Undermining User Privacy on Mobile Devices Using AI

Berk Gulmezoglu
Worcester Polytechnic Institute
bgulmezoglu@wpi.edu

Andreas Zankl
Fraunhofer AISEC
andreas.zankl@aisec.fraunhofer.de

M. Caner Tol
Middle East Technical University
caner.tol@metu.edu.tr

Saad Islam
Worcester Polytechnic Institute
sislam@wpi.edu

Thomas Eisenbarth
University of Lübeck
thomas.eisenbarth@uni-luebeck.de

Berk Sunar
Worcester Polytechnic Institute
sunar@wpi.edu

ABSTRACT

Over the past years, literature has shown that attacks exploiting the microarchitecture of modern processors pose a serious threat to user privacy. This is because applications leave distinct footprints in the processor, which malware can use to infer user activities. In this work, we show that these inference attacks can greatly be enhanced with advanced AI techniques. In particular, we focus on profiling the activity in the last-level cache (LLC) of ARM processors. We employ a simple Prime+Probe based monitoring technique to obtain cache traces, which we classify with deep learning methods including convolutional neural networks. We demonstrate our approach on an off-the-shelf Android phone by launching a successful attack from an unprivileged, zero-permission app in well under a minute. The app detects running applications, opened websites, and streaming videos with up to 98% accuracy and a profiling phase of at most 6 seconds. This is possible, as deep learning compensates measurement disturbances stemming from the inherently noisy LLC monitoring and unfavorable cache characteristics. In summary, our results show that thanks to advanced AI techniques, inference attacks are becoming alarmingly easy to execute in practice. This once more calls for countermeasures that confine microarchitectural leakage and protect mobile phone applications, especially those valuing the privacy of their users.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; **Software and application security**; **Side-channel analysis and countermeasures**; • **Computing methodologies → Machine learning**.

KEYWORDS

Artificial Intelligence; Machine Learning; User Privacy; Activity Inference; Cache Attack; Microarchitecture; ARM; Mobile Device

ACM Reference Format:
Berk Gulmezoglu, Andreas Zankl, M. Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. 2019. Undermining User Privacy on Mobile Devices Using AI. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019, Auckland, New Zealand.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3321705.3329804

AsiaCCS '19, July 9–12, 2019, Auckland, New Zealand

ACM ISBN 978-1-4503-6752-3/19/07…$15.00
https://doi.org/10.1145/3321705.3329804

1 INTRODUCTION

In 2017, more than 2 billion Android devices were used monthly [44]. The security and privacy of the applications deployed on these devices are therefore of great relevance. The Android operating system consequently employs a variety of protection mechanisms. Apps run in sandboxes, inter-process communication is regulated, and users have some degree of control via the permission system. The majority of these features protects against software-based attacks and logical side-channel attacks. The processor hardware, however, also constitutes an attack surface. In particular, the shared processor cache heavily speeds up the execution of applications. As a side effect, each application leaves a footprint in the cache that can be profiled by others. These footprints, in turn, contain sensitive information about the application activity. Jana et al. [26] showed that browsing activity yields unique memory footprints that allow the inference of accessed websites. Oren et al. [36] demonstrated that these footprints can be observed in the cache even from JavaScript code distributed by a malicious website. While these attacks have succeeded based on a solid amount of engineering, the increasing complexity of applications, operating systems (OS), and processors make their implementation laborious and cumbersome. Yet, studying side-channel attacks is important to protect security and privacy critical applications in the long term. We believe that machine learning (ML), especially deep learning (DL) techniques such as convolutional neural networks, assist in making side-channel analysis significantly more scalable. Deep learning reduces the human effort by efficiently extracting relevant information from noisy and complex observations. At the same time, it introduces a new risk as attacks become more potent and easier to implement in practice. We demonstrate this risk and compile a malicious Android application, which, despite having no privileges or permissions, can infer user activities across application and OS boundaries. With the app, we are able to detect other running applications with up to 98% confidence. With this information, we focus on activities that happen within an application. We detect visited websites in Google Chrome and identify videos that are streamed in the Netflix and YouTube applications. Those inferences are possible by analyzing simple last-level cache (LLC) observations of at most 6 seconds with machine learning algorithms. The entire attack succeeds in well under a minute and reveals sensitive information about the mobile phone user. None of the currently employed protection mechanisms prevent our attack, as the LLC is shared between different processes and can be monitored from user space. Our cache profiling technique is based on the Prime+Probe attack [45], which relies on cache eviction to monitor the LLC. Cache eviction, in turn, requires sets of memory addresses that map to a single cache set.

In contrast to previous work, we compile these address sets with a novel algorithm that succeeds even for imprecise timing sources, random line replacement policies, and without access to physical memory addresses. This introduces a certain amount of noise in the cache observations. We counter this effect by applying machine learning, and in particular deep learning, to the observations. We also compare the performance of the employed machine learning techniques. While support vector machines (SVMs) and stacked autoencoders (SAEs) struggle during classification, convolution neural networks (CNNs) succeed in efficiently extracting distinct features and, thus, surpass the other techniques. As CNNs have recently gained attention in the field of side-channel analysis, we explain our parameter selection and compare it to related work. For the implementation of our attack, we neither require the target phone to be rooted nor the malicious application to have certain privileges or permissions. On our test device, a Nexus 5X, the Android OS is up-to-date and all security patches are installed. The malicious code runs in the background, requires no human contribution during the attack, and draws little attention due to the short profiling phase.

Our Contribution. In summary, we

- propose an inference attack on mobile devices that works without privileges, permissions, or special OS interfaces.
- find eviction sets with a novel dynamic timing test that works even with imprecise timing sources, random line replacement policies, and virtual addresses only.
- classify cache observations using machine learning (SVM, SAE, CNN) and thereby infer running applications, opened websites, and streaming videos.
- achieve classification rates of up to 98% with a profiling phase of at most 6 seconds and an overall attack time of well under a minute.

The rest of the paper is organized as follows: Section 2 provides a brief background on cache profiling and machine learning. Section 3 explains the proposed inference attack in detail. Section 4 presents the results of our experiments followed by a discussion in Section 5. Section 6 gives an overview of previous work and compares our results with other techniques. Section 7 concludes our work.

2 BACKGROUND

This section provides a brief introduction to the employed cache profiling and machine learning techniques.

2.1 Cache Profiling

The cache of modern processors consists of multiple levels. Higher levels are small and often private to processor cores, while lower levels are larger and shared among cores. The last-level cache (LLC) is the last stage before external memory (e.g. RAM). Among general purpose processors, set-associative caches are common. These caches are split into a number of cache sets, each of which contains a number of cache lines (equal to the associativity). While every address is deterministically mapped to a cache set, the exact line the corresponding data will be stored on is chosen by a replacement policy. ARM-based application processors mostly employ random selection policies, while Intel x86 processors often implement variants of least-recently used (LRU). Throughout the cache, data is stored on fixed-size cache lines of typically 64 bytes.

Prime+Probe. The cache is a resource that is competitively shared between executing threads. This means that the cache activity of each thread influences the runtime of all other threads. A malicious application can alter its cache footprint and time its execution such that it learns what other applications are executing. This is the basis for cache attacks and often referred to as *cache profiling*. Tromer et al. [45] proposed a prominent profiling technique called *Prime+Probe*. In the prime step, the adversary fills one or more cache sets with own data. This is done by accessing a set of addresses that all map to the same cache set. This set of addresses is called *eviction set*. After a short waiting period, the adversary measures how long it takes to re-accesses all addresses in the eviction set. If no other thread placed data in the monitored cache set, this re-access cycle will be fast. In contrast, if one or more cache lines got replaced in the meantime, the re-access cycle will trigger line replacements and, thus, will be slower. As a result, the timing measurements of the adversary reflect the activity in the cache.

2.2 Machine Learning

The following paragraphs introduce support vector machines, stacked autoencoders, and convolutional neural networks.

Support Vector Machines (SVMs). SVMs construct a classifier by mapping training data into a higher dimensional space, where distinct features can efficiently be separated. This separation is achieved with a hyperplane that maximizes the margin between the classes. A regularization parameter allows a tunable degree of misclassification while finding the hyperplane.

Stacked Autoencoders (SAEs). An autoencoder (AE) is a type of neural network that can be trained to reconstruct an input. The network consists of an encoder function $h = f(x)$ that extracts distinct features from the input x and a decoder unit that reconstructs the original data $r = g(h)$. The network is trained such that the error between r and x is minimized. Stacked AEs are constructed by combining multiple AEs sequentially. The idea behind SAEs is to learn only useful input features instead of learning an exact copy of the input. With a softmax layer at the end, SAEs can be used as classifiers for supervised learning.

