12-7-2008

Nymble: Blocking Misbehaving Users in Anonymizing Networks

Patrick P. Tsang
Dartmouth College

Apu Kapadia
MIT Lincoln Laboratory

Cory Cornelius
Dartmouth College

Sean W. Smith
Dartmouth College

Nymble: Blocking Misbehaving Users in Anonymizing Networks*†

Patrick P. Tsang‡, Apu Kapadia§, Cory Cornelius¶ and Sean W. Smith‖

Dartmouth Computer Science Technical Report
TR2008-637

Dec 7, 2008

Abstract

Anonymizing networks such as Tor allow users to access Internet services privately by using a series of routers to hide the client's IP address from the server. The success of such networks, however, has been limited by users employing this anonymity for abusive purposes such as defacing popular websites. Website administrators routinely rely on IP-address blocking for disabling access to misbehaving users, but blocking IP addresses is not practical if the abuser routes through an anonymizing network. As a result, administrators block *all* known exit nodes of anonymizing networks, denying anonymous access to misbehaving and behaving users alike. To address this problem, we present Nymble, a system in which servers can "blacklist" misbehaving users, thereby *blocking users without compromising their anonymity*. Our system is thus agnostic to different servers' definitions of misbehavior — servers can blacklist users for whatever reason, and the privacy of blacklisted users is maintained.

Keywords: anonymous blacklisting, privacy, revocation

*This research was supported in part by the NSF, under grant CNS-0524695, and the Bureau of Justice Assistance, under grant 2005-DD-BX-1091. The views and conclusions do not necessarily reflect the views of the sponsors.

†Nymble first appeared in a PET '07 paper [JKTS07]. This paper presents a significantly improved construction and a complete rewrite and evaluation of our (open-source) implementation.

‡Department of Computer Science, Dartmouth College, USA. E-mail: patrick@cs.dartmouth.edu

§MIT Lincoln Laboratory, USA. E-mail: akapadia@ll.mit.edu. This research was performed while he was at Dartmouth College.

¶Department of Computer Science, Dartmouth College, USA. E-mail: Cory.T.Cornelius@Dartmouth.EDU

‖Department of Computer Science, Dartmouth College, USA. E-mail: sws@cs.dartmouth.edu

Contents

1 Introduction **4**
- 1.1 Our solution . . . 5
- 1.2 Contributions of this paper . . . 5

2 An Overview to Nymble **6**
- 2.1 Resource-based blocking . . . 6
- 2.2 The Pseudonym Manager . . . 7
- 2.3 The Nymble Manager . . . 7
- 2.4 Time . . . 7
- 2.5 Blacklisting a user . . . 8
- 2.6 Notifying the user of blacklist status . . . 8

3 Security Model **9**
- 3.1 Goals and threats . . . 9
- 3.2 Trust assumptions . . . 10

4 Preliminaries **10**
- 4.1 Notation . . . 10
- 4.2 Cryptographic primitives . . . 10
- 4.3 Data structures . . . 11
- 4.4 Communication channels . . . 16

5 Our Nymble Construction **16**
- 5.1 System setup . . . 16
- 5.2 Server registration . . . 17
- 5.3 User registration . . . 18
- 5.4 Credential acquisition . . . 18
- 5.5 Nymble-connection establishment . . . 19
- 5.6 Service provision and access logging . . . 21
- 5.7 Auditing and filing for complaints . . . 21
- 5.8 Blacklist update . . . 22
- 5.9 Periodic update . . . 23

6 Evaluation **24**
- 6.1 Security . . . 24
- 6.2 Performance . . . 24

7 Security Formalism **28**
- 7.1 Oracles . . . 28
- 7.2 Game-Accountability . . . 30
- 7.3 Game-Non-Frameability . . . 32
- 7.4 Game-Anonymity . . . 33

8 Security Analysis **34**
8.1 Accountability . . . 34
8.2 Non-Frameability . . . 35
8.3 Anonymity . . . 35
8.4 Across multiple linkability windows . . . 37

9 Discussion **37**

10 Conclusions **38**

List of Figures

1 The Nymble system architecture . . . 6
2 The life cycle of a misbehaving user . . . 8
3 Nymble's matrix of trust assumptions . . . 10
4 A summary of all the data structures in Nymble . . . 12
5 Evolution of seeds and nymbles . . . 13
6 A chain of daisies . . . 16
7 Different types of channels utilized in Nymble . . . 18
8 The *Nymble-connection Establishment* protocol . . . 20
9 The marshaled size of various Nymble data structures . . . 26
10 Nymble's performance at the NM, the user and the server . . . 27
11 The universe of all honest registered users . . . 33

1 Introduction

Anonymizing networks such as Crowds [RR98] and Tor [DMS04] route traffic through independent nodes in separate administrative domains to hide a client's IP address. Unfortunately, some users have misused such networks — under the cover of anonymity, users have repeatedly defaced popular websites such as Wikipedia. Since website administrators cannot blacklist individual malicious users' IP addresses, they blacklist the *entire* anonymizing network. Such measures eliminate malicious activity through anonymizing networks at the cost of denying anonymous access to behaving users. In other words, a few "bad apples" can spoil the fun for all. (This has happened repeatedly with Tor.[1])

There are several solutions to this problem, each providing some degree of accountability. In *pseudonymous credential systems* [Cha90, Dam88, HS06, LRSW99], users are required to log into websites using pseudonyms, which can be added to a blacklist if a user misbehaves. Unfortunately, this approach results in *pseudonymity for all users*, and weakens the anonymity provided by the underlying anonymizing network.

Anonymous credential systems such as Camenisch and Lysyanskaya's systems [CL01, CL04] use *group signatures* for anonymous authentication Basic group signatures [ACJT00, BSZ05, CvH91] allow servers to revoke a misbehaving user's anonymity by *complaining* to a *group manager*. In these schemes, servers must query the group manager for every authentication, and this lack of scalability makes it unsuitable for our goals. *Traceable signatures* [KTY04, vABHO06] allow the group manager to release a trapdoor that allows *all* signatures generated by a particular user to be traced; such an approach does not provide the *backward unlinkability* [NF05] that we desire, where a user's accesses before the complaint remain anonymous. Backward unlinkability allows for what we call *subjective blacklisting*, where servers can blacklist users for whatever reason since the privacy of the blacklisted user is not at risk. In contrast, approaches without backward unlinkability need to pay careful attention to when and why a user must have all their connections linked, and users must also worry about whether their (mis)behaviors will be judged fairly.

Subjective blacklisting is also better suited to servers such as Wikipedia, where misbehaviors such as questionable edits to a webpage, are hard to define in mathematical terms. In some systems, misbehavior can indeed be defined precisely. For instance, double-spending of an "e-coin" is considered a misbehavior in anonymous e-cash systems [Bra93, Cha82], following which the offending user is deanonymized. Unfortunately, such systems work for only narrow definitions of misbehavior — it is difficult to map more complex notions of misbehavior onto "double spending" or related approaches [TFS04].

With *dynamic accumulators* [CL02, Ngu05, TX03], a revocation operation results in a new accumulator and public parameters for the group, and *all other existing users' credentials must be updated*, and it is thus difficult to manage in practical settings. *Verifier-local revocation (VLR)* [BS01, AST02, BS04] fixes this shortcoming by requiring the server ("verifier") to perform only local updates during revocation. Unfortunately, VLR requires heavy computation at the server that is linear in the size of the blacklist. For example, for a blacklist with 1,000 entries, each authentication would take tens of seconds,[2] a prohibitive cost in practice. In contrast, our scheme takes the server about one millisecond per authentication, which is several thousand times faster

[1] The *Abuse FAQ for Tor Server Operators* lists several such examples at http://tor.eff.org/faq-abuse.html.en.

[2] In the construction due to Boneh and Shacham [BS04], computation at the server involves $3 + 2|BL|$ pairing operations, each of which takes tens of ms, according to a benchmark from: http://crypto.stanford.edu/pbc.

than VLR.

Lastly, any practical deployment of a credential scheme must address the *Sybil attack* [Dou02], where a malicious user can potentially assume multiple identities.

1.1 Our solution

We present a secure system called *Nymble*, which provides all the following properties: anonymous authentication, backward unlinkability, subjective blacklisting, fast authentication speeds, rate-limited anonymous connections, revocation auditability (where users can verify whether they have been blacklisted), and also addresses the Sybil attack to make its deployment practical.[3]

In Nymble, users acquire an ordered collection of *nymbles*, a special type of pseudonym, to connect to websites. Without additional information, these nymbles are computationally hard to link,[4] and hence using the stream of nymbles simulates anonymous access to services. Websites, however, can blacklist users by obtaining a *seed* for a particular nymble, allowing them to link future nymbles from the same user — those used before the complaint remain unlinkable. Servers can therefore blacklist anonymous users without knowledge of their IP addresses while allowing behaving users to connect anonymously. Our system ensures that users are aware of their blacklist status before they present a nymble, and disconnect immediately if they are blacklisted. Although our work applies to anonymizing networks in general, we consider Tor for purposes of exposition. In fact, any number of anonymizing networks can rely on the same Nymble system, blacklisting anonymous users regardless of their anonymizing network(s) of choice.

1.2 Contributions of this paper

Our research makes the following contributions:

- **Blacklisting anonymous users.** We provide a means by which servers can blacklist users of an anonymizing network while maintaining their privacy.
- **Practical performance.** A system such as ours will see widespread adoption only if its performance is acceptable at the server. Our protocol makes use of inexpensive symmetric cryptographic operations to significantly outperform the alternatives.
- **Open-source implementation.** With the goal of contributing a workable system, we have built an open-source implementation of Nymble, which is publicly available.[5] We provide performance statistics to show that our system is indeed practical.

A preliminary *work-in-progress* version of this paper (suggesting the use of trusted hardware) was presented at the Second Workshop on Advances in Trusted Computing [TKS06]. In our paper presented at the Privacy Enhancing Technologies Symposium [JKTS07], we eliminated the trusted hardware from our design, and outlined the initial version of our system. This paper represents a significantly revised system, including major enhancements and a thorough evaluation of our protocol, and an open-source release of our system for general use. Some of the authors of this paper

[3] We do not claim to solve the Sybil attack, which is a challenging problem, but we do provide a solution that represents a tradeoff between blacklistability and practicality.

[4] Two nymbles are *linked* if one can infer that they belong to the same user with probability better than random guessing.

[5] The Nymble Project. http://www.cs.dartmouth.edu/~nymble.

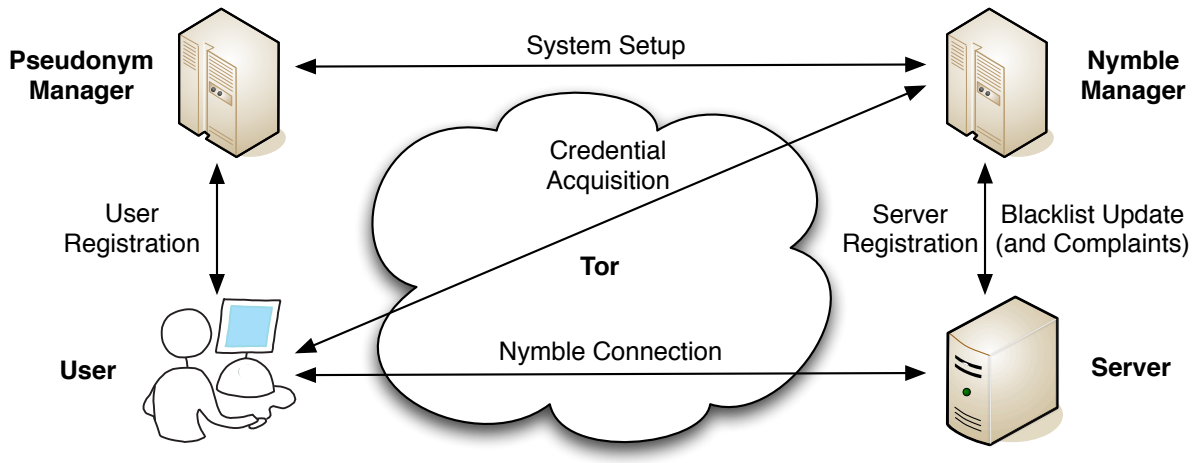


Figure 1: The Nymble system architecture showing the various modes of interaction. Users interact with the PM directly, and with the NM and servers though the anonymizing network. All interactions not involving the user take place directly.

have published two anonymous authentication schemes, BLAC [TAKS07] and PEREA [TAKS08], which eliminate the need for a trusted third party for revoking users. While BLAC and PEREA provide better privacy by eliminating the TTP, Nymble provides authentication rates that are several orders of magnitude faster than BLAC and PEREA (see Section 6). Nymble thus represents a practical solution for blocking misbehaving users of anonymizing networks.

2 An Overview to Nymble

We now present a high-level overview of the Nymble system, and defer the entire protocol description and security analysis to subsequent sections.

2.1 Resource-based blocking

Our system provides servers with a means to block misbehaving users of an anonymizing network. Blocking a particular *user*, however, is a formidable task since that user could possibly acquire several identities (called the *Sybil* attack [Dou02]). The Nymble system binds *nymbles* to *resources* that are sufficiently difficult to obtain in great numbers. For example, we have used IP addresses as the resource in our implementation, but our scheme generalizes to other resources such as email addresses, identity certificates, trusted hardware, and so on.

We note that if two users can prove access to the same resource (e.g., if an IP address is reassigned), they will obtain the same stream of nymbles. We will address the practical issues related with resource-based blocking in Section 9, along with other examples of resources that would be useful in limiting the Sybil attack.

At this point we emphasize that creating Sybil-free credentials is an independent research topic in itself and we do not claim to solve the Sybil attack in this paper. This problem is faced by *any credential system* [Dou02, LSM06], and we suggest some promising approaches based on resource-based blocking since we aim to create a real-world deployment.

