstrates the algorithm with a nine-address candidate set $(C_1, C_2, \ldots, C_9)$ and a target cache with associativity $W = 2$. Initially, we set $i = 1$, $LB = 0$, $UB = N = 9$, and $n = \lfloor (UB+LB)/2 \rfloor = 4$ (Step ①). Because the first $n = 4$ addresses cannot evict T_a, we set $LB = n = 4$ and update n to $\lfloor (UB + LB)/2 \rfloor = 6$ (Step ②). With the updated n, the first $n = 6$ addresses now can evict T_a, so we set $UB = n = 6$ and update $n = 5$ (Step ③). This process is repeated until $UB = LB + 1 = 6$ (Step ④). At this point, C_6 is found to be a congruent address, which is saved to the front of the list by swapping it with C_1. Then, we increment i to 2, set $LB = i - 1 = 1$ without changing UB (Step ⑤), and repeat the binary search (Steps ⑤–⑦). The algorithm finishes once W congruent addresses are found (Step ⑧), which form a minimal eviction set for T_a.

Backtracking mechanism. When the $TestEviction$ returns a false-positive result due to environmental noise, our algorithm can incorrectly set UB to a value smaller than τ. As a result, the binary search may incorrectly identify a non-congruent address as a congruent one. This erroneous state is detected if the first UB addresses cannot evict T_a after the binary search for the iteration finishes. To recover from this state, we gradually increase UB with a large stride until the first UB addresses can evict T_a and restart the binary search.

Comparison to existing algorithms. Unlike Prime+Scope, our algorithm uses parallel $TestEviction$. As discussed in

Section 4.3, parallel *TestEviction* is at least an order of magnitude faster than sequential *TestEviction*. Therefore, our algorithm is faster than Prime+Scope.

Compared to group testing, both our algorithm and group testing can use parallel *TestEviction*. Assume that we use the number of memory accesses as a proxy for execution time. Using our algorithm, it takes $O(\log N)$ parallel *TestEviction* executions to find a tipping point, where N is the candidate set size. Since each parallel *TestEviction* needs to make $O(N)$ memory accesses, it takes $O(N \log N)$ accesses to find one congruent address. As we need to find W congruent addresses, the end-to-end execution requires $O(WN \log N)$ accesses. In contrast, group testing requires $O(W^2N)$ accesses. Therefore, whether group testing or our algorithm makes fewer accesses, and consequently executes faster, depends on the specific values of W and $\log N$.

As an intuitive comparison, the ratio of the number of accesses made by group testing over our algorithm is estimated by $O(W/\log N)$. Since we use $N = 3UW$ (Section 4.2), we rewrite the ratio as $O(W/\log(UW))$. This suggests that in caches with high associativity (i.e., a large W), group testing *tends* to make more accesses than our algorithm. This is supported by our experiments in Section 5.3.

5.3 Evaluating Our Optimizations

We evaluate group testing, Prime+Scope, and our binary search-based algorithm with candidate filtering in both the Cloud Run and local environments. We use the same methodology as the experiment in Section 4.2, except for reducing the time limit of constructing one eviction set to 100 ms (because of candidate filtering). Each algorithm is evaluated in three scenarios: (1) SingleSet, where we construct a single eviction set for a randomly chosen SF set; (2) PageOffset, where we construct eviction sets for all SF sets at a randomly chosen page offset; and (3) WholeSys, where we construct eviction sets for all SF sets in the system. Our experiments include 88,350 and 1,000 measurements per algorithm in the cloud and local environments, respectively, in SingleSet; 8,835 and 100 in PageOffset; and 1,767 and 20 in WholeSys.

Results. Table 4 lists the success rate and execution time of each algorithm under different scenarios in both the Cloud Run and local environments. The execution time measures *both candidate filtering and address pruning*. As we find that Ps and PsOp have similar success rates and execution times after applying candidate filtering, Table 4 only shows the one with the shortest average execution time, and calls it PsBst. We call our binary search-based algorithm BinS.

The SingleSet scenario in Table 4 is directly comparable to the scenario in Table 3. Table 4 shows the effectiveness of candidate filtering on Cloud Run, as it leads to significantly shortened execution times. For example, the average execution time of GtOp is reduced from 512 ms to 27.2 ms. The resulting success rate also increases substantially. Indeed, for GtOp, it goes from 56.0% to 97.7%.

Recall that the average execution time comprises both candidate filtering and addresses pruning. In the SingleSet scenario, it can be shown that candidate filtering on Cloud Run takes on average 22.3 ms, which dominates the execution time. As a result, the average execution times are similar across all algorithms. As will be shown in Section 5.3.1, the portion of the execution time spent on candidate filtering drastically decreases when building numerous eviction sets in the PageOffset and WholeSys scenarios.

Next, consider PageOffset. All the algorithms experience increases in average execution times as they go from the local to the Cloud Run environments. Comparing group testing to our algorithm on Cloud Run, we see that Gt and GtOp take 92% and 38% more time to build eviction sets on average, as we find that Gt and GtOp make 162% and 52% more memory accesses than BinS. As for PsBst, it takes on average 57% more time than BinS, due to its use of the sequential *TestEviction*.

The results for WholeSys are qualitatively similar to PageOffset, except for larger drops in success rates as we go from the local to the Cloud Run environments. Still, while the average success rates of Gt, GtOp, PsBst, and BinS on Cloud Run are 88.1%, 90.5%, 91.7%, and 92.6%, respectively, the medians are 96.7%, 98.5%, 99.4%, and 99.1%, respectively.

To summarize, the combination of candidate filtering and our binary search-based algorithm offers significant performance improvements over the *well-optimized* state-of-the-art algorithms. On Cloud Run, they reduce the time to construct eviction sets for all SF sets in the system from an expected duration of 14.6 hours (Section 4.2) to a mere 2.4 minutes (last column of Table 4), with a median success rate of 99.1%. These improvements make the LLC Prime+Probe attack in the cloud feasible.

5.3.1 Overhead of Candidate Filtering.

As indicated before, it takes 22.3 ms to complete one candidate filtering on Cloud Run. This time includes constructing one L2 eviction set and using it to filter candidates. While this time dominates the execution time when constructing a *single* eviction set (Section 5.3), the same filtered candidates can be reused to construct many eviction sets for LLC/SF sets that are mapped to the same L2 set. For example, in the 28-slice Skylake-SP processor used in our Cloud Run evaluation, constructing the 896 LLC/SF sets in the PageOffset scenario requires only 16 candidate filtering executions, which takes 435 ms on average. This execution time makes up a small portion of the total execution time in PageOffset (2.87 s in Table 4).

In the WholeSys scenario, a naive process would build eviction sets for all 1,024 L2 sets and execute candidate filtering 1,024 times. We optimize the process by exploiting the following property of the L2: if addresses A and B are congruent in L2, then $A' = A + \delta$ and $B' = B + \delta$ are also congruent in L2—as long as the δ is small enough such that A and A' belong to the same page, and B and B' belong to the same page [41, 56, 63].