Convolutional Neural Networks (CNNs). CNNs consist of neurons that are interconnected and grouped into layers. Each neuron computes a weighted sum of its inputs using a (non-) linear activation function. Those inputs either stem from the actual inputs to the network or from previous layers. A typical CNN comprises multiple layers of neurons. *Convolution* layers are the core of the CNN. They consist of filters that are slid over the width and height of the input to learn any two-dimensional patterns. The activation functions in the neurons thereby extract distinct features from each input. For efficiency, the neurons are connected only to a local region of the input. The *depth* of the convolution layer defines the number of filters and the *stride* controls how fast they are moved over the input. *Pooling* layers apply a filter to the input and forward only the maximum coefficient from every subregion. This reduces the size of the input and avoids over-fitting, as only the most dominant features from the convolution layer are passed to the rest of the

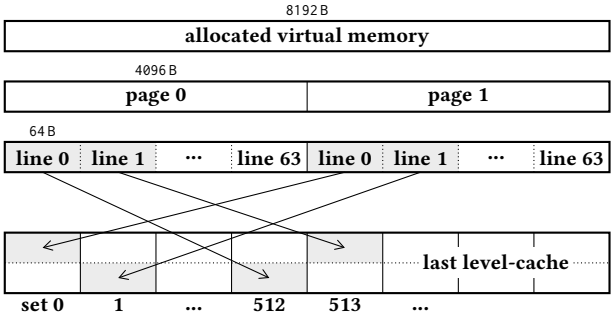


Figure 1: Mapping of virtual memory to cache sets.

network. *Dense* or fully-connected layers learn additional relations between the activations of the previous layers. The final layer of the network is a *loss* layer with a size of $1 \times L$, where L is the number of labels or classes. A back-propagation algorithm decreases the loss value by updating the weights in the network. For each classification, the network provides a *probability estimate* that indicates the confidence of the model for a predicted class. The sum of all estimates equals to 1.

3 INFERENCE ATTACK

The threat model of our proposed inference attack assumes that a mobile device user installs a malicious application from an app store on Android. This happens regularly, as malicious apps offer benign functionality to disguise malicious background activities (e.g. hidden crypto currency mining). The malicious code needed for our attack operates from user space and does not need any app permissions. This means that we neither require a rooted phone, nor ask the user for certain permissions, nor rely on any exploits, e.g., to escalate privileges or to break out of sandboxes. Furthermore, we do not rely on features or programming interfaces that might not be available on all Android versions. The sole task of our malicious code is to profile the LLC and classify victim activities with pre-trained ML/DL models. Once the LLC profiles have been gathered, the models are queried to infer sensitive information.

3.1 Attack Outline

The proposed inference attack consists of two main phases. In the **training phase**, the attacker creates ML/DL models on a training device that is similar to the target device. Ideally, the processor and operating system are identical on both devices. The models are created by recording raw LLC profiles of target applications, websites, and videos, followed by preparing the feature vectors, and training the ML/DL algorithms with them. The trained models are then directly integrated into the malicious application, which is subsequently published in the app store. In the **attack phase**, the malicious app prepares eviction sets for profiling the LLC on the target device. Subsequently, the LLC sets are profiled in a Prime+Probe manner and the feature vectors are extracted. Finally, the feature vectors are classified with the pre-trained models to infer opened applications, visited websites, and streamed videos. All steps of the attack phase are lightweight and can be executed in the background without drawing notable attention.

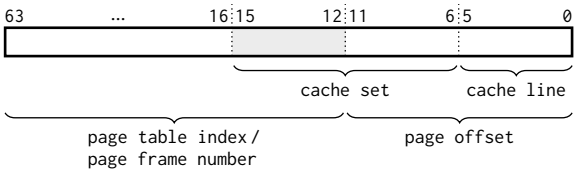


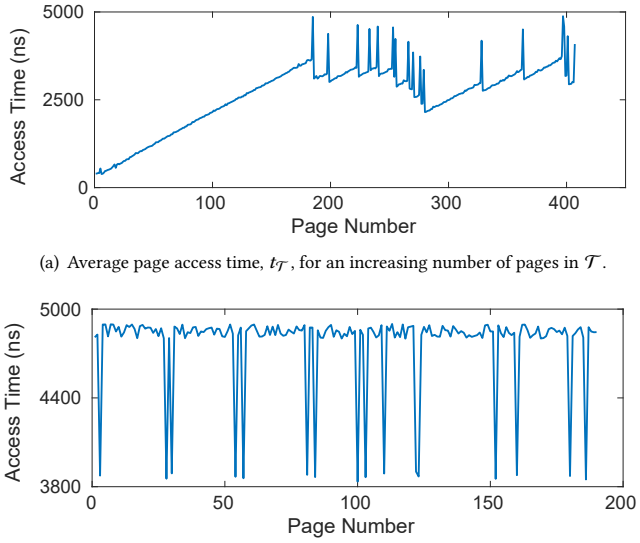
Figure 2: Virtual/physical address and its interpretation.

3.2 Finding Eviction Sets

Once deployed, the first task of the malicious app is to find eviction sets on the target device. An eviction set is a group of memory addresses that map to the same cache set. These addresses are called *set-congruent*. Figure 1 illustrates the problem of finding set-congruent addresses and forming eviction sets. The graphic shows a block of virtual memory that is backed by two fixed-size memory pages. In the figure, we assume a common page size of 4 KiB. As soon as an address within the block is accessed, the corresponding memory content is brought into the processor cache. Since the cache manages data on fixed-size cache lines, one access will cache multiple bytes. We assume a common cache line size of 64 bytes. The illustrated cache is set-associative and holds multiple lines per cache set. Memory that is brought into the cache is deterministically assigned to a cache set. For last-level caches, this assignment is commonly derived from physical addresses that are unavailable to most user space applications. Figure 2 illustrates the link between virtual and physical addresses, and how they are interpreted by the cache. The most significant bits of each virtual address are the page table index, while the least significant bits are the page offset. A page size of 4 KiB yields $log_2(4096) = 12$ offset bits. The page table index is used to lookup an entry in the page tables that contains the page frame number. The page frame number and the page offset form the physical address. The page offset bits are thereby identical in both virtual and physical address. The least significant bits of the physical address are then used by the cache controller for basic indexing. The lowest bits are used to address a byte on a cache line, while the subsequent bits determine the cache set in which the address will be placed. For 1024 cache sets, 10 bits are used as cache set index. As shown in Figure 2, these bits do not fit entirely within the page offset (highlighted in gray). Therefore, the exact cache set cannot be determined from the virtual address, as the most significant index bits are unknown. This complicates the mapping of virtual addresses to cache sets and may cause consecutive pages to map to completely different parts of the cache, as indicated in Figure 1. Finding eviction sets from user space is therefore a non-trivial problem. To fully solve it, one must *(a)* group virtual addresses according to cache sets, and *(b)* obtain the correct order such that group 0 maps to cache set 0 and so on. We refer to this as an *ordered* mapping between virtual addresses and cache sets. In practice, one must obtain physical address bits to derive this mapping, e.g., by gaining elevated privileges [29] or by exploiting additional vulnerabilities [14]. Alternatively, it is possible to find an *unordered* mapping that only fulfills *(a)*. This can be achieved with search algorithms that perform simple timing measurements and thereby find groups of set-congruent virtual addresses. While the algorithms do not reveal which group corresponds to which cache

Algorithm 1 Finding eviction sets.

```
1:  T = {}
2:  t_old = 0
3:  for i from 1 to n do
4:      add(T, i)
5:      t_T = access(T, r)
6:      if (t_T − t_old) > τ_jump then
7:          E ← {}
8:          for all p in T do
9:              t_p = access(T \ {p}, r)
10:             if (t_T − t_p) > τ_jump then
11:                 add(E, p)
12:             end if
13:         end for
14:         report(E)
15:         remove(T, E)
16:         t_old = access(T, r)
17:     else
18:         t_old = t_T
19:     end if
20: end for
```



(a) Average page access time, $t_{\mathcal{T}}$, for an increasing number of pages in $\mathcal{T}$.

(b) Average page access time, t_p, used to filter an eviction set from $\mathcal{T}$.

Figure 3: Plots of (a) $t_{\mathcal{T}}$ and (b) t_p, as used in Algorithm 1.

set, they can be run entirely from user space. In literature, the study by Vila et al. [48] investigates this type of search algorithms. The authors give a comprehensive overview of previous approaches, but limit their evaluation to Intel processors. We discuss approaches relevant to this work in the following paragraph and refer the interested reader to this study for further information.

Lipp et al. [29] compile eviction sets from physical addresses which they obtain from the *pagemap* file that is present on many Linux systems. After their work was published, Android restricted the access to *pagemap* entries from user space.[1] Irazoqui et al. [24] and Gruss et al. [15] rely on *huge pages*, i.e. pages with typical sizes of > 1 MiB. Huge pages increase the page offset and thereby reveal the missing bits that determine the cache set. Oren et al. [36] and Bosman et al. [4] rely on special page allocation mechanisms in web browsers and operating systems that simplify the eviction set search. Genkin et al. [9] build eviction sets from sandboxed code within a web browser, while only relying on virtual addresses. Yet, they still require a precise and low-noise timing source to distinguish cache hit and miss. In contrast to these previous works, we propose a search algorithm that neither relies on physical addresses (whether obtained from pagemap, huge pages, or elsewhere), nor on certain features of memory allocators, nor on a precise timing source. Our approach for finding eviction sets is purely based on virtual addresses and robust against imprecise and noisy timing sources. In addition, we found our approach to be resilient against the random line replacement policy implemented in many ARM application processors.