2.2 The Pseudonym Manager

The user must first contact the *Pseudonym Manager (PM)* and demonstrate control over a resource; for IP-address blocking, the user is required to connect to the PM directly (i.e., not through a known anonymizing network), as shown in Figure 1. We assume the PM has knowledge about Tor routers, for example, and can ensure that users are communicating with it directly.[6] Pseudonyms are deterministically chosen based on the controlled resource, ensuring that the same pseudonym is always issued for the same resource.

Note that the user *does not* disclose what server he or she intends to connect to, and therefore the user's connections are anonymous to the PM. The PM's duties are limited to mapping IP addresses (or other resources) to pseudonyms. As we will explain, the user contacts the PM only once per *linkability window* (e.g., once a day).

2.3 The Nymble Manager

After obtaining a pseudonym from the PM, the user connects to the *Nymble Manager (NM)* through the anonymizing network, and requests nymbles for access to a particular server (such as Wikipedia). Nymbles are generated using the user's pseudonym and the server's identity. The user's connections, therefore, are pseudonymous to the NM (as long as the PM and the NM do not collude) since the NM knows only the pseudonym-server pair, and the PM knows only the IP address-pseudonym pair. Note that due to the pseudonym assignment by the PM, nymbles are bound to the user's IP address and the server's identity.

To provide the requisite cryptographic protection and security properties (e.g., users should not to be able to fabricate their own nymbles), the NM encapsulates nymbles within *nymble tickets*. Servers wrap seeds into *linking tokens* and therefore we will speak of linking tokens being used to link future nymble tickets. The importance of these constructs will become apparent as we proceed.

2.4 Time

Nymble tickets are bound to specific time periods. As illustrated in Figure 2, in our system time is divided into linkability windows of duration $\mathcal{W}$, each of which is split into L time periods of duration $\mathcal{T}$ (i.e., $\mathcal{W} = L * \mathcal{T}$). We will refer to time periods and linkability windows chronologically as $t_1, t_2, \ldots, t_L$ and w_1, w_2, ... respectively. While a user's access *within* a time period is tied to a single nymble ticket, the use of different nymble tickets *across* time periods grants the user anonymity between time periods — smaller time periods provide users with higher rates of anonymous authentication, and likewise longer time periods rate-limit the number of misbehaviors from a particular user before he or she is blocked. For example, $\mathcal{T}$ could be set to 5 minutes, and $\mathcal{W}$ to 1 day (and thus $L = 288$). The linkability window serves two purposes — it allows for *dynamism* since resources such as IP addresses can get reassigned to different well-behaved users, making it undesirable to blacklist such resources indefinitely, and it ensures *forgiveness* of misbehavior after a certain period of time.

We assume that all entities, the PM, NM, servers, and users are time synchronized (for example, with `time.nist.gov` via the Network Time Protocol (NTP)), and can thus calculate the current

[6] Note that if a user connects through an unknown anonymizing network or proxy, the security of our system is no worse than that provided by real IP-address blocking, where the user could have used an anonymizing network unknown to the server.

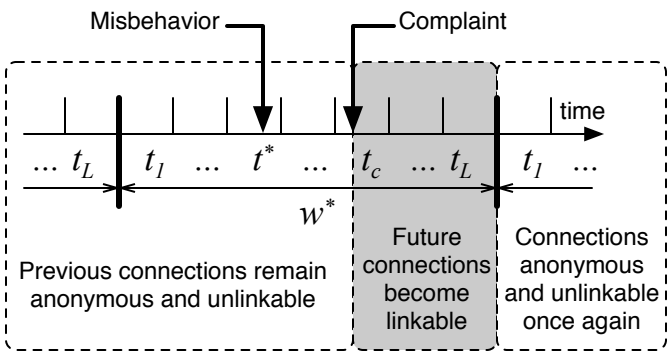


Figure 2: The life cycle of a misbehaving user. If the server complains in time period t_c about a user's connection in t^*, the user becomes linkable starting in t_c. Note that the complaint in t_c can include nymble tickets from only t_{c-1} and earlier.

linkability window and time period. Time-synchronization protocols such as NTP reduce but do not eliminate clock skews. Nymble does not attempt to defend against remote device-fingerprinting attacks, which attempt to reduce the anonymity of users by, e.g., inferring their clock skews [KBC05]. We leave such defenses to future work.

2.5 Blacklisting a user

If a user misbehaves, the server may link any future connection from this user within the current linkability window (e.g., the same day). Consider Figure 2 as an example: A user connects and misbehaves at a server during time period t^* within linkability window w^*. The server later detects this misbehavior and complains to the NM in time period t_c ($t^* < t_c \leq t_L$) of the same linkability window w^*. As part of the complaint, the server presents the nymble ticket of the misbehaving user and obtains the corresponding seed from the NM. The server is then able to link future connections by the user in time periods $t_c, t_c + 1, \ldots, t_L$ of the same linkability window w^* to the complaint. Therefore, users are blacklisted for the rest of the day, for example (the linkability window), once the server has complained about that user. Note that the user's connections in $t_1, t_2, \ldots, t^*, t^* + 1, \ldots, t_c$ remain unlinkable (i.e., including those since the misbehavior and until the time of complaint). Even though misbehaving users can be blocked from making connections in the future, the users' past connections remain unlinkable. This property allows the NM to be agnostic of servers' complaints; servers can subjectively judge users for any reason because users' privacy is maintained. We now describe how users are notified of their blacklisting status.

2.6 Notifying the user of blacklist status

Users who make use of anonymizing networks expect their connections to be anonymous. If a server obtains a seed for that user, however, it can link that user's subsequent connections (we emphasize that the user's previous connections remain anonymous to the server). It is of utmost importance, then, that users be notified of their blacklisting status before they present a nymble ticket to a server. In our system, the user can download the server's blacklist and verify whether she is on the blacklist. If so, the user disconnects immediately (the server learns that "some user disconnected probably because he or she has been blacklisted").

Since the blacklist is cryptographically signed by the NM, the authenticity of the blacklist is easily verified if the blacklist was updated in the current time period (only one update to the blacklist per time period is allowed). If the blacklist has not been updated in the current time period, the NM provides servers with "daisies" every time period so that users can verify the freshness of the blacklist ("blacklist from time period t_{old} is fresh as of time period t_{now}"). As we will discuss later, these daisies are elements of a hash chain, and provide a lightweight alternative to digital signatures. Using digital signatures and daisies, we thus ensure that race conditions are not possible in verifying the freshness of a blacklist. A user is guaranteed that he or she will not be linked if the user verifies the integrity and freshness of the blacklist before sending his or her nymble ticket.

3 Security Model

Nymble aims for four security goals. We provide informal definitions here; a detailed formalism is given in Section 7, which explains how these goals must also resist coalition attacks.

3.1 Goals and threats

An entity is *honest* when its operations abide by the system's specification. An honest entity can be *curious*: it attempts to infer knowledge from its own information (e.g., its secrets, state, and protocol communications). An honest entity becomes *corrupt* when it is compromised by an attacker, and hence reveals its information at the time of compromise, and operates under the attacker's full control, possibly deviating from the specification.

3.1.1 Blacklistability

Blacklistability assures that any honest server can indeed block misbehaving users. Specifically, if an honest server complains about a user that misbehaved in the current linkability window, the complaint will be successful and the user will *not* be able to *"nymble-connect,"* i.e., establish a Nymble-authenticated connection, to the server successfully in subsequent time periods (following the time of complaint) of that linkability window.

3.1.2 Rate-limiting

Rate-limiting assures any honest server that *no* user can successfully nymble-connect to it more than once within any single time period.

3.1.3 Anonymity

A user is *legitimate* according to a server if she has not been blacklisted by the server, and has not exceeded the rate limit of establishing Nymble-connections. Honest servers must be able to differentiate between legitimate and illegitimate users.

Anonymity protects the anonymity of honest users, regardless of their legitimacy according to the (possibly corrupt) server; the server cannot learn any more information beyond whether the user behind (an attempt to make) a nymble-connection is legitimate or illegitimate.

Who	Whom	How	What
Servers	PM & NM	honest	*Blacklistability* & *Rate-limiting*
Users	PM	honest	*Anonymity*
Users	NM	honest & not curious	*Anonymity*
Users	PM & NM	honest	*Non-frameability*

Figure 3: Nymble's matrix of trust assumptions: *Who* trusts *whom* to be *how* for *what* guarantee.

3.1.4 Non-frameability

Non-frameability guarantees that any honest user who is legitimate according to an honest server can nymble-connect to that server. This prevents an attacker from framing a legitimate honest user, e.g., by getting the user blacklisted for someone else's misbehavior.

3.2 Trust assumptions

We allow the servers and the users to be corrupt and controlled by an attacker. Not trusting these entities is important because encountering a corrupt server and/or user is a realistic threat. Nymble must still attain its goals under such circumstances.

Nymble makes several assumptions on *who* trusts *whom* to be *how* for *what* guarantee. We represent these trust assumptions as a matrix in Figure 3. Should a trust assumption become invalid, Nymble will not be able to provide the corresponding guarantee. For example, a corrupt NM alone can undermine *Blacklistability* by issuing credentials to users without a valid pseudonym.

4 Preliminaries

4.1 Notation

The notation $a \in_R S$ represents an element drawn uniformly at random from non-empty set S. $\mathbb{N}_0$ is the set of non-negative integers, and $\mathbb{N}$ is the set $\mathbb{N}_0 \backslash \{0\}$. $s[i]$ is the i-th element of list s. $s||t$ is the concatenation of (the unambiguous encoding of) lists s and t. The empty list is denoted by $\emptyset$. We sometimes treat lists of tuples as dictionaries. For example, if L is the list $((\mathsf{Alice}, 1234), (\mathsf{Bob}, 5678))$, then $\mathsf{L}[\mathsf{Bob}]$ denotes the tuple $(\mathsf{Bob}, 5678)$. If A is a (possibly probabilistic) algorithm, then $A(x)$ denotes the output when A is executed given the input x. $a := b$ means that b is assigned to a.

4.2 Cryptographic primitives

Nymble uses the following building blocks (concrete instantiations are suggested in Section 6):

- Secure *cryptographic hash functions*. These are one-way, collision-resistant and pseudo-random functions [BR93] that compute a digest of a message. Denote the range of the hash functions by $\mathcal{H}$.
- Secure *message authentication* (MA) [BCK96]. It consists of the key generation ($\mathsf{MA.KeyGen}$), and the message authentication code (MAC) computation ($\mathsf{MA.Mac}$) algorithms. Denote the domain of MACs by $\mathcal{M}$.

Algorithm 1 `PMCreatePseudonym`

Input: $(uid, w) \in \mathcal{H} \times \mathbb{N}$
Persistent state: $pmState \in \mathcal{S}_P$
Output: $pnym \in \mathcal{P}$

1: Extract $nymKey_P, macKey_{NP}$ from $pmState$
2: $nym := \mathsf{MA.Mac}(uid||w, nymKey_P)$
3: $mac := \mathsf{MA.Mac}(nym||w, macKey_{NP})$
4: **return** $pnym := (nym, mac)$

Algorithm 2 `NMVerifyPseudonym`

Input: $(pnym, w) \in \mathcal{P} \times \mathbb{N}$
Persistent state: $nmState \in \mathcal{S}_N$
Output: $b \in \{\textbf{true}, \textbf{false}\}$

1: Extract $macKey_{NP}$ from $nmState$
2: $(nym, mac) := pnym$
3: **return** $mac \stackrel{?}{=} \mathsf{MA.Mac}(nym||w, macKey_{NP})$

- Secure *symmetric-key encryption* (Enc) [BDJR97]. It consists of the key generation (Enc.KeyGen), encryption (Enc.Encrypt), and decryption (Enc.Decrypt) algorithms. Denote the domain of ciphertexts by Γ.
- Secure *digital signatures* (Sig) [GMR88]. It consists of the key generation (Sig.KeyGen), signing (Sig.Sign), and verification (Sig.Verify) algorithms. Denote the domain of signatures by Σ.

4.3 Data structures

We now describe several important data structures used by Nymble (Figure 4 provides a convenient summary).

4.3.1 Pseudonyms

The PM issues pseudonyms to users. A pseudonym *pnym* has two components *nym* and *mac*: *nym* is a pseudo-random mapping of the user's identity (e.g., IP address),[7] the linkability window w for which the pseudonym is valid, and the PM's secret key $nymKey_P$; *mac* is a MAC that the NM uses to verify the integrity of the pseudonym. Algorithms 1 and 2 describe the procedures of creating and verifying pseudonyms.

4.3.2 Seeds and nymbles

A *nymble* is a pseudo-random number, which serves as an identifier for a particular time period. Nymbles (presented by a user) across periods are unlinkable unless a server has blacklisted that user. Nymbles are presented as part of a nymble ticket, as described next.

[7] In Nymble, identities (users' and servers') are encoded into a fixed-length string using a cryptographic hash function.