Table 4. Eviction set construction effectiveness of various algorithms under different configurations. The number of eviction sets may vary between local and cloud because the experiments use machines with different number of slices.

Env.	Metrics	SingleSet # Ev sets: Local=1, Cloud=1				PageOffset # Ev sets: Local=704, Cloud=896				WholeSys # Ev sets: Local=45,056, Cloud=57,344			
		Gt	GtOp	PsBst	BinS	Gt	GtOp	PsBst	BinS	Gt	GtOp	PsBst	BinS
Quiescent Local	Succ. Rate	99.3%	99.5%	99.2%	99.9%	98.6%	99.2%	99.4%	99.5%	99.0%	99.1%	99.5%	99.5%
	Avg. Time	15.2 ms	14.7 ms	14.7 ms	14.1 ms	1.95 s	1.48 s	3.02 s	1.04 s	103.6 s	79.6 s	175.0 s	50.1 s
	Stddev Time	3.1 ms	2.6 ms	0.8 ms	2.2 ms	0.72 s	0.17 s	2.48 s	0.16 s	16.1 s	7.9 s	72.7 s	5.5 s
	Med. Time	14.7 ms	14.4 ms	14.5 ms	13.9 ms	1.77 s	1.43 s	1.39 s	1.00 s	96.8 s	76.9 s	185.6 s	48.9 s
Cloud Run	Succ. Rate	96.7%	97.7%	97.2%	98.1%	95.6%	97.4%	98.4%	98.0%	88.1%	90.5%	91.7%	92.6%
	Avg. Time	28.8 ms	27.2 ms	33.2 ms	26.6 ms	5.51 s	3.95 s	4.51 s	2.87 s	301.1 s	212.6 s	244.4 s	142.4 s
	Stddev Time	14.4 ms	10.8 ms	21.4 ms	11.6 ms	2.62 s	1.90 s	2.72 s	1.58 s	63.0 s	52.1 s	58.9 s	34.8 s
	Med. Time	25.1 ms	24.7 ms	26.7 ms	23.9 ms	4.94 s	3.52 s	3.85 s	2.53 s	290.1 s	200.4 s	229.6 s	134.2 s

Exploiting this property, we first construct 16 eviction sets for all L2 sets at page offset `0x0`. Then, we use each eviction set to generate a filtered candidate set at page offset `0x0`. Finally, we can derive a new filtered candidate set at page offset δ by adding δ to each candidate address of the filtered candidate set at page offset `0x0`. As a result, the WholeSys scenario requires only 16 L2 eviction set constructions and candidate filtering executions. The time of completing candidate filtering (435 ms) is negligible compared to the total execution time in WholeSys (142.4 s in Table 4).

5.3.2 Other Intel Server Platforms and Target Caches.

As discussed in Section 5.2, group testing tends to incur a higher execution overhead over our binary search-based algorithm when the cache associativity increases. To illustrate this trend, we measure the performance of eviction set construction on Ice Lake-SP, which features caches with higher associativity than in Skylake-SP. Specifically, Ice Lake-SP has a 16-way SF and a 20-way L2 cache, whereas Skylake-SP has a 12-way SF and a 16-way L2 cache.

Because we do not see Ice Lake-SP being used on Cloud Run, we measure the performance on local quiescent Skylake-SP and Ice Lake-SP machines. The Skylake-SP machine utilized is the same as in prior experiments. The Ice Lake-SP machine uses an Intel Xeon Gold 5320, which has 26 LLC/SF slices. For each machine and algorithm, we measure the time to construct a single SF or L2 eviction set 1000 times. Candidate filtering is enabled for SF eviction set construction, but its time is *not* included in our measurements.

First, we consider constructing eviction sets for the SF. Gt, GtOp, and BinS take, on average, 2.23 ms, 1.77 ms, and 1.17 ms, respectively, to construct a single eviction set for the 12-way SF of Skylake-SP. The same process takes Gt, GtOp, and BinS on average 3.81 ms, 3.07 ms, and 1.68 ms, respectively, for the 16-way SF of Ice Lake-SP. As we go from Skylake-SP to Ice Lake-SP, the ratio Gt/BinS and GtOp/BinS changes from 1.91 and 1.51 to 2.27 and 1.83, respectively.

Similarly, Gt, GtOp, and BinS take, on average, 2.49 ms, 1.90 ms, and 1.33 ms, respectively, to construct a single eviction set for the 16-way L2 of Skylake-SP. The same process takes Gt, GtOp, and BinS on average 14.48 ms, 8.16 ms, and 2.28 ms, respectively, for the 20-way L2 of Ice Lake-SP. As we go from Skylake-SP to Ice Lake-SP, the ratio Gt/BinS and GtOp/BinS changes from 1.87 and 1.43 to 6.35 and 3.58, respectively.

6 Monitoring Memory Accesses & Identifying Target Cache Sets

Eviction set construction is the first step of an end-to-end LLC attack (Step 1 in Table 1). In this section, we improve the remaining steps with two new techniques. First, Section 6.1 introduces *Parallel Probing*, which enables the monitoring of victim memory accesses with high time resolution. This technique optimizes Step 2 (identify target sets) and Step 3 (exfiltrate information) in Table 1 for the noisy cloud environment. Second, Section 6.2 leverages *Power Spectral Density* [83] from signal processing to easily identify the victim's target cache set. This technique optimizes Step 2 in Table 1 for the noisy cloud environment.

6.1 Parallel Probing for Memory Access Monitoring

Given a cache set to monitor, the attacker can detect memory accesses to that set with Prime+Probe (Section 2.1). It is vital that *both* prime and probe latencies are short. A short probe latency enables the attacker to monitor when accesses occurs at a high time resolution [71]. A short prime latency allows the attacker to quickly prepare the monitored cache set for detecting the next access. In a noisy cloud environment, where a cache set may be frequently accessed by processes of other tenants, failure to prime the set in a timely manner can increase the chance of missing the victim's accesses.

To minimize the probe latency, Prime+Scope [71] primes a specific line from the eviction set to become the *eviction candidate (EVC)*, which is the line to be evicted when a new line needs to be inserted into the set. This method enables the attacker to check only if the EVC remains cached. Further, since the EVC can be cached in L1, the probe latency becomes minimal, leading to a high time resolution. However, this comes at the cost of using a slower and more complex priming pattern to prepare the replacement state [71], which can reduce monitoring effectiveness in a noisy environment.

Table 5. Prime and probe latencies of two Prime+Scope strategies and parallel probing on Cloud Run. The host processors' frequency is 2 GHz.

