Nymble: Blocking Misbehaving Users in Anonymizing Networks

Patrick P. Tsang, Apu Kapadia, *Member, IEEE,* Cory Cornelius, and Sean W. Smith

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.

Index Terms—anonymous blacklisting, privacy, revocation

1 Introduction

Anonymizing networks such as Tor [18] 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* [14], [17], [23], [28], users 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 anonymizing network.

Anonymous credential systems [10], [12] employ *group signatures*. Basic group signatures [1], [6], [15] allow servers to revoke a misbehaving user's anonymity by *complaining* to a *group manager*. Servers must query the group manager for every authentication, and thus lacks scalability. *Traceable signatures* [26] 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* [30] 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 worry about whether their 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 [8], [13], 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 [32].

With *dynamic accumulators* [11], [31], a revocation operation results in a new accumulator and public parameters for the group, and *all other existing users' credentials must be updated*, making it impractical. *Verifier-local revocation (VLR)* [2], [7], [9] 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,

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 [24]. This paper presents a significantly improved construction and a complete rewrite and evaluation of our (open-source) implementation (see Section 2.7 for a summary of changes).

- *Patrick Tsang, Cory Cornelius and Sean Smith are with the Department of Computer Science, Dartmouth College, Hanover NH, USA.*
- *Apu Kapadia is with Indiana University Bloomington, IN, USA.*

Manuscript received xxxx.

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

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 than VLR. We believe these low overheads will incentivize servers to adopt such a solution when weighed against the potential benefits of anonymous publishing (e.g., whistle-blowing, reporting, anonymous tip lines, activism, and so on.).[3]

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 [19] to make its deployment practical.

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.** 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.

2. In Boneh and Shacham's construction [7], computation at the server involves $3+2|BL|$ pairing operations, each of which takes tens of ms (see benchmarks at http://crypto.stanford.edu/pbc).

3. http://www.torproject.org/torusers.html.en

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.

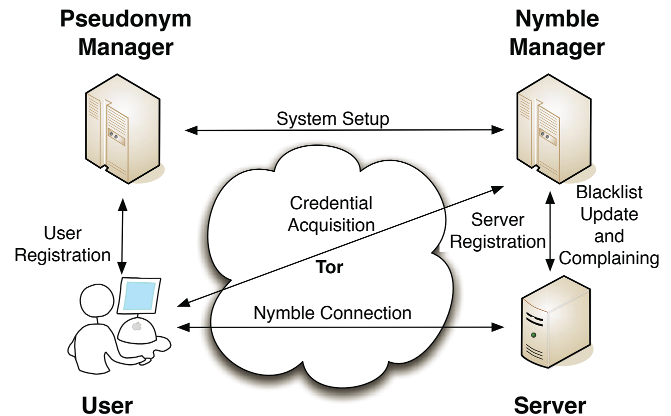


Fig. 1. The Nymble system architecture showing the various modes of interaction. Note that users interact with the NM and servers though the anonymizing network.

Some of the authors of this paper have published two anonymous authentication schemes, BLAC [33] and PEREA [34], 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.

We note that an extended version of this article is available as a technical report [16].

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

To limit the number of identities a user can obtain (called the *Sybil* attack [19]), 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, and trusted hardware. We address the practical issues related with resource-based blocking in Section 8, and suggest other alternatives for resources.

We do not claim to solve the Sybil attack. This problem is faced by *any credential system* [19], [27], 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 must 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 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). A user's requests to the NM are therefore pseudonymous, and nymbles are generated using the user's pseudonym and the server's identity. These nymbles are thus specific to a particular user-server pair. Nevertheless, as long as the PM and the NM do not collude, the Nymble system cannot identify which user is connecting to what server; the NM knows only the pseudonym-server pair, and the PM knows only the user identity-pseudonym pair.

