Improving Wireless Privacy with an Identifier-Free Link Layer Protocol

Ben Greenstein[†] Damon McCoy[*] Jeffrey Pang[‡]
Tadayoshi Kohno[§] Srinivasan Seshan[‡] David Wetherall[†§]

[†]Intel Research Seattle [*]University of Colorado [‡]Carnegie Mellon University [§]University of Washington

benjamin.m.greenstein@intel.com damon.mccoy@colorado.edu jeffpang@cs.cmu.edu
yoshi@cs.washington.edu srini@cmu.edu david.wetherall@intel.com

ABSTRACT

We present the design and evaluation of an 802.11-like wireless link layer protocol that obfuscates all transmitted bits to increase privacy. This includes explicit identifiers such as MAC addresses, the contents of management messages, and other protocol fields that the existing 802.11 protocol relies on to be sent in the clear. By obscuring these fields, we greatly increase the difficulty of identifying or profiling users from their transmissions in ways that are otherwise straightforward. Our design, called *SlyFi*, is nearly as efficient as existing schemes such as WPA for discovery, link setup, and data delivery despite its heightened protections; transmission requires only symmetric key encryption and reception requires a table lookup followed by symmetric key decryption. Experiments using our implementation on Atheros 802.11 drivers show that *SlyFi* can discover and associate with networks faster than 802.11 using WPA-PSK. The overhead *SlyFi* introduces in packet delivery is only slightly higher than that added by WPA-CCMP encryption (10% vs. 3% decrease in throughput).

Categories and Subject Descriptors

C.2 [**Computer Systems Organization**]: Computer-Communication Networks; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Design, Security

Keywords

privacy, anonymity, wireless, 802.11

1. INTRODUCTION

Wireless capabilities are rapidly spreading beyond mobile computers to everyday consumer devices ranging from cellphones and personal health monitors to game controllers and digital cameras. This evolving wireless ecosystem is increasingly pervasive and personal in its usage compared to laptops, and it heightens privacy risks that are already significant compared to wired networks. Wireless links are more exposed than their wired counterparts because transmissions are broadcast and can be received by anyone within radio range. Without sophisticated wiretapping hardware or access to network cables, third parties that are not intended recipients may eavesdrop on conversations using only commodity radios and off-the-shelf software. To counter this threat, it is common practice to use standards such as WPA for 802.11 to encrypt packet contents.

Unfortunately, even the best practices for data confidentiality as used in 802.11 today leave users vulnerable to straightforward attacks on their privacy that reveal who they are and what they are doing to any party within radio range. For example, it is well-known that MAC addresses, which are sent in the clear with existing encryption schemes, serve as handles that can be used to identify users and track their locations [16, 18]. Even without MAC addresses, recent work has demonstrated that users can often be identified and linked with confidential information such as their location history using management information that is sent in the clear [15, 22]. More generally, 802.11 facilitates user tracking and inventorying attacks that are conceptually identical to RFID threats [19], which have prompted much public concern over privacy.

These attacks work because third parties can observe low-level identifiers such as addresses and network names in transmissions that map readily to high-level identifiers such as identities. The key problem that this paper addresses is the complete removal of explicit identifiers from wireless transmissions for parties other than the intended recipients; existing schemes obscure identifiers only within packet payloads. By removing all identifiers, privacy is strengthened because attacks that rely on explicit identifiers will fail.

To combat these attacks, other research has proposed periodically changing specific identifiers, e.g., that MAC addresses be changed every session or when idle [16, 17]. But concealing specific fields leaves open the possibility of tracking and inventorying by other fields that have not been protected. Furthermore, even within a session, sequences of encrypted packets, which remain linked by an explicit and consistent address, can reveal sensitive information about their contents. For example, a distinct pattern of packet

MobiSys'08, June 17–20, 2008, Breckenridge, Colorado, USA.

sizes and timings is sometimes sufficient to identify the keys a user types [28], the web pages he views [30], the videos he watches [26], the languages he speaks [33], and the applications he runs [34]. As far as we are aware, our work is the first to provide a complete design and prototype for removing all explicit identifiers from wireless links.

The obvious difficulty with simply removing identifiers is that they play key roles in the efficient operation of existing protocols. For example, a destination address (802.11) or connection identifier (WiMAX) allows a device to decide whether it is the destination of a message by using a simple compare operation. Mechanisms such as ARP that translate between addresses at different layers rely on identifiers that persist for significant periods of time. And the process of service discovery and rendezvous, such as with an available access point (AP), requires a service to announce its existence with an explicit, recognizable identifier or for a client to probe for it. Either way, the process relies on the transmission of a service name explicitly.

This paper presents *SlyFi*, an 802.11-like protocol that encrypts entire packets to remove explicit identifiers while retaining efficiency comparable to 802.11 with WPA. No explicit information in *SlyFi* messages can be used by third parties to link them together. We show that all features that rely on identifiers—service discovery, packet filtering, and address binding—can be supported without exposing them. Different mechanisms are used for service discovery and subsequent data transfers, but in both cases a device can determine whether it is the recipient of a message with lightweight table look-ups. We have implemented *SlyFi* on commodity 802.11 NICs and our experiments show that *SlyFi*'s performance impact is modest. In particular, we show that a *SlyFi* client can discover and associate with services even faster than 802.11 with WPA using PSK authentication. *SlyFi*'s overhead results in a throughput degradation that is only slightly greater than that of WPA with CCMP encryption (10% vs. 3%).

The rest of this paper is organized as follows. §2 presents the requirements of a solution and an overview of *SlyFi*. §3 presents the design of two main mechanisms it uses, while §4 discusses practical details and our prototype implementation. §5 presents performance evaluation results. §6 discusses related work and §7 concludes.

2. PROBLEM AND SOLUTION OVERVIEW

Our goal is to build a wireless link layer protocol that allows clients and services to communicate without exposing identifiers to third parties. This section outlines the threat model we consider. We then discuss our security requirements and the challenges in meeting them and present an overview of *SlyFi*, an efficient identifier-concealing link layer protocol based on 802.11.

2.1 Threat Model

Attack. The previous section outlined three types of attacks enabled by low-level identifiers not obscured by existing security mechanisms: the inventorying, tracking, and profiling of users and their devices. Users can be subjected to these attacks without their knowledge because an adversary can carry them out without being visibly or physically present. In addition, users are vulnerable even when using the best existing security practices, such as WPA. Thus, these attacks violate common assumptions about privacy. The effectiveness of these attacks is dependent on an adversary's ability to link packets sent at different times to the same device. The easiest way for adversaries to link packets is by observing the same low-level identifier in each.

	10 ms	100 ms	1 sec	1 min	1 hr
SIGCOMM 2004	1.4	3.2	7.6	24.7	80.1
OSDI 2006	4.6	9.0	20.6	60.8	221.3
UCSD 2006	2.4	7.1	17.9	76.6	176.6

Table 1—Mean number of devices that send or receive 802.11 data packets at different time intervals at two conferences (SIGCOMM [25], OSDI [11]) and one office building (UCSD [12]). Intervals with no data packets are ignored. UCSD has observations from multiple monitors.