Strategy	Prime Latency (mean ± std. deviation)	Probe Latency (mean ± std. deviation)
PS-Flush	6,024 ± 990 cycles	94 ± 0.7 cycles
PS-Alt	2,777 ± 735 cycles	
Parallel	1,121 ± 448 cycles	118 ± 0.7 cycles

Our solution. We discover that, due to the high memory-level parallelism supported by modern processors, simply probing *with overlapped accesses* all the W lines of a minimal eviction set (Section 2.2) results in a probe latency only slightly higher than that of Prime+Scope. The advantage of this *parallel probing* method is that it allows us to prime the cache set without preparing any replacement state. Therefore, parallel probing works irrespective of the replacement policy used by the target cache, which can be unknown or quite complex [43, 74, 90, 98].

Evaluating Parallel Probing. We conduct a covert-channel experiment similar to the one done by Purnal et al. [71] to evaluate two different Prime+Scope strategies and our parallel probing. In the experiment, we create a sender and a receiver thread that agree on a target SF set. The sender thread accesses the target set at a fixed time interval, while the receiver thread uses Prime+Scope or parallel probing to detect accesses to the target set. For a sender's access issued at time t, if the receiver detects an access at time $t' \in (t, t+\epsilon)$, where ϵ is an error bound, we say that the sender's access is detected by the receiver. We use $\epsilon = 500$ cycles (or 250 ns).

We conduct this experiment on Cloud Run with varying access intervals. In each experiment, the sender thread accesses the target SF set 2,000 times. We measure the percentage of the sender's accesses that are detected by the receiver—i.e., the *detection rate*. We also collect the probe and prime latencies and exclude outliers that are above 20,000 cycles, as an interrupt or context switch likely occurred during the operation. The experiment is done on different hosts on different days and at different times of day. We repeat the experiment 10 times on each host, totaling 4,070 measurements.

For Prime+Scope, we evaluate two prime strategies discussed by Purnal et al. [71]. The first strategy (PS-Flush) is to load, flush, and sequentially reload the eviction set. The second strategy (PS-Alt) is to perform an alternating pointer-chase using *two* eviction sets. More details of these strategies is found in [71]. For our parallel probing technique (Parallel), we use a prime strategy that simply traverses the eviction set 12 times with overlapped accesses.

Table 5 lists the prime and probe latencies of each strategy. The table reveals that the average probe latency of Parallel is only 24 cycles higher than that of Prime+Scope, yet Parallel exhibits a substantially lower prime latency.

The benefit of this reduced prime latency is depicted in Figure 6, which shows the average detection rate for different

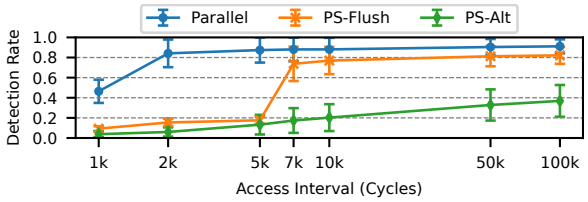


Figure 6. Detection rate of each monitoring strategy with various access interval. The x-axis employs a logarithmic scale. The error bars represent the standard deviations.

access intervals. With a 2k-cycle access interval, Parallel achieves an average detection rate of 84.1%, while PS-Flush and PS-Alt reach average detection rates of 15.4% and 6.0%, respectively. The low detection rates of PS-Flush and PS-Alt are primarily due to their long prime latencies.

Even when the access interval is sufficiently long for all strategies to complete priming, Parallel still maintains the highest detection rate. With a 100k-cycle access interval, Parallel, PS-Flush, and PS-Alt attain average detection rates of 91.1%, 82.1%, and 36.9%, respectively. To understand why, we inspected a random subset of the detected memory access traces. In PS-Flush, we observe that missed detections mainly result from noisy accesses made by other tenants to the monitored cache set, occurring just before the sender's access. After the receiver detects the noisy access, it is unable to finish priming before the sender accesses the set.

In PS-Alt, although the receiver initially detects the sender's accesses, it often later fails to prime the monitored line as the EVC, leading to many missed detections. We believe this might be due to the SF replacement states being altered by background accesses, resulting in failing to prepare the EVC.

6.2 Power Spectral Density for Set Identification

To identify the target cache sets (Step 2 in Table 1), the attacker can collect a short memory access trace from each potential target cache set *while the victim is executing*. The attacker then applies signal processing techniques to determine whether a given memory access trace has any characteristic that resembles what is expected from a given target cache set. Prior work has considered characteristics such as the number of accesses in the trace or the access pattern [41, 56]. These characteristics can be hard to identify in the cloud due to the high level of environmental noise.

Our solution. Our insight is that a victim program's accesses to the target cache set are often periodic in a way that the attacker expects, while this is not the case for the background accesses. Therefore, we propose to process the access traces in the frequency domain, where it is easier to spot the expected periodic patterns. Specifically, we estimate the *Power Spectral Density* (PSD) [83] of each memory access trace using Welch's method [96]. PSD measures the "strength" of the signal at different frequencies [83]. If the access trace is collected from the target set where the victim makes periodic accesses, we will observe peaks in the

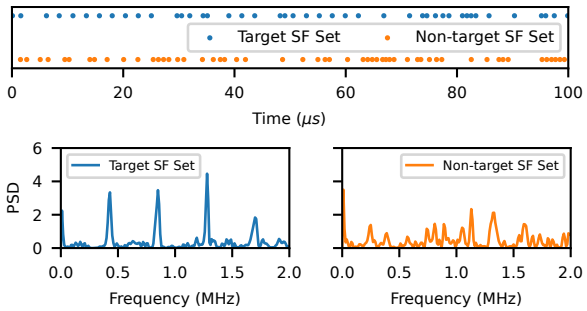


Figure 7. The top plot shows traces of memory access to the target SF set (top trace) and the non-target SF set (bottom trace) collected on Cloud Run. The two bottom plots show the power spectral density of the two traces.

trace's PSD around the expected victim-access frequencies. If, instead, the trace is not collected from the target set, it will have a PSD without the expected peaks.

Example. To demonstrate our proposal, we collect an access trace from a target SF set of a victim program and another trace from a non-target SF set, and compare the PSD of both traces. In this example, the victim executes an ECDSA implementation [62] that will be described in Section 7.1. In this implementation, the victim processes each individual secret bit in a loop. The victim accesses the target SF set when an iteration starts and, if the secret bit being processed in the iteration is zero, it also accesses the set in the midpoint of the iteration. The execution of each iteration takes a mostly fixed time duration of about 9,700 cycles on a 2 GHz Skylake-SP machine on Cloud Run. Because of the access that may occur in the midpoint of an iteration, the victim's accesses to the target set have a period of about 4,850 cycles. Therefore, we expect to observe a peak in the PSD at the frequency of $f = 2\,\text{GHz}/4{,}850 \approx 0.41\,\text{MHz}$.

The top plot of Figure 7 shows two 100 µs memory access traces collected on a 2 GHz Skylake-SP machine on Cloud Run. The blue dots at the top are the observed accesses to the target SF set; while the orange dots at the bottom are the observed accesses to the non-target SF set. For both traces, we see similar numbers of accesses: 50 accesses to the target set and 48 to the non-target set. It is difficult to interpret these two patterns.