To provide the requisite cryptographic protection and security properties, 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, 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 $w_1, w_2, \ldots$ 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, while longer time periods allow servers to 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 allows for *dynamism* since resources such as IP addresses can get reassigned and it is undesirable to blacklist such resources indefinitely, and it ensures *forgiveness* of misbehavior after a certain period of time. We assume all entities are time synchronized (for example, with time.nist.gov via the Network Time Protocol (NTP)), and can thus calculate the current linkability window and time period.

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.

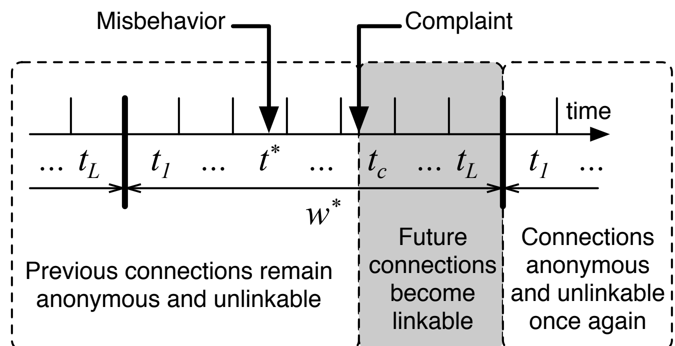


Fig. 2. The life cycle of a misbehaving user. If the server complains in time period t_c about a user's connection in t^*, the user becomes linkable starting in t_c. The complaint in t_c can include nymble tickets from only t_{c-1} and earlier.

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, once the server has complained about a user, that user is blacklisted for the rest of the day, for example (the linkability window). 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, thus providing backward unlinkability and subjective blacklisting.

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. It is of utmost importance, then, that users be notified of their blacklist status before they present a nymble ticket to a server. In our system, the user can download the server's blacklist and verify her status. If blacklisted, the user disconnects immediately.

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 discussed in Section 4.3.4, 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.

2.7 Summary of updates to the Nymble protocol

We highlight the changes to Nymble since our conference paper [24]. Previously, we had proved only the privacy properties associated with *nymbles* as part of a two-tiered hash chain. Here we prove security *at the protocol level*. This process gave us insights into possible (subtle) attacks against privacy, leading us to redesign our protocols and refine our definitions of privacy. For example, users are now either *legitimate* or *illegitimate*, and are anonymous within these sets (see Section 3). This redefinition affects how a user establishes a "Nymble connection" (see Section 5.5), and now prevents the server from distinguishing between users who have already connected in the same time period and those who are blacklisted, resulting in larger anonymity sets.

A thorough protocol redesign has also resulted in several optimizations. We have eliminated blacklist version numbers and users do not need to repeatedly obtain the current version number from the NM. Instead servers obtain proofs of freshness every time period, and users directly verify the freshness of blacklists upon download. Based on a hash-chain approach, the NM issues lightweight *daisies* to servers as proof of a blacklist's freshness, thus making blacklist updates highly efficient. Also, instead of embedding seeds, on which users must perform computation to verify their blacklist status, the NM now embeds a unique identifier $nymble^*$, which the user can directly recognize. Lastly, we have compacted several datastructures, especially the servers' blacklists, which are downloaded by users in each connection, and report on the various sizes in detail in Section 6. We also report on our open-source implementation of Nymble, completely rewritten as a C library for efficiency.

3 Security Model

Nymble aims for four security goals. We provide informal definitions here; a detailed formalism can be found in our technical report [16], 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.

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.

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

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. This property assumes each user has a single unique identity. When IP addresses are used as the identity, it is possible for a user to "frame" an honest user who later obtains the same IP address. Non-frameability holds true only against attackers with different *identities* (IP addresses).

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.

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. With regard to the PM and NM, Nymble makes several assumptions on *who* trusts *whom* to be *how* for *what* guarantee. We summarize 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 PM or NM can violate *Blacklistability* by issuing different pseudonyms or credentials to blacklisted users. A dishonest PM (resp. NM) can *frame* a user by issuing her the pseudonym (resp. credential) of another user who has already been blacklisted. To undermine the *Anonymity* of a user, a dishonest PM (resp. NM) can first impersonate the user by cloning her pseudonym (resp. credential) and then attempt to authenticate to a server—a successful attempt reveals that the user has already made a connection to the server during the time period. Moreover, by studying the

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

Fig. 3. *Who* trusts *whom* to be *how* for *what* guarantee.

complaint log, a curious NM can deduce that a user has connected more than once if she has been complained about two or more times. As already described in Section 2.3, the user must trust that at least the NM or PM is honest to keep the user and server identity pair private.

