Privacy Implications of Ubiquitous Caching in Named Data Networking Architectures

Technical Report TR-iSecLab-0812-001

Tobias Lauinger
Northeastern University,
Boston, USA

Nikolaos Laoutaris
Telefónica Research,
Barcelona, Spain

Pablo Rodriguez
Telefónica Research,
Barcelona, Spain

Thorsten Strufe
Technische Universität
Darmstadt, Germany

Ernst Biersack
Eurécom,
Sophia-Antipolis, France

Engin Kirda
Northeastern University,
Boston, USA

ABSTRACT

Content is at the heart of next-generation Internet architectures such as Content-Centric Networking (CCN): Instead of routing location-based messages to end hosts, the network transmits location-independent, named content objects. Such data objects can (and are envisioned to) be cached in arbitrary network nodes. In this technical report, we discuss several privacy attacks related to the ubiquitous presence of caching in CCN: Attackers can monitor access to specific content objects by other users connected to the same cache, they can discover the names of objects stored in the cache, and they can duplicate entire data flows from and to other users of the cache. We identify the architectural features and protocol functions that make these attacks possible, and we recommend measures to mitigate cache-based attacks.

1. INTRODUCTION

A range of named data networking (NDN) architectures have been proposed to address performance and security issues of the current Internet architecture. Many proposals advocate addressing content instead of addressing end hosts, enabling innovative routing strategies and providing the potential for ubiquitous caching at the network layer. However, with any new component or protocol feature that an architecture proposal introduces, it potentially introduces new security issues as well.

In this technical report, we discuss undesirable side effects of ubiquitous caching with respect to privacy. Caching can achieve reductions in upstream bandwidth and content retrieval delays. Yet, the performance increase comes at the cost of a reduction in privacy due to caches keeping transient communication traces.

Caches have been exploited for privacy attacks in other contexts, e.g. DNS caches [13] and Web browser caches [9]. However, NDN architectures tend to exacerbate the privacy risks of caches: Currently deployed caches, such as DNS caches, Web caches and CDNs, are application-specific and shared by a relatively large number of users. Many NDN architectures, on the other hand, replace application-specific caches with general-purpose network-layer caches. General-purpose caches increase the attack surface for privacy attacks by putting *any* communication at risk. Furthermore, such caches can be part of any network node; they can potentially be shared by fewer users. Fewer users sharing a cache increases the information gained in privacy attacks because there remains less uncertainty as to which user requested certain information.

We introduce three privacy attacks based on caches: *Request monitoring* allows an attacker to track any access to a given content object. Given prior information, an attacker can use this technique to confirm a hypothesis about the victim. The *object discovery* attack lets attackers gain an overview of the data objects that are transiting through a cache, and the *flow cloning* attack can be used to replicate an entire data flow to another machine. Note that all these attacks can be carried out by any legitimate user of a cache. To the best of our knowledge, we are the first to describe the latter two attacks; the former attack is novel at least in the context of NDN architectures.

In the absence of a real-world deployment of a NDN network with real user data, it is challenging to quantify the severity of possible attacks. Several design and deployment parameters have direct influence on the feasibility of attacks. For instance, if the system is used to disseminate only static, high-popularity content, the risk of privacy breaches is likely to be low. Similarly, the supported protocol primitives, the placement of caches and decisions about *what* to cache *for how long* directly impact the scope and feasibility of attacks.

Although it may be too early to assess the real-world impact of such attacks, it is time to identify what architectural and operational choices enable these attacks and how countermeasures could be designed. Therefore, the purpose of this technical report is not so much to quantitatively analyse the attacks; we leave this for future work. Rather, our goal is to raise awareness for the consequences of ubiquitous caches on user privacy.

In order to allow for a focussed discussion, we concentrate on Content-Centric Networking (CCN) as a representative of NDN architectures. In this technical report, we identify three cache-based privacy attacks in CCN, and we design minimally invasive countermeasures that allow users to mark content as sensitive and prevent these attacks. While our discussion focusses on CCN, we believe that the overall class of attacks also applies to other NDN proposals that make use of ubiquitous caching.

2. BACKGROUND

A range of named data networking architectures have been proposed; for a survey, see Choi et al. [7]. Content-Centric Networking [12] is among the more recent approaches and has already generated a body of follow-up work by various research groups.

2.1 Content-Centric Networking

For the purpose of our discussion in this technical report, CCN consists of the following main features:

Named content: Each chunk of data transmitted in the network is identified by a globally unique name. Names address content (including its hash), but not its location. Names consist of one or more components similar to a URL, but without any protocol, port, or host name specification. Large objects of content may be split into smaller chunks, each with an individual name, but a common prefix.

Pull-based communication model: CCN specifies two types of network messages: Interest and Data messages. In order to obtain a content object with a given name, a network entity sends an Interest message. In response to an Interest, the network delivers at most one Data message.

Prefix matching: Any Data message delivered must match the preceding Interest. A Data message matches an Interest if the name in the Interest is a prefix of (or identical to) the name in the Data message.

Router functionality: Any router (or other network device) may contain a cache for content objects. When an incoming Interest is received, a router first checks whether a matching object is cached locally. In case of a cache miss, the Interest can be forwarded to adjacent routers. When forwarding Interests, routers keep transient state to identify the interface for backward forwarding of incoming Data responses.