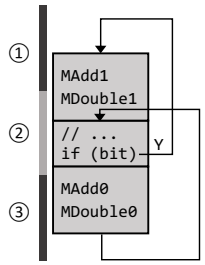
The bottom plots show the PSD of the access traces collected from the target set (left) and the non-target set (right). In the PSD for the target set, we clearly see a peak at the base frequency $f = 0.41\,\text{MHz}$ and at multiples of f. In contrast, in the PSD for the non-target set, we see no significant peaks at the expected frequency.

7 Demonstrating an End-to-End Attack

In this section, we demonstrate the combination of our techniques discussed in Sections 5 and 6 by mounting an end-to-end, cross-tenant attack in Cloud Run. Our demonstration uses a vulnerable implementation of Elliptic Curve Digital Signature Algorithm (ECDSA) [44] from OpenSSL 1.0.1e [62]

```
for bit in k {
  if (bit) {
    MAdd(x1,z1,x2,z2); // MAdd1
    MDouble(x2,z2); // MDouble1
  } else {
    MAdd(x2,z2,x1,z1); // MAdd0
    MDouble(x1,z1); // MDouble0
  }
  // ...
}
```

(a) Simplified code snippet.



(b) Memory layout.

Figure 8. Simplified vulnerable code snippet (left) and its memory layout in VA space (right). Each thick vertical line represents a cache line. The control-flow edge that exits the loop is omitted in the right figure.

as an example victim. While this implementation is deprecated, we use it solely as a vehicle to illustrate our techniques.

7.1 Attack Outline

The vulnerable ECDSA implementation that we target uses the Montgomery ladder technique [45] to compute on the nonce k, an ephemeral key that changes with each signing. The attacker can derive the private key used for signing by extracting some bits of k across multiple signing operations [1, 17, 27, 37, 59, 61, 103]. Thus, the attacker's goal is to learn as many bits of k as possible. Our demonstration targets curve `sect571r1`, which uses a 571-bit nonce.

Similar to prior work [39, 56, 103], we assume the attacker knows the memory layout of the library used by the victim. This assumption generally holds, as victims often install and use libraries whose binaries are publicly released. Moreover, as we are targeting a victim web service (Section 3), we assume the library is loaded once at the victim container startup time and uses the same VA-PA mapping throughout the container's lifetime.

Figure 8a shows a simplified version of the Montgomery ladder implementation [62] that we are targeting. The code iterates through each bit of the nonce k and calls functions `MAdd` and `MDouble` with different arguments depending on the value of the bit. This implementation is resilient to end-to-end timing, as it executes the same sequence of operations regardless of the bit value. However, it has secret-dependent control flow. Since each side of the branch resides on a different cache line, the program fetches different cache lines based on the value of the nonce bit. As a result, the attacker can infer each individual nonce bit by monitoring code fetch accesses to these cache lines tracked by the SF.

Figure 8b shows the memory layout of the vulnerable code snippet in VA space, compiled with the default build options and static linkage. Each thick vertical line represents a different cache line. Given this layout, one approach is to monitor accesses to cache line ②. Line ② is used by the `if` statement, which is executed at the beginning of an iteration. As a result, the code fetch accesses made by the `if` statement serve as a "clock" and mark the iteration boundaries.

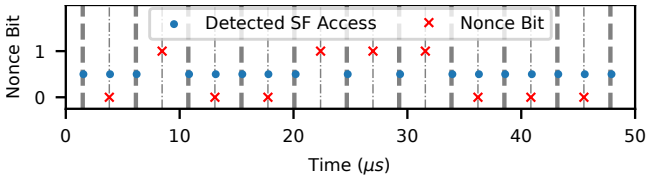


Figure 9. A snippet of memory accesses to the target SF set collected on Cloud Run. Dots are detected accesses, and crosses are the nonce bit k values (1 or 0).

Cache line ② is also utilized by the true direction of the branch. When the control flow takes the true direction and `MAdd1` is executing, Prime+Probe will evict line ②. As the control flow returns from `MAdd1` and is about to call `MDouble1`, the program needs to fetch line ②, creating one access in the midpoint of the iteration. Then, while `MDouble1` is executing, Prime+Probe evicts cache line ② again, triggering a code fetch access when returning from `MDouble1` and executing the `if` statement.

Therefore, we observe two accesses to line ② per iteration if the bit value is 1, and one access to line ② if the bit value is 0. It should be noted that, although line ② slightly overlaps with the beginning of the `else` block, we will *not* observe an extra access if the bit value is 0. This is because the overlapped region is executed immediately after the `if` statement, and the interval is too brief to be detected.

In practice, when we collect a trace of the memory accesses to the target SF set to which cache line ② maps, we also want to collect the ground truth of nonce bit k and iteration boundaries for validation purpose. This requires some slight instrumentation of the binary, a practice also seen in prior work [30, 108]. The instrumentation is purely for validation purpose and it is not necessary for the attack. However, due to the instrumentation, the layout of the code changes, and it is easier to monitor the cache line corresponding to the `else` direction. The reasoning is similar to the explanation for line ②, but we now observe the additional memory access at the midpoint of an iteration when the bit value is 0, not 1.

We collect the trace of memory accesses to the target SF set (using the techniques of Section 6), the ground truth of nonce bit k, and iteration boundaries on Cloud Run, while the victim code is executing. Figure 9 shows a short snippet of the trace that happens to contain no noisy accesses made by other tenants. In the figure, thick dashed vertical lines represent the ground truth for iteration boundaries, and thin dashed vertical lines represent halves of iterations. Dots are detected accesses, and crosses are the nonce bit k values (1 or 0). Iterations where bit k value is 0 have two accesses. From the trace, we can easily read the nonce bits.

It takes only about 9,700 cycles on Cloud Run to execute one iteration of the Montgomery ladder loop that we target. Thus, when the nonce bits have a sequence of continuous zeros, the attacker needs to detect a sequence of accesses that are 4,850 cycles apart. As shown in Table 5, the prime pattern of Prime+Scope's [71] PS-Flush takes on average 6,024 cycles to complete, while the PS-Alt pattern has a low detection rate (Figure 6). As a result, the Prime+Scope versions either frequently miss memory accesses or report an access as occurring at a time different from when the actual access occurs. In contrast, our *Parallel Probing* strategy takes on average only 1,121 cycles to execute (Table 5) and thus accurately detects the memory accesses in ECDSA.

7.2 Finding the Target Cache Set with PSD

We apply our PSD method to identify the victim's target SF set on Cloud Run. To obtain the ground truth, we run the victim and attacker programs in the same container. The attacker `mmap`s the victim program so that the attacker can access the target line. Then, when the attacker identifies an eviction set that might correspond to the target SF set, the attacker can validate it by checking whether the eviction set indeed evicts the target line.