Thus, our goal is to limit two forms of *linkability*: First, information should not be provided in individual packets that explicitly links the packets to the identities of the sender or intended receiver. Second, to prevent the profiling, fingerprinting, and tracking of sequences of related packets, packets from the same sender should not be linkable to each other, irrespective of whether any one of them may be linked explicitly to its source. In other words, when there are k potential devices and an adversary observes a packet, he should only be able to infer that the packet is from (or to) one of those k devices, not which one. Profiling a device's packet sequences would be more difficult even at short timescales if many devices are active simultaneously. Table 1, which shows the average number of active devices observed at different time intervals, shows that there are indeed many simultaneously active devices in three 802.11 traces.

Potential Victims. The aforementioned attacks are damaging to both wireless clients, such as laptops, and wireless services, such as APs, particularly since the distinction between client and service devices is becoming increasingly blurred; e.g., a client game station sometimes provides wireless service to others as an ad hoc AP. Thus, we want to limit the linkability of packets transmitted by both clients and services.

We assume that clients and services have (possibly shared) cryptographic keys prior to communication. These keys can be obtained in the same way as in existing secure 802.11 and Bluetooth networks. For example, devices can leverage traditional credentials from trusted authorities (e.g., for RADIUS authentication) or bootstrap symmetric keys using out-of-band pairing techniques [31]. We believe that most private services will be known beforehand (e.g., a home 802.11 AP) and can bootstrap keys using these methods. Nonetheless, in previous work [22] we also proposed methods to privately bootstrap keys with unknown services by leveraging transitive trust relationships.

The mere possession of cryptographic keys does not immediately yield satisfactory solutions, however, as clients and services have limited computational resources. As a consequence, solutions should not enable denial of service attacks that exploit this limitation. For example, simply encrypting the entirety of a packet is not sufficient if a receiver can not quickly determine whether it is the intended recipient or not. This is because an adversary would then be able to exhaust a device's computational resources by broadcasting "junk" packets that the device would expend a non-trivial amount of resources to discard.

Adversary. We are concerned with limiting the packet linking ability of *third parties*, i.e., parties other than the original sender or intended recipient of those packets. For example, packets sent between an 802.11 client and an 802.11 AP are exposed to anyone within radio range, but only the client and service should be able to link them together. We are not concerned with preventing the service from linking together the client's packets (or vice versa), as techniques used to hide a client's identity from a service in wired networks (e.g., [14]) are also applicable in wireless networks.

We assume adversaries have commodity 802.11 radios and are able to observe all transmitted packets, but they are not privy to the cryptographic keys that clients and services have prior to communication. As with most practical systems, we assume that adversaries are computationally bounded and thus can not successfully attack standard cryptosystems such as RSA, ElGamal, and AES.

Limitations. *SlyFi*'s removal of low-level identifiers makes it much more difficult for third parties to link packets together or to a particular user, thus improving privacy. Nonetheless, packet sizes, packet timings, and physical layer information may still sometimes act as side channels that link packets together. Defending against these attacks is outside the scope of this paper. However, without explicit identifiers linking together packets, it becomes a more difficult probabilistic task to separate the transmissions of different sources. Such attacks are less accessible as they usually require sophisticated attackers [32] or non-commodity hardware [23].

2.2 Security Requirements

We want to be able to deliver a message from A to B without identifiers, but still ensure that B can verify it was sent by A. More formally, consider a procedure F that computes $c \leftarrow F(A, B, p)$, where A and B are the identities of the sender and recipient, respectively, p is the original message payload, and c is the result which A transmits. (Shared cryptographic key state is an additional, implicit input to F, but we omit it here for brevity.) We want F to have the following four properties. We denote security properties in this paper using small caps.

STRONG UNLINKABILITY. To protect against tracking and profiling attacks, a sequence of packets should not be linkable. More formally, any party other than A or B that receives $c_1 = F(A, B, p_1)$ and $c_2 = F(A, B, p_2)$ should not be able to determine that the sender or receiver of c_1 or c_2 are the same. In particular, this implies that c_1 and c_2 must not contain consistent identifiers. We note that some packet types, such as discovery messages, are less vulnerable to short-term profiling and thus only need to be unlinkable at coarser timescales to prevent long-term tracking. Consequently, we outline a relaxed version of this property in §3.3 to efficiently handle these packets.

AUTHENTICITY. To restrict the discovery of services to authorized clients and prevent spoofing and man-in-the-middle attacks, recipients should be able to verify a message's source. More formally, B should be able to verify that A was the author of c and that it was constructed recently (to prevent replay attacks).

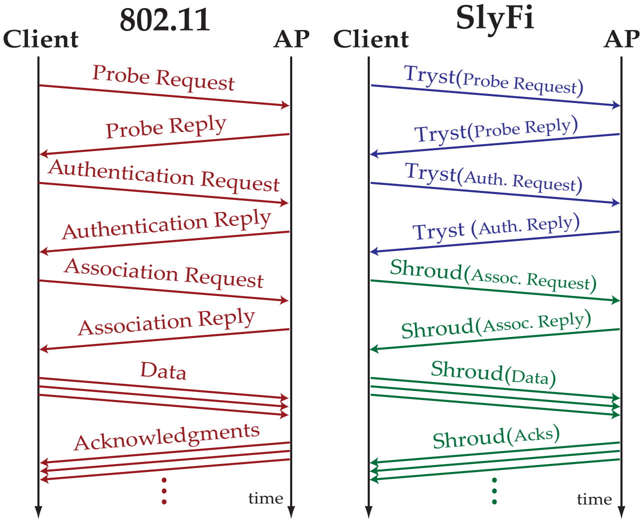

Figure 1—The SlyFi protocol.

CONFIDENTIALITY. No party other than A or B should be able to determine the contents of p. In contrast to existing wireless confidentiality schemes, not even fields and addresses in the header should be decipherable by third parties.

MESSAGE INTEGRITY. Finally, as with existing 802.11 security schemes, receivers should be able to detect if messages were tampered with by third parties. More formally, B should be able to derive p from c and verify that it was not altered after transmission.

2.3 Challenges

The principal approach to concealing 802.11 client identities has been to use MAC address pseudonyms [16, 18]. Pseudonym proposals do not meet our strong unlinkability requirement because all packets sent under one pseudonym are trivially linkable. Moreover, the use of pseudonyms does not conceal other information in headers, such as capabilities, that can be used to link packets together [21]. Furthermore, the proposals focus on data delivery alone, and do not address important network functions, such as authentication and service discovery.

Prior approaches are limited because meeting all our security requirements while maintaining important wireless functionality is nontrivial. Consistent destination addresses allow devices to quickly filter messages intended for others so efficient data transport is difficult without them. Moreover, cryptographic authenticity is difficult to provide without identifiers. Message recipients typically need to know which cryptographic key to use to verify a message, and it is hard to tell the recipient which one without explicitly identifying it. Finally, removing identifiers completely from the process of service discovery is hard because wireless clients and services typically rendezvous by broadcasting an agreed upon identifier. A service might be willing to expose its identifier through announcements to save potential clients from having to expose it in probes. No such straightforward solution exists to conceal both client and service identities.

2.4 System Overview

In light of the shortcomings of existing solutions, we introduce the *SlyFi* protocol that meets our security requirements using two identity-concealing mechanisms, *Tryst* and *Shroud*, while providing functionality similar to 802.11. Before describing these mechanisms, we first give an overview of *SlyFi* in this section.