Content security: Each content object carries its hash value as part of its name in order to ensure data integrity. Data authenticity is achieved by digitally signing content objects. Confidentiality and access control are implemented by encrypting content objects and distributing the keys.

2.2 Caches

With respect to caching, we should distinguish two different kinds of policies: Caching policies and replacement policies.

A *caching policy* [14] reasons about whether an object should be considered for caching. For instance, such a policy could attempt to exclude objects that are unlikely to be shared between users by not caching any interactive traffic. A policy that coordinates caching decisions between different caches in a topology to avoid overlap also falls into this category. The default policy is to cache everything.

A *replacement policy* decides which object should be evicted from the otherwise full cache to accommodate a new object. Traditional policies are FIFO, LRU, LFU, or random policies such as the ones discussed in [18]. As of version 0.6.0, the cache in PARC's CCNx prototype[1] implements an approximated FIFO policy that periodically eliminates the oldest objects when the cache is above capacity.

3. ATTACK MODEL

The network setup that we consider in this technical report is composed of an arbitrary topology of caches with a shared gateway cache at the edge of the network. Such a gateway cache may correspond to a DSLAM, for instance, and has around $100 - 1000$ end users directly connected to it. This cache is the users' only active Internet access point, and users do not share with each other any local caches that they might have in their home networks. That is, all non-local requests are routed through the gateway cache. In the following, we will abstract from the rest of the network and consider only this gateway cache.

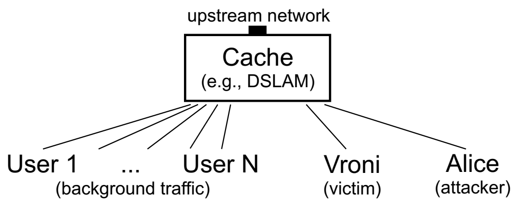


Figure 1: Attack model: The attacker is an unprivileged user of the same cache as the victim.

We assume that the gateway cache works independently (it does not explicitly cooperate with other caches located upstream or downstream) and its caching policy allows it to cache any type of data. As cache replacement policies, we consider FIFO, LRU, and random replacement. Since the replacement policy needs to operate at line speed, more complex replacement policies or policies requiring more state (such as LFU) may not be applicable [3, 18].

In our scenario, both the attacker *Alice* and the victim *Vroni* are connected to the same gateway cache (see Figure 1). Note that Alice is a regular user of the cache. Attacks that require privileged administrative access to infrastructure, such as backbone network cables and routers, are out of the scope of this technical report.

4. REQUEST MONITORING ATTACK

The request monitoring attack leverages caches to monitor access to predefined objects. The attack is based on two main assumptions: (1) Vroni requests a low-popularity and privacy-sensitive object O at time t_o.[2] This results in a cached copy of O, and (2) Alice can find out that O is cached. The goal of the attack is for Alice to find out *if*, *when* and *how often* there is an access to O through the cache.

As in other privacy attacks [4], low-popularity objects imply a higher privacy risk, and some auxiliary information is needed to carry out the attack: Alice uses prior knowledge about Vroni to define the name of an object O (or a list of objects) that Vroni might request. If Alice detects an access to O, the low popularity of O implies a high probability that the request was made by Vroni and not by another user of the cache.

For example, Alice might already know that Vroni is the only person in the neighbourhood who speaks Russian, but Alice does not know Vroni's political convictions. Alice can compile a list of objects found on Russian web sites and classify them according to the political spectrum. Every time Alice observes a local access to these objects, she can reasonably assume that Vroni requested the object. By

[1]PARC CCNx, available at `http://www.ccnx.org/`.

[2]Note that we assume only a *single* access to O. While in theory, object access could be modelled as a frequency f_o, our single access assumption acknowledges the fact that f_o might be very low in practice, such as one request per day or per week.

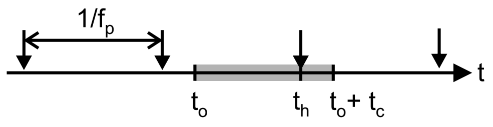


Figure 2: Noninvasive cache probing: The attacker checks the cache with frequency f_p (downward arrows). The victim accesses object O at time t_o; the area filled in grey denotes the time during which O is cached. The attacker detects O at t_h.

comparing with her list of classified objects, Alice can infer Vronis political preferences.

In another scenario, Alice knows that Vroni is the neighbourhood's only customer of a given bank. By continuously monitoring access to an object found on the bank's web site, Alice can detect when Vroni does online banking. Alice can use this information for social engineering attacks. For instance, Alice might call Vroni while she is logged into her online banking account to tell her that the bank detected a security problem when Vroni logged in, and that Vroni should reveal her password so that the problem can be fixed.

Alternatively, Alice might know that in the neighbourhood, Vroni is the only player of a certain online role-playing game. If Alice detects that Vroni accesses the game very frequently, she might conclude that Vroni has an addiction problem.

In this section, we assume that Alice has already used prior knowledge to select O. In Section 5, we describe an attack that allows Alice to discover what types of objects are transiting through her cache, which is useful if she has less knowledge about its user population.

4.1 Non-Invasive Cache Probing

In order to explain the principles and requirements of cache probing, we initially make the simplifying assumption of *non-invasive* probing. That is, Alice can send a request to the cache and find out whether an object is cached without modifying the state of the cache.

To find out when a user of the cache requests O, Alice sends periodic requests, so-called probes, to the cache and records whether O was found in the cache (as shown in Figure 2). To carry out the attack, Alice needs to know how often to send probes, the probing frequency f_p.