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 $\mathsf{L[Bob]}$ 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 and collision-resistant functions that resemble random oracles [5]. Denote the range of the hash functions by $\mathcal{H}$.
- Secure *message authentication* (MA) [3]. These consist of the key generation (MA.KeyGen), and the message authentication code (MAC) computation (MA.Mac) algorithms. Denote the domain of MACs by $\mathcal{M}$.
- Secure *symmetric-key encryption* (Enc) [4]. These consist 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) [22]. These consist 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

Nymble uses several important data structures:

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** $mac \stackrel{?}{=} \mathsf{MA.Mac}(nym||w, macKey_{NP})$

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.

As shown in Figure 4, *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.

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

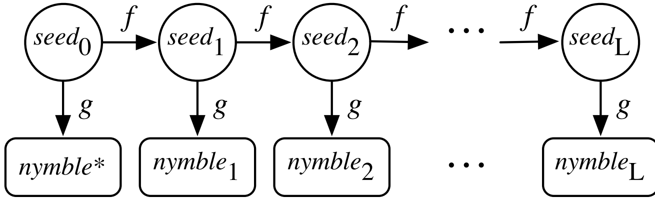


Fig. 4. Evolution of seeds and nymbles. Given $seed_i$ it is easy to compute $nymble_i, nymble_{i+1}, \ldots, nymble_L$, but not $nymble^*, nymble_1, \ldots, nymble_{i-1}$.

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

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. Upon a complaint, the NM extracts the user's seed and issues 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.

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

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 $macKey_{NS}, macKey_N, seedKey_N, encKey_N$ 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: **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** $mac_N \stackrel{?}{=} \mathsf{MA.Mac}(content, macKey_N)$

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.

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 [20], [29] to certify that "blacklist from time period t is still fresh."

For each complaint, the NM 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, $daisy_5 = h(daisy_6)$.

As outlined later (in Algorithm 13), $target$ is set to $daisy_t$. Now, until the next update to the blacklist, the

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** $(nymble^* \stackrel{?}{\in} blist)$

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 in a *linking-list*, and updates each token on the list every time period. When a user presents a nymble ticket, 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.

4.4 Communication channels

Nymble utilizes three types of communication channels, namely type-*Basic*, -*Auth* and -*Anon* (Figure 6).

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.

8. The Transport Layer Security Protocol Version 1.2. IETF RFC 5246.
9. While we acknowledge the existence of attacks on Tor's anonymity, we assume Tor provides perfect anonymity [21] for the sake of arguing Nymble's own anonymity guarantee.

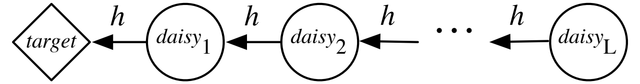


Fig. 5. Given $daisy_i$ it is easy to verify the freshness of the blacklist by applying h i times to obtain *target*. Only the NM can compute the next $daisy_{i+1}$ in the chain.

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** $cert := (t, daisy, t, mac, sig)$

5 Our Nymble Construction

5.1 System setup

During setup, 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. $macKey_{NP}$ 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.

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: $target := h^{(t_d - t_s)}(daisy)$
5: $content := sid||t_s||w||target||blist$
6: **return** 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** $mac \stackrel{?}{=}$ MA.Mac$(content, macKey_N)$

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. 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 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.

1) The user extracts $pnym$ from $usrState$ and sends the pair $(pnym, sid)$ to the NM.