The *SlyFi* link layer is designed to replace 802.11 for managed wireless connectivity between clients and APs. The privacy protecting mechanisms of the protocol explicitly protect all bits transmitted by the link layer. A client wishing to join and send data to a *SlyFi* network sends a progression of messages similar to 802.11 (Figure 1). Instead of sending these messages in the clear, they are encapsulated by the two identity-hiding mechanisms we describe in §3.

A client first transmits probes, encapsulated by Tryst, to discover nearby APs it is authorized to use. A probe is encrypted such that: 1) only the client and the networks named in the probe can learn the probe's source, destination, and contents, and 2) messages encapsulated for a particular *SlyFi* AP sent at different times cannot be linked by their contents. An AP that receives a probe verifies that it was created by an authorized user and sends an encrypted reply, indicating its presence to that client. If the client wishes to establish a link to the AP, it sends an authentication request, also encapsulated by Tryst, containing session information including keys for subsequent data transmission, which are used to bootstrap Shroud. Obviously, *SlyFi* APs cannot send clear-text beacons if they wish to protect service identities. However, they may do so if they wish to announce themselves publicly. Such a public announcement could immediately be followed by a confidential authentication request from an interested client, and thus would not compromise client privacy.

After a link has been established by an authentication response, Shroud is used to conceal the addresses and contents of future messages delivered on the link. An eavesdropper can not use the contents of any two messages protected by Shroud to link them to the same sender or receiver.

Both Tryst and Shroud essentially encrypt the entire contents of each message, including addresses normally found in the header. The essential differences between them arise due to the different requirements of discovery, link establishment, and data transfer.

3. IDENTIFIER-FREE MECHANISMS

Identifiers are used in wireless protocols for two general functions: 1) as a handle by which to discover a service and establish a link to it, and 2) to address packets on a link and allow unintended recipients to ignore packets efficiently. Tryst and Shroud address each of these functions, respectively. To motivate our mechanisms, we first describe two straw man mechanisms that meet our security requirements, but are inefficient. We then discuss Tryst and Shroud, which are enabled by minor relaxations of these requirements or additional assumptions made possible by their intended uses. We conclude the section by discussing how *SlyFi* can still support other protocol functions, such as higher layer binding.

To illustrate each mechanism we consider the scenario when A sends a message p to B. Each mechanism consists of three key elements: the *bootstrapping* of cryptographic keys that the sender and receiver require to compute the procedure F (described in §2.2); the *construction* of $c \leftarrow F(A, B, p)$ by the sender; and the *message filtering* by a receiver to determine if c is intended for him.

3.1 Straw Man: Public Key Mechanism

We first sketch public key, a mechanism based on a protocol that Abadi and Fournet [7] prove meet the aforementioned security requirements.

Bootstrapping. This mechanism assumes that A and B each have a public/private key pair and each have the public keys of the other.

Construction. We sketch this mechanism here, but refer the reader to the first protocol discussed in [7] for details. To provide authenticity, A digitally signs the statement $s = \{A, B, T\}$ where T is the current time. A message header is constructed as an encryption of s and the digital signature, using B's public key. By using a public key encryption scheme that does not reveal which key is used, such as ElGamal [9], identities of neither sender nor intended recipient are revealed.[1] In addition, this achieves strong unlinkability because the ElGamal encryption scheme is randomized so each encrypted header appears random. The payload can be encrypted via conventional means (e.g., as described later in §3.3).

Message filtering. When B receives a message, he will attempt to decrypt this header. If the decryption fails (i.e., the result does not include the statement $\{j, B, T\}$, for a known identity j), the message is not intended for B and can be discarded. If decryption succeeds, B then checks the signature and the time to verify that the message was recently generated by j before accepting it.

Although this protocol achieves the security properties we desire, it is slow because it uses public key cryptography. In particular, on AP and consumer electronics hardware, a single private key decryption can take over 100 milliseconds—several orders of magnitude greater than the time required to transmit the message (see §5). Since B must attempt to decrypt the header for every message he receives whether he is the intended recipient or not, he can be backlogged just by processing ambient background traffic.

3.2 Straw Man: Symmetric Key Mechanism

Next we sketch symmetric key, a similar mechanism based on symmetric keys that addresses this pitfall.

Bootstrapping. This mechanism assumes that A and B share a symmetric key.

Construction. Using symmetric keys shared only between A and B, we can use a construction intuitively similar to public key. A encrypts the statement s using symmetric encryption such as AES-CBC. We can omit A and B from s since it is implied by the use of their symmetric key. A then computes a message authentication code (MAC) over the encrypted value so B can verify its authenticity. A random initialization vector (IV) is used so that the resulting cipher text and MAC appear random and thus are unlinkable to any other message.

Message filtering. Upon receipt of a message, B verifies the MAC in the header using the same key A used to construct the message. If the MAC does not verify, then this message is not for B and he can discard it.

[1] In practice, we would still use RSA for faster signatures; we just require each party to have both ElGamal and RSA key pairs.

Symbol	Definition
I	The length of each Tryst time interval.
T, T_0, T_i	Respectively, the current time, the time Tryst keys were bootstrapped, and the start of time interval i: $T_0 + i \cdot I$.
k_p	A one-time use key for encrypting a payload.
$k_{AB}^{Enc}, k_{AB}^{MAC}, k_{AB}^{addr}$	Long-term keys to encrypt, MAC, and compute addresses for Tryst messages sent from A to B.
$k_{s:AB}^{Enc}, k_{s:AB}^{MAC}$	Session keys to encrypt and MAC Shroud messages sent from A to B.
$\mathsf{AES}_k\,(b)$	Encipher single 128-bit block b with key k using the AES cipher.
$\mathsf{AES\text{-}CBC}_{k,i}\,(m)$	Encrypt m with symmetric key k and 128 bit IV i using the AES cipher in CBC mode.
$\mathsf{AES\text{-}CMAC}_k\,(m)$	Construct 128-bit message authentication code (MAC) of m with key k using AES-CMAC [29].
$\mathsf{SHA1}_{128}\,(m)$	Return first 128 bits of a cryptographic hash of m.

Table 2—Cryptographic terminology used in §3.3–§4. All keys are 128 bits. When a key k is only used once, an IV is not required for $\mathsf{AES\text{-}CBC}_{k,i}\,(m)$, so we abbreviate it as $\mathsf{AES\text{-}CBC}_k\,(m)$. We use PKCS5 padding for m when it is of variable length.

Of course, since the message does not indicate to B which key was used to generate the MAC—indeed it cannot, or it will no longer be unlinkable—and B has a symmetric key for each client from whom he can receive messages, B must try all these keys to verify the MAC. There is locality when keys are used (e.g., A may know that he expects a reply from B after sending a message to him) so we can sort keys in most-recently-used order, but, for messages not intended for B, he must try all keys before discarding them. Thus, filtering is inefficient for clients or APs that have many keys.

3.3 Discovery and Binding: Tryst

We now describe Tryst, the mechanism we use for transmitting discovery and binding messages such as 802.11 probes and authentication messages. Tryst builds upon the symmetric key straw man, but leverages the following properties of these messages in order to enable efficient message processing:

> *Infrequent Communication.* Individual devices send discovery and binding messages infrequently. For example, 802.11 clients send probes only when they are searching for an AP and send authentication messages only at the beginning of a session or when roaming between APs.
>
> *Narrow Interface.* Unlike data packets, which can contain arbitrary contents, there are very few different messages that are used for discovery and binding. Thus, it is unlikely that their evolution at short time scales exposes many sensitive side channels of information when individual messages are not decipherable. It is only the ability to link these messages together at long time scales (e.g., hours or days) that threatens location privacy.