If Alice knows that O remains cached for at least t_c time units after being requested by a user, t_c implies a lower bound on the probing frequency $f_p \geq 1/t_c$ to achieve a 100 % detection rate. In this case, Alice detects an access within one probing cycle. That is, if a cache hit has been detected at time t_h, Alice concludes that the actual access time t_o was at most $1/f_p$ earlier.

Whether there is a lower bound on the cache time t_c depends on the replacement policy of the cache. LRU has such a lower bound, the *characteristic time* [5]: Under stationary object popularity distribution and constant incoming request rate, each object remains in the cache for approximately the same time after the last cache hit. In fact, LRU can be modelled as a FIFO queue where a cached object is removed and reinserted at each cache hit. The characteristic time is simply the time it takes for an object to propagate from the tail to the head of the list, given there are no hits to the object. We extend the definition of the characteristic time to FIFO and random replacement policies: Both policies are insensitive to cache hits; their characteristic time is the expected lifetime of an object in the cache.

Alice can measure t_c by artificially inserting an object into the cache and observing for how long it remains cached (see Figure 3(a) and Algorithm 1): She selects a "fresh" object that is currently not cached and that no one but her will request, and inserts the object into the cache. Alice then repeatedly queries the cache to find out whether the object is still cached and updates the estimated values for the lower and upper bound of the cache's characteristic time accordingly. For the measurement, the request frequency f_m depends on the desired precision of the estimation, and on Alice's (approximate) prior knowledge of the order of magnitude of t_c. If Alice has no prior knowledge of the order of magnitude of t_c, she can start with, say, one request every ten seconds and adjust the frequency if the estimated bounds of t_c are not satisfactory.

Algorithm 1 Measurement of t_c (Non-Invasive)

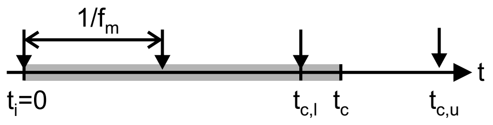
```
 1: $o$ = /exclusive/unused/object          ▷ object name
 2: request_with_insert($o$)                ▷ insert object into cache
 3: $t_i$ = current_timestamp()             ▷ insertion time
 4: $t_{c,l} = 0$                           ▷ lower bound on $t_c$
 5: $t_{c,u} = \infty$                      ▷ upper bound on $t_c$

 6: loop
 7:     if request_from_cache($o$) == CACHE_HIT then
 8:         $t_{c,l}$ = current_timestamp() $- t_i$
 9:         sleep for $1/f_m$ seconds
10:     else
11:         $t_{c,u}$ = current_timestamp() $- t_i$
12:         break
13:     end if
14: end loop
15: return $[t_{c,l};\ t_{c,u}]$
```

In practice, t_c is a probabilistic value, and it evolves over time as the object popularity and request rate vary. To accommodate this, Alice can repeat her measurement periodically and possibly scale down $t_{c,l}$ (or scale up $t_{c,u}$) so that the probability of an object being evicted from the cache before $t_{c,l}$ (or after $t_{c,u}$) is negligible. Even if the cache's replacement policy has no clearly defined t_c, Alice can choose lower and upper bounds that are "safe" for most cases (at the cost of higher f_p).

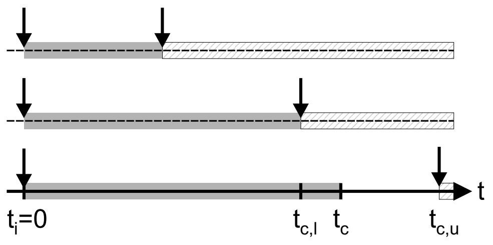
To estimate a lower bound for the characteristic time in the worst case, assume that a LRU or FIFO cache has a capacity of N_c objects and receives aggregate requests for N_r objects per second, but all without any cache hit. In this case, the performance of LRU degrades to FIFO and the characteristic time is $t_c = N_c/N_r$ seconds. (If there are cache hits, the characteristic time of a LRU cache will be higher than this lower bound because the cache's object eviction rate decreases.) For instance, for a cache size of 1 GB and an incoming upstream bandwidth of 100 Mbit/s, the characteristic time is at least $\frac{1 \cdot 2^{30} \cdot 8}{100 \cdot 10^6}\, s \approx 85.9\, s$. Consequently, the minimum probing frequency $f_p \approx 0.012$ Hz is reasonably low and the attack is feasible.

Non-invasive cache probing is possible only under certain circumstances. In practice, Alice's requests might modify the state of the cache in two ways:

- In case of a *cache miss*, the Interest will be forwarded to



(a) The attacker inserts an object at time $t_i = 0$, checks its availability with frequency f_m, and obtains the lower and upper bounds $t_{c,l}$ and $t_{c,u}$.



(b) The attacker inserts three objects at time $t_i = 0$. In each iteration, she checks the availability of a different object. The effects of the invasive probing (cache lifetime extension or insertion of the object, as in LRU) are shown in the light grey shaded area.

Figure 3: Measuring the characteristic time t_c for (a) noninvasive and (b) invasive probing. For clarity of presentation, we set $t_i = 0$ in this figure.

other routers. If a Data response is received, the data might be inserted into the cache. To avoid this, Alice may use the *scope* field in CCN's Interest messages to prevent an Interest from being forwarded to another router.