Algorithm 1 outlines our approach for finding eviction sets. Prior to execution, we assume that n memory pages have been requested and are available as a memory pool. Note that we do not pose any requirements on the memory pages, thus, our algorithm works with any page size, including, but not limited to, 4 KiB. Since we want to evict the entire LLC, we need to choose n such that the requested memory area is large enough to fill it. In our experiments, we request a memory area that is twice as large as the LLC. This turned out to be sufficient for deriving all eviction sets. Algorithm 1 iterates over the allocated memory pages in sequential order. Each unused page is first added to a temporary eviction set $\mathcal{T}$ (line 4). Next, the first byte of each page in $\mathcal{T}$ is accessed and the average time $t_{\mathcal{T}}$ of this access cycle is measured. The parameter r determines how often the access cycle is repeated. In each cycle, all pages in $\mathcal{T}$ are accessed once. The overall timing is then divided by r to obtain the average. This is useful to account for imprecise timing sources and different replacement policies. A detailed discussion of r is given later in this section. The access time $t_{\mathcal{T}}$ is then compared with the time from the previous loop cycle (line 6), where $\mathcal{T}$ was one page smaller. If the time difference is higher than a threshold τ_{jump}, then there is a systematic contention in a cache set. In other words, the pages in $\mathcal{T}$ entirely fill one cache set and cause a line replacement in the process. This is illustrated in Figure 3(a), which shows the average access time $t_{\mathcal{T}}$ over an increasing number of pages in $\mathcal{T}$. As long as no set contention occurs, the average timings increase steadily. Once a contention happens, the timing peaks. Each peak in the plot indicates that one cache set is completely filled.

After a set contention is detected, Algorithm 1 iterates over all pages in $\mathcal{T}$, temporarily excludes one of them from $\mathcal{T}$, accesses this reduced set, and stores the average access time in t_p (line 9). If the time difference between $t_{\mathcal{T}}$ and t_p is again bigger than τ_{jump}, then the excluded page p belongs to the eviction set. This is illustrated in Figure 3(b), which shows the average access time t_p for all candidate pages p in $\mathcal{T}$. As soon as a candidate is part of the eviction set, the systematic set contention vanishes and the access time t_p drops. Each drop in the plot therefore indicates a member of the eviction set. Those pages are then added to the final eviction set $\mathcal{E}$, which is reported on line 14. The entries in $\mathcal{E}$ are subsequently removed from $\mathcal{T}$, before the outer loop continues to add unused pages to

[1] https://source.android.com/security/bulletin/2016-03-01

Algorithm 2 Removing duplicates.

```
1:  F = {}
2:  B = {1..m}
3:  while notempty(B) do
4:      f = pop(B)
5:      for s in B do
6:          t_E = access([onepage(E_f), E_s], r)
7:          if (t_E > τ_jump) then
8:              remove(B, s)
9:          end if
10:     end for
11:     add(F, f)
12: end while
```

$\mathcal{T}$. Once the outer loop reaches n, the reported eviction sets are expanded. This procedure is outlined in the following paragraph.

Eviction Set Expansion and Duplicates. Each of the m eviction sets reported by Algorithm 1 contains a list of memory pages. Since one page fits more than one cache line, we can derive multiple evictions sets from one $\mathcal{E}$. This is indicated in Figure 1. With 4 KiB pages and 64-byte cache lines, there are 64 lines on one page. As memory pages are contiguous physical memory, we know that those 64 lines belong to 64 consecutive cache sets. Hence, we can derive 64 adjacent eviction sets from one $\mathcal{E}$ (the first being $\mathcal{E}$ itself) by simply adding multiples of 64 to the start address of the pages. Depending on how large we chose n, it can happen that Algorithm 1 reports more than one eviction set for each cache set. We therefore need to check all reported eviction sets for duplicates and remove them. This procedure is outlined in Algorithm 2. It starts by storing the indices of all m reported eviction sets from Algorithm 1 in the list $\mathcal{B}$ for bookkeeping (line 2). As long as $\mathcal{B}$ is not empty, the first index is removed and assigned to f (line 4). The algorithm then iterates over all remaining indices s and accesses the corresponding eviction sets $\mathcal{E}_f$ and $\mathcal{E}_s$. In particular, one page in $\mathcal{E}_f$ and all pages in $\mathcal{E}_s$ are accessed consecutively, and the whole process is repeated r times. If the average timing $t_{\mathcal{E}}$ of these access cycles is larger than a threshold τ_{jump}, then $\mathcal{E}_f$ and $\mathcal{E}_s$ map to the same cache set. If this happens, the affected index s is removed from $\mathcal{B}$ (line 8), and the iteration continues. After all eviction sets have been tested, the index in f is added to the final list $\mathcal{F}$ (line 11). With each loop, $\mathcal{B}$ is shrinking as duplicate eviction sets are removed. Once $\mathcal{B}$ is empty, $\mathcal{F}$ contains the list of unique eviction set indices.

Timer Precision and Noise. Both algorithms 1 and 2 are designed to compensate imprecise and noisy timing sources. Although previous works [9, 40] suggest that accurate timers can be crafted even in environments that restrict access to high-precision timing sources, this engineering effort can be saved here. We believe this adds to the practicality of our attack. The precision and noise compensation in our algorithms is done by tuning the parameter r, as well as the threshold τ_{jump}. r defines the number of access cycles, i.e., how often a selection of memory pages is accessed. τ_{jump} defines how the timings of these access cycles are evaluated. In Algorithm 1, the accesses on line 5 will typically cause cache hits until the gathered pages trigger a systematic set contention. The difference between $t_{\mathcal{T}}$ and t_{old} will therefore be in the order of t_{miss}, where t_{miss} is the duration of a cache miss. Similarly, the accesses on line 9 will cause cache hits, if the candidate page p is part of the eviction set. In this case, the difference between $t_{\mathcal{T}}$ and t_p will again be around t_{miss}. Therefore, τ_{jump} can initially be set slightly smaller than t_{miss}. Adjustments can be made subsequently based on experimental data. The choice of r depends on the precision of the timer and the measurement quality. In general, r should be set such that $r \cdot t_{miss}$ is larger than the precision of the timer. It can be increased further, if high levels of noise are encountered, e.g., due to high system load. The choices for τ_{jump} and r also hold for Algorithm 2, where the accesses on line 6 will typically cause cache hits until $\mathcal{E}_f$ and $\mathcal{E}_s$ are duplicates. When this happens, a systematic set contention will occur, as the chosen page from $\mathcal{E}_f$ will be evicted by $\mathcal{E}_s$. In our experiments, we set r between 900 and 1000, and τ_{jump} to 500. The timer available on our test device, a Nexus 5X, provides a precision of 52 ns. This corresponds to approximately 100 clock cycles. In many related attacks (e.g. [49]), where timers typically have clock cycle accuracy, this rather low resolution would already introduce difficulties. In our approach, we simply tune r to compensate the low resolution.

Line Replacement Policies. We can also use r to compensate the effects of replacement policies. This is because the parameter r causes repetitive accesses to cache lines, which signals the cache controller that the accessed lines are of heightened interest and should not be replaced. For least-recently-used (LRU) policies, this is obviously beneficial, as unrelated cache activity will less likely interfere with the eviction set finding. But also random replacement policies benefit, because averaging over r access cycles attenuates the effect of unintended line replacements that happen due to random line selection. Our experiments outlined in Section 4 show that the choice of r as stated above is sufficient to compensate the effects of the random line replacement found on our test device.

Implementation and Limitations. Only few requirements have to be satisfied to find eviction sets with our approach. Memory must be allocated and accessed, and the accesses must be timed. Memory pages can be of arbitrary size and the timing source can be coarse-grained. This allows our algorithm to be implemented in user space and, thus, in a plethora of environments beyond mobile devices, e.g., desktop computers and cloud servers. Even sandboxes and virtual machines are typically no obstacle, enabling remote attacks, e.g., from JavaScript. The limitation of our approach is that the exact mapping of eviction sets to cache sets remains unknown. However, this is not a deficiency but a direct consequence of not knowing physical addresses. This choice makes our approach more practical and, thanks to the application of machine learning, still allows successful inference attacks.

Performance. We evaluated algorithms 1 and 2 on our test device with the parameters stated previously. The targeted last-level cache is 16-way set-associative and contains 1024 cache sets. It implements a random replacement policy and features 64-byte cache lines. Requested memory pages have a standard size of 4 KiB. Based on 1000 evaluation runs, algorithms 1 and 2 successfully yield all eviction sets with an average runtime of 20 seconds. Note that during our inference attack, eviction sets must be found only once and remain unchanged until the malicious app is restarted.

3.3 Post-processing and Feature Vectors

With the eviction sets obtained from algorithms 1 and 2, the last-level cache can be profiled in a traditional Prime+Probe [45] manner. This is done by filling each cache set with the corresponding eviction set (prime), before re-filling it immediately afterwards (probe). High levels of activity in the cache set will increase the probing time, whereas low levels will keep it low. Each probing time constitutes a *measurement sample* of the cache set. In our experiments, we measure n_T samples per cache set. The overall activity profile of the last-level cache, in short *LLC profile*, consists of the samples of all cache sets. Before these samples are classified by the machine learning algorithms, they are post-processed and converted to feature vectors. The following list outlines the post-processing steps.

(1) Elimination of timing outliers with a threshold τ_O. All outliers are replaced with the sample median. In our experiments, τ_O is set to 5 μs .
(2) Conversion of timing samples to binary representation. A threshold τ_H decides whether a sample value is *low* or *high*. In our experiments, τ_H is set to 750 ns.
(3) Sample compression by grouping high samples. Bursts of consecutive high samples are reduced to a single high value.

The removal of outliers reduces noise in the measurements. The binary representation simplifies interfacing with the machine learning algorithms and distills cache activity to two categories: *high* and *low*. Sample compression reduces data complexity while keeping the essential information of whether there was high or low activity. It also alleviates the effect of random replacement policies, as it compensates self-eviction during the probe step. From the post-processed measurement samples we derive three different feature vectors that are outlined in the following paragraphs.