Scanning strategy. Since the attacker knows the VA of the target cache line of the ECDSA victim, they only need to construct eviction sets for SF sets at the page offset of the target line and scan only those sets—i.e., it is the PageOffset scenario. To approximate the WholeSys scenario, we also measure the effectiveness of our approach by scanning cache sets at every page offset in a *random* order.

The ECDSA victim program spends only about 25% of its execution time running the vulnerable code. Therefore, there is a high chance that the attacker cannot detect the target set, as they may collect the traces while the victim is not executing the vulnerable code—a problem known as de-synchronization. Hence, the attacker repeatedly scans all possible sets until detecting the target set or timeout. We set the timeouts for PageOffset and WholeSys to 60 s and 900 s, respectively. Time spent on eviction set construction is not counted towards the timeout. Implementation details are presented in the extended version of this paper [112].

Evaluation setup. We conduct this experiment on Cloud Run at different times of day, totaling 357 measurements for PageOffset and 207 measurements for WholeSys. For WholeSys, we deem the scan successful if it manages to locate the cache set accessed by the either side of the branch, as accesses made by either side can disclose the nonce k.

Results. Table 6 lists the key metrics of finding the target cache set using the PSD method. Given our timeout configurations, 94.1% and 73.9% of the scanning attempts find the target set under PageOffset and WholeSys, respectively. The lower success rate under WholeSys is mainly because we can only scan each SF set fewer times within the timeout period, leading to more failures due to the de-synchronization problem. Averaged among successful scans, it takes 6.1 s and 179.7 s to find the target set under PageOffset and WholeSys, respectively. Finally, we scan from 762 sets/s to 831 sets/s. The scanning speed can be improved by using multiple threads to scan cache sets in parallel.

Table 6. Performance of identifying the target cache set.

Metric	PageOffset	WholeSys
Success Rate	94.1%	73.9%
Average Success Time	6.1 s	179.7 s
Std. Deviation of Success Time	6.9 s	177.4 s
95% Percentile Success Time	16.1 s	546.6 s
Average Scan Rate	831 sets/s	762 sets/s

7.3 End-to-End Nonce Extraction

Putting all the pieces together, we demonstrate end-to-end, *cross-tenant* nonce k extractions on Cloud Run. In this demonstration, the attacker first successfully co-locates their attack container with the victim container [111]. Then, the attacker builds the eviction sets and finds the target set using the PSD method, while sending requests to trigger victim executions. Once the target set is identified, the attacker triggers the victim execution 10 more times to steal the different nonces used in each execution.

To process the memory access trace, we train a random forest classifier [66, 67] to predict if a detected memory access corresponds to an iteration boundary. To filter out false-positive boundary predictions, we consider only boundary pairs that are $8k$ to $12k$ cycles apart, as this is the duration variation that we expect from a single iteration on these hosts. From each pair of predicted neighboring boundaries, we recover the nonce bit in the iteration by checking if there is an extra access in the middle of the iteration.

We attempt end-to-end nonce k extractions under the PageOffset scenario on 52 pairs of co-located containers on Cloud Run. We identify a potential target set and observe a signal in 47 of them. Within the 470 traces collected from these 47 victims, we extract an average of 68% (or a median value of 81%) of the nonce bits. Among these recovered bits, our average bit error rate is 3%. The full attack, which includes constructing eviction sets, identifying the target SF set, and collecting 10 traces, takes an average of 19 seconds.

8 Related Work

Side-channel attacks in cloud. Ristenpart et al. [75] examined the placement of virtual machines on physical hosts within AWS and developed techniques to achieve co-location. Zhang et al. [107] employed Flush+Reload for a cross-tenant attack on a Platform-as-a-Service (PaaS) cloud. However, Flush+Reload is no longer feasible in modern clouds [3, 79]. İnci et al. [39] in 2015 conducted a Prime+Probe attack on AWS EC2 to extract RSA keys, using a reverse-engineered LLC slice hash function and huge pages to build eviction sets. Their attack is long running, relies on huge pages, and targets an inclusive LLC—all of which are incompatible with modern cloud environments.

Mitigations to cache-based side-channel attacks. Defenses can be broadly categorized into two types. The first type, partition-based solutions [22, 23, 47, 53, 76, 88, 94, 110], blocks attacks by partitioning the cache between different tenants. However, this approach often requires complex hardware design and results in high execution overhead. The second type, randomization-based defenses [54, 55, 72, 73, 77, 80, 85, 95, 97], focuses on obfuscating the victim's cache usage. While this method offers high performance, it fails to provide comprehensive security guarantees.

Eviction set construction. Algorithms for constructing eviction sets have received significant attention [34, 56, 70, 81, 90, 101]. However, most approaches are developed and evaluated in a quiescent local environment. Besides the group testing [73, 90] and Prime+Scope [71] algorithms discussed in Section 2.2.1, Prime+Prune+Probe (PPP) [70] exploits the LRU replacement policy to defeat randomized caches by minimizing memory accesses. CTPP [101], which is concurrent to our work, builds on PPP by integrating it with Prime+Scope. Based on the evaluation in CTPP [101], the success rates of both PPP and CTPP fall to almost zero when a single memory-intensive SPEC 2006 benchmark [36], such as `mcf`, runs in the background. Using the average LLC access rate as a metric, the cache activity caused by `mcf` is only about 10% of what we observed on Cloud Run. Lastly, Guo et al. [34] exploited a non-temporal prefetch instruction to accelerate eviction set construction on Intel inclusive LLCs, but found this technique inapplicable to Intel non-inclusive LLCs.

Prime+Probe techniques. Prior arts [26, 51] also used parallel probing in their Prime+Probe implementations [25, 52]. However, to our knowledge, we are the first to study the parallel probing strategy to strike a good balance between probe and prime latency. Oren et al. [63] processed memory access traces in the frequency domain to fingerprint websites.

9 Conclusion

In this paper, we presented an end-to-end, cross-tenant LLC Prime+Probe attack on a vulnerable ECDSA implementation in the public FaaS Google Cloud Run environment. We showed that state-of-the-art eviction set construction algorithms are ineffective on Cloud Run. We then introduced L2-driven candidate address filtering and a binary search-based algorithm for address pruning to speed-up eviction set construction. Subsequently, we introduced parallel probing to monitor victim memory accesses with high time resolution. Finally, we leveraged power spectral density to identify the victim's target cache set in the frequency domain. Overall, we extract a median value of 81% of the secret ECDSA nonce bits from a victim container in 19 seconds on average.

Ethical considerations. We limited our attempts to exfiltrate information from only victims under our control. We monitored just one SF set of the host at a time, thus minimizing potential performance interference with other tenants.

Acknowledgments

We thank our shepherd, Yuval Yarom. This work was funded in part by an Intel RARE gift; by ACE, one of the 7 centers in JUMP 2.0, an SRC program sponsored by DARPA; and by NSF grants 1942888, 1954521, 1956007, 2154183, and 2107470.