- In case of a *cache hit*, depending on the replacement policy, the lifetime of the requested object in the cache might be affected. While FIFO and random replacement are insensitive to cache hits, LRU would (conceptually) reinsert the requested object into the queue and thereby increase its lifetime in the cache by at least the characteristic time.

Consequently, local scope with FIFO (as currently implemented in the CCNx prototype) or random replacement allow non-invasive cache probing, but LRU does not.

4.2 Invasive Cache Probing

As alluded to above, the requirements for non-invasive cache probing might not be satisfied in a real-world deployment of CCN. Therefore, we now assume that every cache miss causes the object to be cached, and that the replacement policy is either FIFO or LRU. In order to successfully monitor access to objects, this scenario requires Alice to solve three challenges: (1) Decide whether a Data response received from the router corresponds to a cache hit or a cache miss, (2) estimate t_c, and (3) detect Vroni's accesses to O despite the possibility of a cache hit being due to Alice's own monitoring requests. In the following, we detail how the algorithms introduced in the previous section can be enhanced to fulfil these requirements.

4.2.1 Detect Cache Hit/Miss

If Alice cannot use the scope field in Interest messages, she can detect cache hits by exploiting the timing side channel: Since the shared cache is only one hop away, Alice can reasonably assume that under comparable local network congestion conditions, every response served by the cache is always faster than data fetched from the upstream network. Furthermore, the distance of only one hop means that the response time distribution of the cache can be expected to have low variance over short time windows. The same does not hold for the latency of objects that are not cached: They may be fetched from unknown locations in the upstream network and have an unpredictable delay. Therefore, Alice's approach is to measure the local cache's response time (or the response time distribution) and to classify a Data response as a cache hit if the response time is close enough to the cache's expected response time, or as a cache miss if the response time is too high.

Algorithm 2 provides a simplified version of the procedure that Alice can use to measure the local cache's response time t_r for cache hits. Alice picks an arbitrary object o and requests it to be sure that it is cached. She then requests o a second time, knowing that it will be served from the cache, and measures the response time t_r. By repeating this procedure, Alice can estimate the expectation and variance of the response time distribution. Since local traffic conditions may vary, Alice can keep measuring response times in regular intervals and keep a moving average and variance estimation. If the response time distribution was normally distributed with an estimated mean $\hat{\mu}$ and variance $\hat{\sigma}$, Alice could conclude that an object was not cached locally if $t_r > \hat{\mu} + 2 \cdot \hat{\sigma}$, for instance. The accuracy of this approach depends on how precise Alice's estimates of the mean and variance are for the instantaneous traffic conditions, and how large the delay is between the local router and the location in the upstream network from which a content object is served in case of a cache miss. Therefore, if the local router's queue length varies quickly and if most cache misses are served by caches located close by in the upstream network, Alice needs to increase the measurement frequency, whereas in the opposite case, she can measure the response times less frequently.

Algorithm 2 Measurement of t_r (Invasive)
```
1: $o$ = <arbitrary object name>
2: request($o$)      ▷ insert object into cache (blocking call)
3: $t$ = current_timestamp()
4: request($o$)      ▷ request object from cache (blocking call)
5: return current_timestamp() − $t$
```

4.2.2 Measure Characteristic Time

For the measurement of the characteristic time in the invasive case, Algorithm 1 still applies if the replacement policy is FIFO because a cache hit does not extend the lifetime of the object in the cache. The only required modification to the algorithm is to use the procedures of Section 4.2.1 to distinguish cache hits and misses.

If the replacement policy is LRU, a cache hit extends the lifetime of the object in the cache and Algorithm 1 does not converge. Therefore, every time a cache hit is detected, Alice needs to assume that the object has been reinserted into the cache, reset the time t_i to the current time, and increase the waiting time between iterations to $1/f_m \cdot j$, where j is the number of the current iteration. The drawback of this approach is a potentially long delay until the measurement terminates.

Alternatively, Alice can parallelise the algorithm by initially inserting a large number of single-use objects (Figure 3(b) and Algorithm 3). In each iteration, she checks for a cache hit/miss of a different object, which bypasses the effects of her invasive requests. For simplicity, we assume that Alice has access to a remote machine that can generate and serve arbitrary objects under a name prefix that Alice controls. Alternatively, Alice could compile a list of existing objects that are very unlikely to be cached or requested.

Algorithm 3 Measurement of t_c (Invasive, LRU)

```
1: $N$ = MAX_ITERATIONS
2: $o(j)$ = /alice/generate/<j>          ▷ object name template
3: for $j = 1;\ j \le N;\ j = j + 1$ do
4:     request($o(j)$)                    ▷ insert object into cache
5:     $t_{i,j}$ = current_timestamp()     ▷ insertion time
6: end for
7: $t_{c,l} = 0$                          ▷ lower bound on $t_c$
8: $t_{c,u} = \infty$                     ▷ upper bound on $t_c$

9: for $j = 1;\ j \le N;\ j = j + 1$ do
10:    if request($o(j)$) == CACHE_HIT then
11:        $t_{c,l}$ = current_timestamp() $- t_{i,j}$
12:        sleep for $1/f_m$ seconds
13:    else
14:        $t_{c,u}$ = current_timestamp() $- t_{i,j}$
15:        break
16:    end if
17: end for
18: return $[t_{c,l};\ t_{c,u}]$
```

4.2.3 Monitor Access to O

With invasive probing, every cache miss causes the requested object to be cached, and in the case of LRU, a cache hit extends the lifetime of the object in the cache. Consequently, Alice must be careful not to measure state changes that she caused herself and incorrectly attribute them to Vroni. In order not to detect a cache hit caused by one of her previous probes, Alice must choose $f_p < 1/t_c$. However, such a probing frequency implies that a request made by Vroni can go unnoticed if it happens shortly after Alice's probe.