Unordered Feature Vector. This feature vector is called *unordered*, because the exact mapping of eviction sets to cache sets is unknown, as explained in Section 3.2. This means that it is unclear which region of the cache a given sample stems from. However, it is possible to determine which region of the memory page a sample belongs to, because the addresses in a given eviction set share a common page offset. We leverage this observation to further compress the feature vector and reduce training complexity. In particular, we sum up the high samples of all cache sets that belong to the same page offset. With 4 KiB pages and 64-byte cache lines there are 64 distinct page offsets. Thus, the final feature vector contains 64 values.

FFT Feature Vector. For this feature vector, the post-processed measurement samples are converted with a fast Fourier transformation (FFT). The purpose of the FFT is to further reduce measurement noise, which has been pointed out by previous work [19, 36]. The FFT turns the n_T time-domain samples per cache set into $\frac{n_T}{2}$ frequency components (excluding the DC component). The employed sampling rate n_S is derived by dividing 1 second by the duration of one Prime+Probe cycle. We reduce the complexity of the $\frac{n_T}{2}$ frequency components by compressing them to n_F final components with a sliding window. These final components are again summed up over all cache sets that belong to the same page offset. Thus, the final feature vector contains $n_F \cdot 64$ values.

Ordered Feature Vector. This feature vector is different from the previous two. It is called *ordered*, because the exact mapping of eviction sets to cache sets is assumed to be known (e.g. from *pagemap* entries). This implies that additional attack steps have been performed (e.g. privilege escalation). Having the exact mapping, the cache sets are profiled in ascending order. But instead of grouping samples per page offset, the feature vector is constructed by concatenating the sum of high samples for each cache set. Thus, the final feature vector contains as many values as there are sets in the cache. The purpose of the *ordered* vector is to evaluate the performance of the previous two feature vectors. In the following Section 4, it is used to mount a *comparison attack* that captures cache activities with high resolution, but at the cost of additional attack steps.

4 EXPERIMENT SETUP AND RESULTS

In total, we conduct three experiments in which our malicious app detects running applications, visited websites, and streamed videos. The attack targets are given in Appendix A. For each machine learning algorithm, we build a multi-class classifier. 90% of the measured LLC profiles are thereby selected randomly for the training phase, while the rest of the data is chosen to evaluate the efficiency of the trained classifier. This 10% holdout approach yields the classification rates that are presented in this section. The rates are thereby based on the most likely label. Throughout the experiments, we observed that 10-fold cross-validation results are consistent with a 10% holdout approach. We also evaluated our classifiers against *unknown* inputs accounting for activity the models have not been trained with. In total, we collected more than 800 GB of cache profiling data to evaluate our inference attack.

4.1 Target Device

We use a Google Nexus 5X with Android v8.0.0 for our experiments. It features four ARM Cortex-A53 and two ARM Cortex-A57 processor cores. The malicious code runs on one of the A57 cores and profiles the LLC in the background. The LLC on the A57 core cluster contains 1024 cache sets. The target applications are launched and transition automatically to the A57 processor cluster. This is because the scheduler assigns resource-hungry processes (e.g. browser or multimedia applications) to the A57 cores to leverage their high performance. During all experiments, the system was connected to the campus wireless network and background processes from the Android OS and other apps were running. The timing source in our malicious app is the POSIX *clock_gettime* system call, which is available on all Android versions as part of the Bionic standard C library [3]. For website inference, we run Google Chrome and for video inference, we run the Netflix and YouTube apps.[2]

4.2 ML/DL Configuration

The following paragraphs discuss the parameter selection of the machine learning algorithms and provide further details about their usage. SVM and SAE classification is implemented with the help of LibSVM [5], whereas CNN classification is done using custom Keras [7] scripts together with the Tensorflow [1] GPU backend.

[2] Chrome v64.0.3282.137, Netflix v6.16.0, YouTube v13.36.50 .

The CNN is trained on a workstation with two Nvidia 1080Ti (Pascal) GPUs, a 20-core Intel i7-7900X CPU, and 64 GB of RAM.

SVM. The *ordered* and *unordered* feature vectors are classified with a linear SVM, while a non-linear SVM is used for the *FFT* feature vector. This is because the FFT is computed with non-linear functions (*cos*, *sin*) and the labels are linearly increasing for the classes. This choice is verified in preliminary experiments. Similarly, we determine that the linear kernel type outperforms radial basis and polynomial options for the *unordered* and *FFT* feature vectors.

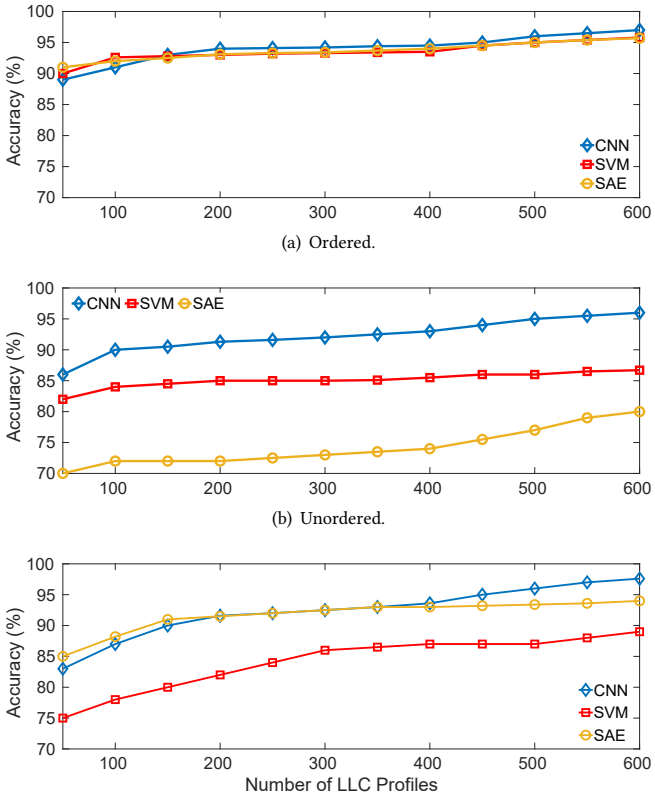
SAE. The SAE is constructed with two hidden layers of 250 and 50 neurons, respectively. The maximum number of epochs is set to 400, since no improvements can be observed afterwards. We decrease the effect of over-fitting by setting the L2 weight regularization parameter to 0.01. The output layer is a softmax layer.

CNN. The CNN consists of two 1-D convolution layers that are followed by maxpooling, dropout, as well as flatten and dense layers. The selection of the layer parameters is done with the help of preliminary experiments. Table 5 in Appendix A shows the parameter space that we explored. Eventually, we selected the parameters that yielded the lowest validation loss (highlighted in bold). The size of the first 1-D convolution layer is varied from 8 to 1024. The lowest validation loss is obtained with a size of 512. Similarly, the size of the second convolution layer is varied between 32 and 256, and eventually fixed to 256. A third convolution layer does not improve classification. The activation function in the convolution layers is set to rectified linear unit (ReLU). The size of the subsequent maxpooling layer is varied from 2 to 8. The default size of 2 yields the best results. The dropout of the following dropout layer is varied between 0.1 and 0.5, and finally set to 0.2. A higher dropout, as for example used in computer vision, adversely affects the classification. Next, the kernel size is adjusted and, out of the values between 3 and 27, a size of 9 achieves the lowest validation loss. A flatten layer shapes the data in our network, before a dense layer with size 200 and *tanh* activation function is appended. Finally, we employ a set of standard choices: the kernel initializers are chosen uniformly at random, an Adam optimizer is used to speed up the training phase, and the batch size is set to 50, as its effect on the classification rate is negligible.

4.3 Evaluation Results

The following sections present the evaluation results for application, website, and video inference.

4.3.1 ***Application Inference.*** For this attack, we target 100 random mobile applications from the Google Play Store, including dating, political, and spy apps. The full list is given in Table 3 in Appendix A. The first 70 apps are used to train and evaluate the machine learning models, while the remaining 30 apps are treated as being unknown. Each app is started and profiled for 1.5 seconds as described in Section 3. Within this time frame, we collect $n_T = 1,500$ measurement samples per cache set. For the FFT computation, the sampling rate $n_S = 1.9\,MHz$ and the number of bins $n_F = 15$. A comparison of the machine learning techniques and feature vectors is given in Figure 4. It contains three sub-plots that each show the classification results of SVM, SAE, and CNN over an increasing number of recorded LLC profiles. Recall that 90% of the recorded profiles are used for training, whereas the rest is



(a) Ordered.

(b) Unordered.

(c) FFT.

Figure 4: Classification results for application inference over an increasing number of LLC profiles for (a) *ordered*, (b) *unordered*, and (c) *FFT* feature vectors.

used to obtain the classification rates shown in the plots. The stated numbers of LLC profiles only reflect the measurement effort for the training phase, which is done offline on a training device. In the attack phase on the target device, recording a single LLC profile is sufficient to conduct a successful inference attack. The same holds for the results shown in figures 8 and 9. Plot 4(a) illustrates the results of the comparison attack, which is based on the *ordered* feature vector. The ordered profiling allows all three classifiers to distinguish applications with high confidence. CNN even achieves a classification rate of 97%. Plot 4(b) shows that classification rates drop, if the LLC profiles are based on the *unordered* feature vector. SAE even falls down to 80%, while CNN remains above 95%. The CNN we designed is therefore least affected by the unknown mapping between eviction and cache sets. As shown in Plot 4(c), the classification rates improve again, if the *FFT* feature vector is used. In particular, CNN and SAE benefit from this transformation, while SVM cannot fully leverage the information in the frequency spectrum. Our CNN reaches a classification rate of 97.8% and is thereby able to fully close the gap to the comparison attack.