References

[1] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. 2020. LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage. In *2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 225–242. https://doi.org/10.1145/3372297.3417268

[2] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium*. 19–28.

[3] Amazon AWS. 2022. The Security Design of the AWS Nitro System - AWS Whitepaper. https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.html.

[4] Amazon AWS. 2023. AWS Lambda - FAQs. https://aws.amazon.com/lambda/faqs/.

[5] Amazon AWS. 2023. Cloud Computing Services - Amazon Web Services (AWS). https://aws.amazon.com/.

[6] Amazon AWS. 2023. Secure and resizable cloud compute - Amazon EC2 - Amazon Web Services. https://aws.amazon.com/ec2/.

[7] Amazon AWS. 2023. Serverless Computing - AWS Lambda - Amazon Web Services. https://aws.amazon.com/lambda/.

[8] Microsoft Azure. 2023. Azure Functions - Serverless Functions in Computing | Microsoft Azure. https://azure.microsoft.com/en-us/products/functions.

[9] Microsoft Azure. 2023. Azure Functions scale and hosting | Microsoft Learn. https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale#timeout.

[10] Microsoft Azure. 2023. Cloud Computing Services | Microsoft Azure. https://azure.microsoft.com/.

[11] Anton Beloglazov and Rajkumar Buyya. 2010. Adaptive Threshold-Based Approach for Energy-Efficient Consolidation of Virtual Machines in Cloud Data Centers. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science (MGC)*. ACM, 4. https://doi.org/10.1145/1890799.1890803

[12] Anton Beloglazov and Rajkumar Buyya. 2010. Energy Efficient Resource Management in Virtualized Cloud Data Centers. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE Computer Society, 826–831. https://doi.org/10.1109/CCGRID.2010.46

[13] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 785–800. https://doi.org/10.1145/3319535.3363194

[14] Samira Briongos, Ida Bruhns, Pedro Malagón, Thomas Eisenbarth, and José Manuel Moya. 2021. Aim, Wait, Shoot: How the CacheSniper Technique Improves Unprivileged Cache Attacks. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 683–700. https://doi.org/10.1109/EUROSP51992.2021.00051

[15] Samira Briongos, Pedro Malagón, José Manuel Moya, and Thomas Eisenbarth. 2020. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *29th USENIX Security Symposium*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1967–1984. https://www.usenix.org/conference/usenixsecurity20/presentation/briongos

[16] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *2019 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 870–887. https://doi.org/10.1109/SP.2019.00066

[17] Jinzheng Cao, Jian Weng, Yanbin Pan, and Qingfeng Cheng. 2023. Generalized Attack on ECDSA: Known Bits in Arbitrary Positions. *Designs, Codes and Cryptography* 91, 11 (2023), 3803–3823. https://doi.org/10.1007/S10623-023-01269-7

[18] Google Cloud. 2023. Cloud Computing Services | Google Cloud. https://cloud.google.com/.

[19] Google Cloud. 2023. Cloud Run: Container to production in seconds | Google Cloud. https://cloud.google.com/run/.

[20] Google Cloud. 2023. Container runtime contract | Cloud Run Documentation | Google Cloud. https://cloud.google.com/run/docs/container-contract.

[21] Google Cloud. 2023. Setting request timeout (services) | Cloud Run Documentation | Google Cloud. https://cloud.google.com/run/docs/configuring/request-timeout#setting.

[22] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2020. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *29th USENIX Security Symposium*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 451–468. https://www.usenix.org/conference/usenixsecurity20/presentation/dessouky

[23] Ghada Dessouky, Emmanuel Stapf, Pouya Mahmoody, Alexander Gruler, and Ahmad-Reza Sadeghi. 2022. Chunked-Cache: On-Demand and Scalable Cache Isolation for Security Architectures. In *29th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/auto-draft-225/

[24] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *26th USENIX Security Symposium*. USENIX Association, 51–67. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen

[25] Catherine Easdon, Michael Schwarz, Martin Schwarzl, and Daniel Gruss. 2022. GitHub - libtea/frameworks. https://github.com/libtea/frameworks/blob/7af59e01/libtea/src/libtea_cache_improved.c#L503.

[26] Catherine Easdon, Michael Schwarz, Martin Schwarzl, and Daniel Gruss. 2022. Rapid Prototyping for Microarchitectural Attacks. In *31st USENIX Security Symposium*. USENIX Association, 3861–3877. https://www.usenix.org/conference/usenixsecurity22/presentation/easdon

[27] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. 2016. Attacking OpenSSL Implementation of ECDSA with a Few Signatures. In *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 1505–1515. https://doi.org/10.1145/2976749.2978400

[28] Alexander Fuerst, Stanko Novakovic, Iñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. 2022. Memory-Harvesting VMs in Cloud Platforms. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 583–594. https://doi.org/10.1145/3503222.3507725

[29] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. 2018. Drive-By Key-Extraction Cache Attacks from Portable Code. In *16th International Conference on Applied Cryptography and Network Security (ACNS) (Lecture Notes in Computer Science, Vol. 10892)*, Bart Preneel and Frederik Vercauteren (Eds.). Springer, 83–102. https://doi.org/10.1007/978-3-319-93387-0_5

[30] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-Aside Buffer: Defeating Cache Side-Channel Protections with TLB Attacks. In *27th USENIX Security Symposium*. USENIX Association, 955–972. https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[31] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) (Lecture Notes in Computer Science, Vol. 9721)*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.). Springer, 300–321. https://doi.org/10.1007/978-3-319-40667-1_15

[32] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) (Lecture Notes in Computer Science, Vol. 9721)*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.). Springer, 279–299. https://doi.org/10.1007/978-3-319-40667-1_14

[33] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *24th USENIX Security Symposium*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 897–912. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss

[34] Yanan Guo, Xin Xin, Youtao Zhang, and Jun Yang. 2022. Leaky Way: A Conflict-Based Cache Covert Channel Bypassing Set Associativity. In *55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 646–661. https://doi.org/10.1109/MICRO56248.2022.00053

[35] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *9th Biennial Conference on Innovative Data Systems Research (CIDR)*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

[36] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News* 34, 4 (2006), 1–17. https://doi.org/10.1145/1186736.1186737

[37] Nick A Howgrave-Graham and Nigel P. Smart. 2001. Lattice Attacks on Digital Signature Schemes. *Designs, Codes and Cryptography* 23 (2001), 283–290.

[38] Mehmet Sinan İnci, Berk Gülmezoğlu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. 2015. Seriously, Get Off My Cloud! Cross-VM RSA Key Recovery in a Public Cloud. *IACR Cryptology ePrint Archive* (2015), 898. http://eprint.iacr.org/2015/898

[39] Mehmet Sinan İnci, Berk Gülmezoğlu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. 2016. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *Cryptographic Hardware and Embedded Systems (CHES) (Lecture Notes in Computer Science, Vol. 9813)*. Springer, 368–388. https://doi.org/10.1007/978-3-662-53140-2_18

[40] Intel. 2023. PerfMon Event - Skylake-X Server Events. https://perfmon-events.intel.com/skylake_server.html.