CCN splits large objects into smaller chunks (the default chunk size is 4 KB). If O is larger than 4 KB, Vroni will request all the chunks at approximately the same time. Consequently, Alice can parallelise her probing algorithm and use a different chunk in each iteration. This trivially guarantees Alice that she does not measure hits due to her own cache insertions. Furthermore, a cache insertion caused by Alice vanishes after at most $t_{c,u}$ time units if there is no request by Vroni. Therefore, Alice knows that she can *reuse* a chunk for probing in future iterations $t_{c,u}$ time units after each request.

To detect Vroni's requests, Alice needs to make sure that at least one probing chunk is not cached during each $1/f_p$ time interval. This implies that Alice needs $m = \lceil t_{c,u} \cdot f_p \rceil + 1$ chunks to carry out the attack for a FIFO replacement policy. For LRU, $m - 1$ chunks are sufficient because Alice can detect lifetime extensions due to a cache hit by Vroni. Note that in the parallelised case, $f_p \ge 1/t_{c,l}$, where $t_{c,l}$ is Alice's estimation of a lower bound of t_c, in order to achieve 100 % detection precision. For instance, if the characteristic time $t_c = 85.9$ s measured with ±10 % error, $t_{c,l} = 77.3$ s, $t_{c,u} = 94.5$ s, $f_p = 0.013$ Hz and $m = 3$ chunks.

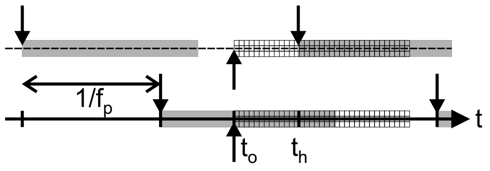


Figure 4: Parallel invasive probing with LRU: The attacker alternates probes between two chunks with frequency f_p (downward arrows). The victim accesses object O at time t_o and the attacker detects a cache hit at time t_h. The area filled in grey denotes the time during which a chunk is cached due to the attacker's probes; the shaded area is caching time due to the victim's request.

Figure 4 and Algorithm 4 illustrate this attack for LRU using $m-1$ chunks: Alice uses the procedures of Sections 4.2.1 and 4.2.2 to detect cache hits and to obtain $t_{c,l}$ and $t_{c,u}$; she then uses these values to compute f_p and m, and compiles a list of chunks $c(0)$, $c(1)$, ..., $c(m-1)$ of the same object O. For the actual probing, Alice cycles through the chunks. If a cache hit is detected, Alice concludes that someone other than herself requested O during the past $1/f_p$ seconds.

Algorithm 4 Parallel Cache Probing (Invasive, LRU)

```
1: $N$ = MAX_ITERATIONS
2: $c(j)$ = /object/chunk/<j>       ▷ chunk name template

3: for $j = 1;\ j \le N;\ j = j + 1$ do
4:     if request($c(j \mod (m-1))$) == CACHE_HIT then
5:         "someone requested $O$"
6:     else
7:         "no one requested $O$"
8:     end if
9:     sleep for $1/f_p$ seconds
10: end for
```

For FIFO, Algorithm 4 can be used with m chunks instead of $m - 1$. The probabilistic nature of a random replacement policy can be handled by increasing f_p. Other replacement policies might incur a significant cost for the attack (in terms of high f_p and a large number of potentially single-use chunks). However, Arianfar et al. [3] argue that only simple policies such as FIFO, LRU and random replacement can be implemented at router line speed.

4.3 Countermeasures

The cache monitoring attack is based on the assumption that an attacker can *predict the names* of privacy sensitive content objects and *detect cache hits* for these objects with *reasonable effort.* From the fact that an object is cached, the attacker concludes that a user of the cache has (recently) requested the object.

Object names could be made unpredictable by using one-time names, by tunnelling requests through a TOR-like system such as Andana [8], for instance. However, generalised use of such a solution would result in increased network traffic because caching is ineffective while data is tunnelled, and transmission delays would increase because the data is typically not delivered on the shortest path.

In order to prevent the detection of cache hits, the scope field in Interest messages could be disabled, and the timing side channel could be made more difficult to exploit by serving each cached object with an artificial delay equal to the delay observed when the object was fetched initially. However, these measures would reduce the functionality of CCN, increase delays, and would make attacks only more difficult, but not impossible.

Other means to increase the cost of attacks would be to use less predictable replacement policies or to place caches only at higher aggregation levels where it is much more difficult for an attacker to make inferences about an individual user. These measures, too, involve a potential reduction of efficiency.

At first, the preceding arguments appear to suggest that there is a trade-off between privacy and performance: Each aforementioned countermeasure would achieve an increase in privacy at the price of a performance loss such as increased response times, lower network efficiency, or reduced protocol functionality. Yet, all these countermeasures would indifferently affect *all* traffic, independent of its privacy requirements. A better way to implement countermeasures is to target only privacy-sensitive traffic without restricting the non-sensitive majority of the traffic.