Based on these two observations, we define a relaxed version of the strong unlinkability property:

LONG-TERM UNLINKABILITY. Let $t(m)$ be the time a message m was generated. Any party other than A or B that receives $c_1 = F(A, B, p_1)$ and $c_2 = F(A, B, p_2)$ should not be able to determine that the sender or receiver of c_1 or c_2 were the same if $|t(c_2) - t(c_1)| > I$, for some time interval I. In practice, I would be several minutes and may be different for each client-service relationship.

Tryst achieves this relaxed form of unlinkability, which is sufficient for discovery and binding messages because very few are likely to be generated during any given interval I. Even if an adversary is able to force multiple discovery messages to be generated during one interval, e.g., by jamming the channel to force all clients to reassociate, the ability to link them together is unlikely to be threatening.

For clarity, we list the cryptographic terminology we use in the subsequent description in Table 2.

Bootstrapping. Similar to symmetric key, A and B each have symmetric keys $k_{AB}^{Enc}, k_{AB}^{MAC}, k_{AB}^{addr}$ for constructing messages from A to B (and another set of keys for B to A). They also remember the time they exchanged these keys as T_0.

Temporary unlinkable addresses. A client A and a service B that share a symmetric key can independently compute the same sequence of unlinkable addresses and thus will at any given time know which address to use to send messages to the other. Specifically:

$$addr_{AB}^{i} = \mathsf{AES}_{k_{AB}^{addr}}\,(i)\,, \quad \text{where } i = \lfloor (T - T_0)/I \rfloor$$

In other words, $addr_{AB}^{i}$ is a function of the ith time interval after key negotiation. The crucial property we leverage is that for any two values $\mathsf{AES}_{k1}\,(i1)$ and $\mathsf{AES}_{k2}\,(i2)$ where $i1 \neq i2$, it is intractable for a third party to determine whether $k1 = k2$, even if $i1$ and $i2$ are known. Thus, these addresses are unlinkable without knowledge of k_{AB}^{addr}.

In practice, B computes $addr_{AB}^{i}$ once at time $T_i = T_0 + i \cdot I$. B maintains a hash table containing the addresses for messages he might receive. At time T_i, he clears the table and inserts the key-value pair ($addr_{jB}^{i}$, j) for each identity j he has keys for, so that he can anticipate messages sent with these addresses and determine that he should use j's keys to process them. When A wants to send a message to B at time T, he also computes $addr_{AB}^{i}$. §4.1 discusses how we deal with clock skew.

Construction. Tryst(A, B, p) is computed as follows (Figure 2):

1. Generate a random key k_p.
2. *header* ← $\{s, mac\}$, where:

$$s = \{addr_{AB}^{i}, \mathsf{AES}_{k_{AB}^{Enc}}\,(k_p)\},$$
$$mac = \mathsf{AES\text{-}CMAC}_{k_{AB}^{MAC}}\,(s)\,.$$

   header proves to B that A is the sender and B the receiver because only A and B have k_{AB}^{Enc} and k_{AB}^{MAC}. Moreover, it proves to B that it was constructed near the current time T because $addr_{AB}^{i}$ is a cryptographic function of T. This provides authenticity.

   To third parties, *mac* appears to be random because it is computed over the encryption of random key k_p, so neither it nor the encipherment of k_p can link it to

$s = \{addr^i_{AB}, \mathsf{AES}_{k^{Enc}_{AB}}(k_p)\}$	$mac = \mathsf{AES\text{-}CMAC}_{k^{MAC}_{AB}}(s)$	$etext = \mathsf{AES\text{-}CBC}_{k_{p1}}(p)$	$emac = \mathsf{AES\text{-}CMAC}_{k_{p2}}(etext)$
32 bytes	16 bytes	variable	16 bytes

Figure 2—Tryst packet format.

other messages. $addr^i_{AB}$ is sent "in the clear" and will be used in all messages sent during time interval T_i, but $addr^{i1}_{AB}$ and $addr^{i2}_{AB}$ for any $i1 \neq i2$ are unlinkable, thus providing long-term unlinkability.

3. $ctext \leftarrow \{etext, emac\}$, where:

$$etext = \mathsf{AES\text{-}CBC}_{k_{p1}}(p)\,,$$
$$emac = \mathsf{AES\text{-}CMAC}_{k_{p2}}(etext)\,.$$

   k_{p1} and k_{p2} are pseudo-random keys derived from k_p (e.g., $k_{p1} = k_p$ and $k_{p2} = \mathsf{SHA1}_{128}(k_p)$). $ctext$ is an encryption of the payload p along with a MAC which, given k_p, verifies that the payload was not altered during transmission. We derive two keys from k_p so that different keys are used for encryption and MAC. Since k_p is random, $ctext$ will be different from previous messages even when an identical payload p was sent before.

4. $c \leftarrow \{header, ctext\}$.

The overhead (64–80 bytes per message) is acceptable since discovery and binding messages are sent infrequently.

Message filtering. Upon reception of such a message, B simply checks his hash table to determine if he has an address $addr^i_{AB}$. If he does, it will be associated with the keys for A, which can be used to verify and decrypt the $header$. If not, he can discard the message. Once $header$ is decrypted, he obtains k_p, which can be used to decrypt and verify $ctext$ to retrieve the original p. In contrast to the straw man mechanisms, this protocol enables devices to discard messages not intended for them efficiently, using hash table lookups instead of costly cryptographic operations.

3.4 Data Transport: Shroud

Tryst is insufficient for identifier-free data transport because data messages are neither infrequent nor do they have a narrow interface. Thus, to defend against side-channel analysis, we want strong unlinkability rather than just long-term unlinkability. Shroud maintains this property efficiently by leveraging a key assumption about data transport:

> *Connected Communication.* Whereas discovery messages are often sent at times when they will not be received, data messages are only sent after a link has been established. Thus, a sender and receiver can assume that, barring message loss, their messages will be received by their intended recipient.

In effect, this assumption enables Shroud to compute a sequence of unlinkable addresses on a per packet basis, as we will describe shortly.

Bootstrapping. We bootstrap Shroud with random session keys $k^{Enc}_{s:AB}, k^{MAC}_{s:AB}$ for messages from A to B. These keys are exchanged in *SlyFi*'s authentication messages (see Figure 1) and thus are protected by Tryst. For reasons that we discuss in the construction below, the same key $k^{Enc}_{s:AB}$ can be used for both address computation and encryption.

Per-packet unlinkable addresses. The only design choice in Tryst that sacrifices strong unlinkability is the use of the same $addr^i_{AB}$ for all packets during time interval i. Thus, we can essentially use Tryst, provided that we can compute addresses $addr^i_{AB}$ per packet rather than per time interval. To do this in Shroud, $addr^i_{AB}$ is computed as a function of the ith *transmission* since link establishment:

$$addr^i_{AB} = \mathsf{AES}_{k^{Enc}_{s:AB}}(i)\,, \quad \text{where } i = \text{transmission number}$$

Since a connection has been established, B will receive every packet sent by A on this link barring message loss, and, hence, B only needs to compute address $i{+}1$ after the receipt of message i; i.e., B computes the address he expects in the next message.