Data structure	Description	Definition	Domain
pnym	Pseudonym	(nym, mac)	$\mathcal{P} \doteq \mathcal{M}^2$
ticket	Ticket	$(time, nymble, ctxt, mac_N, mac_{NS})$	$\mathcal{T} \doteq \mathbb{N} \times \mathcal{H} \times \Gamma \times \mathcal{M}^2$
cred	Credential	$(nymble^*, tickets)$	$\mathcal{D} \doteq \mathcal{H} \times \mathcal{T}^L$
blist	Blacklist	$(nymble_1^*, \ldots, nymble_n^*)$	$\mathcal{B}_n \doteq \mathcal{H}^n, n \in \mathbb{N}_0$
cert	Blacklist cert.	$(time_d, daisy, time_s, mac, sig)$	$\mathcal{C} \doteq \mathbb{N} \times \mathcal{H} \times \mathbb{N} \times \mathcal{M} \times \Sigma$
token	Linking token	$(seed, nymble)$	$\mathcal{N} \doteq \mathcal{H}^2$
pmState	PM's state	$(nymKey_P, macKey_{NP})$	$\mathcal{S}_P \doteq \mathcal{K}_{\mathsf{Mac}}^2$
nmKeys	NM's keys	$(macKey_{NP}, macKey_N, seedKey_N$ $encKey_N, decKey_N, signKey_N, verKey_N)$	$\mathcal{K} \doteq \mathcal{K}_{\mathsf{Mac}}^3 \times \mathcal{K}_{\mathsf{Encrypt}} \times \mathcal{K}_{\mathsf{Decrypt}} \times$ $\mathcal{K}_{\mathsf{Sign}} \times \mathcal{K}_{\mathsf{Verify}}$
nmEntry	NM's entry	$(sid, macKey_{NS}, daisy_L, time_{\text{lastUpd}})$	$\mathcal{E}_N \doteq \mathcal{H} \times \mathcal{K}_{\mathsf{Mac}} \times \mathcal{H} \times \mathbb{N}$
nmState	NM's state	$(keys, nmEntries)$	$\mathcal{S}_N \doteq \mathcal{K} \times {\mathcal{E}_N}^n, n \in \mathbb{N}_0$
svrState	Server's state	$(sid, macKey_{NS}, blist, cert, seen\text{-}tickets,$ $cmplnt\text{-}tickets, lnkng\text{-}tokens, time_{\text{lastUpd}})$	$\mathcal{S}_S \doteq \mathcal{H} \times \mathcal{K}_{\mathsf{Mac}} \times \mathcal{B}_k \times \mathcal{C} \times \mathcal{T}^\ell \times$ $\mathcal{T}^m \times \mathcal{N}^n \times \mathbb{N}, k, \ell, m, n \in \mathbb{N}_0$
usrEntry	User's entry	$(sid, cred, ticketDisclosed)$	$\mathcal{E}_U \doteq \mathcal{H} \times \mathcal{D} \times \{\mathbf{true}, \mathbf{false}\}$
usrState	User's state	$(pnym, usrEntries)$	$\mathcal{S}_U \doteq \mathcal{P} \times {\mathcal{E}_U}^n, n \in \mathbb{N}_0$

Figure 4: A summary of all the data structures in Nymble.

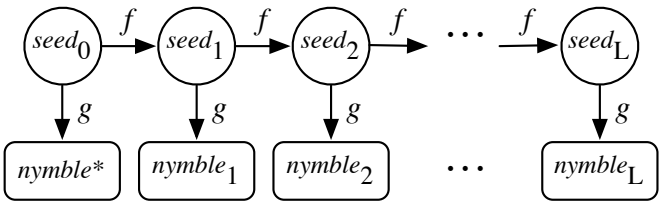


Figure 5: Evolution of seeds and nymbles. Given $seed_i$ it is easy to compute $nymble_i, nymble_{i+1}, \ldots, nymble_L$, but not $nymble^*, nymble_1, \ldots, nymble_{i-1}$.

As shown in Figure 5, *seeds* evolve throughout a linkability window using a *seed-evolution function* f; the seed for the next time period ($seed_{next}$) is computed from the seed for the current time period ($seed_{cur}$) as

$$seed_{next} = f(seed_{cur}).$$

The nymble ($nymble_t$) for a time period t is evaluated by applying the *nymble-evaluation function* g to its corresponding seed ($seed_t$), i.e.,

$$nymble_t = g(seed_t).$$

The NM sets $seed_0$ to a pseudo-random mapping of the user's pseudonym *pnym*, the (encoded) identity *sid* of the server (e.g., domain name), the linkability window w for which the seed is valid, and the NM's secret key $seedKey_N$. Seeds are therefore specific to user-server-window combinations. As a consequence, a seed is useful only for a particular server to link a particular user during a particular linkability window.

In our Nymble construction, f and g are two distinct cryptographic hash functions. Hence, it is easy to compute future nymbles starting from a particular seed by applying f and g appropriately, but infeasible to compute nymbles otherwise. Without a seed, the sequence of nymbles appears unlinkable, and honest users can enjoy anonymity. Even when a seed for a particular time period is obtained, all the nymbles prior to that time period remain unlinkable.

4.3.3 Nymble tickets and credentials

We now describe how *nymbles* are wrapped into *tickets* for authentication when connecting to a server.

A *credential* contains all the nymble tickets for a particular linkability window that a user can present to a particular server. Algorithm 3 describes the following procedure of generating a credential upon request.

A *ticket* contains a nymble specific to a server, time period, and linkability window. *ctxt* is encrypted data that the NM can use during a complaint involving the nymble ticket. In particular, *ctxt* contains the first nymble ($nymble^*$) in the user's sequence of nymbles, and the seed used to generate that nymble. As will be explained later, upon a complaint, the NM can extract the user's seed and issue it to the server by evolving the seed, and $nymble^*$ helps the NM to recognize whether the user has already been blacklisted.

The MACs mac_N and mac_{NS} are used by the NM and the server respectively to verify the integrity of the nymble ticket as described in Algorithms 4 and 5. As will be explained later, the NM will need to verify the ticket's integrity upon a complaint from the server.

Algorithm 3 `NMCreateCredential`

Input: $(pnym, sid, w) \in \mathcal{P} \times \mathcal{H} \times \mathbb{N}$
Persistent state: $nmState \in \mathcal{S}_N$
Output: $cred \in \mathcal{D}$

1: Extract $macKey_{NP}, macKey_N, seedKey_N, encKey_N$ from $keys$ in $nmState$
2: $seed_0 := f(\mathsf{Mac}(pnym||sid||w, seedKey_N))$
3: $nymble^* := g(seed_0)$
4: **for** t from 1 to L **do**
5: $\quad seed_t := f(seed_{t-1})$
6: $\quad nymble_t := g(seed_t)$
7: $\quad ctxt_t := \mathsf{Enc.Encrypt}(nymble^*||seed_t, encKey_N)$
8: $\quad ticket'_t := sid||t||w||nymble_t||ctxt_t$
9: $\quad mac_{N,t} := \mathsf{MA.Mac}(ticket'_t, macKey_N)$
10: $\quad mac_{NS,t} := \mathsf{MA.Mac}(ticket'_t||mac_{N,t}, macKey_{NS})$
11: $\quad tickets[t] := (t, nymble_t, ctxt_t, mac_{N,t}, mac_{NS,t})$
12: **end for**
13: **return** $cred := (nymble^*, tickets)$

Algorithm 4 `NMVerifyTicket`

Input: $(sid, t, w, ticket) \in \mathcal{H} \times \mathbb{N}^2 \times \mathcal{T}$
Persistent state: $svrState$
Output: $b \in \{\textbf{true}, \textbf{false}\}$

1: Extract $macKey_N$ from $keys$ in $nmState$
2: $(\cdot, nymble, ctxt, mac_N, mac_{NS}) := ticket$
3: $content := sid||t||w||nymble||ctxt$
4: **return** $mac_N \stackrel{?}{=} \mathsf{MA.Mac}(content, macKey_N)$

4.3.4 Blacklists

A server's blacklist is a list of $nymble^*$'s corresponding to all the nymbles that the server has complained about. Users can quickly check their blacklisting status at a server by checking to see whether their $nymble^*$ appears in the server's blacklist (see Algorithm 6).

Blacklist integrity It is important for users to be able to check the integrity and freshness of blacklists, because otherwise servers could omit entries or present older blacklists and link users without their knowledge. The NM signs the blacklist (see Algorithm 7), along with the server identity sid, the current time period t, current linkability window w, and $target$ (used for freshness, explained soon), using its signing key $signKey_N$. As will be explained later, during a complaint procedure, the NM needs to update the server's blacklist, and thus needs to check the integrity of the blacklist presented by the server. To make this operation more efficient, the NM also generates a MAC using its secret key $macKey_N$ (line 3). At the end of the signing procedure, the NM returns a blacklist certificate (line 6), which contains the time period for which the certificate was issued, a $daisy$ (used for freshness, explained soon), mac and sig. Algorithms 8 and 9 describe how users and the NM can verify the integrity and freshness of blacklists.

Algorithm 5 `ServerVerifyTicket`

Input: $(t, w, ticket) \in \mathbb{N}^2 \times \mathcal{T}$
Persistent state: *svrState*
Output: $b \in \{\textbf{true}, \textbf{false}\}$

1: Extract $sid, macKey_{NS}$ from *svrState*
2: $(\cdot, nymble, ctxt, mac_N, mac_{NS}) := ticket$
3: $content := sid||t||w||nymble||ctxt||mac_N$
4: **return** $mac_{NS} \stackrel{?}{=} \mathsf{MA.Mac}(content, macKey_{NS})$

Algorithm 6 `UserCheckIfBlacklisted`

Input: $(sid, blist) \in \mathcal{H} \times \mathcal{B}_n$, $n, \ell \in \mathbb{N}_0$
Persistent state: $usrState \in \mathcal{S}_U$
Output: $b \in \{\textbf{true}, \textbf{false}\}$

1: Extract $nymble^*$ from *cred* in $usrEntries[sid]$ in *usrState*
2: **return** $(nymble^* \stackrel{?}{\in} blist)$

Blacklist freshness If the NM has signed the blacklist for the current time period, users can simply verify the digital signature in the certificate to infer that the blacklist is both valid (not tampered with) and fresh (since the current time period matches the time period in the blacklist certificate). To prove the freshness of blacklists every time period, however, the servers would need to get the blacklists digitally signed every time period, thus imposing a high load on the NM. To speed up this process, we use a hash chain [EGM89, Mic02, HJP05] to certify that "blacklist from time period t is still fresh."

Each time the NM responds to a complaint request, it generates a new random seed $daisy_L$ for a hash chain corresponding to time period L. It then computes $daisy_{L-1}, daisy_{L-2}, \ldots, daisy_t$ up to current time period t by successively hashing the previous daisy to generate the next with a cryptographic hash function h. For example, $daisy_5 = h(daisy_6)$. Note if the NM reveals $daisy_5$, one can compute $daisy_4, \ldots, daisy_1$ by applying the hash function successively, but $daisy_6, \ldots, daisy_L$ remain secret to the NM.

As outlined later (in Algorithm 13), *target* is set to $daisy_t$. Now, until the next update to the blacklist, the NM need only release daisies for the current time period instead of digitally signing the blacklist. Given a certificate from an older time period and $daisy_t$ for current time period t, users can verify the integrity and freshness of the blacklist by computing the target from $daisy_t$.

4.3.5 Complaints and linking tokens

A server complains to the NM about a misbehaving user by submitting the user's nymble ticket that was used in the offending connection. The NM returns a seed, from which the server creates a *linking token*, which contains the seed and the corresponding nymble.

Each server maintains a list of linking tokens created as above, called the *linking-list*, and updates each token on the list every time period. When a user presents a nymble ticket for making a nymble-connection, the server checks the nymble within the ticket against the nymbles in the linking-list entries. A match indicates that the user has been blacklisted.

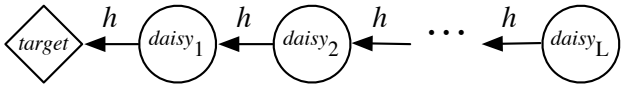


Figure 6: A chain of daisies. Given $daisy_i$ it is easy to verify the freshness of the blacklist by applying h i times to obtain *target*. Only the NM can compute the next $daisy_{i+1}$ in the chain.

Algorithm 7 `NMSignBL`

Input: $(sid, t, w, target, blist) \in \mathcal{H} \times \mathbb{N}^2 \times \mathcal{H} \times \mathcal{B}_n$, $n \in \mathbb{N}_0$
Persistent state: $nmState \in \mathcal{S}_N$
Output: $cert \in \mathcal{C}$

1: Extract $macKey_N, signKey_N$ from *keys* in *nmState*
2: $content := sid||t||w||target||blist$
3: $mac := \mathsf{MA.Mac}(content, macKey_N)$
4: $sig := \mathsf{Sig.Sign}(content, signKey_N)$
5: $daisy := target$
6: **return** $cert := (t, daisy, t, mac, sig)$

4.4 Communication channels

Nymble distinguishes and utilizes three types of communication channels over which protocols are executed, namely type-*Basic*, -*Auth* and -*Anon*, depending on the channels' characteristics as shown in Figure 7.

We assume that a public-key infrastructure (PKI) such as X.509 is in place, and that the NM, the PM and all the servers in Nymble have obtained a PKI credential from a well-established and trustworthy CA. (We stress that the users in Nymble, however, need not possess a PKI credential.) These entities can thus realize type-*Basic* and type-*Auth* channels to one another by setting up a TLS[8] connection using their PKI credentials.

All users can realize type-*Basic* channels to the NM, the PM and any server, again by setting up a TLS connection. Additionally, by setting up a TLS connection over the Tor anonymizing network,[9] users can realize a type-*Anon* channel to the NM and any server.

5 Our Nymble Construction

5.1 System setup

To set up the Nymble system, the NM and the PM interact as follows.

1. The NM executes `NMInitState`() (see Algorithm 10) and initializes its state *nmState* to the algorithm's output.

2. The NM extracts $macKey_{NP}$ from *nmState* and sends it to the PM over a type-*Auth* channel.

[8] The Transport Layer Security Protocol Version 1.2. IETF RFC 5246. http://tools.ietf.org/rfc/rfc5246.txt.

[9] While we acknowledge the existence of attacks on Tor's anonymity, we assume Tor provides perfect anonymity [FJS07] for the sake of arguing Nymble's own anonymity guarantee.

Algorithm 8 `VerifyBL`

Input: $(sid, t, w, blist, cert) \in \mathcal{H} \times \mathbb{N}^2 \times \mathcal{B}_n \times \mathcal{C}$, $n \in \mathbb{N}_0$
Output: $b \in \{\textbf{true}, \textbf{false}\}$

1: $(t_d, daisy, t_s, mac, sig) := cert$
2: **if** $t_d \neq t \vee t_d < t_s$ **then**
3: **return false**
4: **end if**
5: $target := h^{(t_d - t_s)}(daisy)$
6: $content := sid||t_s||w||target||blist$
7: **return** $\mathsf{Sig.Verify}(content, sig, verKey_N)$

Algorithm 9 `NMVerifyBL`

Input: $(sid, t, w, blist, cert) \in \mathcal{H} \times \mathbb{N}^2 \times \mathcal{B}_n \times \mathcal{C}$, $n \in \mathbb{N}_0$
Persistent state: $nmState \in \mathcal{S}_N$
Output: $b \in \{\textbf{true}, \textbf{false}\}$

1-6: Same as lines 1–6 in `VerifyBL`
7: Extract $macKey_N$ from *keys* in *nmState*
8: **return** $mac \stackrel{?}{=} \mathsf{MA.Mac}(content, macKey_N)$

$macKey_{NP}$ is a shared secret between the NM and the PM, so that the NM can verify the authenticity of pseudonyms issued by the PM.