Intuitively, privacy-sensitive objects must be (locally) unpopular at the time of the request: As an object becomes more popular, its "entropy" decreases in that revealing its existence in a cache leaks less potentially compromising information. Furthermore, it might be difficult to attribute a popular object to an individual user, given that several users might potentially have expressed an interest in the object. Therefore, countermeasures against privacy leaks may focus on locally unpopular objects as a superset of the privacy-sensitive objects.

From a performance point of view, it is usually undesirable to cache (locally) unpopular objects because cache hits are unlikely. Ideally, a caching policy should exclude locally unpopular content from being cached. Such a policy would improve both privacy *and* performance. However, designing a caching policy that can give privacy guarantees (and ensure that a sensitive, unpopular object is indeed *never* cached) is a challenging task. For instance, an object requested 100 times by the same user might be privacy-sensitive, while it is probably less sensitive if requested once by 100 different users of the same cache. Furthermore, it must be difficult for an attacker to establish a fake popularity for a privacy-sensitive object: If an artificially generated popularity allows the object to be cached, the attacker might still be able to carry out the request monitoring attack.

Until such a caching policy is available, privacy-sensitive content could be "flagged" to prevent it from being cached. While such a flag could be set by the content publisher, the potential privacy issues tend to arise on the receiver side; therefore, it seems more adequate to leave this decision to the receiver. End hosts could, for instance, selectively route flagged Interests (and receive the corresponding data) through a tunnel such as Andana.

An alternative approach that applies if end users trust their ISP (and do not mind letting their ISP know what content they consider sensitive) may be to optionally specify a Nonce in Interest messages. A Nonce is a number chosen by the user so that is not easily predictable by the attacker, e.g. a random number with sufficient entropy. If set, routers should use this Nonce to isolate Interests: Interests for the same name, but with different Nonce are forwarded and cached independently. Data requested with the Nonce field set should be cached only for very short time periods or not at all; a cache hit may occur only when the Nonce matches.[3][4] This would allow a user to define for each object whether it is considered privacy sensitive and should be retrieved privately. All other objects achieve the usual performance.

The challenge for the latter two approaches is to educate users about the need for privacy protection, and to provide them with an easy-to-use and reliable mechanism to flag Interests. Flagging too many Interests degrades network performance, while flagging too few Interests might lead to privacy attacks.

A more elaborate discussion of potential countermeasures against request monitoring can be found in [16].

5. OBJECT DISCOVERY ATTACK

The previous attack allowed Alice to monitor access to an object with good time precision, but required prior knowledge of the name to be monitored. In this section, we describe an attack that allows Alice to inspect the contents of a cache and discover the cached objects.

Alice uses two specific features of CCN—prefix matching and exclusion patterns in Interest messages. Prefix matching allows Alice to discover a new object with a name that she has not previously known. Exclusion patterns allow Alice to force the cache to return an item that she has not yet seen. Taken in combination, these two characteristics permit Alice to recursively enumerate the contents of a cache: First, Alice requests data for the prefix of the name subspace to be explored, e.g. the root prefix `/`; the cache will reply with an arbitrary data object that satisfies this constraint. In the following iterations, Alice puts the name of the discovered item on the exclusion list and sends the next request.

Algorithm 5 contains an implementation of this approach. The procedure ENUMERATE-LEVEL discovers all the name components in the cache that are directly adjacent to the given prefix. For instance, if the cache contains `/my/first/object`, `/my/second/object` and `/something/unrelated`, then the procedure call ENUMERATE-LEVEL(`/my/`) will return the set {`first`, `second`}. The procedure ENUMERATE-SUBSPACE uses this functionality to discover all objects in the name hierarchy subtree defined by the prefix p.

The above algorithm operates on chunks. The algorithm can be enhanced (using CCN's naming conventions) to fetch only one chunk per object. By operating on content objects instead of chunks, the overhead could be reduced significantly. However, Alice would still need to fully download one chunk (4 KB plus headers) of each discovered data object. This limits the speed at which she can send new requests. Furthermore, the contents of the cache evolve continuously as new objects are inserted and old objects are evicted. Therefore, it is unrealistic to obtain a consistent snapshot of a large

[3]Note that this use of the Nonce field is different from what the CCNx specification envisions.

[4]This mechanism could be abused by attackers to fill up router caches with many copies of the same object by requesting it under different Nonces. However, attackers can also cache useless information or send large numbers of Interests to upstream caches by simply requesting objects having different names. Therefore, this use of the Nonce field would not increase CCN's attack surface. Furthermore, processing the Nonce could be restricted to routers close to end users and be disabled deeper in the core network.