Of course, message loss in wireless networks is common, so we would like to be able to tolerate the loss of w consecutive losses for some w. Thus, on receipt of message i, a receiver computes the $(i+1)$th to $(i+w)$th addresses and inserts them all into its hash table (removing all addresses $j \leq i$). Note that, except for the first message received (e.g., the association request or reply in *SlyFi*), which requires the computation of w addresses, only one additional address needs to be computed for each subsequent packet sent to B, unless there are message losses; B performs no address computation for packets destined for other devices that it overhears. §4.2 discusses how we choose w and perform link layer retransmissions.

Construction. With per-packet unlinkable addresses, we could use the Tryst construction and achieve the desired security properties and filter packets efficiently. However, we can make additional optimizations. Shroud(A, B, p) is computed as follows (Figure 3):

1. $header \leftarrow addr^i_{AB}$.

   Unlike in Tryst, no Shroud address will ever appear in two different messages; thus no one can successfully record and replay them. As a consequence, $addr^i_{AB}$ itself proves to B that A sent the message to B and that it was message transmission i, which B expects. This provides authenticity. Each message will have a different address and addresses are strongly unlinkable.

2. $ctext \leftarrow \{etext, emac\}$, where:

$$etext = \mathsf{AES\text{-}CBC}_{k^{Enc}_{s:AB}, header}(p)\,,$$
$$emac = \mathsf{AES\text{-}CMAC}_{k^{MAC}_{s:AB}}(header, etext)\,.$$

   In Tryst, we use a random key to perform the encryption to ensure that the encrypted payload and MAC are unlinkable to previous messages even if their contents are the same. Since each Shroud address is pseudo-random and is used only once, $header$ effectively serves as a random nonce that we can use as an IV to the encryption of the payload. This ensures

$header = addr^i_{AB}$	$etext = \textsf{AES-CBC}_{k^{Enc}_{s:AB}, header}\,(p)$	$emac = \textsf{AES-CMAC}_{k^{MAC}_{s:AB}}\,(header, etext)$
16 bytes	variable	16 bytes

Figure 3—Shroud packet format.

that *etext* is unlinkable to previous messages even if their contents are the same and we use the same key $k^{Enc}_{s:AB}$ for encryption. Similarly, we include *header* in the computation of *emac* to ensure that it is unlinkable to previous messages even if p is null.

3. $c \leftarrow \{header, ctext\}$.

We note that $addr^i_{AB}$ implies the Ethernet addresses in p so they can be removed. Therefore, Shroud's additional 32 bytes of overhead can, in practice, be reduced to only 15–30 bytes per packet, depending on the 1–16 bytes the PKCS5 padding scheme adds to align *etext* to 16-byte AES blocks.

Message filtering. As in Tryst, B determines whether a message is for him by looking up $addr^i_{AB}$ in the hash table containing his precomputed addresses. In fact, since the address is located in the same position in both Tryst and Shroud packets, there is no need to distinguish the two message types and a single hash table can be shared by both. The value associated with each address key in the hash table will indicate whether it should be demultiplexed to Tryst or Shroud.

3.5 Other Protocol Functions

Tryst and Shroud make the crucial elements of a link layer protocol—discovery, binding, and data transport—identifier-free, but other protocol functions must be supported as well. In this section, we explain how *SlyFi* can support these functions without introducing identifiers.

Broadcast. Shroud supports identifier-free broadcast transmissions in managed mode. Broadcasted frames are encrypted with a key and sequence number that are shared by the AP and all clients on the local network. As in 802.11's managed mode, a client forwards frames to the AP that it wishes the AP to broadcast. In Shroud, the transmission to the AP is protected by the per-client shared key used for unicast transmissions. (A client optionally may divulge his identity to all associated stations by including his source address.) Upon reception at the AP, the frame is decrypted and then re-encrypted with the shared broadcast key. The shared key and current sequence number are managed by the AP and conveyed to each of its clients during association. Although *SlyFi* currently does not support broadcast key revocation, we believe we can apply a scheme similar to that of 802.11i [6]; this is is a topic of future work.

Binding to higher layer identifiers. There is often a need to bind higher layer names to link layer addresses. For example, ARP binds Ethernet addresses to IP addresses. Obviously, we do not want to have to re-establish this binding for every Shroud address change. Instead, we have the AP negotiate with each client a pseudonym address that remains consistent for that session, but that is not sent in actual messages. The client informs the AP of its IP to pseudonym binding whenever its IP address changes. Thus, the AP can answer all ARPs.

Announcement. Beacons are broadcasted in the clear to announce an 802.11 AP. While *SlyFi* does not prevent beacons, an AP that wants to hide its presence obviously cannot use them. To discover APs in *SlyFi*, a client must have the necessary Tryst keys to probe for it. We do not believe this is a hindrance, since existing secure 802.11 networks already require secure out-of-band channels to exchange keys before association.

Time synchronization. Beacons are also used to convey timestamps so that clients and APs can synchronize their clocks. With synchronized clocks, clients need only turn on their radios at designated times to receive packets when operating in low power modes. Since only clients on the local network need to synchronize their clocks, this information can be encrypted using the broadcast encryption key described above.

Roaming. Clients sometimes use probes or beacons after association to search for better APs to roam to. Using Tryst to send these probes might be expensive if a client sends them frequently. However, these APs are usually in the same administrative domain and thus could share a broadcast key, which could be used to encrypt these messages instead of using Tryst. In addition, Shroud session state could be migrated between APs in advance, similar to how WPA pre-authentication is performed.

Coexistence. Our implementation of *SlyFi* can coexist with normal 802.11 devices because we encapsulate *SlyFi* messages in management frames that normal 802.11 devices ignore (see §4.3). The medium access protocol is unchanged.[2] Thus, *SlyFi* can be deployed incrementally. In a mixed environment, a *SlyFi*-enabled client can first search for a *SlyFi*-enabled AP using Tryst probes. If no such AP is found, then a client willing to fall-back to a normal 802.11 AP can listen for beacons and associate normally.

4. IMPLEMENTATION DETAILS

This section discusses practical considerations involved in implementing Tryst, Shroud, and our *SlyFi* prototype.

4.1 Tryst: Practical Considerations

Clock skew. In practice A and B will not have perfectly synchronized clocks. To allow for clock skew up to $k \cdot I$ between devices, B should anticipate the addresses that may be used for any messages sent in the time range $[T_{i-k}, T_{i+k}]$ at time T_i. Thus, he also inserts (or keeps) $addr^{i-k}_{jB}, addr^{i-k+1}_{jB}, \ldots, addr^{i+k}_{jB}$ into the table for all identities j for which he has keys. Note that messages sent by A will still only use the address $addr^i_{jB}$ for one time interval of

[2] We do not yet support RTS/CTS because our software implementation is not fast enough to perform filtering at the timescale required, but we note that RTS/CTS is rarely used in actual managed networks.

length I. B will simply accept messages with that address for longer.