[41] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES. In *2015 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 591–604. https://doi.org/10.1109/SP.2015.42

[42] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a Minute! A fast, Cross-VM Attack on AES. In *Research in Attacks, Intrusions and Defenses - 17th International Symposium (RAID) (Lecture Notes in Computer Science, Vol. 8688)*, Angelos Stavrou, Herbert Bos, and Georgios Portokalidis (Eds.). Springer, 299–319. https://doi.org/10.1007/978-3-319-11379-1_15

[43] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel S. Emer. 2010. High Performance Cache Replacement using Re-reference Interval Prediction (RRIP). In *37th International Symposium on Computer Architecture (ISCA)*. ACM, 60–71. https://doi.org/10.1145/1815961.1815971

[44] Don Johnson, Alfred Menezes, and Scott A. Vanstone. 2001. The Elliptic Curve Digital Signature Algorithm (ECDSA). *International Journal of Information Security* 1, 1 (2001), 36–63. https://doi.org/10.1007/S102070100002

[45] Marc Joye and Sung-Ming Yen. 2002. The Montgomery Powering Ladder. In *Cryptographic Hardware and Embedded Systems (CHES) (Lecture Notes in Computer Science, Vol. 2523)*, Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar (Eds.). Springer, 291–302. https://doi.org/10.1007/3-540-36400-5_22

[46] Mehmet Kayaalp, Nael B. Abu-Ghazaleh, Dmitry V. Ponomarev, and Aamer Jaleel. 2016. A High-Resolution Side-Channel Attack on Last-Level Cache. In *53rd Annual Design Automation Conference (DAC)*. ACM, 72:1–72:6. https://doi.org/10.1145/2897937.2897962

[47] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel S. Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 974–987. https://doi.org/10.1109/MICRO.2018.00083

[48] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. NetCAT: Practical Cache Attacks from the Network. In *2020 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 20–38. https://doi.org/10.1109/SP40000.2020.00082

[49] Bo Li, Jianxin Li, Jinpeng Huai, Tianyu Wo, Qin Li, and Liang Zhong. 2009. EnaCloud: An Energy-Saving Application Live Placement Approach for Cloud Computing Environments. In *IEEE International Conference on Cloud Computing (CLOUD)*. IEEE Computer Society, 17–24. https://doi.org/10.1109/CLOUD.2009.72

[50] Linux. 2022. Core Scheduling; The Linux Kernel Documentation. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/core-scheduling.html.

[51] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 549–564. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp

[52] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. GitHub - IAIK/armageddon. https://github.com/IAIK/armageddon/blob/96ebc2d8/libflush/libflush/eviction/eviction.c#L266.

[53] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos V. Rozas, Gernot Heiser, and Ruby B. Lee. 2016. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 406–418. https://doi.org/10.1109/HPCA.2016.7446082

[54] Fangfei Liu and Ruby B. Lee. 2014. Random Fill Cache Architecture. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 203–215. https://doi.org/10.1109/MICRO.2014.28

[55] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. 2016. Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks. *IEEE Micro* 36, 5 (Sept. 2016), 8–16. https://doi.org/10.1109/MM.2016.85

[56] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 605–622. https://doi.org/10.1109/SP.2015.43

[57] Mulong Luo, Wenjie Xiong, Geunbae Lee, Yueying Li, Xiaomeng Yang, Amy Zhang, Yuandong Tian, Hsien-Hsin S. Lee, and G. Edward Suh. 2023. AutoCAT: Reinforcement Learning for Automated Exploration of Cache-Timing Attacks. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 317–332. https://doi.org/10.1109/HPCA56546.2023.10070947

[58] John D McCalpin. 2021. *Mapping Addresses to L3/CHA Slices in Intel Processors.* Technical Report. Texas Advanced Computing Center, University of Texas at Austin.

[59] Gabrielle De Micheli, Rémi Piau, and Cécile Pierrot. 2020. A Tale of Three Signatures: Practical Attack of ECDSA with wNAF. In *12th International Conference on Cryptology in Africa (AFRICACRYPT) (Lecture Notes in Computer Science, Vol. 12174)*, Abderrahmane Nitaj and

Amr M. Youssef (Eds.). Springer, 361–381. https://doi.org/10.1007/978-3-030-51938-4_18

[60] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. 2019. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations. *International Journal of Parallel Programming* 47, 4 (2019), 538–570.

[61] Phong Q Nguyen and Igor E Shparlinski. 2003. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Designs, Codes and Cryptography* 30, 2 (2003), 201–217.

[62] OpenSSL. 2011. Montgomery Ladder Implementation of OpenSSL 1.0.1e. https://github.com/openssl/openssl/blob/46ebd9e3/crypto/ec/ec2_mult.c#L268.

[63] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *2015 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 1406–1418. https://doi.org/10.1145/2810103.2813708

[64] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *The Cryptographers' Track at the RSA Conference 2006 (CT-RSA) (Lecture Notes in Computer Science, Vol. 3860)*, David Pointcheval (Ed.). Springer, 1–20. https://doi.org/10.1007/11605805_1

[65] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. 2021. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *30th USENIX Security Symposium*, Michael D. Bailey and Rachel Greenstadt (Eds.). USENIX Association, 645–662. https://www.usenix.org/conference/usenixsecurity21/presentation/paccagnella

[66] Mahesh Pal. 2005. Random Forest Classifier for Remote Sensing Classification. *International Journal of Remote Sensing* 26, 1 (2005), 217–222.

[67] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830. https://doi.org/10.5555/1953048.2078195

[68] Colin Percival. 2005. Cache Missing for Fun and Profit. https://www.daemonology.net/papers/cachemissing.pdf.

[69] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium*. USENIX Association, 565–581. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl

[70] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. 2021. Systematic Analysis of Randomization-Based Protected Cache Architectures. In *42nd IEEE Symposium on Security and Privacy (S&P)*. IEEE, 987–1002. https://doi.org/10.1109/SP40001.2021.00011

[71] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. 2021. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 2906–2920. https://doi.org/10.1145/3460120.3484816

[72] Moinuddin K. Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 775–787. https://doi.org/10.1109/MICRO.2018.00068

[73] Moinuddin K. Qureshi. 2019. New Attacks and Defense for Encrypted-Address Cache. In *46th International Symposium on Computer Architecture (ISCA)*, Srilatha Bobbie Manne, Hillery C. Hunter, and Erik R. Altman (Eds.). ACM, 360–371. https://doi.org/10.1145/3307650.3322246

[74] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel S. Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *34th International Symposium on Computer Architecture (ISCA)*. ACM, 381–391. https://doi.org/10.1145/1250662.1250709

[75] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *2009 ACM Conference on Computer and Communications Security (CCS)*, Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis (Eds.). ACM, 199–212. https://doi.org/10.1145/1653662.1653687

[76] Gururaj Saileshwar, Sanjay Kariyappa, and Moinuddin K. Qureshi. 2021. Bespoke Cache Enclaves: Fine-Grained and Scalable Isolation from Cache Side-Channels via Flexible Set-Partitioning. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 37–49. https://doi.org/10.1109/SEED51797.2021.00015

[77] Gururaj Saileshwar and Moinuddin K. Qureshi. 2021. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In *30th USENIX Security Symposium*, Michael D. Bailey and Rachel Greenstadt (Eds.). USENIX Association, 1379–1396. https://www.usenix.org/conference/usenixsecurity21/presentation/saileshwar

[78] Michael Schwarz, Moritz Lipp, and Daniel Gruss. 2018. JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks. In *25th Annual Network and Distributed System Security Symposium (NDSS)*.