Algorithm 5 Object Discovery

```
 1: procedure ENUMERATE-LEVEL($p$)                    ▷ name prefix $p$
 2:         ▷ returns name components directly after prefix
 3:     assert $p = /p_0/p_1/\dots/p_i/$      ▷ enumerate level $i+1$
 4:     $E = \emptyset$                      ▷ name component exclusion set
 5:     loop
 6:         $r = \text{request}(p, E)$            ▷ with prefix matching and
 7:                             ▷ name component exclusion at level $i+1$
 8:         if r.status == CACHE_MISS then
 9:             return $E$
10:         else
11:             $n = \text{r.name}$ ▷ found object satisfying $p$ and $E$
12:             assert $n = /n_0/n_1/\dots/n_j$ with $j \geq i$
13:             assert $p_0 = n_0,\ p_1 = n_1,\ \dots,\ p_i = n_i$
14:             if $j > i$ then
15:                 assert $\forall e \in E : n_{i+1} \neq e$
16:                 $E = E \cup \{n_{i+1}\}$
17:             else
18:                 return $E$    ▷ $p$ is no prefix but full name
19:             end if
20:         end if
21:     end loop
22: end procedure

23: procedure ENUMERATE-SUBSPACE($p$)   ▷ name prefix $p$
24:     ▷ returns subtree with object names and all prefixes
25:     $E = \text{ENUMERATE-LEVEL}(p)$
26:     $F = \emptyset$                           ▷ full names in subspace
27:     for all $e \in E$ do
28:         $f = p/e/$        ▷ append last component to prefix
29:         $F = F \cup \{f\}$
30:         $F = F \cup \text{ENUMERATE-SUBSPACE}(f)$
31:     end for
32:     return $F$
33: end procedure
```

cache. However, the attack can be used to quickly explore smaller name subspaces by starting the algorithm with a longer name prefix. For instance, instead of enumerating the entire cache ($p =$ `/`), Alice could restrict the enumeration to the much smaller subspace $p =$ `/savings-bank/boston/`. Alternatively, Alice can use just the procedure ENUMERATE-LEVEL($p =$ `/`) at the root level to gain an overview of which protocols and services are currently being used, and recursively enumerate only those subspaces that look interesting to her. The output of this step can be used to prepare a more targeted attack such as the request monitoring attack.

A similar version of this "attack" is implemented in the tool `ccnls` included in the CCNx distribution. We have verified in the current version 0.6.0 of CCNx that prefix matching and name exclusion lists are indeed enabled for in-memory caches and not only for persistent content repositories.[5] Because of the privacy implications of this attack, the CCN primitives used by the tool `ccnls` should be disabled in real-world deployments: Prefix matching should be restricted to making forwarding decisions, and not be applied when matching cached objects (except for the last, implicit component of the name, the content hash). Name exclusion lists in Interests should not be honoured by caches, or should be closely monitored: The attack can potentially be detected because the algorithm tends to generate long name exclusion lists.

[5]This behaviour can be verified by requesting an object from a repository, shutting down the repository, and using `ccnpeek` with a prefix of the object's name to fetch it from the cache.

6. DATA FLOW CLONING ATTACK

Some applications, such as Voice-over-CCN [11], exchange interactive data flows on top of CCN. Since the objects of the data flow are cached by default, Alice can attempt to replicate and reconstruct the original data flow. Even if the data itself is encrypted, side channels such as message sizes and timing might leak information that can be exploited. For instance, Wright et al. detected phrases in encrypted VoIP calls [21] with varying packet sizes, and Chen et al. extracted medical conditions, search terms and tax information from HTTPS-protected Web browsing sessions [6].

The Voice-over-CCN prototype uses the naming scheme `/domain/user/call-id/rtp/sequence-number` for the voice data exchanged during a call. Once Alice knows the prefix of a specific ongoing call instance, such as `/tid/vroni/1234/rtp/` detected with an object discovery attack, she can predict the names of future packets, request them in the same way as Vroni, and obtains a copy of the Data object sequence.

If a flow's content names (and sequence numbers) are encrypted following the scheme `/routing-prefix/{encrypted}`, Alice can use the object discovery attack to enumerate the cached name subspace under `/routing-prefix/` and request a *new* object at least as fast as Vroni. While this technique allows Alice to receive the Data objects in the same order (and nearly at the same time) as Vroni, the absence of clear-text sequence numbers means that Alice cannot detect out-of-order transmissions. If the same routing prefix is shared by several ongoing data flows to different users of the cache, it will be very difficult (or impossible) for Alice to tell the flows apart. Consequently, in addition to the countermeasures discussed in the previous sections, interactive data flows should encrypt as much of the name as possible.

7. RELATED WORK

The request monitoring attack on CCN was inspired by DNS snooping [10], a similar attack on DNS that probes DNS resolvers for cached domain name mappings. This attack was later refined [20, 1, 17] to infer the access rate to domain names from the time during which a name is not cached. Krishnan and Monrose [13] showed how to exploit the DNS prefetching features of modern Web browsers. By probing DNS caches, they can detect Web searches for a given keyword with an accuracy of 85 %.

The scope of the CCN request monitoring attack is larger than attacks based on DNS because the attack can potentially affect any protocol on top of CCN, not just domain name to IP address mappings. Furthermore, CCN caches can be located in arbitrary network devices, meaning that in practice a CCN cache might be shared by fewer users than a DNS cache, and that attackers can extract more precise information about individual users. Another difference to DNS is that the entries in a DNS cache are rather small and tend to be evicted due to expiry. In contrast, CCN caches typically contain larger objects that are evicted due to cache replacement, which makes it necessary to estimate the cache's characteristic time. Lastly, the DNS attacks all make use of an iterative query mode similar to non-invasive probing in CCN. In this technical report, we provide additional attack algorithms for request monitoring in the case where non-invasive probing is disabled.

Felten and Schneider [9] showed how malicious Web sites can probe client-side web browser caches to find out if the user has visited another, unrelated web site. In contrast to cache probing using DNS or CCN, this attack requires interaction by the user (i.e., visiting the malicious web site).