3. The PM generates $nymKey_P$ by running $\mathsf{Mac.KeyGen}()$ and initializes its state *pmState* to the pair $(nymKey_P, macKey_{NP})$.

4. The NM publishes $verKey_N$ in *nmState* in a way that the users in Nymble can obtain it and verify its integrity at any time (e.g., during registration).

5.2 Server registration

To participate in the Nymble system, a server with identity *sid* initiates a type-*Auth* channel to the NM, and registers with the NM according to the *Server Registration* protocol below. Each server may register at most once in any linkability window.

1. The NM makes sure that the server has not already registered: If $(sid, \cdot, \cdot) \in nmEntries$ in its *nmState*, it terminates with failure; it proceeds otherwise.

2. The NM reads the current time period and linkability window as t_{now} and w_{now} respectively, and then obtains a *svrState* by running (see Algorithm 11)

$$\texttt{NMRegisterServer}_{nmState}(sid, t_{now}, w_{now}).$$

3. The NM appends *svrState* to its *nmState*, sends it to the *Server*, and terminates with success.

4. The server, on receiving *svrState*, records it as its state, and terminates with success.

Type	Initiator	Responder	Link
Basic	–	Authenticated	Confidential
Auth	Authenticated	Authenticated	Confidential
Anon	Anonymous	Authenticated	Confidential

Figure 7: Different types of channels utilized in Nymble.

Algorithm 10 `NMInitState`

Output: $nmState \in \mathcal{S}_N$

1: $macKey_{NP} :=$ Mac.KeyGen()
2: $macKey_N :=$ Mac.KeyGen()
3: $seedKey_N :=$ Mac.KeyGen()
4: $(encKey_N, decKey_N) :=$ Enc.KeyGen()
5: $(signKey_N, verKey_N) :=$ Sig.KeyGen()
6: $keys := (macKey_{NP}, macKey_N, seedKey_N,$
7: $\quad encKey_N, decKey_N, signKey_N, verKey_N)$
8: $nmEntries := \emptyset$
9: **return** $nmState := (keys, nmEntries)$

In *svrState*, $macKey_{NS}$ is a key shared between the NM and the server for verifying the authenticity of nymble tickets; $time_{\text{lastUpd}}$ indicates the time period when the blacklist was last updated, which is initialized to t_{now}, the current time period at registration.

5.3 User registration

A user with identity *uid* must register with the PM once each linkability window before acquiring a credential during that linkability window. To do so, the user initiates a type-*Basic* channel to the PM, followed by the *User Registration* protocol described below.

1. The PM checks if the user is allowed to register. In our current implementation, a user's identity is her IP address. The PM infers the registering user's IP address from the communication channel, and makes sure that the IP address does not belong to a known Tor exit node. If this is not the case, the PM terminates with failure.

2. Otherwise, the PM reads the current linkability window as w_{now}, and runs

$$pnym := \texttt{PMCreatePseudonym}_{pmState}(uid, w_{now}).$$

   The PM then gives *pnym* to the user, and terminates with success.

3. The user, on receiving *pnym*, sets her state *usrState* to $(pnym, \emptyset)$, and terminates with success.

5.4 Credential acquisition

To establish a Nymble-connection to a server, a user must provide a valid ticket, which is acquired as part of a credential from the NM. To acquire a credential for server *sid* during the current linkability window, a registered user initiates a type-*Anon* channel to the NM, followed by the *Credential Acquisition* protocol below.

Algorithm 11 `NMRegisterServer`

Input: $(sid, t, w) \in \mathcal{H} \times \mathbb{N}^2$
Persistent state: $nmState \in \mathcal{S}_N$
Output: $svrState \in \mathcal{S}_S$

$(keys, nmEntries) := nmState$
$macKey_{NS} := \mathsf{Mac.KeyGen}()$
$daisy_L \in_R \mathcal{H}$
$nmEntries' := nmEntries || (sid, macKey_{NS}, daisy_L, t)$
$nmState := (keys, nmEntries')$
$target := h^{(L-t+1)}(daisy_L)$
$blist := \emptyset$
$cert := \mathtt{NMSignBL}_{nmState}(sid, t, w, target, blist)$
$svrState := (sid, macKey_{NS}, blist, cert, \emptyset, \emptyset, \emptyset, t)$
return $svrState$

1. The user extracts *pnym* from *usrState* and sends the pair $(pnym, sid)$ to the NM.
2. The NM reads the current linkability window as w_{now}. It makes sure the user's *pnym* is valid: If
$$\mathtt{NMVerifyPseudonym}_{nmState}(pnym, w_{now})$$
returns **false**, the NM terminates with failure; it proceeds otherwise.
3. The NM runs
$$\mathtt{NMCreateCredential}_{nmState}(pnym, sid, w_{now}),$$
which returns a credential *cred*. The NM sends *cred* to the user and terminates with success.
4. The user, on receiving *cred*, creates $usrEntry := (sid, cred, \textbf{false})$, appends it to its state *usrState*, and terminates with success.

5.5 Nymble-connection establishment

To establish a connection to a server *sid*, the user initiates a type-*Anon* channel to the server, followed by the *Nymble-connection establishment* protocol described below, which is also illustrated in Figure 8.

5.5.1 Blacklist validation

1. The server sends $\langle blist, cert \rangle$ to the user, where *blist* is its blacklist for the current time period and *cert* is the certificate on *blist*. (We will describe how the server can update its blacklist soon.)
2. The user reads the current time period and linkability window as $t_{now}^{(U)}$ and $w_{now}^{(U)}$ and assumes these values to be current for the rest of the protocol.
3. For freshness and integrity the user checks if
$$\mathtt{VerifyBL}_{usrState}(sid, t_{now}^{(U)}, w_{now}^{(U)}, blist, cert) = \textbf{true}.$$

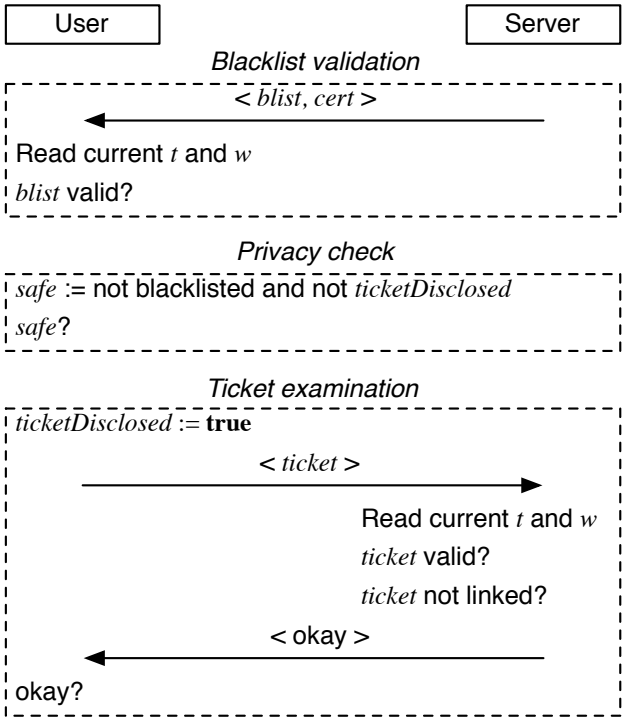


Figure 8: The *Nymble-connection Establishment* protocol.

If not, she terminates the protocol with failure.

5.5.2 Privacy check

Since multiple connection-establishment attempts by a user to the same server within the same time period can be linkable, the user keeps track of whether she has already disclosed a ticket to the server in the current time period by maintaining a boolean variable *ticketDisclosed* for the server in her state.

Furthermore, since a user who has been blacklisted by a server can have her connection-establishment attempts linked to her past establishment, the user must make sure that she has not been blacklisted thus far.

Consequently, if *ticketDisclosed* in $usrEntries[sid]$ in the user's *usrState* is **true**, or

$$\texttt{UserCheckIfBlacklisted}_{usrState}(sid, blist) = \textbf{false},$$

then it is unsafe for the user to proceed with the protocol; the user terminates the protocol with failure.

5.5.3 Ticket examination

1. The user sets *ticketDisclosed* in $usrEntries[sid]$ in *usrState* to **true**. She then sends $\langle ticket \rangle$ to the server, where *ticket* is $ticket[t_{now}^{(U)}]$ in *cred* in $usrEntries[sid]$ in *usrState*.

Algorithm 12 `ServerLinkTicket`

Input: $ticket \in \mathcal{T}$
Persistent state: $svrState \in \mathcal{S}_S$
Output: $b \in \{\textbf{true}, \textbf{false}\}$

Extract *lnkng-tokens* from *svrState*
$(\cdot, nymble, \cdots) := ticket$
for all $i = 1$ to $|lnkng\text{-}tokens|$ **do**
  if $(\cdot, nymble) = lnkng\text{-}tokens[i]$ **then**
    return true
  end if
end for
return false

Note that the user discloses *ticket* for time period $t_{now}^{(U)}$ after verifying *blist*'s freshness for $t_{now}^{(U)}$. This procedure avoids the situation in which the user verifies the current blacklist just before a time period ends, and then presents a newer *ticket* for the next time period.

2. On receiving $\langle ticket \rangle$, the server reads the current time period and linkability window as $t_{now}^{(S)}$ and $w_{now}^{(S)}$ respectively. The server then checks that:

   - *ticket* is fresh, i.e., $ticket \notin slist$ in server's state.
   - *ticket* is valid, i.e., on input $(t_{now}^{(S)}, w_{now}^{(S)}, ticket)$, the algorithm `ServerVerifyTicket` returns **true**. (See Algorithm 5.)
   - *ticket* is not linked (in other words, the user has not been blacklisted), i.e.,

     $$\texttt{ServerLinkTicket}_{svrState}(ticket) = \textbf{false}.$$

     (See Algorithm 12.)

3. If any of the checks above fails, the server sends $\langle \mathsf{goodbye} \rangle$ to the user and terminates with failure. Otherwise, it adds *ticket* to *slist* in its state, sends $\langle \mathsf{okay} \rangle$ to the user and terminates with success.

4. On receiving $\langle \mathsf{okay} \rangle$, the user also terminates with success.

5.6 Service provision and access logging

If both the user and the server terminate with success in the *Nymble-connection Establishment* described above, the server may start serving the user over the same channel. The server records *ticket* and logs the access during the session for a potential complaint in the future.

5.7 Auditing and filing for complaints

If at some later time the server desires to blacklist the user behind a Nymble-connection, during the establishment of which the server collected *ticket* from the user, the server files a complaint by appending *ticket* to *cmplnt-tickets* in its *svrState*.

Filed complaints are batched up. They are processed during the next blacklist update (to be described next).

5.8 Blacklist update

Servers update their blacklists for the current time period for two purposes. First, as mentioned earlier, the server needs to provide the user with its blacklist (and blacklist certificate) for the current time period during a Nymble-connection establishment. Second, the server needs to be able to blacklist the misbehaving users by processing the newly filed complaints (since last update).

The procedure for updating blacklists (and their certificates) differs depending on whether complaints are involved. At a high level, when there is no complaint (i.e., the server's *cmplnt-tickets* is empty), blacklists stay unchanged; the certificates need only a "light refreshment." When there are complaints, on the other hand, new entries are added to the blacklists and certificates need to be regenerated.

Our current implementation employs "lazy" update: the server updates its blacklist upon its first Nymble-connection establishment request in a time period.

5.8.1 Without complaints

1. The server with identity *sid* initiates a type-*Auth* channel to the NM, and sends a request to the NM for a blacklist update.

2. The NM reads the current time period as t_{now}. It extracts t_{lastUpd} and $daisy_L$ from *nmEntry* for *sid* in *nmState*. If t_{lastUpd} is t_{now}, the server has already updated its blacklist for the current time period, and the NM hence terminates the protocol as failure.

3. Otherwise, the NM updates t_{lastUpd} to t_{now}. It computes $daisy' := h^{(L-t_{now}+1)}(daisy_L)$ and sends $(t_{now}, daisy')$ to the server.

4. The server replaces t_d and *daisy* in *cert* in *blist* in its *svrState* with t_{now} and $daisy'$ respectively.

5.8.2 With complaints

1. The server with identity *sid* initiates a type-*Auth* channel to the NM and sends (*blist*, *cert*, *cmplnt-tickets*) from its *svrState* as a blacklist update request.

2. The NM reads the current time period as $t_{now}^{(N)}$. It runs

   $$\texttt{NMHandleComplaints}_{nmState}$$

   on input $(sid, t_{now}, w_{now}, blist, cert, cmplnt\text{-}tickets)$. (See Algorithm 15.) If the algorithm returns $\perp$, the NM considers the update request invalid, in which case the NM terminates the protocol as failure.

3. Otherwise, the NM relays the algorithm's output $(blist', cert', seeds)$, to the server.

4. The server updates its state *svrState* as follows. It replaces *blist* and *cert* with $blist||blist'$ and $cert'$ respectively, and sets *cmplnt-tkts* to $\emptyset$. For each $seed \in seeds$, the server creates a *token* as $(seed, g(seed))$ and appends it to *lnkng-tokens*. Finally, the server terminates with success.