A further performance metric for the three machine learning techniques is shown in Figure 5. It displays the receiver operating characteristic (ROC) curves for the *FFT* feature vector. For multi-class classification, ROC curves are computed for each class against all remaining classes (1 × N-1). The final ROC curve is then the

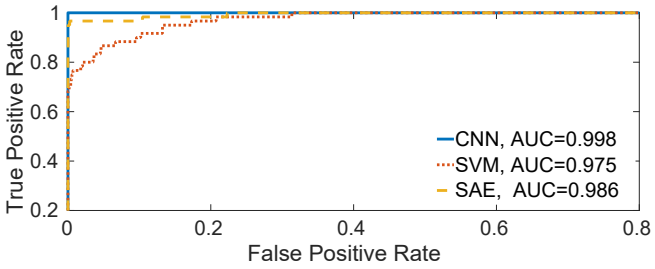


Figure 5: Average receiver operating characteristic (ROC) curves for SVM, SAE, CNN during application inference.

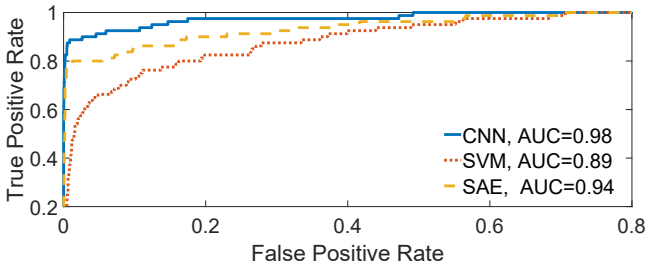


Figure 6: Average receiver operating characteristic (ROC) curves for SVM, SAE, CNN during website inference.

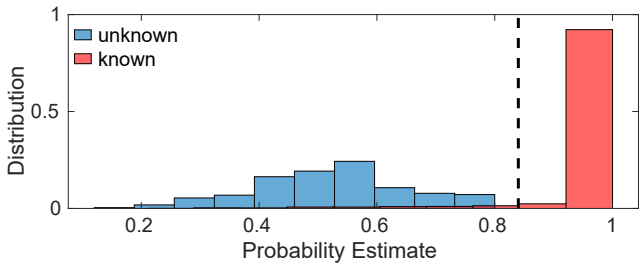


Figure 7: Probability estimates from the CNN softmax layer while classifying known and unknown apps.

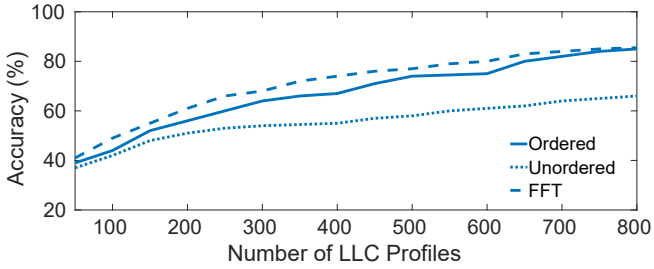


Figure 8: Website classification with our CNN for *ordered* (solid), *unordered* (dotted), and *FFT* (dashed) feature vectors.

average over all computed ROC curves. Figure 5 also provides the area under the curve (AUC) values in the plot legend. The higher the AUC, the less the machine learning technique suffers from false positives. While all three classifiers produce low false positive rates, CNN outperforms SVM and SAE. Based on the results of the application detection, we conclude that the CNN is the most suitable classifier for our inference attack. For website and video inference, we will therefore only present the results of the CNN.

Unknown Applications. The inherent nature of supervised learning is to recognize events that are similar to those used in the training phase. In practice, however, events may occur that the model has never been trained with. This also applies to our inference attack. Naturally, we cannot train our models with all existing applications on the app store. In fact, we want to focus only on apps that are of interest. Hence, we need a way to recognize and filter apps we have not trained yet. We achieve this by monitoring the probability estimates obtained from the softmax layer. Recall that we train only 70 apps out of the 100 that are given in Table 3. When we classify all 100 apps on our target device, we obtain the probability estimates shown in Figure 7. All known apps yield a high probability estimate close to 1, whereas unknown apps yield estimates that are significantly lower. We thus label each classification that yields a probability estimate below a threshold to be *unknown*. This threshold can be tuned according to attack requirements. A low value ensures that no application is missed during the attack. However, this leads to the detection of apps that have not been executed (false positives). A high threshold increases the confidence that all detected apps have actually been running. However, this comes at the cost of misclassifying known apps to be unknown (false negatives). As a general rule, we recommend to set the threshold at the intersection of the probability distributions obtained from the softmax layer. In our experiments, we chose a threshold of 0.84, which is illustrated as a dashed line in Figure 7. With this approach, our inference attack works reliably even in the presence of unknown applications on the target device.

4.3.2 ***Website Inference.*** The results in the previous section illustrate that our malicious app can reliably detect running applications with high confidence. Once a browser is detected, the app tries to infer websites that are currently viewed. For this attack, we target 100 different websites that are visited in Google Chrome. The list of websites is given in Table 4 in Appendix A. To emphasize that browsing histories are sensitive information, the list includes news, social media, political, and dating websites. For each website, we profile the LLC for 1.5 seconds and again obtain $n_T = 1,500$ samples per cache set. The features vectors are constructed in the same way as for application detection. Figure 8 shows the CNN classification results for all three feature vectors over an increasing number of LLC profiles. Similar to application inference, the *FFT* results match and slightly overshoot the results of the comparison attack. With a classification rate of 86%, the CNN is able to infer viewed websites with satisfactory confidence. The classification rate is lower compared to application inference, because loading and rendering websites leaves a weaker footprint in the last-level cache than opening apps. The ROC curves for the *FFT* feature vector are shown in Figure 6. The AUC values in the plot legend again illustrate that our CNN yields the lowest number of false positives. The CNN classifier and the *FFT* feature vector are therefore the best choices for website inference.

Unknown Websites. As previously, we train the CNN with only 70 websites and subsequently classify all 100 websites from Table 4. The probability estimates of the softmax layer are similar to the application inference and are thus not shown for the sake of brevity.

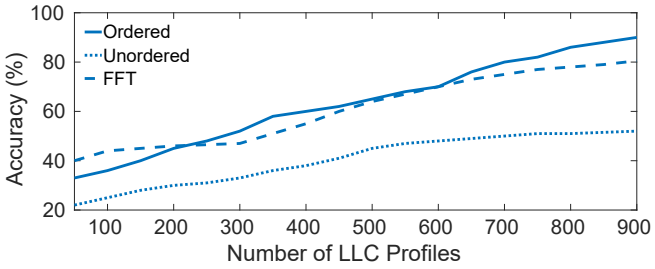


Figure 9: Video classification with our CNN for *ordered* (solid), *unordered* (dotted), and *FFT* (dashed) feature vectors.

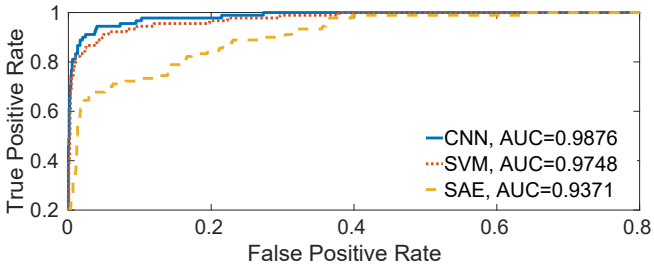


Figure 10: Average receiver operating characteristic (ROC) curves for SVM, SAE, CNN during video inference.

4.3.3 ***Video Inference.*** Similar to website inference, our malicious app also tries to detect videos that are being streamed in the Netflix and YouTube applications. We therefore target a total of 20 videos, which are given in Table 2 in Appendix A. In contrast to previous evaluations, we increase the profiling phase to 6 seconds. This is because the LLC footprint of videos is significantly less distinct compared to applications and websites. Within the extended profiling phase, we collect $n_T = 6,000$ samples per cache set. For the FFT computation, the number of bins, n_F, is increased to 60. Due to the high number of feature values, the size of the first convolution layer in our CNN is increased to 1024. The rest of the feature vectors are constructed in the same way as for application and website inference.

Figure 9 shows the CNN classification results for all three feature vectors over an increasing number of LLC profiles. The *FFT* results again match the comparison attack, but eventually fall behind by 10%. With a classification rate of 80%, our inference attack is able to infer streaming videos with moderate success. We believe that the LLC profiles do not contain enough information to distinguish multiple videos, as video processing is a rather homogeneous task. In addition, parts of the video decoding are typically outsourced to the GPU, which further reduces the cache footprint. Regarding the ROC curves, which are shown in Figure 10, the CNN again outperforms SVM and SAE. Due to the reduced success rate for video classification, we skipped the evaluation of unknown videos. Yet, we expect it to follow the same trend as for application and website inference.

5 DISCUSSION

The previous section shows that modern machine learning techniques enable successful inference attacks even when simple cache profiling methods are employed. Throughout our experiments, frequency-domain transforms of LLC profiles yield high success rates when being classified by a CNN. The FFT thereby reduces the noise in the measurements, while the CNN distills consistent features despite the lacking order with which the cache sets are profiled. The resulting classification closely matches the comparison attack that is based on precisely ordered LLC profiles obtained with the help of additional attack steps. Clearly, an adversary can omit these steps when using our inference attack. The limit of our attack becomes apparent when the LLC activity is less distinct or faints. While applications and websites can reliably be inferred, the accuracy drops for video classification. This could be improved by increasing the profiling time or including other side-channels such as GPU activity. Nevertheless, the results clearly indicate that a carefully crafted and well-trained CNN enables inference attacks that are robust, easy to implement, and therefore practical.