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

Fig. 6. 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: $encKey_N, decKey_N, signKey_N, verKey_N)$
8: $nmEntries := \emptyset$
9: **return** $nmState := (keys, nmEntries)$

2) The NM reads the current linkability window as w_{now}. It makes sure the user's $pnym$ is valid: If

   $$\texttt{NMVerifyPseudonym}_{nmState}(pnym, w_{now})$$

   returns **false**, the NM terminates with failure; it proceeds otherwise.
3) The NM runs

   $$\texttt{NMCreateCredential}_{nmState}(pnym, sid, w_{now}),$$

   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.

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

   $$\texttt{VerifyBL}_{usrState}(sid, t_{now}^{(U)}, w_{now}^{(U)}, blist, cert) = \textbf{true}.$$

   If not, she terminates the protocol with failure.

Algorithm 11 `NMRegisterServer`

Input: $(sid, t, w) \in \mathcal{H} \times \mathbb{N}^2$
Persistent state: $nmState \in \mathcal{S}_N$
Output: $svrState \in \mathcal{S}_S$
1: $(keys, nmEntries) := nmState$
2: $macKey_{NS} :=$ Mac.KeyGen()
3: $daisy_L \in_R \mathcal{H}$
4: $nmEntries' := nmEntries || (sid, macKey_{NS}, daisy_L, t)$
5: $nmState := (keys, nmEntries')$
6: $target := h^{(L-t+1)}(daisy_L)$
7: $blist := \emptyset$
8: $cert :=$ $\texttt{NMSignBL}_{nmState}(sid, t, w, target, blist)$
9: $svrState := (sid, macKey_{NS}, blist, cert, \emptyset, \emptyset, \emptyset, t)$
10: **return** $svrState$

Algorithm 12 `ServerLinkTicket`

Input: $ticket \in \mathcal{T}$
Persistent state: $svrState \in \mathcal{S}_S$
Output: $b \in \{$**true**, **false**$\}$
1: Extract $lnkng\text{-}tokens$ from $svrState$
2: $(\cdot, nymble, \cdots) := ticket$
3: **for all** $i = 1$ to $|lnkng\text{-}tokens|$ **do**
4:   **if** $(\cdot, nymble) = lnkng\text{-}tokens[i]$ **then**
5:     **return** **true**
6: **return** **false**

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

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

then it is unsafe for the user to proceed and the user sets $safe$ to **false** and terminates the protocol with failure.[10]

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 $usrState$.
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 $\langle$goodbye$\rangle$ to the user and terminates with failure. Otherwise, it adds $ticket$ to $slist$ in its state, sends $\langle$okay$\rangle$ to the user and terminates with success.
4) On receiving $\langle$okay$\rangle$, the user terminates in success.

10. We note that a nymble-authenticated *session* may be long-lived, where actions within the session are linkable. It is the establishment of multiple sessions within a time period that is disallowed.

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 $cmplnt\text{-}tickets$ 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. When there is no complaint (i.e., the server's $cmplnt\text{-}tickets$ 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. Since these updates are certified for integrity and freshness at the granularity of time periods, multiple updates

within a single time period are disallowed (otherwise servers could send users stale blacklists).

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 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 $(blist, cert, cmplnt\text{-}tickets)$ 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 $cmplnt\text{-}tkts$ to $\emptyset$. For each $seed \in seeds$, the server creates a $token$ as $(seed, g(seed))$ and appends it to $lnkng\text{-}tokens$. Finally, the server terminates with success.

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 7–12). Finally, the algorithm prepares an answer to the update request by invoking `NMComputeBLUpdate` and `NMComputeSeeds` (see Algorithm 14) (lines 13–16).

`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.