Scoped broadcast. A client may want to send the same discovery message to multiple services (e.g., to discover any one of them). To do this, A constructs one *header* for each intended recipient, but includes the same k_p in each *header*; e.g., he sends $\{header1, \ldots, headerN, ctext\}$. Hence, any party that can interpret any one *header* can obtain k_p and decrypt the payload. However, each party can only interpret the *header* intended for them, so the identities of the other parties remain obscured.

Forward security. One concern is that k_{AB}^{addr} is stored for a long time and if it is compromised, an adversary could compute the addresses of all messages that A ever sent to B. We mitigate this risk by computing a new key each day using a forward-secure pseudorandom bit generator [10]; i.e., the key for day j: $k_{AB}^{addr(j)} \leftarrow \mathsf{SHA1}_{128}\left(k_{AB}^{addr(j-1)}\right)$. Both A and B discard the old key and use the new key for computing addresses. The address computation remains the same, but an adversary that obtains $k_{AB}^{addr(j)}$ would only be able to compute addresses for days j and after.

Side-channel attacks. If an adversary knows the sender or intended recipient of a Tryst probe, the presence or absence of a reply may reveal additional information. For example, an adversary can replay a probe at another location to see if the recipient responds. However, these attacks can only be performed for the short time interval that an address is valid and can be mitigated by simple countermeasures. For example, since an adversary cannot distinguish the content of a response from any other message, if random delays were added to probe responses, an adversary might lose them in the noise of frequent background traffic. In addition, receivers can cache valid probe and authentication requests that they receive for the duration they are valid and ignore replays of those messages. We did not implement these countermeasures since these attacks assume adversaries already know the sender or intended recipient of a message, which can not be learned from the message's contents alone.

4.2 Shroud: Practical Considerations

Choosing w. w determines the number of consecutive packet losses Shroud can endure. In practice, burst losses of more than 50 packets are extremely rare on usable links [24] so we use $w = 50$. A larger burst loss will result in a higher level timeout and require re-establishing the link. The overhead required to maintain these addresses is not prohibitive; even a heavily loaded AP with 256 clients (the max supported by the standard MadWifi driver [3]) requires only 1MB. Most clients, which only have one association at a time, could easily check message addresses in hardware with no more delay or energy than existing NICs. We show in §5.4 that even software filtering incurs little overhead.

Acknowledgments. Every unicast 802.11 data packet is acknowledged by the receiver to manage message loss. In principle, link-layer acknowledgments can simply acknowledge the address of the received Shroud packet, since the sender knows the last address used. However, our current implementation is in software and thus is unable to send this ack within the 16 microseconds allotted to it. Therefore, we currently use software acks that selectively acknowledge cumulative windows of data packets. Each acknowledgment and message retry is processed anew by Shroud.

4.3 Prototype Implementation

We implemented *SlyFi* in C++ using the Click Modular Router [20], incorporating Tryst and Shroud with its existing 802.11 implementation (which is by the authors of Roofnet [4]). Since existing 802.11 NICs will not send frames without proper 802.11 headers, each Tryst or Shroud message is encapsulated in an "anonymous" 802.11 header, i.e., one with constant fields and addresses. NICs are placed in promiscuous mode so that they receive all these frames and perform filtering in software. We use the cryptographic routines in libgcrypt [2] and ran our software as a Linux kernel module.

5. PERFORMANCE EVALUATION

We evaluate two key areas. First, we examine how quickly we can discover and set up a link with Tryst. A quick link establishment improves usability both by reducing the delay before communication can begin and by preventing noticeable interruptions when roaming between APs. Second, we examine the performance penalty incurred when using Shroud to deliver data traffic. In general, we find that *SlyFi* performs comparably to 802.11 using WPA and substantially out-performs the straw man mechanisms we discussed.

5.1 Comparison Protocols

We compare our *SlyFi* implementation to the following baseline protocols and alternatives:

wifi-open. The baseline implementation of 802.11 without WEP or WPA in Click. *SlyFi* uses the same components, simply encapsulating the original packets where needed, allowing us to make a direct comparison to a software implementation without our mechanisms.

wifi-open-driver. The 802.11 implementation in the MadWifi driver/firmware [3]. We compare to this second baseline since wifi-open has additional overhead when used for data transport, which we discuss in §5.4. Neither wifi-open nor wifi-open-driver meet any of our security requirements.

wifi-wpa. A baseline implementation of 802.11 with WPA, which provides authentication, message integrity, and confidentiality, but not unlinkability as messages still include Ethernet addresses and network names. We use the standard WPA client and AP implementations on Linux [1], which run on top of wifi-open-driver, so wifi-wpa does not incur the overhead mentioned above. We run wifi-wpa using PSK user authentication and CCMP encryption. PSK is the most widely used standard in small private networks. CCMP is comparable to *SlyFi*'s payload encryption, as both are built around AES. However, wifi-wpa performs AES operations using dedicated hardware on the 802.11 NIC, while *SlyFi* performs it in software. To compensate, we also evaluate *SlyFi* with simulated hardware we discuss in §5.4.

public key. The straw man alternative to Tryst for discovery and link setup discussed in §3.1.

symmetric key. The other straw man alternative to Tryst discussed in §3.2. public key and symmetric key still use Shroud once a link is established (i.e., they only replace Tryst in Figure 1).

armknecht. A previous 802.11 frame encryption proposal [8] that is an alternative to Shroud for data transport.[3] Like Shroud, armknecht computes per-packet addresses, but only for the next packet it expects, so it would perform comparably when there is no packet loss or competing traffic. However, a receiver that receives a message without one of its known addresses performs a number of cryptographic operations comparable to symmetric key before discarding it. This is because it treats a packet it does not have an address for as an indication of potential loss and uses these operations to try to recover from it. In contrast, Shroud simply precomputes more addresses to manage loss.

5.2 Setup

We deploy these protocols on a number of Soekris net4801 low-power devices [5]. These devices have hardware comparable to common 802.11 APs and embedded consumer devices. While laptops have more powerful hardware, we demonstrate that our mechanisms are usable even on more constrained devices. In our experiments, we designate each device as either an AP or a client.

Each device has a 266 Mhz 586-class Geode processor, 256 MB of RAM, 1 GB of flash storage, and one CM9 Atheros 802.11a/b/g miniPCI card. Each device runs a minimal version of Linux 2.6.16.13. 802.11 frames are sent and received from a raw 802.11 radiotap device created by the standard MadWifi driver. We operate on 802.11a channel 40 to avoid interference from more common 802.11b/g devices. To make a fair comparison, management frames in all protocols are transmitted at the base rate (6Mbps), as is dictated by the 802.11 standard, while data frames are transmitted at the peak rate (54Mbps).

5.3 Discovery and Link Setup Performance

To evaluate how long a client would need wait before it can start transferring data, we measure the *link setup time*, defined as the delay between when a client begins probing for APs and when it can deliver packets on the established link. In all the protocols except for wifi-wpa, packets can be delivered once an *association response* message is received (see Figure 1). wifi-wpa has an additional key negotiation phase after association.

The parameters that impact link setup time are: the number of client accounts on an AP, the number of networks that each client probes for, and the amount of background probing traffic that is overheard by APs and clients. Our results show that, in contrast to public key and symmetric key, Tryst has faster link setup times than wifi-wpa and scales as gracefully as wifi-open when varying each of these parameters. Moreover, the cost of periodically computing addresses is trivial. Unless otherwise indicated, each data point presented in this section is the mean of 30 trials.