Algorithm 13 `NMComputeBLUpdate`

Input: $(sid, t, w, blist, cmplnt\text{-}tickets) \in \mathcal{H} \times \mathbb{N}^2 \times \mathcal{B}_n \times \mathcal{T}^m$
Persistent state: $nmState \in \mathcal{S}_N$
Output: $(blist', cert') \in \mathcal{B}_m \times \mathcal{C}$

1: $(keys, nmEntries) := nmState$
2: $\begin{pmatrix} \cdot, macKey_N, seedKey_N, \\ encKey_N, \cdot, signKey_N, \cdot \end{pmatrix} := keys$
3: **for** $i = 1$ to m **do**
4: $\quad(\cdot, \cdot, ctxt, \cdot, \cdot) := cmplnt\text{-}tickets[i]$
5: $\quad nymble^* || seed := \mathsf{Decrypt}(ctxt, decKey_N)$
6: $\quad$**if** $nymble^* \in blist$ **then**
7: $\quad\quad blist'[i] \in_R \mathcal{H}$
8: $\quad$**else**
9: $\quad\quad blist'[i] := nymble^*$
10: $\quad$**end if**
11: **end for**
12: $daisy'_L \in_R \mathcal{H}$
13: $target' := h^{(L-t+1)}(daisy'_L)$
14: $cert' := \mathtt{NMSignBL}(sid, t, w, target', blist||blist')$
15: Replace $daisy_L$ and t_{lastUpd} in $nmEntries[sid]$ in $nmState$ with $daisy'_L$ and by t respectively
16: **return** $(blist', cert')$

We now explain what `NMHandleComplaints` does. The algorithm first checks the integrity and freshness of the blacklist (lines 2–6) and that the NM hasn't already updated the server's blacklist for the current time period. It then checks if all complaints are valid for some previous time period during the current linkability window (lines 8–13). Finally, the algorithm prepares an answer to the update request by invoking `NMComputeBLUpdate` and `NMComputeSeeds` (see Algorithm 14) (lines 16–19).

`NMComputeBLUpdate` (see Algorithm 13) creates new entries to be appended to the server's blacklist. Each entry is either the actual $nymble^*$ of the user being complained about if the user has not been blacklisted already, or a random *nymble* otherwise. This way, the server cannot learn if two complaints are about the same user, and thus cannot link the Nymble-connections to the same user. `NMComputeSeeds` (see Algorithm 14) uses the same trick when computing a seed that enables the server to link a blacklisted user.

5.9 Periodic update

5.9.1 Per time period

At the end of each time period that is not the last of the current linkability window, each registered server updates its *svrState* by running (see Algorithm 8)

$$\mathtt{ServerUpdateState}_{svrState}(),$$

which prepares the linking-token-list for the new time period. Each entry is updated by evolving the seed and computing the corresponding nymble.

Algorithm 14 `NMComputeSeeds`

Input: $(t, blist, cmplnt\text{-}tickets) \in \mathbb{N} \times \mathcal{B}_n \times \mathcal{T}^m$
Persistent state: $nmState \in \mathcal{S}_N$
Output: $seeds \in \mathcal{H}^m$

Extract $decKey_N$ from $keys$ in $nmState$
for all $i = 1$ to m **do**
  $(t', nymble, ctxt, \cdots) := cmplnt\text{-}tickets[i]$
  $nymble^* || seed := \mathsf{Enc.Decrypt}(ctxt, decKey_N)$
  if $nymble^* \in blist$ **then**
    $seeds[i] \in_R \mathcal{H}$
  else
    $seeds[i] := f^{(t-t')}(seed)$
  end if
end for
return $seeds$

Each registered user sets *ticketDisclosed* in every *usrEntry* in *usrState* to **false**, signaling that the user has not disclosed any ticket in the new time period.

5.9.2 Per linkability window

At the beginning of each linkability window, all the entities, i.e., the PM, the NM, the servers and the users erase their state and start afresh. In other words, the NM and the PM must re-setup Nymble for the new current linkability window and all servers and users must re-register if they still want to use Nymble.

6 Evaluation

6.1 Security

We state the following theorem regarding the security of our Nymble construction. Its proof will be given in Section 8.

Theorem 1 Our Nymble construction has *Blacklistability*, *Rate-limiting*, *Anonymity* and *Non-frameability*, provided that the listed trust assumptions hold true, and the cryptographic primitives used are secure. □

6.2 Performance

6.2.1 Implementation and experimental setup

We implemented Nymble as a C++ library along with Ruby and JavaScript bindings. We chose Ruby because of our familiarity with the language, and JavaScript because it is the de facto language for Firefox extensions, which we use for the client-side interface. One could, however, easily compile bindings for any of the languages (such as Python, PHP, and Perl) supported by the Simplified Wrapper and Interface Generator (SWIG) for example. We utilize OpenSSL as it supports all the cryptographic primitives that we need.

Algorithm 15 `NMHandleComplaints`

Input: $(sid, t, w, blist, cert, cmplnt\text{-}tickets) \in \mathcal{H} \times \mathbb{N}^2 \times \mathcal{B}_n \times \mathcal{C} \times \mathcal{T}^m$
Persistent state: $nmState \in \mathcal{S}_N$
Output: $(blist', cert', seeds) \in \mathcal{B}_m \times \mathcal{C} \times \mathcal{H}^m$

Extract $time_{\text{lastUpd}}$ from $nmEntries[sid]$ in $nmState$
$b_1 := (time_{\text{lastUpd}} < t)$
$b_2 :=$
  $\texttt{NMVerifyBL}_{nmState}(sid, time_{\text{lastUpd}}, w, blist, cert)$
if $\neg(b_1 \wedge b_2)$ **then**
  return $\perp$
end if
for all $i = 1$ to m **do**
  $ticket := cmplnt\text{-}tickets[i]$; $(\tilde{t}, \cdots) := ticket$
  $b_{i1} := \tilde{t} < t$
  $b_{i2} := \texttt{NMVerifyTicket}_{nmState}(sid, \tilde{t}, w, ticket)$
  if $\neg(b_{i1} \wedge b_{i2})$ **then**
    return $\perp$
  end if
end for
$(blist', cert') :=$
  $\texttt{NMComputeBLUpdate}_{nmState}(sid, t, w, blist, cert)$
$seeds :=$
  $\texttt{NMComputeSeeds}_{nmState}(t, blist, cmplnt\text{-}tickets)$
return $(blist', cert', seeds)$

We use SHA-256 for the cryptographic hash functions; HMAC-SHA-256 for the message authentication MA; AES-256 in CBC-mode for the symmetric encryption Enc; and 2048-bit RSASSA-PSA for the digital signatures Sig. We chose RSA over DSA for digital signatures because of its faster verification speed — in our system, verification occurs more often than signing.

We evaluated our system on a 2.2 GHz Intel Core 2 Duo Macbook Pro with 4 GB of RAM. The PM, the NM, and the server were implemented as Mongrel (Ruby's version of Apache) servers. The user portion was implemented as a Firefox 3 extension in JavaScript with bindings to the Nymble C++ library wrapped as an XPCOM component. We evaluated protocol performance and data-structure size. For each experiment relating to protocol performance, we ran the experiment 10 times and averaged the results. The evaluation of data-structure sizes is the byte count of the marshalled data structures that would be sent over the network.

6.2.2 Experimental results

Figure 9 shows the size in bytes of the various data structures. The X-axis represents the number of entries in each data structure — complaints in the blacklist update request, tickets in the credential (equal to L, the number of time periods in a linkability window), nymbles in the blacklist, tokens and seeds in the blacklist update response, and nymbles in the blacklist. For example, a linkability window of 1 day with 5 minute time periods equates to $L = 288$. The size of a credential in this case is about 59 KB. The size of a blacklist update request with 50 complaints is roughly 11 KB,

Algorithm 16 `ServerUpdateState`

Persistent state: $svrState \in \mathcal{S}_S$

1: Extract *lnkng-tokens* from *svrState*
2: **for all** $i = 1$ to $|lnkng\text{-}tokens|$ **do**
3: $(seed, nymble) := lnkng\text{-}tokens[i]$
4: $seed' := f(seed)$; $nymble' := g(seed')$
5: $tokens'[i] := (seed', nymble')$
6: **end for**
7: Replace *lnkng-tokens* in *svrState* with $tokens'$
8: Replace *seen-tickets* in *svrState* with $\emptyset$

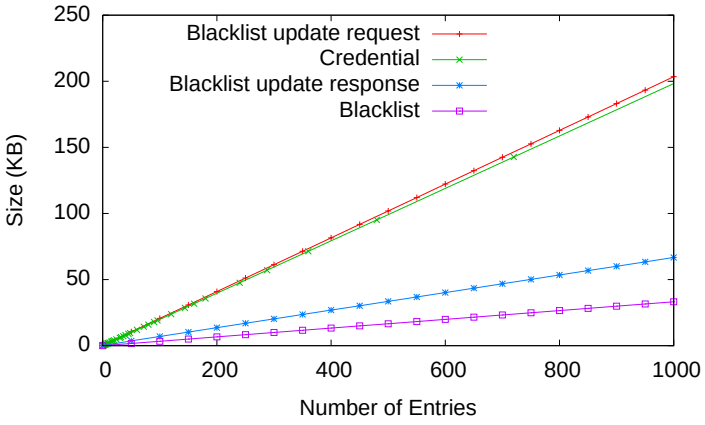


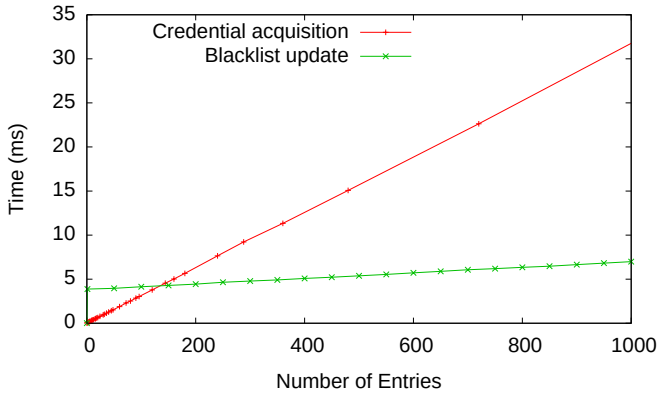
Figure 9: The marshaled size of various Nymble data structures. The X-axis refers to the number of entries — complaints in the blacklist update request, tickets in the credential, tokens and seeds in the blacklist update response, and nymbles in the blacklist.

whereas the size of a blacklist update response for the same amount of complaint requests is only about 4 KB. The size of a blacklist (downloaded by users before each connection) with 500 nymbles is 17 KB.

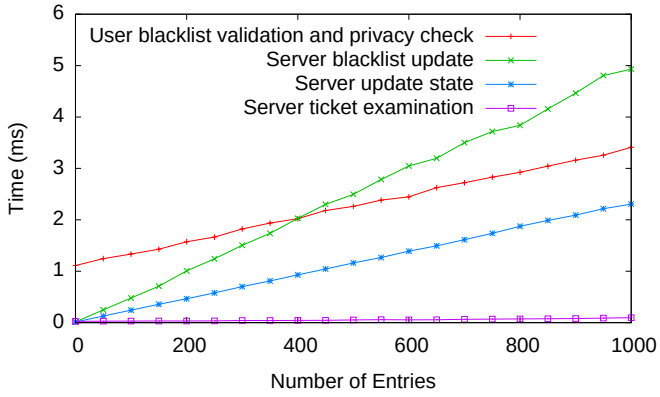
In general, each structure grows linearly as the number of entries increases. Credentials and blacklist update requests grow at the same rate because a credential is a collection of tickets which is more or less what is sent as a complaint list when the server wishes to update its blacklist. In our implementation we use Google's Protocol Buffers to (un)marshal these structures because it is cross-platform friendly and language-agnostic.

Figure 10(a) shows the amount of time it takes the NM to perform various protocols. It takes about 9 ms to create a credential when $L = 288$. Note that this protocol occurs only once every linkability window for each user wanting to connect to a particular server. For blacklist updates, the initial jump in the graph corresponds to the fixed overhead associated with signing a blacklist. To execute the update blacklist protocol with 500 complaints it takes the NM about 54 ms. However, when there are no complaints, it takes the NM on average less than a millisecond to update the daisy.

Figure 10(b) shows the amount of time it takes the server and user to perform various protocols. These protocols are relatively inexpensive by design, i.e., the amount of computation performed



(a) Blacklist updates take several milliseconds and credentials can be generated in 9 ms for the suggested parameter of L=288.



(b) The bottleneck operation of server ticket examination is less than 1 ms and validating the blacklist takes the user only a few ms.

Figure 10: Nymble's performance at (a) the NM and (b) the user and the server when performing various protocols.

by the users and servers should be minimal. For example, it takes less than 3 ms for a user to execute a security check on a blacklist with 500 nymbles. Note that this figure includes signature verification as well, and hence the fixed-cost overhead exhibited in the graph. It takes less than a millisecond for a server to perform authentication of a ticket against a blacklist with 500 nymbles. We implemented these protocols as a linear search through a blacklist's list of nymbles because of the negligible overhead even for lists with thousands of entries. More efficient data structures (such as a hash table) do not seem necessary, but they would protect against timing attacks, and we leave that for future work. Every time period, a server must update its state and blacklist. Given a linking list with 500 entries, the server will spend less than 2 ms updating the linking list. If the server were to issue a blacklist update request with 500 complaints, it would take less than 3 ms for the server to update its blacklist.

7 Security Formalism

To formalize the security goals of Nymble, we model an environment in which the adversary can interact in virtually the same way as a real-world attacker would with the Nymble system. Each security goal is defined as a game in which the adversary aims to win; a Nymble construction achieves a security goal *if and only if* no (efficient) adversary can win in the corresponding game with probability non-negligibly greater than a particular value.

To argue the security of our Nymble construction, we adopt the "reductionist" approach: we assume that a game-winning Adversary $\mathfrak{A}$ exists, and then construct—and hence show the existence of—a Probabilistic Poly-Time (PPT) algorithm called the Simulator $\mathfrak{S}$, which can be used to violate some facts or well-established assumptions, thereby arriving at a contradiction.

We formalize the security of Nymble assuming that there is only one linkability window. In Section 8, we discuss how the security arguments apply across linkability windows.