Attack in Numbers. The pre-trained CNN model is approximately 24 MiB large. Together with the attack code, this yields a total app size of 25 MiB. The other ML models are significantly smaller. If the number of target classes increases, the size of the models grows linearly. When the app is launched, it first creates the eviction sets required for LLC profiling. As stated in Section 3.2, this takes 20 seconds on average. Recording one LLC profile takes at most 6 seconds. The subsequent classification is also a matter of seconds. The work by Ignatov et al. [22] is a useful reference to assess the performance of CNN classification on Android phones. On the Nexus 5X, all CNN classification benchmarks finish in under 13 seconds, yielding a total attack time of well under a minute.

Attack Portability. Our inference attack is not limited to the device and scenario presented in this work. The attack components are flexible and can be ported easily. The *eviction set algorithms* presented in Section 3.2 are generic and can be adapted to other environments with appropriate choices of r and τ_{jump}. The algorithms are robust against changes in cache size, number of sets, associativity, and replacement policy. They will therefore find eviction sets not only on ARM-based mobile devices but also on x86 systems. With the *unordered* and *FFT* feature vectors introduced in Section 3.3, the exact mapping of eviction sets to cache sets is not required for an attack. This has multiple advantages. First, the attack does not require physical addresses and can be launched entirely from user space. Second, it is agnostic to the page size of the system and works with pages from less than 4 KiB to multiple MiB. Third, the attack can be launched without additional and complex attack steps (e.g. [14]) that would increase the attack effort and lower the practicality. The designed *convolutional neural network* is a suitable fit for the resulting cache observations. It allows to distill the cache footprint of virtually any activity occurring on the target system. Future research may study detecting exact versions of applications or input events such as swipes, touches, or the like. The CNN presented in this work is a good starting point for any new attack scenario. Since a fine-tuning of the parameters may be necessary, Table 5 can be consulted for sensible parameter ranges. In summary, our inference attack is versatile and constitutes a threat not only to mobile applications, but also to virtual machines and containers on servers and any desktop software. We consider the exploration of other attack scenarios as future work.

Countermeasures. The inference attack proposed in this work has two fundamental requirements. First, it relies on the mutual eviction of cache lines from the last-level cache. This eviction can be impaired by cache flushing [10], cache partitioning [31], scheduling [46] and line replacement policies [25]. However, most of these approaches require changes to the processor hardware or introduce substantial performance overhead. The second requirement is to time memory accesses. While disabling access to timers or overlaying them with noise [21] complicates attacks, these strategies seem far from sustainable, as timing sources can be crafted artificially even in restrictive execution environments [9, 40]. Another approach is to craft adversarial examples against DL based classification models [11]. Inci et al. [23] recently showed that CNN-based side-channel attacks can be prevented by adding specially crafted noise to performance counters. This approach could also be adopted by applications running on mobile devices. A general defense strategy is the detection of ongoing attacks, e.g., by monitoring the memory access behavior of programs [50]. However, most detection approaches are probabilistic and, thus, suffer from false positives and false negatives. The generic nature of our inference attack renders it extremely difficult to defend against, especially without dedicated support from the processor hardware. For a further discussion of relevant countermeasures we refer the interested reader to the survey by Ge et al. [8].

6 RELATED WORK

Our inference attack relates to previous work in the areas of website and application inference, cache attacks on ARM-based devices, and machine learning in the context of side-channel attacks. The following sections discuss these relations in more detail.

6.1 Website and Application Inference

In literature, the inference of visited websites has been investigated from many perspectives. Vila et al. [47] use shared event loops to infer opened websites from the server side. Panchenko et al. [37] use traffic analysis to detect visited websites in the Tor network. Zhang et al. [51] exploit iOS APIs to infer visited websites and running applications. Spreitzer et al. [42, 43] obtain distinct features from the *procfs* filesystem and use Android APIs to infer opened web pages and applications. Lee et al. [27] exploit uninitialized GPU memory pages to detect websites, while Naghibijouybari et al. [35] exploit OpenGL APIs and GPU performance counters for this task. Gulmezoglu et al. [20] observe hardware performance events of modern processors to infer visited websites. Diao et al. [6] infer applications through system interrupts. Jana and Shmatikov [26] demonstrate that websites leave a distinct memory footprint in the browser application. Oren et al. [36] as well as Gruss et al. [13] demonstrate that opened websites and their individual elements can be inferred from cache observations taken from a malicious JavaScript applet. Shusterman et al. [41] extend this work by inferring websites from JavaScript with simple last-level cache profiles that are classified by convolutional neural networks and long short-term memory. As this is concurrent work to ours, we provide a closer comparison later in this section. Gulmezoglu et al. [19] use cache observations to detect running applications in co-located virtual machines. In this work, we also use measurements of cache activity to infer running applications, visited websites, and streamed videos. The comparison of our results with other attacks is given in Table 1. It shows that GPU-, network traffic-, and operating system-based attacks achieve higher success rates for website classification than our inference attack. However, our attack does not require access to GPU, network, or OS APIs, which can be restricted or easily monitored. We rely on simple memory accesses and coarse-grained timing measurements, which are difficult to restrict and monitor. Compared to the LLC-based attack by Oren et al. [36], our success rates are higher, even though we classify significantly more websites. Compared to the results by Shusterman et al. [41], we achieve similar classification rates. At the same time, we relax the attacker model by not only compensating imprecise timing sources but also random cache replacement policies. For application detection, the success rate of our approach is, to the best of our knowledge, the highest one in literature.

Table 1: Related website and application inference attacks.

	Attack	Attack Vector	Acc. (%)	Classes
Websites	**Our**	**Last-level Cache (LLC)**	**85.8**	**70**
	[36]	LLC	82.1	8
	[41]	LLC	86.1	100
	[20]	CPU Performance Events	84.0	30
	[35]	GPU Performance Events	93.0	200
	[27]	Uninitialized GPU Memory	95.4	100
	[26]	Scheduling Statistics	78.0	100
	[47]	Shared Event Loops	76.7	500
	[37]	Traffic Analysis	92.5	100
	[51]	iOS APIs	68.5	100
	[43]	Java-based Android API	89.4	20
	[42]	ProcFS Leaks	94.0	20
Applications	**Our**	**LLC**	**97.8**	**70**
	[19]	LLC	78.5	40
	[51]	iOS APIs	89.0	120
	[43]	Java-based Android API	85.6	20
	[6]	Interrupt Handling	87.0	100
	[42]	ProcFS Leaks	96.0	100

***Comparison with Shusterman et al.* [41].** Shusterman et al. present a similar inference attack than the one proposed in this work. The LLC profiling is based on traces of cache activity (called *memorygrams*) that are obtained from repeatedly accessing a buffer as large as the LLC. The time to access the entire buffer then relates to the activity in the LLC, which is used to infer websites. In contrast, we build eviction sets to profile individual parts of the LLC. This provides a more fine-grained view on the cache activity. Shusterman et al. profile the LLC for 30 seconds, while our profiling phase is only 1.5 seconds. The authors further choose their CNN parameters based on the success rate, while we select the parameters based on validation loss. Using the validation loss makes the trained model more robust, as success rates increase with over-fitting. This leads to differences in the parameter selection, in particular regarding the number of convolution layers, the kernel size, and the pooling size. While Shusterman et al. use 3 convolution layers, a varying kernel size for each layer, and a pooling size of 4, we train our model with 2 convolution layers, a constant kernel size per layer, and a pooling size of 2. Furthermore, we incorporate the CNN design guidelines by Prouff et al. [38] (see Section 6.3). In summary, both Shusterman et al. and this work propose inference attacks that share a

common goal, but differ regarding approach and attack environment. Shusterman et al. launch their attack from JavaScript on Intel CPUs, while we conduct our attack on Android and ARM. Yet, the achieved classification rates for Google Chrome are comparable for both environments. We believe this emphasizes that inference attacks of this kind are a practical, cross-platform threat.

6.2 Cache Attacks on ARM

Most cache attacks known today either use dedicated flush instructions [16, 18, 49] or targeted thrashing of cache sets [9, 17, 24, 45] to observe cache activity. While many techniques have been proposed for x86 processors, Lipp et al. [29] demonstrated the feasibility of attacks also on ARM processors, which complicate attacks with random replacement policies, exclusive and non-inclusive cache hierarchies, and internal line locking mechanisms [12]. In this work, we show that despite these challenges, simple LLC observations are sufficient to infer user activity on ARM-based mobile devices. Unlike previous work, we pair these simple observations with advanced machine learning techniques and thereby alleviate attack difficulties on ARM processors.

***Comparison with Lipp et al.* [29].** Lipp et al. perform multiple cache attacks on ARM devices, including Prime+Probe [45], the attack technique employed in this work. For this reason, we compare the Prime+Probe technique by Lipp et al. to ours. In particular, we set up an experiment, in which we try to classify the first 20 websites from Table 4 in Appendix A. We obtain the Prime+Probe code from the GitHub repository [30] by Lipp et al. and run the eviction strategy evaluator on the ARM Cortex-A57. The strategy 22-1-6 yields the highest eviction rate of 98%. The code by Lipp et al. uses *pagemap* entries to find eviction sets (thus requiring root privileges), while we employ algorithms 1 and 2 that work without elevated privileges. The profiling phase for the website classification is 1.5 seconds. We collect 800 LLC profiles for each website, and use 90% as training data and the rest as test data. We then derive the *ordered* and *FFT* feature vectors, as described in Section 3.3. We omit the *unordered* feature vector, as it yielded lower accuracies than the other ones throughout our experiments. While the approach by Lipp et al. achieves classification rates of 90% and 85% (*ordered* and *FFT*), our approach yields 93% and 94%. Thus, our profiling technique achieves higher classification rates while requiring no root privileges to find eviction sets.