Keys per service. An AP maintains one key for each client it has an account for, and the total number of keys can impact link setup time. Real networks manage various numbers of client accounts; e.g., home networks will likely have less than a dozen, while the wireless network at the Intel Research lablets, a fairly small organization, has 721. Carnegie Mellon University's wireless network, which may be representative of a large organization, has 36,837 at the time of this writing.

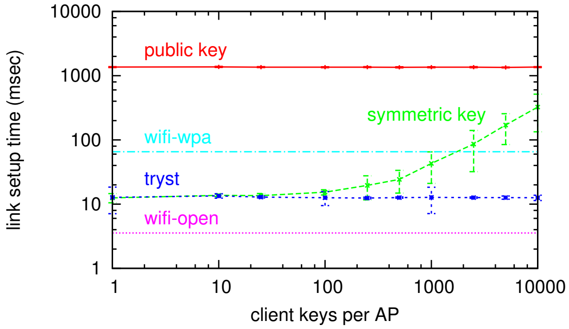

Figure 4—Association delay as the number of keys per AP varies. The client probes for 1 AP. Error bars indicate one standard deviation.

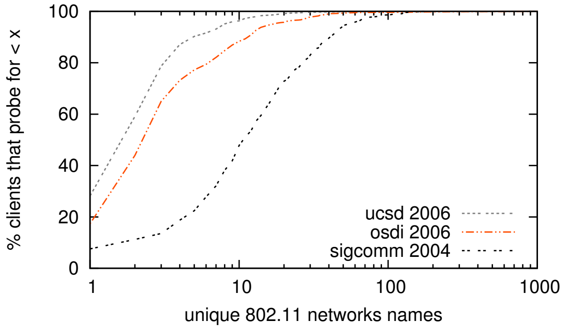

Figure 5—CDF of the number of unique network names probed for by each client in three empirical 802.11 traces.

Figure 4 shows the link setup time for a client that sends one probe in search of a nearby AP as we vary the number of keys per AP. Before each probe, the AP has its keys sorted in random order, and thus, the performance of the symmetric key protocol degrades with the number of keys, since it must check a *discovery* message against all keys until it finds one that successfully validates the MAC on the header. The other protocols have setup times that are independent of the number of keys per AP. Note however, that the public key protocol is still more expensive than all the others, even when the AP has 10,000 keys. Furthermore, although Tryst imposes some overhead over the wifi-open protocol, it has link setup times that are less than wifi-wpa and that, at ∼15 ms, are below the variance in Internet round trip times.

Probes per client. Since private APs cannot send beacons, a client may need to probe for several different networks to figure out which one is present. In 802.11, these probes usually contain the names of networks with which the client has previously associated. Figure 5 shows a cumulative distribution function of the number of unique network names probed for by clients in three wireless traces (described in Table 1). While most users probe for a small number of networks, at least 4% of users in all traces probe for more than 10 and some probe for more than 100.[4] There-

[3] Our implementation uses AES as the cipher.

[4] Users in the SIGCOMM trace probed for more networks because each SIGCOMM AP had a different network name and the network often was unavailable, prompting clients to send probes for names deeper into their list of networks. We ignored broadcast and random network names.

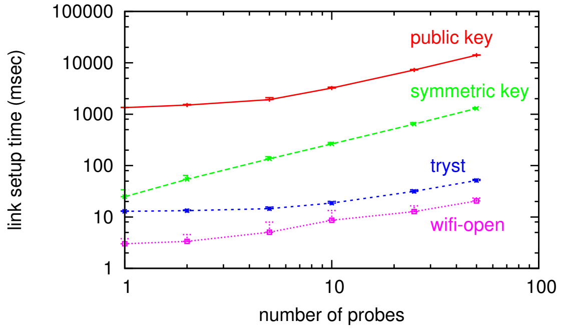

Figure 6—Link setup time as the number of probes each client sends varies. The AP has 500 keys. Error bars indicate one standard deviation.

	probing	openauth	associate	wpa-key	total
public key	886.1	895.2	146.2	NA	1927.6
symmetric key	120.2	8.6	6.9	NA	135.6
tryst	3.3	5.1	6.2	NA	14.5
wifi-open	1.4	1.5	2.2	NA	5.1
wifi-wpa	0.1	6.9	0.8	57.5	65.3

Table 3—Breakdown of link setup time for a client that probes for 5 different networks and an AP with 500 keys. Times are in milliseconds. Each phase corresponds to request/response messages in Figure 1, except wpa-key, which involves 2 round trips after association to derive session keys in wifi-wpa.

fore, it is important that link setup time does not grow substantially with the number of probes.

Figure 6 shows the link setup time as a function of the number of different probes a client sends, which are sent as fast as possible. The number of keys per AP is fixed at 500. For a fair comparison, Tryst sends a separate message for each probe instead of using scoped broadcast (discussed in §3.3). We omit the line for wifi-wpa because the standard probing behavior is different and incurs more delays. If it also sent probes as fast as possible, it would have scaling behavior similar to the wifi-open line since the probes they send are the same.

Although all protocols scale with the number of probes sent, as there is overhead in processing them and limited bandwidth in the medium, the slopes of the two straw man protocols are steeper, indicating that they incur more overhead per probe. The slopes of the Tryst and wifi-open lines are similar, and both have setup times of at most ∼50 ms even when clients send 50 probes.

Performance breakdown. Table 3 shows the breakdown of link setup time for clients that send 5 probes and APs with 500 keys. The public key protocol spends most of its time in the first two phases, since it must process most public key encryptions, decryptions, and signature checks here. These operations are two orders of magnitude slower than the symmetric key analogs. Nonetheless, the symmetric key protocol still spends significant time in the probing phase, because when the AP first receives a probe, it may try to verify the MAC with all its keys. Subsequent phases are faster because both the client and the AP re-sort their keys in MRU order, so the expected number of keys they must try before finding the right one decreases appreciably.

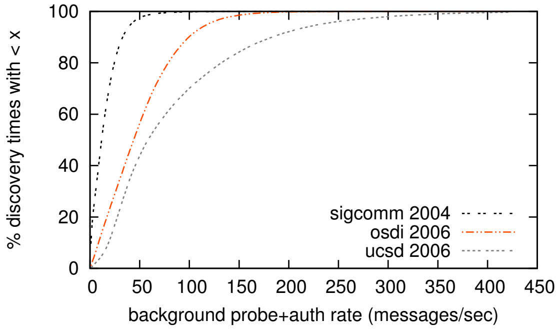

Figure 7—CDF of background probe and authentication messages observed each second where discovery was taking place in each of three 802.11 traces. We only count times when there was at least one probe (i.e., times when discovery was taking place).

Tryst has similar performance to the symmetric key protocol during the last three phases because the number of cryptographic operations is identical. However, the first phase is much faster because the AP looks up the address in a hash table to determine which key to use to verify the message. The open authentication and association phases take slightly longer because they involve computing the initial w Shroud addresses. Even when performing these operations, in addition to standard 802.11 processing, the time it takes for *SlyFi* using Tryst to setup a link is less than 10 ms more than that of wifi-open, which provides no authentication or confidentiality. Moreover, it is faster than wifi-wpa.[5]

Note that if a client did not know the particular wireless frequency a network was located on, it would spend more time in the probing phase because it would have to wait on each channel to see if a probe response arrives. This waiting time is configured to be 20–200 ms in 802.11.