7.1 Oracles

To simulate an environment for the Adversary $\mathfrak{A}$, the Simulator $\mathfrak{S}$ maintains a set of states and operates a set of oracles to answer arbitrary and possibly adaptive queries to those oracles made by $\mathfrak{A}$. $\mathfrak{S}$ maintains the following state: sets $\mathbf{U}_H$, $\mathbf{U}_C$, $\mathbf{S}_H$, $\mathbf{S}_C$, $\mathbf{C}$, $\mathbf{T}$ and $\mathbf{B}$, initially set to empty, and integers I_S, I_U, I_C, I_T and I_B, initially set to one. Also, $\mathfrak{S}$ maintains the current time period t_{now}, initially set to one.

The following specifies how $\mathfrak{S}$ operates the oracles to answer the queries made by $\mathfrak{A}$.

- **The User (resp. Server) Registration Oracle**, $\mathcal{O}_{UR}$ (resp. $\mathcal{O}_{SR}$), allows the adversary to have an honest user (resp. server) register into the system. Upon query with input id, $\mathfrak{S}$ simulates an execution of the *User* (resp. *Server*) *Registration* protocol between an honest user (resp. server) with identity id and the NM (resp. PM). Let π be the protocol communications transcript. If the protocol terminates with failure at the PM (resp. NM), $\mathfrak{S}$ returns $(\perp, \pi)$. Otherwise $\mathfrak{S}$ indexes the newly registered user (resp. server) as $i := I_U$ (resp. $j := I_S$), updates $\mathbf{U}_H := \mathbf{U}_H \cup \{(i, id)\}$ (resp. $\mathbf{S}_H := \mathbf{S}_H \cup \{(j, id)\}$), increments I_U (resp. I_S), and returns (i, π) (resp. (j, π)).

- **The Corrupt-User (resp. Corrupt-Server) Registration Oracle**, $\mathcal{O}_{CUR}$ (resp. $\mathcal{O}_{CSR}$), allows a corrupt user (resp. server) to register into the system. Upon query with input id, $\mathfrak{S}$ interacts with $\mathfrak{A}$ according to the *User* (resp. *Server*) *Registration* protocol, in which $\mathfrak{A}$ plays the role of the user (resp. server) with identity id and $\mathfrak{S}$ plays the role of the PM (resp. NM). If the protocol terminates with failure at the PM (resp. NM), $\mathfrak{S}$ returns $\perp$. Otherwise, $\mathfrak{S}$ indexes the newly registered user (resp. server) as $i := I_U$ (resp. $j := I_S$), updates $\mathbf{U}_C := \mathbf{U}_C \cup \{(i, id)\}$ (resp. $\mathbf{S}_C := \mathbf{S}_C \cup \{(j, id)\}$), increments I_U (resp. I_S), and returns i (resp. j).

  We disallow the adversary to corrupt an honest user or server when the concerned entity is currently involved in one or more other oracle queries. This restriction imposes little reduction on the adversary's capability because, in reality, it is highly likely that an attacker who manages to corrupt a user or a server in the middle of a protocol run has the ability to corrupt the entity a little bit earlier, before that protocol run has started.

- **The User (resp. Server) Corruption Oracle**, $\mathcal{O}_{UC}$ (resp. $\mathcal{O}_{SC}$), allows the adversary to corrupt an honest registered user (resp. server). Upon query with input i (resp. j), if $(i, \cdot) \notin \mathbf{U}_H$ (resp. $(j, \cdot) \notin \mathbf{S}_H$), $\mathfrak{S}$ returns $\perp$. Otherwise $\mathfrak{S}$ removes (i, id) from $\mathbf{U}_H$ (resp. (j, id) from $\mathbf{S}_H$), adds (i, id) to $\mathbf{U}_C$ (resp. (j, id) to $\mathbf{S}_C$), and returns the current state of user i (resp. server j).

- **The Credential Acquisition Oracle**, $\mathcal{O}_{CA}$, allows the adversary to have an honest registered user acquire a credential. Upon query with input (i, sid), if $(i, \cdot) \notin \mathbf{U}_H$, $\mathfrak{S}$ returns $\perp$. Otherwise, $\mathfrak{S}$ simulates an execution of the *Credential Acquisition* protocol between honest user i and the NM, in which the user desires to acquire a credential for connecting to a server *sid*. Let π be the protocol communications transcript. If the protocol terminates with failure at the NM, $\mathfrak{S}$ returns $(\perp, \pi)$. Otherwise, $\mathfrak{S}$ indexes the newly issued credential as $k := I_C$, updates $\mathbf{C} := \mathbf{C} \cup \{(k, i, sid)\}$, increments I_C, and returns (k, π).

- **The Credential Acquisition With Corrupt-User Oracle**, $\mathcal{O}_{CACU}$, allows a corrupt user to acquire a credential. Upon query with input *sid*, $\mathfrak{S}$ interacts with $\mathfrak{A}$ according to the *Credential Acquisition* protocol, in which $\mathfrak{A}$ plays the role of a user acquiring a credential to connect to a server *sid* and $\mathfrak{S}$ plays the role of the NM. If the protocol terminates with failure at the NM, $\mathfrak{S}$ returns $\perp$. Otherwise $\mathfrak{S}$ indexes the newly issued credential as $k := I_C$, updates $\mathbf{U}_C := \mathbf{U}_C \cup \{(k, \perp, sid)\}$, increments I_C, and returns k.

- **The Connection Establishment Oracle**, $\mathcal{O}_{CE}$, allows the adversary to have an honest user establish a Nymble-connection to an honest server. Upon query with input (i, sid), if $(i, \cdot) \notin \mathbf{U}_H$ or $(\cdot, sid) \notin \mathbf{S}_H$, $\mathfrak{S}$ returns $\perp$. Otherwise, $\mathfrak{S}$ simulates an execution of the *Nymble-connection Establishment* protocol between user i and server j, where j is such that $(j, sid) \in \mathbf{S}_H$ and is unique.

  Let π be the protocol communications transcript, the boolean b_T be whether $\texttt{ExtractTicket}(\pi) \neq \perp$, and the booleans b_U and b_S be whether the protocol terminates with success at the user and the server respectively. $\mathfrak{S}$ indexes the connection establishment as $\ell := I_T$, sets $res := (\ell, t_{now}, b_T, b_U, b_S, \pi)$, updates $\mathbf{T} := \mathbf{T} \cup \{(i, j, sid, res)\}$, increments I_T, and finally returns *res*.

  In the above, we have written $\texttt{ExtractTicket}(\pi)$ to denote the ticket sent by the user during the run. If the user terminated before sending a ticket, then $\texttt{ExtractTicket}(\pi)$ is by definition the special symbol $\perp$.

- **The Connection Establishment With Corrupt-User Oracle**, $\mathcal{O}_{CECU}$, allows a corrupt user to establish a Nymble-connection to an honest server. Upon query with input *sid*, if $(\cdot, sid) \notin \mathbf{S}_H$, $\mathfrak{S}$ returns $\perp$. Otherwise, $\mathfrak{S}$ interacts with $\mathfrak{A}$ according to the *Nymble-connection Establishment* protocol, in which $\mathfrak{A}$ plays the role of the user and $\mathfrak{S}$ plays the role of server j, where j is such that $(j, sid) \in \mathbf{S}_H$ and is unique.

  Let π be the protocol communications transcript, the boolean b_T be whether $\texttt{ExtractTicket}(\pi) \neq \perp$, and the boolean b_S be whether the protocol terminates with success at the server. $\mathfrak{S}$ indexes the connection establishment as $\ell := I_T$, sets $res := (\ell, t_{now}, b_T, \perp, b_S, \pi)$, updates $\mathbf{T} := \mathbf{T} \cup \{(i, j, sid, res)\}$, increments I_T, and returns *res*.

- **The Connection Establishment With Corrupt-Server Oracle**, $\mathcal{O}_{CECS}$, allows the adversary to have an honest user establish a Nymble-connection to a corrupt server. Upon query

with input (i, sid), if $(i, \cdot) \notin \mathbf{U}_H$, $\mathfrak{S}$ returns $\perp$. Otherwise $\mathfrak{S}$ interacts with $\mathfrak{A}$ according to the *Nymble-connection Establishment* protocol, in which $\mathfrak{S}$ plays the role of honest user i and $\mathfrak{A}$ plays the role of the server sid.

Let π be the protocol communications transcript, the boolean b_T be whether $\texttt{ExtractTicket}(\pi) \neq \perp$, and the boolean b_U be whether the protocol terminates with success at the user. $\mathfrak{S}$ indexes the connection establishment as $\ell := I_T$, sets $res := (\ell, t_{now}, b_T, b_U, \perp, \pi)$, updates $\mathbf{T} := \mathbf{T} \cup \{(i, \perp, sid, res)\}$, increments I_T, and returns res.

- **The Blacklist Update Oracle**, $\mathcal{O}_{BU}$, allows the adversary to have an honest server update its blacklist. Upon query with input $(sid, \mathcal{T})$, if $(\cdot, sid) \notin \mathbf{S}_H$ or $(\cdot, \cdot, sid, (\cdot, \cdot, \mathbf{true}, \cdot, \mathbf{true}, \pi)) \notin \mathbf{T}$ for some π such that $\texttt{ExtractTicket}(\pi) \in \mathcal{T}$, $\mathfrak{S}$ returns $\perp$. Otherwise, $\mathfrak{S}$ simulates an execution of the *Blacklist Update* protocol between honest server j, where j is such that $(j, sid) \in \mathbf{S}_H$, and the NM, in which server j updates its blacklist with a set of newly complained tickets $\mathcal{T}$.

  Let π be the protocol communications transcript and the booleans b_S and b_N be whether the protocol terminates with success at the server and at the NM respectively. $\mathfrak{S}$ indexes the blacklist update as $\mu := I_B$, sets $res := (\mu, t_{now}, b_S, b_N, \pi)$, updates $\mathbf{B} := \mathbf{B} \cup \{(j, \perp, \mathcal{T}, res)\}$, increments I_B, and returns res.

- **The Blacklist Update With Corrupt-Server Oracle**, $\mathcal{O}_{BUCS}$, allows a corrupt server to update its blacklist. Upon query with input $(sid, \mathcal{T})$, $\mathfrak{S}$ interacts with $\mathfrak{A}$ according to the *Blacklist Update* protocol, in which $\mathfrak{A}$ plays the role of the server sid who desires to update its blacklist with a set of newly complained tickets $\mathcal{T}$, and $\mathfrak{S}$ plays the role of the NM.

  Let π be the protocol communications transcript and the boolean b_N be whether the protocol terminates with success at the NM. $\mathfrak{S}$ indexes the blacklist update as $\mu := I_B$, sets $res := (\mu, t_{now}, b_S, b_N, \pi)$, updates $\mathbf{B} := \mathbf{B} \cup \{(\perp, sid, \mathcal{T}, res)\}$, increments I_B, and returns res.

- **The Time Oracle**, $\mathcal{O}_T$, allows the adversary to elapse time. Upon query, if $t_{now} < L$, $\mathfrak{S}$ simulates, for all $i \in \mathbf{U}_H$ and $j \in \mathbf{S}_H$, an execution of the *Periodic Update* algorithm for user i and server j, increments t_{now}, and finally returns the incremented t_{now}. Otherwise (i.e., $t_{now} = L$), $\mathfrak{S}$ returns $\perp$.

7.2 Game-Accountability

A Nymble construction has the security properties *Blacklistability* and *Rate-limiting* if no PPT adversary $\mathfrak{A}$ can win, with non-negligible probability (in the size of the security parameters), in *Game-Accountability* played against the Simulator $\mathfrak{S}$ as defined below.

1. **Setup Phase.** $\mathfrak{S}$ plays the role of the NM and the PM and executes the *System Setup* protocol (Section 5.1) on a sufficiently large λ. $\mathfrak{S}$ keeps *nmState* and *pmState* secret and gives all public parameters to $\mathfrak{A}$.

2. **Probing Phase.** $\mathfrak{A}$ may arbitrarily and adaptively query all the oracles.

3. **End Game Phase.** $\mathfrak{A}$ declares to end the game.

Denote by Q the sequence of oracle queries made by $\mathfrak{A}$ throughout the game in chronological order. $\mathfrak{A}$ wins in the game if one or more of the following criteria is met:

- **Criterion 1.** Some honest registered server terminated with success in two or more runs of the *Nymble-connection Establishment* protocol during the same time period (according to the server's clock), even though they were all initiated by the same honest registered user.

  Formally, there exist i, sid and t such that Q contains the subsequence:

$$Q' = \begin{pmatrix} (\ell, t, \cdot, \cdot, \mathbf{true}, \cdot) & \leftarrow & \mathcal{O}_{CE}(i, sid), \\ (\ell', t, \cdot, \cdot, \mathbf{true}, \cdot) & \leftarrow & \mathcal{O}_{CE}(i, sid) \end{pmatrix},$$

  where $y \leftarrow \mathcal{O}(x)$ denotes a query to $\mathcal{O}$ on input x that resulted in output y.

- **Criterion 2.** Some run of the *Blacklist Update* protocol (with or without complaints) between some honest registered user and some honest registered server terminated with failure at one or both ends.

  Formally, there exists sid, $\mathcal{T}$, b_S and b_N such that $b_S \wedge b_N = \mathbf{false}$ and Q contains:

$$(\cdot, \cdot, \mathbf{false}, \cdot, \cdot) \leftarrow \mathcal{O}_{BU}(sid, \mathcal{T}).$$

- **Criterion 3.** Some honest registered server terminated with success in some run of the *Nymble-connection Establishment* protocol initiated by some honest registered user, even though the server had by then already terminated with success in some run of the *Blacklist Update* protocol, in which the server complained about the user (by including in the update request some ticket disclosed by the user).