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: $daisy'_L \in_R \mathcal{H}$
11: $target' := h^{(L-t+1)}(daisy'_L)$
12: $cert' := \texttt{NMSignBL}(sid, t, w, target', blist||blist')$
13: Replace $daisy_L$ and t_{lastUpd} in $nmEntries[sid]$ in $nmState$ with $daisy'_L$ and by t respectively
14: **return** $(blist', cert')$

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: $seeds \in \mathcal{H}^m$

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

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 7)

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

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

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

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: $(blist', cert', seeds) \in \mathcal{B}_m \times \mathcal{C} \times \mathcal{H}^m$

1: Extract $time_{\text{lastUpd}}$ from $nmEntries[sid]$ in $nmState$
2: $b_1 := (time_{\text{lastUpd}} < t)$
3: $b_2 :=$
4: $\quad$ $\texttt{NMVerifyBL}_{nmState}(sid, time_{\text{lastUpd}}, w, blist, cert)$
5: **if** $\neg(b_1 \wedge b_2)$ **then**
6: $\quad$ **return** $\perp$
7: **for all** $i = 1$ to m **do**
8: $\quad$ $ticket := cmplnt\text{-}tickets[i]$; $(\tilde{t}, \cdots) := ticket$
9: $\quad$ $b_{i1} := \tilde{t} < t$
10: $\quad$ $b_{i2} := \texttt{NMVerifyTicket}_{nmState}(sid, \tilde{t}, w, ticket)$
11: $\quad$ **if** $\neg(b_{i1} \wedge b_{i2})$ **then**
12: $\quad\quad$ **return** $\perp$
13: $(blist', cert') :=$
14: $\quad$ $\texttt{NMComputeBLUpdate}_{nmState}(sid, t, w, blist, cert)$
15: $seeds :=$
16: $\quad$ $\texttt{NMComputeSeeds}_{nmState}(t, blist, cmplnt\text{-}tickets)$
17: **return** $(blist', cert', seeds)$

Algorithm 16 `ServerUpdateState`

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

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

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 Performance Evaluation

We implemented Nymble and collected various empirical performance numbers, which verify the linear (in the number of "entries" as described below) time and space costs of the various operations and data structures.

6.1 Implementation and experimental setup

We implemented Nymble as a C++ library along with Ruby and JavaScript bindings. 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 for all the cryptographic primitives.

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

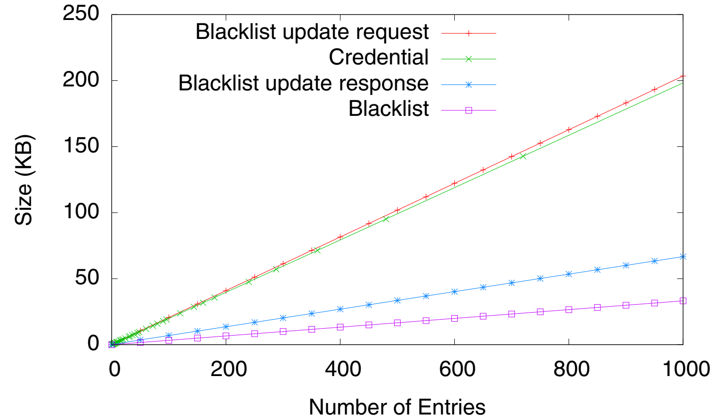


Fig. 7. The marshaled size of various Nymble data structures. The X-axis refers to the number of entries—complaints in the blacklist update request, tickets in the credential, tokens and seeds in the blacklist update response, and nymbles in the blacklist.

Sig. We chose RSA over DSA for digital signatures because of its faster verification speed—in our system, verification occurs more often than signing.

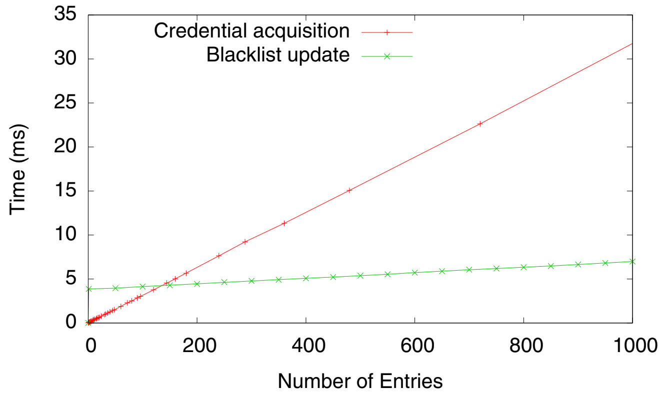
We evaluated our system on a 2.2 GHz Intel Core 2 Duo Macbook Pro with 4 GB of RAM. The PM, the NM, and the server were implemented as Mongrel (Ruby's version of Apache) servers. The user portion was implemented as a Firefox 3 extension in JavaScript with XPCOM bindings to the Nymble C++ library. For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