In the context of CCN, initial security-related research was concerned with content authenticity (and implicitly the prevention of cache poisoning) by digitally signing the binding between names and content [12, 19]. Further work considered "privileged" adversaries such as governments or ISPs that compromise their users' privacy: Andana [8] is a TOR-like onion routing service that can provide users with anonymity. Arianfar et al. [2] described a mechanism against censorship that provides users with plausible deniability. In this technical report, we consider a different type of adversary, that is, an unprivileged user spying on her neighbours. This setting allows for more "lightweight" solutions (see Section 4.3), of which a system such as Andana can be a part.

8. CONCLUSION

In this technical report, we have described a range of high-level attack concepts that exploit the ubiquitous caches in CCN. While application-level caches have been exploited for attacks in other systems, the scope of such attacks becomes much larger when all traffic transits through network-level caches in a named data networking architecture such as CCN.

Caches provide for lower latencies and reduce the required upstream bandwidth, but they also keep transient communication traces that can be exploited for attacks. Arguably, most privacy-sensitive objects have low (local) popularity and should not be subject to caching in order to improve the cache hit rate. Therefore, performance *and* privacy can be improved by caching policies that exclude low-popularity content. However, there are several challenges concerning the secure implementation of this approach in practice. In the meantime, a countermeasure against privacy attacks could be to allow users to "flag" content that they deem sensitive and to handle it separately from the other content.

Other attacks in the context of CCN (as well as an early description of the attacks from this paper) can be found in [15]; [16] contains more discussion about countermeasures against the request monitoring attack.

9. REFERENCES

[1] H. Akcan, T. Suel, and H. Brönnimann. Geographic web usage estimation by monitoring DNS caches. In S. Boll, C. Jones, E. Kansa, P. Kishor, M. Naaman, R. Purves, A. Scharl, and E. Wilde, editors, *LocWeb*, volume 300 of *ACM International Conference Proceeding Series*, pages 85–92. ACM, 2008.

[2] S. Arianfar, T. Koponen, B. Raghavan, and S. Shenker. On preserving privacy in content-oriented networks. In *ICN '11*, Aug. 2011.

[3] S. Arianfar, P. Nikander, and J. Ott. On content-centric router design and implications. In *ReARCH '10*. ACM, Nov. 2010.

[4] J. Calandrino, A. Kilzer, A. Narayanan, E. Felten, and V. Shmatikov. "You might also like:" Privacy risks of collaborative filtering. In *2011 Symposium on Security and Privacy*. IEEE, May 2011.

[5] H. Che, Y. Tung, and Z. Wang. Hierarchical Web caching systems: Modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications*, 20(7):1305 – 1314, Sept. 2002.

[6] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *2010 Symposium on Security and Privacy*. IEEE, May 2010.

[7] J. Choi, J. Han, E. Cho, T. Kwon, and Y. Choi. A survey on content-oriented networking for efficient content delivery. *IEEE Communications Magazine*, 49(3):121–127, 2011.

[8] S. DiBenedetto, P. Gasti, G. Tsudik, and E. Unzun. ANDaNA: Anonymous named data networking application. In *NDSS 2012*, Feb. 2012.

[9] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *CCS '00*. ACM, Nov. 2000.

[10] L. Grangeia. DNS cache snooping, Feb. 2004.

[11] V. Jacobson, D. K. Smetters, N. H. Briggs, M. F. Plass, P. Stewart, J. D. Thornton, and R. L. Braynard. VoCCN: Voice-over content-centric networks. In *ReArch '09*. ACM, Dec. 2009.

[12] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *CoNEXT '09*. ACM, Dec. 2009.

[13] S. Krishnan and F. Monrose. DNS prefetching and its privacy implications: When good things go bad. In *LEET '10*. Usenix, Apr. 2010.

[14] N. Laoutaris, S. Syntila, and I. Stavrakakis. Meta algorithms for hierarchical web caches. In *2004 IEEE International Conference on Performance, Computing, and Communications*, pages 445 – 452, 2004.

[15] T. Lauinger. Security & scalability of content-centric networking. Master's thesis, Eurécom, Sophia-Antipolis, France and Technische Universität Darmstadt, Germany, Sept. 2010.

[16] T. Lauinger, N. Laoutaris, P. Rodriguez, T. Strufe, E. Biersack, and E. Kirda. Privacy risks in named data networking: What is the cost of performance? Editorial note. *ACM SIGCOMM Comput. Commun. Rev.*, 42(5), Oct. 2012.

[17] M. A. Rajab, F. Monrose, A. Terzis, and N. Provos. Peeking through the cloud: DNS-based estimation and its applications. In S. M. Bellovin, R. Gennaro, A. D. Keromytis, and M. Yung, editors, *ACNS*, volume 5037 of *LNCS*, pages 21–38. Springer Verlag, 2008.

[18] D. Rossi and G. Rossini. Caching performance of content centric networks under multi-path routing (and more). Technical report, Télécom ParisTech, July 2011.

[19] D. Smetters and V. Jacobson. Securing network content. Technical report, PARC, Oct. 2009.

[20] C. E. Wills, M. Mikhailov, and H. Shang. Inferring relative popularity of Internet applications by actively querying DNS caches. In *IMC '03*. ACM, 2003.

[21] C. V. Wright, L. Ballard, S. E. Coull, F. Monrose, and G. M. Masson. Spot me if you can: Uncovering spoken phrases in encrypted VoIP conversations. In *IEEE Symposium on Security and Privacy*, pages 35–49. IEEE Computer Society, 2008.