  Formally, there exist i, sid, π, $\mathcal{T}$, t and t' such that $\texttt{ExtractTicket}(\pi) \in \mathcal{T}$, $t < t'$ and Q contains the subsequence:

$$Q' = \begin{pmatrix} (\cdot, \cdot, \cdot, \cdot, \mathbf{true}, \pi) & \leftarrow & \mathcal{O}_{CE}(i, sid), \\ (\cdot, t, \mathbf{true}, \cdot, \cdot) & \leftarrow & \mathcal{O}_{BU}(sid, \mathcal{T}), \\ (\cdot, t', \cdot, \cdot, \mathbf{true}, \cdot) & \leftarrow & \mathcal{O}_{CE}(i, sid) \end{pmatrix}.$$

- **Criterion 4.** Some honest server terminated with success in some run of the *Nymble-connection Establishment* protocol initiated by the adversary, even though, for each of the then corrupt users, the server had by then already (i) terminated with success in some run of the *Nymble-connection Establishment* protocol during the same time period (according to the server's clock) that was initiated by the user, or (ii) terminated with success in some run of the *Blacklist Update* protocol, in which the server complained about the user (by including in the update request some ticket disclosed by the user).[10]

  Formally, there exist sid, m, n, $(t_i, \mathcal{T}_i, \tilde{t}_i, \pi_i)$ for $i = 1$ to n and t_{n+1} such that $m, n \geq 0$, $m + n > |\mathcal{U}_C|$, $\tilde{t}_i < t_{i+1}$ for all $i = 1$ to n, $\texttt{ExtractTicket}(\pi_i) \in \mathcal{T}_i$ for all $i = 1$ to n, and Q

[10] While the user might still have been honest during the concerned disclosure, we assume that the user were already corrupt. We can make this assumption because it gives the adversary no less power in the attack.

contains the subsequences:

$$Q' = \left(\begin{array}{lcl} (\cdot, t_1, \cdot, \cdot, \mathbf{true}, \pi_1) & \leftarrow & \mathcal{O}_{CECU}(sid), \\ (\cdot, \tilde{t}_1, \mathbf{true}, \cdot, \cdot) & \leftarrow & \mathcal{O}_{BU}(sid, \mathcal{T}_1), \\ (\cdot, t_2, \cdot, \cdot, \mathbf{true}, \pi_2) & \leftarrow & \mathcal{O}_{CECU}(sid), \\ (\cdot, \tilde{t}_2, \mathbf{true}, \cdot, \cdot) & \leftarrow & \mathcal{O}_{BU}(sid, \mathcal{T}_2), \\ & \vdots & \\ (\cdot, t_n, \cdot, \cdot, \mathbf{true}, \pi_n) & \leftarrow & \mathcal{O}_{CECU}(sid), \\ (\cdot, \tilde{t}_n, \mathbf{true}, \cdot, \cdot) & \leftarrow & \mathcal{O}_{BU}(sid, \mathcal{T}_n), \end{array} \right),$$

and

$$Q'' = \left(\begin{array}{lcl} (\cdot, t_{n+1}, \cdot, \cdot, \mathbf{true}, \cdot) & \leftarrow & \mathcal{O}_{CECU}(sid), \\ (\cdot, t_{n+1}, \cdot, \cdot, \mathbf{true}, \cdot) & \leftarrow & \mathcal{O}_{CECU}(sid), \\ & \vdots & \\ (\cdot, t_{n+1}, \cdot, \cdot, \mathbf{true}, \cdot) & \leftarrow & \mathcal{O}_{CECU}(sid) \end{array} \right).$$

7.3 Game-Non-Frameability

A Nymble construction has *Non-frameability* if no PPT adversary $\mathfrak{A}$ can win, with non-negligible probability (in the size of the security parameters), in *Game-Non-Frameability* played against the Simulator $\mathfrak{S}$ as defined below.

1. **Setup.** Same as in *Game-Accountability*.
2. **Probing phase.** Same as in *Game-Accountability*.
3. **End game phase.** $\mathfrak{A}$ declares to end the game.

Denote by Q the sequence of oracle queries made by $\mathfrak{A}$ throughout the game in chronological order. $\mathfrak{A}$ wins in the game if:

- **Criterion.** Some run of the *Nymble-connection establishment* protocol between some honest registered user and some honest registered server in some time period terminated with failure at one or both ends, even though the user had not by then disclosed a ticket to the server during the time period (according to the user's clock), and the server had not by then terminated with success in some run of the *Blacklist Update* protocol in which the server complained about the user (by including in the update request some ticket disclosed by the user).

  Formally, there exist i, sid and t such that:

  - Q contains $(\cdot, t, \cdot, b_U, b_S, \cdot) \leftarrow \mathcal{O}_{CE}(i, sid)$ for some $b_U \wedge b_S = \mathbf{false}$,
  - Q does *not* contain any $(\cdot, t, \cdot, \cdot, \mathbf{true}, \cdot) \leftarrow \mathcal{O}_{CE}(i, sid)$, and
  - if Q contains, for some π, $\mathcal{T}$ and $\tilde{t}$,

    $$(\cdot, \cdot, \cdot, \cdot, \mathbf{true}, \pi) \leftarrow \mathcal{O}_{CE}(i, sid) \text{ and } (\cdot, \tilde{t}, \mathbf{true}, \cdot, \cdot) \leftarrow \mathcal{O}_{BU}(sid, \mathcal{T}),$$

    then $\texttt{ExtractTicket}(\pi) \notin \mathcal{T}$ or $\tilde{t} \geq t$.

7.4 Game-Anonymity

7.4.1 The anonymity sets

In Section 3.1.3, we have informally talked about the "legitimacy" of a honest registered user. Here we give a formal definition.

Denote by Q the sequence of oracle queries made by $\mathfrak{A}$ *thus far* during the game defining *Anonymity* (to be described soon) in chronological order. We say that an honest registered user i^* is *illegitimate* at time period t^* according to the (possibly corrupt or impersonated) server with identity sid^* if at least one of the following criteria holds true:

- **Criterion 1.** The user had by then disclosed a ticket to the server in some run of the *Nymble-connection Establishment* protocol during time period t^* (according to the user's clock).

  Formally, Q contains, for some $\tau \neq \perp$,

  $$(\cdot, t^*, \mathbf{true}, \cdot, \cdot, \cdot) \leftarrow \mathcal{O}_{CE/CECS}(i^*, sid^*).$$

- **Criterion 2.** The server had by then terminated with success in some run of the *Blacklist Update* protocol, in which the server complained about the user (by including a ticket disclosed by the user).

  Formally, there exist π, $\mathcal{T}$, t such that $\texttt{ExtractTicket}(\pi) \in \mathcal{T}$, $t < t^*$ and Q contains

  $$\begin{array}{lcl} (\cdot, \cdot, \mathbf{true}, \cdot, \cdot, \pi) & \leftarrow & \mathcal{O}_{CE/CECS}(i^*, sid^*), \quad \text{and} \\ (\cdot, t, \cdot, \mathbf{true}, \cdot) & \leftarrow & \mathcal{O}_{BU/BUCS}(sid^*, \mathcal{T}). \end{array}$$

Thus, an honest registered user i^* is *legitimate* at time period t^* according to (possibly corrupt or impersonated) server sid^* if she is not *illegitimate*.

Figure 11 depicts the concept.

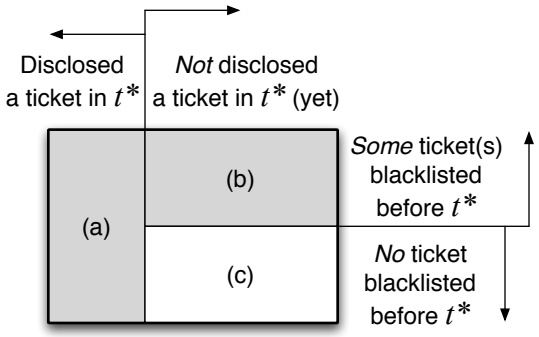


Figure 11: The universe of all honest registered users is partitioned into two anonymity sets according to the behavior of both the users and the (possibly malicious) server. Regions (a) and (b) together represent the anonymity set of the *illegitimate* users; region (c) represents that of the *legitimate* users.

7.4.2 The game

A Nymble construction has *Anonymity* if no PPT adversary $\mathfrak{A}$ can win, with probability non-negligibly (in the size of the security parameters) greater than 1/2, in *Game-Anonymity* played against the Simulator $\mathfrak{S}$ as defined below.

1. **Setup.** Same as in *Game-Accountability*.
2. **Probing Phase I.** Same as in *Game-Accountability*.
3. **Challenge Phase.** $\mathfrak{A}$ decides on $i_0^* \neq i_1^*$ and sid^* such that the following criterion holds:
   - **Criterion.** Either both users i_0^* and i_1^* are *legitimate*, or they are both *illegitimate*, according to the server with identity sid^*.

   $\mathfrak{A}$ queries $\mathcal{O}_{CE}(\perp, sid^*)$ or $\mathcal{O}_{CECS}(\perp, sid^*)$, i.e., without specifying i. $\mathfrak{S}$ flips a fair coin $\hat{b} \in_R \{0,1\}$ and answers the query assuming $i = i_{\hat{b}}$.
4. *Probing Phase II.* Same as Probing Phase I, with the additional restriction that the criterion above must still hold.
5. *End Game Phase.* $\mathfrak{A}$ returns guess $\tilde{b} \in \{0,1\}$ on $\hat{b}$.

$\mathfrak{A}$ wins in the game if $\tilde{b} = \hat{b}$.

8 Security Analysis

We sketch the proof of Theorem 1.

8.1 Accountability

The honesty of the PM and the security of HMAC together imply that a coalition of c users, each with a unique identity, can get at most c pseudonyms that will be verified to be valid at the NM. Since the NM is also honest, these c valid pseudonyms enable each user to acquire a credential from the NM for connecting to a given server. Thus, given a time period, the coalition has at most c tickets that will be verified by the server to be valid.

Due to the honesty of both the server and the NM and the security of the HMAC, there does not exist a valid ticket that is not one of those c tickets. Since reused tickets will always result in a failed connection establishment at an honest server, the coalition can successfully establish at most c connections at the server in any time period, each time using a different one of those c tickets, regardless of the server's blacklisting.

Next, we observe that if the NM and the server are honest, then the server can always successfully blacklist the ticket used in a connection establishment in which the server terminated with success. The reason is the following. An honest verifier terminates with success in a connection establishment protocol only if the ticket is verified to be valid, which will also be verified by the NM to be valid, as long as HMAC and digital signature are secure. Since an honest server complains only about valid tickets presented to it, the NM will always terminate with success during a blacklist update.

Now it suffices to show that, for each of those c tickets, call it $ticket^*$, if the owner of the ticket, say user i^*, owns another ticket, call it $ticket'$, that was used in a previous successful connection establishment, index it k', to an honest server j^*, and $ticket'$ has been blacklisted in some time period t', then establishing a connection to server j^*, index it k^*, during any time period $t^* > t'$ by presenting $ticket^*$ will fail.

Let us assume the contrary that connection establishment k^* was successful. Since connection establishments k' and k^* were successful, $ticket'$ and $ticket^*$ must be valid. The nymble in each of them was thus correctly computed according to user i^* identity. The two nymbles are thus two nodes on a hash chain, separated by a distance of $(t^* - t')$. Now since the NM is honest and the server has successfully blacklisted tkt' and updates its linking list honestly, the `ServerLinkTicket` will return fail on input $ticket^*$. The establishment k^* should have failed, and hence the contradiction.

8.2 Non-Frameability

Assume the contrary that the adversary successfully framed honest user i^* with respect to honest server j^* in time period t^*. Then user i^* failed to establish a connection, index it k^*, to server j in time period t^*, and yet the user was legitimate according to the server at that moment, i.e., user i^* had not yet disclosed a ticket to server j^* in time period t^*, and the tickets disclosed by user i^* had not been blacklisted successfully by the server up until the beginning of time period t^*.

Since the establishment k^* failed and both user i^* and server j^* were honest, the ticket presented, call it $ticket^*$, during the establishment had been seen by server j^*, or it was invalid or linked. By arguments similar to those in *Blacklistability*, an adversary who has not corrupted user i^* cannot forge user i^*'s tickets. Thus, server j^* had not seen $ticket^*$. Also, since the PM, the NM, user i^* and server j^* were honest, $ticket^*$ must have been valid. Consequently, $ticket^*$ was linked.

The fact that $ticket^*$ was linked implies that there exists an entry $(seed^*, nymble^*)$ in server j^*'s linking list such that the nymble in $ticket^*$ equals $nymble^*$. The existence of the entry means that the honest server j^* had blacklisted a ticket, call it $ticket_b$, and complained about it in a successful blacklist update protocol in some time period $t_u < t^*$ such that the returned $seed_u$ evolves to $seed^*$. Since the NM was honest and the blacklist update was successful, $ticket_b$ was valid and it can thus be similarly argued that it was created by the honest NM for a particular user $\tilde{i}$.

If $\tilde{i} \neq i^*$, then user $\tilde{i}$'s $seed_0$ is different from user i^*'s $seed_0$ so long as the PM is honest, and yet the two $seed_0$'s evolve to the same $seed^*$, which contradicts the collision-resistance property of the evolution function. Now we have $\tilde{i} = i^*$. That is, $ticket_b$ is a ticket given to user i^*. As argued before, no attacker can forge $ticket_b$ without corrupting user i^*. Hence, $ticket_b$ was the ticket presented by user i^* in a successful connection establishment to server j^*. This contradicts the assumption that user i^* had not been blacklisted before t^*.

8.3 Anonymity

We argue the anonymity of our Nymble construction in two cases. In the first, the adversary attempts to distinguish between two users i_0^* and i_1^* of his choice behind a connection establishment, index it as k^*, to a possibly corrupt server j^* also of his choice, who are both illegitimate according to server j^* chosen by the adversary, at some point of time in some time period t^*, also of the adversary's choice. In the second case, the two chosen users are both legitimate according to the server. We show that the adversary cannot succeed in either case.

8.3.1 Distinguishing between two illegitimate users

We argue that the two chosen illegitimate users will react indistinguishably.

Observe that all honest users execute the *Nymble-connection Establishment* protocol in exactly the same manner up until the end of the *Blacklist validation* stage (Section 5.5.1). It suffices then to show that every illegitimate user will evaluate *safe* to **false**, and hence terminate the protocol with failure at the end of the *Privacy check* stage (Section 5.5.2).