6.2 Experimental results

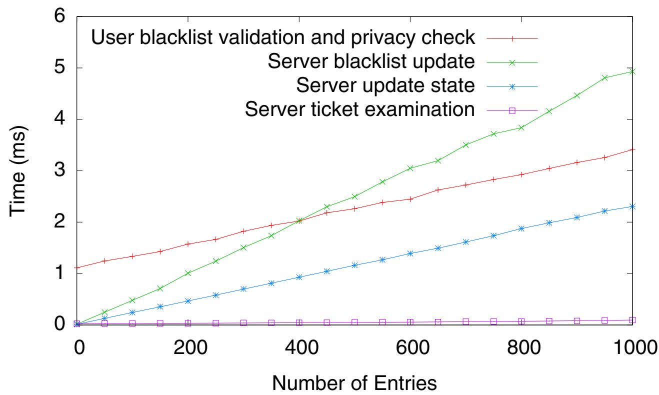
Figure 7 shows the size of the various data structures. The X-axis represents the number of entries in each data structure—complaints in the blacklist update request, tickets in the credential (equal to L, the number of time periods in a linkability window), nymbles in the blacklist, tokens and seeds 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 $L = 288$.[11] 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, whereas the size of a blacklist update response for 50 complaints 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

11. The security-related tradeoffs for this parameter are discussed in Section 2.4. The performance tradeoffs are apparent in this section (see Figures 7 and 8(a) for credential size and acquisition times).



(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.

Fig. 8. Nymble's performance at (a) the NM and (b) the user and the server when performing various protocols.

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 8(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 8(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 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. Every time period (e.g., every 5 minutes), 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.

To measure the latency perceived by an authenticating user, we simulated a client authenticating to a server with 500 blacklist entries. We simulated two scenarios, with the PM, NM and server (a) on the local network and (b) on a remote machine (48 ms round trip time).[12] On average it took a total of 470 ms for the full protocol on the local network and 2001 ms for the remote case: acquiring a pseudonym (87 ms local; 307 ms remote) and credential (107 ms; 575 ms), acquiring the blacklist and the server checking if the user is blacklisted (179 ms; 723 ms), and finally authenticating (97 ms; 295 ms).[13]

7 Security Analysis

Theorem 1: Our Nymble construction has *Blacklistability*, *Rate-limiting*, *Non-frameability* and *Anonymity* provided that the trust assumptions in Section 3.2 hold true, and the cryptographic primitives used are secure. □

We summarize the proof of Theorem 1. Please refer to our technical report [16] for a detailed version.

7.1 Blacklistability

An honest PM and NM will issue a coalition of c unique users at most c valid credentials for a given server. Because of the security of HMAC, only the NM can issue valid tickets, and for any time period the coalition has at most c valid tickets, and can thus make at most c connections to the server in any time period regardless of the server's blacklisting. It suffices to show that if each of the c users has been blacklisted in some previous time period of the current linkability window, the coalition cannot authenticate in the current time period k^*.

Assume the contrary that connection establishment k^* using one of the coalition members' $ticket^*$ was successful even though the user was blacklisted in a previous time period k'. Since connection establishments k' and k^* were successful, the corresponding tickets $ticket'$ and $ticket^*$ must be valid. Assuming the security of digital signatures and HMAC, an honest server can always contact an honest NM with a valid ticket and the NM will successfully terminate during the blacklist update. Since the server blacklisted the valid $ticket'$ and updates its linking list honestly, the `ServerLinkTicket` will return fail on input $ticket^*$, and thus the connection k^* must fail, which is a contradiction.

12. The remote machine was a 256 slice from Slicehost running Ubuntu 4.09, and provide an indication of performance on the network.