6.3 Machine Learning and SCAs

Side-channel attacks (SCAs) typically rely on signal processing and statistics to infer information from observations. Since 2011, advanced machine learning approaches were introduced to side-channel literature. Lerman et al. [28] use random forests (RFs), SVMs, and self-organizing maps (SOMs) to compare the effectiveness of machine learning techniques against template attacks. Later, Gulmezoglu et al. [19] showed that SVM-based approaches can be used to extract features from FFT components obtained from cache traces. Martinasek et al. [33, 34] showed that basic neural network techniques can recover AES keys with a 96% success rate. With the increasing popularity of deep learning, corresponding techniques were also studied for SCAs. In 2016, Maghrebi et al. [32] compared four deep learning techniques with template attacks while attacking an unprotected AES implementation using power consumption. In 2017, Schuster et al. [39] showed that encrypted streams can be used to classify videos with CNNs. Gulmezoglu et al. [20] used hardware performance events to classify websites visited on a personal computer using SVM and CNN.

***Comparison with Prouff et al.* [38].** It is important to follow a systematic approach when choosing parameters of CNNs. Prouff et al. studied the parameter selection of MLP and CNN in the context of side-channel attacks. There are in total 4 rules to follow according to their work. The first one states that consecutive convolution layers should have the same parameters. The second rule is that pooling layers should have a dimension of 2. The third rule is that the number of filters in a convolution layer should be higher than the one of the previous layer. The fourth rule states that all convolution layers should have the same kernel size. While we implement rules 1, 2 and 4, rule number 3 does not apply to our experiments. Instead, the number of filters decreases for each convolution layer. In addition, we do not exhaustively explore batch sizes and optimization methods, since they do not significantly affect the validation loss in our case.

7 CONCLUSION

Inference attacks undermine our privacy by revealing our most secret interests, preferences, and attitudes. Unfortunately, modern processors, which constitute the core of our digital infrastructure, are particularly vulnerable to these attacks. Footprints in the processor cache allow the inference of running applications, visited websites, and streaming videos. Above all, the advances in machine learning, especially the concepts behind deep learning, significantly lower the bar of successfully implementing inference attacks. Our work demonstrates that it is possible to execute an inference attack without privileges, permissions, or access to special programming interfaces and peripherals. The simple nature of the attack code makes a comprehensive defense extremely difficult. This simplicity is paired with the careful application of deep learning. Interferences such as measurement noise, misalignment, or unfavorable processor features are thereby conveniently compensated. The comparison with concurrent work furthermore indicates that inference attacks of this kind are ubiquitous and succeed across runtime environments and processing hardware. For applications that value the privacy of their users, protection against inference attacks is therefore of utmost importance. A comprehensive solution, however, seems to require a closer collaboration between hardware manufacturers, operating system designers, and application developers.

ACKNOWLEDGMENTS

We would like to thank our shepherd Martina Lindorfer as well as the anonymous reviewers for their valuable feedback. This work is supported by the National Science Foundation, under grants CNS-1618837 and CNS-1814406. Berk Gulmezoglu is also supported by the WPI PhD Global Research Award 2017.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 265–283.

[2] Alexa Internet Inc. 2018. The top 500 sites on the web. http://www.alexa.com/topsites. Last accessed 2018-01-01.

[3] Android Open Source Project. 2008. Bionic Initial Contribution. https://android.googlesource.com/platform/bionic/+/a27d2baa. Last accessed 2019-01-21.

[4] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *IEEE Symposium on Security and Privacy, San Jose, USA, May 22-26, 2016*. 987–1004.

[5] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM TIST* 2, 3 (2011), 27:1–27:27.

[6] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. 2016. No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. In *IEEE Symposium on Security and Privacy, San Jose, USA, May 22-26, 2016*. 414–432.

[7] Chollet F. et al. 2018. Keras. https://keras.io. Last accessed 2019-01-21.

[8] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering* 8, 1 (2018), 1–27.

[9] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. 2018. Drive-By Key-Extraction Cache Attacks from Portable Code. In *Applied Cryptography and Network Security - ACNS 2018, Proceedings*. 83–102.

[10] Michael Misiu Godfrey and Mohammad Zulkernine. 2013. A Server-Side Solution to Cache-Based Side-Channel Attacks in the Cloud. In *2013 IEEE Sixth International Conference on Cloud Computing, Santa Clara, CA, USA, June 28 - July 3, 2013*. IEEE Computer Society, 163–170.

[11] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[12] Marc Green, Leandro Rodrigues Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. 2017. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In *26th USENIX Security Symposium, Vancouver, BC, Canada, August 16-18, 2017*. 1075–1091.

[13] Daniel Gruss, David Bidner, and Stefan Mangard. 2015. Practical Memory Deduplication Attacks in Sandboxed Javascript. In *ESORICS 2015 - Proceedings Part I, Vienna, Austria, September 21-25, 2015 (Lecture Notes in Computer Science)*, Vol. 9326. Springer, 108–122.

[14] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *Proceedings of ACM SIGSAC CCS 2016, Vienna, Austria, October 24-28, 2016*. 368–379.

[15] 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 2016*. Springer, 300–321.

[16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment - DIMVA 2016*. Springer, 279–299.

[17] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *24th USENIX Security Symposium, Washington, D.C., USA, August 12-14, 2015*. 897–912.

[18] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In *32nd IEEE Symposium on Security and Privacy, 22-25 May 2011, Berkeley, California, USA*. 490–505.

[19] Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. 2017. Cache-Based Application Detection in the Cloud Using Machine Learning. In *AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*. 288–300.

[20] Berk Gülmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. 2017. PerfWeb: How to Violate Web Privacy with Hardware Performance Events. In *ESORICS 2017 - Proceedings Part II, Oslo, Norway, September 11-15, 2017 (Lecture Notes in Computer Science)*, Vol. 10493. Springer, 80–97.

[21] Wei-Ming Hu. 1991. Reducing Timing Channels with Fuzzy Time. In *IEEE Symposium on Security and Privacy*. 8–20.

[22] Andrey Ignatov, Radu Timofte, William Chou, Ke Wang, Max Wu, Tim Hartley, and Luc Van Gool. 2018. AI Benchmark: Running Deep Neural Networks on Android Smartphones. *CoRR* abs/1810.01109 (2018). arXiv:1810.01109 http://arxiv.org/abs/1810.01109

[23] Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2018. DeepCloak: Adversarial Crafting As a Defensive Measure to Cloak Processes. *arXiv preprint arXiv:1808.01352* (2018).

[24] Gorka Irazoqui, 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, SP 2015*. 591–604.

[25] 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 *Proceedings of ISCA 2010, June 19-23, 2010, Saint-Malo, France*. 60–71.

[26] Suman Jana and Vitaly Shmatikov. 2012. Memento: Learning Secrets from Process Footprints. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 143–157.

[27] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *IEEE Symposium on Security and Privacy, SP , Berkeley, CA, USA, May 18-21, 2014*. 19–33.

[28] Liran Lerman, Gianluca Bontempi, and Olivier Markowitch. 2015. A machine learning approach against a masked AES - Reaching the limit of side-channel attacks with a learning model. *J. Cryptographic Engineering* 5, 2 (2015), 123–139.

[29] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium, Austin, TX, USA, August 10-12, 2016*. 549–564.

[30] Lipp M. and others. 2018. ARMageddon. https://github.com/IAIK/armageddon. Last accessed 2019-01-21.

[31] 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 2016, Barcelona, Spain, March 12-16, 2016*. 406–418.

[32] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. 2016. Breaking Cryptographic Implementations Using Deep Learning Techniques. In *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*. 3–26.

[33] Zdenek Martinasek, Jan Hajny, and Lukas Malina. 2013. Optimization of Power Analysis Using Neural Network. In *CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers*. 94–107.

[34] Zdenek Martinasek and Vaclav Zeman. 2013. Innovative method of the power analysis. *Radioengineering* 22, 2 (2013), 586–594.

[35] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael B. Abu-Ghazaleh. 2018. Rendered Insecure: GPU Side Channel Attacks are Practical. In *Proceedings of ACM SIGSAC CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2139–2153.

[36] 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 *Proceedings of ACM SIGSAC CSS 2015, Denver, CO, USA, October 12-16, 2015*. 1406–1418.

[37] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. 2016. Website Fingerprinting at Internet Scale. In *Proceedings of NDSS 2016, San Diego, USA, February 21-24, 2016*.

[38] Emmanuel Prouff, Remi Strullu, Ryad Benadjila, Eleonora Cagli, and Cécile Dumas. 2018. Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database. *IACR Cryptology ePrint Archive* 2018 (2018), 53. http://eprint.iacr.org/2018/053

[39] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. 2017. Beauty and the Burst: Remote Identification of Encrypted Video Streams. In *26th USENIX Security Symposium, Vancouver, BC, Canada, August 16-18, 2017*. 1357–1374.

[40] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017*. 247–267.