Let us first consider an illegitimate user who has already disclosed a ticket during a connection establishment, index it as k', to the same server at a time prior to—but in the same period as—the current connection establishment k^*. In both establishments, the users correctly identify server j^* due to the authenticity of the channel. Hence, the boolean *ticketDisclosed* for the server has been set to **true** during establishment k' and thus *safe* is evaluated to **false** during establishment k^*.

Now, an illegitimate user who has never disclosed a ticket during the same time period must have, by definition, one or more of her disclosed tickets successfully blacklisted by server j^* before t^*. Let $ticket'$ be one such ticket. Hence, server j^* executed a blacklist update protocol before t^* in which $ticket'$ was contained in the list of tickets under complaint and NM terminated with success. Since NM terminated with success, $ticket'$ was valid. If a blacklist update with a complaint against $ticket'$ terminated with success at the NM, then server j^* will have a signed blacklist, call it $blist_c$, containing the user's $nymble^*$ for the time period in which the update happened.

Now, if we assume the contrary that *safe* is evaluated to **true** during the establishment k^*, then the user's $nymble^*$ is not in the given blacklist, call it $blist^*$. Since $blist^*$ is verified by the user to be valid, server j^* must have obtained it via a blacklist update protocol, index it as ℓ, in which the NM terminated with success (as otherwise the digital signature would be forgeable, or the hash in the daisy chain could be inverted). This implies that, in update ℓ, server j^* has supplied a blacklist verified to be valid by the NM for some earlier time period.

Since the user's $nymble^*$ appeared on $blist_c$, and an honest NM never deletes entries in the blacklist during a blacklist update protocol, $nymble^*$ also appeared on $blist^*$, which contradicts the assumption.

8.3.2 Distinguishing between two legitimate users

We show that the two chosen users will react indistinguishably. We first argue that any legitimate user i^* will proceed to the *Ticket examination* stage in the *Nymble-connection establishment* protocol. We then argue that the adversary can extract no information on the user's identity from the ticket that the user discloses.

The authenticity of the channel implies that the user always knows the correct identity of the server she is establishing a connection to. As a result, if the user is legitimate according to sever j^*, the boolean *ticketDisclosed* for the server remains **false**.

Now, *safe* is evaluated to **true** if `UserCheckIfBlacklisted` returns **false**. We explain why this is. Assume the contrary that the algorithm returns **true**. Then the blacklist, call it $blist^*$, given to the user contains the user's $nymble^*$. By arguments similar to those we have used, this implies that the user has actually had at least one of her disclosed tickets blacklisted by the server, which contradicts the fact that the user is legitimate.

Now, let us investigate the composition of a ticket. Since each of the two MACs in the ticket is a deterministic function output of *sid*, t, w, *nymble*, *ctxt*, $macKey_N$, $macKey_{NS}$, an adversary can learn from the MACs no information other than those objects, among which only *nymble* and *ctxt*

depend on the user's identity. Since the adversary does not know the decryption key, the CCA2 security of the encryption implies that *ctxt* reveals no information about its underlying plaintext and thus the user's identity to the adversary.

Finally, we argue why the *nymble* in the ticket also reveals no information about the user's identity to the adversary. Suppose the ticket is valid for time period t^*. Then by now it should be straightforward to see that the adversary cannot have obtained any seed of the user at time period t^* or before. Under the Random Oracle model, the Simulator can correctly simulate the hash function g from $seed_{t^*}$ to $nymble_{t^*}$ of the user. And hence the result.

8.4 Across multiple linkability windows

With multiple linkability windows, our Nymble construction still has *Accountability* and *Non-frameability* because each ticket is valid for and only for a specific linkability window; it still has *Anonymity* because pseudonyms are an output of a collision-resistant function that takes the linkability window as input.

9 Discussion

IP-address blocking By picking IP addresses as the resource for limiting the Sybil attack, our current implementation closely mimics IP-address blocking that many servers on the Internet rely on. There are, however, some inherent limitations to using IP addresses as the scarce resource. If a user can obtain multiple IP addresses she can circumvent nymble-based blocking and continue to misbehave. We point out, however, that this problem exists in the absence of anonymizing networks as well, and the user would be able to circumvent regular IP-address based blocking by using multiple IP addresses. Some servers alleviate this problem with subnet-based IP blocking, and while it is possible to modify our system to support subnet-blocking, new privacy challenges emerge; a more thorough description of subnet-blocking is left for future work.

Other resources Users of anonymizing networks such as Tor would be reluctant to use resources that directly reveal their identity (e.g., passports or a national PKI). Email addresses could provide more privacy, but provide weak blacklistability guarantees because users can easily create new email addresses. Other possible resources include client puzzles [JB99] and e-cash, where users are required to perform a certain amount of computation or pay money to acquire a credential. Another alternative is for the PM to send an SMS message to the user's mobile phone. These approaches would limit the number of credentials obtained by a single individual by raising the cost of acquiring credentials.

Server-specific linkability windows An enhancement would be to provide support to vary $\mathcal{T}$ and $\mathcal{L}$ for different servers. As described, our system does not support varying linkability windows, but does support varying time periods. This is because the PM is not aware of the server the user wishes to connect to, yet it must issue pseudonyms specific to a linkability window. We do note that the use of resources such as client puzzles or e-cash would eliminate the need for a PM, and users could obtain Nymbles directly from the NM. In that case, server-specific linkability windows could be used.

Side-channel attacks While our current implementation does not fully protect against side-

channel attacks, we have used caution to mitigate the risks. We have implemented various algorithms in a way that their execution time leaks little information that cannot already be inferred from the algorithm's output.[11] Also, since a confidential channel does not hide the size of the communication, we have constructed the protocols so that each kind of protocol message is of the same size regardless of the identity or current legitimacy of the user. Finally, we have paid attention to some potential leakage of the user's information such as when setting up a TLS connection, during which cipher-suite parameters are exchanged in the clear.

10 Conclusions

We have proposed and built a comprehensive credential system called Nymble, which can be used to add a layer of accountability to any publicly known anonymizing network. Servers can blacklist misbehaving users while maintaining their privacy, and we show how these properties can be attained in a way that is practical, efficient, and sensitive to needs of both users and services.

We hope that our work will increase the mainstream acceptance of anonymizing networks such as Tor, which has thus far been completely blocked by several services because of users who abuse their anonymity.

Acknowledgments

Peter C. Johnson and Daniel Peebles helped in the early stages of prototyping. We are grateful for the suggestions and help from Roger Dingledine and Nick Mathewson.

[11] We acknowledge that timing attacks may still allow the fingerprinting of users based on their response times.

References

[ACJT00] Giuseppe Ateniese, Jan Camenisch, Marc Joye, and Gene Tsudik. A Practical and Provably Secure Coalition-Resistant Group Signature Scheme. In *CRYPTO*, LNCS 1880, pages 255–270. Springer, 2000.

[AST02] Giuseppe Ateniese, Dawn Xiaodong Song, and Gene Tsudik. Quasi-Efficient Revocation in Group Signatures. In *Financial Cryptography*, LNCS 2357, pages 183–197. Springer, 2002.

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. In *CRYPTO*, LNCS 1109, pages 1–15. Springer, 1996.

[BDJR97] Mihir Bellare, Anand Desai, E. Jokipii, and Phillip Rogaway. A Concrete Security Treatment of Symmetric Encryption. In *FOCS*, pages 394–403, 1997.

[BR93] Mihir Bellare and Phillip Rogaway. Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73. ACM Press, 1993.

[Bra93] Stefan Brands. Untraceable Off-line Cash in Wallets with Observers (Extended Abstract). In *CRYPTO*, LNCS 773, pages 302–318. Springer, 1993.

[BS01] Emmanuel Bresson and Jacques Stern. Efficient Revocation in Group Signatures. In *Public Key Cryptography*, LNCS 1992, pages 190–206. Springer, 2001.

[BS04] Dan Boneh and Hovav Shacham. Group Signatures with Verifier-Local Revocation. In *ACM Conference on Computer and Communications Security*, pages 168–177. ACM, 2004.

[BSZ05] Mihir Bellare, Haixia Shi, and Chong Zhang. Foundations of Group Signatures: The Case of Dynamic Groups. In *CT-RSA*, LNCS 3376, pages 136–153. Springer, 2005.

[Cha82] David Chaum. Blind Signatures for Untraceable Payments. In *CRYPTO*, pages 199–203, 1982.

[Cha90] David Chaum. Showing Credentials without Identification Transfeering Signatures between Unconditionally Unlinkable Pseudonyms. In *AUSCRYPT*, LNCS 453, pages 246–264. Springer, 1990.

[CL01] Jan Camenisch and Anna Lysyanskaya. An Efficient System for Non-transferable Anonymous Credentials with Optional Anonymity Revocation. In *EUROCRYPT*, LNCS 2045, pages 93–118. Springer, 2001.

[CL02] Jan Camenisch and Anna Lysyanskaya. Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials. In *CRYPTO*, LNCS 2442, pages 61–76. Springer, 2002.

[CL04] Jan Camenisch and Anna Lysyanskaya. Signature Schemes and Anonymous Credentials from Bilinear Maps. In *CRYPTO*, LNCS 3152, pages 56–72. Springer, 2004.

[CvH91] David Chaum and Eugène van Heyst. Group Signatures. In *EUROCRYPT*, pages 257–265, 1991.

[Dam88] Ivan Damgård. Payment Systems and Credential Mechanisms with Provable Security Against Abuse by Individuals. In *CRYPTO*, LNCS 403, pages 328–335. Springer, 1988.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. In *Usenix Security Symposium*, pages 303–320, August 2004.

[Dou02] John R. Douceur. The Sybil Attack. In *IPTPS*, LNCS 2429, pages 251–260. Springer, 2002.

[EGM89] Shimon Even, Oded Goldreich, and Silvio Micali. On-Line/Off-Line Digital Schemes. In *CRYPTO*, LNCS 435, pages 263–275. Springer, 1989.

[FJS07] Joan Feigenbaum, Aaron Johnson, and Paul F. Syverson. A Model of Onion Routing with Provable Anonymity. In *Financial Cryptography*, LNCS 4886, pages 57–71. Springer, 2007.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

[HJP05] Yih-Chun Hu, Markus Jakobsson, and Adrian Perrig. Efficient Constructions for One-Way Hash Chains. In *ACNS*, volume LNCS 3531, pages 423–441, 2005.

[HS06] Jason E. Holt and Kent E. Seamons. Nym: Practical Pseudonymity for Anonymous Networks. Internet Security Research Lab Technical Report 2006-4, Brigham Young University, June 2006.

[JB99] Ari Juels and John G. Brainard. Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks. In *NDSS*. The Internet Society, 1999.

[JKTS07] Peter C. Johnson, Apu Kapadia, Patrick P. Tsang, and Sean W. Smith. Nymble: Anonymous IP-Address Blocking. In *Privacy Enhancing Technologies*, LNCS 4776, pages 113–133. Springer, 2007.

[KBC05] Tadayoshi Kohno, Andre Broido, and K. C. Claffy. Remote physical device fingerprinting. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 211–225, Washington, DC, USA, 2005. IEEE Computer Society.

[KTY04] Aggelos Kiayias, Yiannis Tsiounis, and Moti Yung. Traceable Signatures. In *EUROCRYPT*, LNCS 3027, pages 571–589. Springer, 2004.

[LRSW99] Anna Lysyanskaya, Ronald L. Rivest, Amit Sahai, and Stefan Wolf. Pseudonym Systems. In *Selected Areas in Cryptography*, LNCS 1758, pages 184–199. Springer, 1999.

[LSM06] B. N. Levine, C. Shields, and N. B. Margolin. A Survey of Solutions to the Sybil Attack. Technical Report Tech report 2006-052, University of Massachusetts Amherst, Oct 2006.

[Mic02] Silvio Micali. NOVOMODO: Scalable Certificate Validation and Simplified PKI Management. In *1st Annual PKI Research Workshop - Proceeding*, April 2002.

[NF05] Toru Nakanishi and Nobuo Funabiki. Verifier-Local Revocation Group Signature Schemes with Backward Unlinkability from Bilinear Maps. In *ASIACRYPT*, LNCS 3788, pages 533–548. Springer, 2005.

[Ngu05] Lan Nguyen. Accumulators from Bilinear Pairings and Applications. In *CT-RSA*, LNCS 3376, pages 275–292. Springer, 2005.

[RR98] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for Web Transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, November 1998.

[TAKS07] Patrick P. Tsang, Man Ho Au, Apu Kapadia, and Sean W. Smith. Blacklistable Anonymous Credentials: Blocking Misbehaving Users Without TTPs. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 72–81, New York, NY, USA, 2007. ACM.

[TAKS08] Patrick P. Tsang, Man Ho Au, Apu Kapadia, and Sean W. Smith. PEREA: Towards Practical TTP-Free Revocation in Anonymous Authentication. In *ACM Conference on Computer and Communications Security*, pages 333–344. ACM, 2008.

[TFS04] Isamu Teranishi, Jun Furukawa, and Kazue Sako. k-Times Anonymous Authentication (Extended Abstract). In *ASIACRYPT*, LNCS 3329, pages 308–322. Springer, 2004.

[TKS06] Patrick P. Tsang, Apu Kapadia, and Sean W. Smith. Anonymous IP-Address Blocking in Tor with Trusted Computing (Work-in-progress). In The Second Workshop on Advances in Trusted Computing (WATC '06 Fall), November 2006.

[TX03] Gene Tsudik and Shouhuai Xu. Accumulating Composites and Improved Group Signing. In *ASIACRYPT*, LNCS 2894, pages 269–286. Springer, 2003.

[vABHO06] Luis von Ahn, Andrew Bortz, Nicholas J. Hopper, and Kevin O'Neill. Selectively Traceable Anonymity. In *Privacy Enhancing Technologies*, LNCS 4258, pages 208–222. Springer, 2006.