13. For perspective, we note that with SSL disabled the total times were 65 ms on the local network and 1086 ms for the remote case.

7.2 Non-Frameability

Assume the contrary that the adversary successfully framed honest user i^* with respect to an honest server in time period t^*, and thus user i^* was unable to connect in time period t^* using $ticket^*$ even though none of his tickets were previously blacklisted. Because of the security of HMAC, and since the PM and NM are honest, the adversary cannot forge tickets for user i^*, and the server cannot already have seen $ticket^*$; it must be that $ticket^*$ was linked to an entry in the linking list. Thus there exists an entry $(seed^*, nymble^*)$ in the server's linking list, such that the nymble in $ticket^*$ equals $nymble^*$. The server must have obtained this entry in a successful blacklist update for some valid $ticket_b$, implying the NM had created this ticket for some 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. Thus we have $\tilde{i} = i^*$. But as already argued, the adversary cannot forge i^*'s $ticket_b$, and it must be the case that i^*'s $ticket_b$ was blacklisted before t^*, which contradicts our assumption that i^* was a legitimate user in time t^*.

7.3 Anonymity

We show that an adversary learns only that *some* legitimate user connected or that *some* illegitimate user's connection failed, i.e., there are two anonymity sets of legitimate and illegitimate users.

Distinguishing between two illegitimate users We argue that any two chosen illegitimate users out of the control of the adversary will react indistinguishably. Since 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 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).

For an illegitimate user (attempting a new connection) who has already disclosed a ticket during a connection establishment earlier in the same time period, $ticketDisclosed$ for the server will have been set to **true** and $safe$ is evaluated to **false** during establishment k^*.

An illegitimate user who has not disclosed a ticket during the same time period must already be blacklisted. Thus the server complained about some previous $ticket^*$ of the user. Since the NM is honest, the user's $nymble^*$ appears in some previous blacklist of the server. Since an honest NM never deletes entries from a blacklist, it will appear in all subsequent blacklists, and $safe$ is evaluated to **false** for the current blacklist. Servers cannot forge blacklists or present blacklists for earlier time periods (as otherwise the digital signature would be forgeable, or the hash in the daisy chain could be inverted).

Distinguishing between two legitimate users The authenticity of the channel implies that a legitimate user knows the correct identity of the server and thus boolean $ticketDisclosed$ for the server remains **false**. Furthermore, `UserCheckIfBlacklisted` returns **false** (assuming the security of digital signatures) and $safe$ is evaluated to **true** for the legitimate user.

Now, in the ticket presented by the user, only $nymble$ and $ctxt$ are functions of 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 the user's identity to the adversary. Finally, since the server has not obtained any seeds for the user, under the Random Oracle model the $nymble$ presented by the user is indistinguishable from random and cannot be linked with other $nymble$s presented by the user. Furthermore, if and when the server complains about a user's $ticket$s in the future, the NM ensures that only one real seed is issued (subsequent seeds corresponding to the same user are random values), and thus the server cannot distinguish between legitimate users for a particular time period by issuing complaints in a future time period.

7.4 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.

8 Discussion

IP-address blocking By picking IP addresses as the resource for limiting the Sybil attack, our current implementation closely mimics IP-address blocking employed by Internet services. There are, however, some inherent limitations to using IP addresses as the scarce resource. If a user can obtain multiple addresses she can circumvent both nymble-based and regular IP-address blocking. Subnet-based blocking alleviates this problem, and while it is possible to modify our system to support subnet-based blocking, new privacy challenges emerge; a more thorough description is left for future work.

Other resources Users of anonymizing networks 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 [25] and e-cash, where users are required to perform a certain amount of computation or pay money to acquire a credential. 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 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. 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.

9 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.

REFERENCES

[1] G. Ateniese, J. Camenisch, M. Joye, and G. Tsudik. A Practical and Provably Secure Coalition-Resistant Group Signature Scheme. In *CRYPTO*, LNCS 1880, pages 255–270. Springer, 2000.