[41] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2018. Robust Website Fingerprinting Through the Cache Occupancy Channel. *CoRR* abs/1811.07153 (2018). arXiv:1811.07153 http://arxiv.org/abs/1811.07153

[42] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. 2018. ProcHarvester: Fully Automated Analysis of Procfs Side-Channel Leaks on Android. In *AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*. 749–763.

[43] Raphael Spreitzer, Gerald Palfinger, and Stefan Mangard. 2018. SCAnDroid: Automated Side-Channel Analysis of Android APIs. In *Proceedings of WiSec 2018, Stockholm, Sweden, June 18-20, 2018*. ACM, 224–235.

[44] The Verge. 2017. Google announces over 2 billion monthly active devices on Android. https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users. Last accessed 2019-01-21.

[45] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology* 23, 1 (2010), 37–71.

[46] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. 2014. Scheduler-based Defenses against Cross-VM Side-channels. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, San Diego, CA, 687–702.

[47] Pepe Vila and Boris Köpf. 2017. Loophole: Timing Attacks on Shared Event Loops in Chrome. In *26th USENIX Security Symposium, Vancouver, BC, Canada, August 16-18, 2017*. 849–864.

[48] Pepe Vila, Boris Köpf, and José Francisco Morales. 2018. Theory and Practice of Finding Eviction Sets. *CoRR* abs/1810.01497 (2018). arXiv:1810.01497 http://arxiv.org/abs/1810.01497

[49] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 719–732.

[50] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. 2016. Memory DoS Attacks in Multi-tenant Clouds: Severity and Mitigation. *CoRR* abs/1603.03404 (2016). arXiv:1603.03404 http://arxiv.org/abs/1603.03404

[51] Xiaokuan Zhang, Xueqiang Wang, Xiaolong Bai, Yinqian Zhang, and XiaoFeng Wang. 2018. OS-level Side Channels without Procfs: Exploring Cross-App Information Leakage on iOS. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*.

A APPENDIX

This section provides complementary information regarding our experiments. It lists the profiled applications, websites, and videos, and gives the parameters that were explored while constructing our convolutional neural network.

A.1 Profiled Applications, Websites, Videos

Table 2 lists the videos that are profiled in our experiments. YouTube videos are chosen from the list of most watched videos on YouTube. Trailers and recaps viewed in the Netflix app are from the series *The House of Cards*. Tables 3 and 4 list the profiled applications and websites. The selection of websites is taken from the Alexa ranking [2].

A.2 CNN Parameter Selection

Table 5 documents the CNN parameter exploration that was conducted prior to our experiments. The final parameters are highlighted in bold.

Table 2: List of profiled videos.

Youtube (left) and Netflix (right) Videos	
1) Despacito	1) Season 1 Trailer
2) See You Again	2) Season 2 Trailer
3) Shape of You	3) Season 3 Trailer
4) Gangnam Style	4) Season 4 Trailer
5) Uptown Funk	5) Season 5 Trailer
6) Sorry	6) Season 1 Recap
7) Sugar	7) Season 2 Recap
8) Shake it Off	8) Season 3 Recap
9) Roar	9) Season 4 Recap
10) Bailando	10) Season 1 Trailer (Extended)

Table 3: List of profiled applications.

Applications		
1) Spotify	35) Reddit	69) OurTime
2) Snapchat	36) Imdb	70) HowAboutWe
3) Instagram	37) Creditkarma	71) Tiktok
4) Facebook	38) Alexa	72) Canva
5) YouTube	39) Yahoo	73) Autolist
6) Chrome	40) Starz	74) Sephora
7) Netflix	41) Zedge	75) Indeed
8) Uber	42) Textnow	76) Marvel
9) Twitter	43) Soundcloud	77) Hinge
10) Bitmoji	44) Booking	78) Daylio
11) Google Drive	45) Duolingo	79) Roku
12) Pandora	46) Tinder	80) Investing
13) NY Times	47) Joom	81) Ifood
14) Pinterest	48) Xbox	82) Fitbit
15) Lyft	49) Shazam	83) Goodrx
16) InBrowser	50) Chase	84) Fastnews
17) Firefox Focus	51) Huffington	85) Touchnote
18) Orfox	52) Breitbart	86) Nike
19) Musical Focus	53) Earspy	87) Sony
20) Wish	54) Ispy	88) Kayak
21) Hulu	55) Spycamera	89) Expedia
22) Workout	56) Mspy	90) Sketch
23) Waze	57) Secretagent	91) PlutoTV
24) Walmart	58) Politico	92) Grubhub
25) Wholefoods	59) TheHill	93) McDonald's
26) Dairy Queen	60) Dailykos	94) Target
27) Discord	61) Infowars	95) Trivia
28) Venmo	62) Match	96) Starbucks
29) Groupon	63) PlentyofFish	97) Horoscope
30) Twitch	64) Zoosk	98) Beetles
31) Yelp	65) eHarmony	99) Glasdoor
32) Letgo	66) Okcupid	100)Tickermaster
33) Iheart	67) Badoo	
34) eBay	68) Christian Mingle	

Table 4: List of profiled websites.

Websites					
1) Google	18) Craigslist	35) Hulu	52) Breitbart	69) BlackPeopleMeet	86) Nginx
2) Facebook	19) Paypal	36) Quora	53) Drudgereport	70) HowAboutWe	87) Springer
3) Wikipedia	20) Apple	37) Salesforce	54) Politico	71) Oracle	88) Apache
4) Amazon	21) Bing	38) Wells	55) The Hill	72) Reuters	89) Flickr
5) Reddit	22) Chase	39) Bank of America	56) Slate	73) BBC	90) Grawatar
6) Yahoo	23) Zillow	40) Stackoverflow	57) Dailykos	74) Nasa	91) Sourceforge
7) Twitter	24) Walmart	41) Guardian	58) Infowars	75) Eventbrite	92) Archive
8) eBay	25) Yelp	42) Forbes	59) Salon	76) Dailymotion	93) Go
9) Netflix	26) Github	43) Dropbox	60) TheBlaze	77) Blogger	94) Wix
10) Linkedin	27) NY Times	44) Mozilla	61) Match	78) Nature	95) Myspace
11) Office	28) Pinterest	45) Soundcloud	62) Plenty of Fish	79) Digg	96) Mysql
12) Cnn	29) Imdb	46) Weebly	63) Zoosk	80) Wiley	97) Time
13) Espn	30) Microsoft	47) Vimeo	64) Okcupid	81) Wired	98) Cnbc
14) Wikia	31) Msn	48) Adobe	65) eHarmony	82) Ted	99) Skype
15) Twitch	32) Fox News	49) Wordpress	66) Badoo	83) Feedburner	100) Alibaba
16) Live	33) Blogspot	50) Tumblr	67) Christian Mingle	84) Oath	
17) Instagram	34) Dailymail	51) Huffington	68) OurTime	85) Ietf	

Table 5: CNN parameter exploration. Final selection highlighted in bold.

Convolution	Max Pooling	Dropout	Kernel Size	Dense	Training Loss	Training Accuracy (%)	Validation Loss	Validation Accuracy (%)
1024	2	0.1	3	50	0.2568	93.54	0.7092	83.04
512	2	0.1	3	50	0.2085	94.23	0.6876	84.03
256	2	0.1	3	50	0.2554	95.01	0.7127	82.75
128	2	0.1	3	50	0.2666	93.56	0.7307	82.78
64	2	0.1	3	50	0.2790	92.31	0.7342	82.68
32	2	0.1	3	50	0.5443	83.23	0.8038	80.56
16	2	0.1	3	50	0.3821	89.04	0.6910	82.38
8	2	0.1	3	50	0.4513	86.94	0.7057	82.29
512-256	2	0.1	3	50	0.2581	92.17	0.6436	84.40
512-128	2	0.1	3	50	0.3725	89.09	0.6510	84.18
512-64	2	0.1	3	50	0.6743	81.12	0.7638	80.93
512-32	2	0.1	3	50	0.4495	87.25	0.7355	81.46
512-256-128	2	0.1	3	50	0.2564	91.24	0.6440	84.55
512-256-64	2	0.1	3	50	0.3345	90.15	0.6609	84.09
512-256-32	2	0.1	3	50	0.3259	90.75	0.6984	81.25
512-256	2	0.1	3	50	0.2581	92.17	0.6436	84.40
512-256	4	0.1	3	50	0.4823	86.48	0.7467	82.45
512-256	8	0.1	3	50	0.5642	84.24	0.7160	81.54
512-256	2	0.2	3	50	0.2581	92.17	0.6436	84.80
512-256	2	0.3	3	50	0.2756	91.24	0.6783	83.34
512-256	2	0.4	3	50	0.2894	90.57	0.6928	82.86
512-256	2	0.5	3	50	0.3184	88.37	0.7293	81.43
512-256	2	0.2	6	50	0.4068	87.65	0.6583	83.45
512-256	2	0.2	9	50	0.3686	89.48	0.6314	85.21
512-256	2	0.2	18	50	0.3079	91.00	0.6794	84.82
512-256	2	0.2	27	50	0.3283	90.06	0.6915	83.87
512-256	2	0.2	9	100	0.3387	90.03	0.6836	83.07
512-256	2	0.2	9	150	0.3208	90.39	0.6456	83.57
512-256	**2**	**0.2**	**9**	**200**	**0.3104**	**91.25**	**0.6218**	**85.76**
512-256	2	0.2	9	250	0.3487	89.74	0.6424	83.38
512-256	2	0.2	9	300	0.3562	88.96	0.6592	82.51
512-256	2	0.2	9	350	0.3859	86.52	0.6834	82.15