[79] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. 2022. Remote Memory-Deduplication Attacks. In *29th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

[80] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu. 2021. Randomized Last-Level Caches Are Still Vulnerable to Cache Side-Channel Attacks! But We Can Fix It. In *42nd IEEE Symposium on Security and Privacy (S&P)*. IEEE, 955–969. https://doi.org/10.1109/SP40001.2021.00050

[81] Wei Song and Peng Liu. 2019. Dynamically Finding Minimal Eviction Sets Can Be Quicker Than You Think for Side-Channel Attacks against the LLC. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. USENIX Association, 427–442. https://www.usenix.org/conference/raid2019/presentation/song

[82] SPEC. 2023. SPEC CPU2006 and CPU2017 Flag Description - Platform settings for New H3C systems. https://www.spec.org/cpu2017/flags/New_H3C-Platform-Settings-V1.3-SKL-RevE.html.

[83] Petre Stoica and Randolph Moses. 2005. *Spectral Analysis of Signals*. Vol. 452. Pearson Prentice Hall Upper Saddle River, NJ.

[84] Simon M. Tam, Harry Muljono, Min Huang, Sitaraman Iyer, Kalapi Royneogi, Nagmohan Satti, Rizwan Qureshi, Wei Chen, Tom Wang, Hubert Hsieh, Sujal Vora, and Eddie Wang. 2018. SkyLake-SP: A 14nm 28-Core Xeon® Processor. In *2018 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 34–36. https://doi.org/10.1109/ISSCC.2018.8310170

[85] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. 2020. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In *27th Annual Network and Distributed System Security Symposium (NDSS)*.

[86] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. 2022. TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering. In *31st USENIX Security Symposium*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 989–1007. https://www.usenix.org/conference/usenixsecurity22/presentation/tatar

[87] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the Next Generation. In *Fifteenth EuroSys Conference 2020*. ACM, 30:1–30:14. https://doi.org/10.1145/3342195.3387517

[88] Daniel Townley, Kerem Arikan, Yu David Liu, Dmitry Ponomarev, and Oguz Ergin. 2022. Composable Cachelets: Protecting Enclaves from Cache Side-Channel Attacks. In *31st USENIX Security Symposium*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 2839–2856. https://www.usenix.org/conference/usenixsecurity22/presentation/townley

[89] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. 2015. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *24th USENIX Security Symposium*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 913–928. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/varadarajan

[90] Pepe Vila, Boris Köpf, and José F. Morales. 2019. Theory and Practice of Finding Eviction Sets. In *2019 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 39–54. https://doi.org/10.1109/SP.2019.00042

[91] Pepe Vila, Boris Köpf, and José F. Morales. 2020. cgvwzq/evsets: Tool for testing and finding minimal eviction sets. https://github.com/cgvwzq/evsets.

[92] Daimeng Wang, Zhiyun Qian, Nael B. Abu-Ghazaleh, and Srikanth V. Krishnamurthy. 2019. PAPP: Prefetcher-Aware Prime and Probe Side-Channel Attack. In *56th Annual Design Automation Conference (DAC)*. ACM, 62. https://doi.org/10.1145/3316781.3317877

[93] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (ATC)*, Haryadi S. Gunawi and Benjamin C. Reed (Eds.). USENIX Association, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[94] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2016. SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In *53rd Annual Design Automation Conference (DAC)*. ACM, 74:1–74:6. https://doi.org/10.1145/2897937.2898086

[95] Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. In *34th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 494–505.

[96] Peter Welch. 1967. The use of fast Fourier transform for the estimation of power spectra: a method based on time averaging over short, modified periodograms. *IEEE Transactions on audio and electroacoustics* 15, 2 (1967), 70–73.

[97] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *28th USENIX Security Symposium*. USENIX Association, 675–692. https://www.usenix.org/conference/usenixsecurity19/presentation/werner

[98] Henry Wong. 2013. Intel Ivy Bridge Cache Replacement Policy. https://blog.stuffedcow.net/2013/01/ivb-cache-replacement/.

[99] Wenjie Xiong and Jakub Szefer. 2020. Leaking Information Through Cache LRU States. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 139–152. https://doi.org/10.1109/HPCA47549.2020.00021

[100] Zhang Xu, Haining Wang, and Zhenyu Wu. 2015. A Measurement Study on Co-residence Threat inside the Cloud. In *24th USENIX Security Symposium*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 929–944. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/xu

[101] Zihan Xue, Jinchi Han, and Wei Song. 2023. CTPP: A Fast and Stealth Algorithm for Searching Eviction Sets on Intel Processors. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. ACM, 151–163. https://doi.org/10.1145/3607199.3607202

[102] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *2019 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 888–904. https://doi.org/10.1109/SP.2019.00004

[103] Yuval Yarom and Naomi Benger. 2014. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive* 2014 (2014), 140.

[104] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium*. USENIX Association, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[105] Yanqi Zhang, Iñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*. ACM, 724–739. https://doi.org/10.1145/3477132.3483580

[106] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM Side Channels and Their Use to Extract Private Keys. In *2012 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 305–316. https://doi.org/10.1145/2382196.2382230

[107] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 990–1003. https://doi.org/10.1145/2660267.2660356

[108] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2022. Binoculars: Contention-Based Side-Channel Attacks Exploiting the Page Walker. In *31st USENIX Security Symposium*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 699–716. https://www.usenix.org/conference/usenixsecurity22/presentation/zhao-zirui

[109] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2023. GitHub - zzrcxb/LLCFeasible. https://github.com/zzrcxb/LLCFeasible/blob/9bd94240/libs/cache/evset.c#L459.

[110] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2023. Untangle: A Principled Framework to Design Low-Leakage, High-Performance Dynamic Partitioning Schemes. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 771–788. https://doi.org/10.1145/3582016.3582033

[111] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2024. Everywhere All at Once: Co-Location Attacks on Public Cloud FaaS. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM. https://doi.org/10.1145/3617232.3624867

[112] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2024. Last-Level Cache Side-Channel Attacks Are Feasible in the Modern Public Cloud (Extended Version). In arXiv.