[2] G. Ateniese, D. X. Song, and G. Tsudik. Quasi-Efficient Revocation in Group Signatures. In *Financial Cryptography*, LNCS 2357, pages 183–197. Springer, 2002.

[3] M. Bellare, R. Canetti, and H. Krawczyk. Keying Hash Functions for Message Authentication. In *CRYPTO*, LNCS 1109, pages 1–15. Springer, 1996.

[4] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A Concrete Security Treatment of Symmetric Encryption. In *FOCS*, pages 394–403, 1997.

[5] M. Bellare and P. 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.

[6] M. Bellare, H. Shi, and C. Zhang. Foundations of Group Signatures: The Case of Dynamic Groups. In *CT-RSA*, LNCS 3376, pages 136–153. Springer, 2005.

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

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

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

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

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

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

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

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

[15] D. Chaum and E. van Heyst. Group Signatures. In *EUROCRYPT*, pages 257–265, 1991.

[16] C. Cornelius, A. Kapadia, P. P. Tsang, and S. W. Smith. Nymble: Blocking Misbehaving Users in Anonymizing Networks. Technical Report TR2008-637, Dartmouth College, Computer Science, Hanover, NH, December 2008.

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

[18] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *Usenix Security Symposium*, pages 303–320, Aug. 2004.

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

[20] S. Even, O. Goldreich, and S. Micali. On-Line/Off-Line Digital Schemes. In *CRYPTO*, LNCS 435, pages 263–275. Springer, 1989.

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

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

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

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

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

[26] A. Kiayias, Y. Tsiounis, and M. Yung. Traceable Signatures. In *EUROCRYPT*, LNCS 3027, pages 571–589. Springer, 2004.

[27] 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.

[28] A. Lysyanskaya, R. L. Rivest, A. Sahai, and S. Wolf. Pseudonym Systems. In *Selected Areas in Cryptography*, LNCS 1758, pages 184–199. Springer, 1999.

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

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

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

[32] I. Teranishi, J. Furukawa, and K. Sako. *k*-Times Anonymous Authentication (Extended Abstract). In *ASIACRYPT*, LNCS 3329, pages 308–322. Springer, 2004.

[33] P. P. Tsang, M. H. Au, A. Kapadia, and S. 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.

[34] P. P. Tsang, M. H. Au, A. Kapadia, and S. W. Smith. PEREA: Towards Practical TTP-Free Revocation in Anonymous Authentication. In *ACM Conference on Computer and Communications Security*, pages 333–344. ACM, 2008.

Patrick Tsang is a Ph.D. candidate in the Computer Science department at Dartmouth College. His research covers security and trust in computers and networks, privacy-enhancing technologies and applied cryptography. He is particularly interested in developing scalable security and privacy solutions for resource-constrained environments. He is a member of the International Association for Cryptologic Research (IACR).

Dr. Apu Kapadia received his Ph.D. from the University of Illinois at Urbana-Champaign and received a four-year High-Performance Computer Science Fellowship from the Department of Energy for his dissertation research. Following his doctorate, Apu joined Dartmouth College as a Post-Doctoral Research Fellow with the Institute for Security Technology Studies (ISTS), and then as a Member of Technical Staff at MIT Lincoln Laboratory. He joined Indiana University Bloomington as an Assistant Professor in the School of Informatics and Computing in 2009. Apu is an active researcher in security and privacy and is particularly interested in anonymizing networks, usable security, security in peer-to-peer and mobile networks, and applied cryptography. He is a member of IEEE and ACM.

Cory Cornelius is a Ph.D. student in the Computer Science department at Dartmouth College. His research interests are security, privacy, networks, and, in particular, the intersection of these fields.

Prof. Sean W. Smith has been active in information security since before there was a Web. Previously a research scientist at Los Alamos National Lab and at IBM, he joined Dartmouth in 2000. Sean has published two books; been granted over a dozen patents; and advised over three dozen Ph.D., M.S., and senior honors theses. Sean was educated at Princeton and CMU, and is a member of Phi Beta Kappa and Sigma Xi.