Background probing traffic. The previous experiments assumed no ambient background traffic during the link setup process. However, due to the ad hoc nature of real wireless deployments, stations and APs often overhear messages that are not destined for them. For example, Figure 7 shows the rate of probe requests, responses, and authentication messages observed by one monitoring point.[6] Although the ambient message rate is generally fairly low, there are times when the rate is over 100 messages per second, due to many clients performing discovery at once. Thus, it is crucial that clients and APs be able to discard these messages quickly.

To evaluate how well *SlyFi* can manage background probe and authentication messages, we examine a client's link setup time as a function of such traffic. To do this, we introduce a third machine that sends background messages at a specified rate destined neither for the client or the AP. These background messages are encapsulated in the protocols we compare, but we precompute them so that their generation

[5] We note that wifi-wpa incurs an unnecessary delay in the open authentication phase, but since the bulk of the time is spent in wpa-key for key computation and exchange, removing this delay would not change the ranking of the total link setup times.

[6] The UCSD trace merged observations from multiple monitoring points, so it observes more traffic at any given time. The OSDI trace contains more users than the SIGCOMM trace and thus observed a higher rate of traffic.

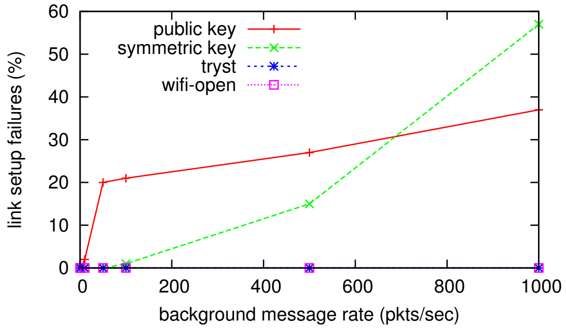

Figure 8—Percentage of 100 link setup attempts that fail to complete within 30 seconds as we vary the rate of background probe traffic not destined for the target AP. The client probes for one AP and the AP has 500 keys.

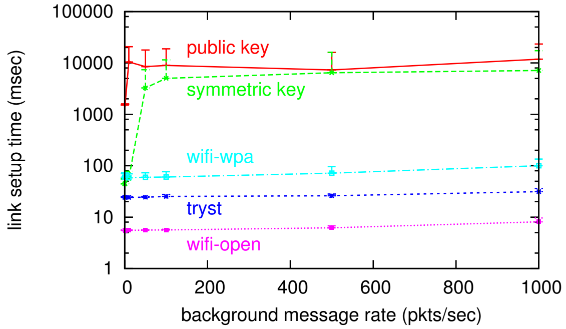

Figure 9—Link setup time for successful attempts as we vary the rate of background probe traffic not destined for the target AP. Error bars indicate one standard deviation.

is able to maintain the specified rate. Each protocol queues up to 10 messages (drop tail) if it is busy processing and each client request is retransmitted once per second. We consider a link setup attempt to fail if it does not complete in 30 seconds. The client probes once for an AP with 500 keys.[7]

Figure 8 shows the percentage of link setup attempts that failed. Due to the processing required by the public key and symmetric key protocols in order to determine whether a message is destined for the receiver, each begins to fail when the background message rate grows. No attempts fail when using Tryst or wifi-open. We omit the line for wifi-wpa because its message retry behavior is different. Figure 9 shows the link setup times for the attempts that succeeded. Note that while the symmetric key protocol is able to cope with message rates of up to 100 messages/second before it begins to fail, its link setup times grow to several seconds, and impact perceived performance, even when the rate is 50 messages/second.

Contention for the medium causes Tryst, wifi-open, and wifi-wpa to each have link setup times that grow slightly as the background message rate increases, but their scaling behavior is gradual and roughly consistent. Tryst's ability to discard background messages quickly enables it to scale gracefully. This property is important not only for dealing with ambient discovery traffic, but also for mitigating the impact of malicious denial of service attacks. With the public key and symmetric key protocols, a malicious device only needs to send a small number of messages to prevent a client from setting up a link.

# keys	1	10	50	100	500	1000	10,000
time (msec)	0.08	0.49	2.3	4.7	24	47	800

Table 4—The mean time to update Tryst addresses for a single time interval as we vary the number of keys.

Address update time. At the beginning of each time interval, a Tryst node precomputes the message addresses it expects to receive to enable quick message filtering. Table 4 shows the time it takes to compute these addresses and update the hash table as we increase the number of keys a device holds. A node computes two addresses per time interval, per key (one for probes and one for authentication messages). Clients, which are unlikely to have more than 100 keys, would spend only a few milliseconds each time interval to update addresses, and time intervals would likely be at least several minutes. Even APs with 10,000 clients would spend less than 1 second.

5.4 Data Transport Performance

We now examine how well Shroud performs at delivering data packets. We begin with a description of microbenchmarks that break down how long Shroud takes to send, filter, and receive packets. Then we present an analysis of packet delivery latency and throughput when a *SlyFi* client and AP are communicating in isolation. Finally, we look at performance in the face of background traffic, and present results describing achievable throughput both as the number of clients managed by the AP and the amount of competing traffic varies.

Simulated hardware encryption. Shroud's cryptography operations are implemented in software, which adversely affects performance. To understand how Shroud would perform with hardware support, we simulate the processing times of that hardware. As a result, we provide measurements both for the software-only version, shroud-sw, as well as for the version with hardware simulation, shroud-hw.

Since both wifi-wpa and Shroud use AES to encrypt and MAC packets, we use wifi-wpa's processing times as an estimate for shroud-hw.[8] We estimate wifi-wpa's cryptographic processing time (including I/O) as the difference in round trip ping delays between wifi-wpa and wifi-open-driver. Measurements suggest the time to encrypt a 1500 byte ICMP packet is ∼16 usec and the time to encrypt a payload-free packet is ∼14 usec. I/O overhead dominates, but there is a small linear scaling factor as the packet size increases. Neither encryption nor MAC computation are parallelizable in Shroud, whereas CCMP's encryption may be parallelized in hardware. Thus, we conservatively estimate that shroud-

[7] Note that in this experiment, the client and AP drivers ran in user level, rather than in the kernel, because when the straw man protocols become overloaded with message processing, the Linux kernel became unresponsive to experimental commands. This imposes a slight overhead on message processing, but is insubstantial compared to each protocol's relative performance.

[8] wifi-wpa uses AES counter mode for payload encryption and AES-CBC for MAC computation, while Shroud uses AES-CBC mode for payload encryption and CMAC (a relative of AES-CBC mode) for the MAC. Counter mode is parallelizable, while AES-CBC is not.

	send		filter		receive		
	sw	hw	sw	hw	sw	hw	
update *addrs*							
(max message loss)	15	14	NA	NA	2047	2003	(50)
(no message loss)	15	14	NA	NA	119	117	(1)
process *etext*	951	16	NA	NA	1541	16	
process *emac*	740	16	NA	NA	740	16	
Shroud total	1821	120	32	32	3290	290	
Click total	1913	215	144	144	3402	407	

Table 5—Breakdown of processing times (see §3.4) for 1500 byte packets for shroud-sw (sw) and shroud-hw (hw). All times are in microseconds. Numbers in parentheses are numbers of address computations.

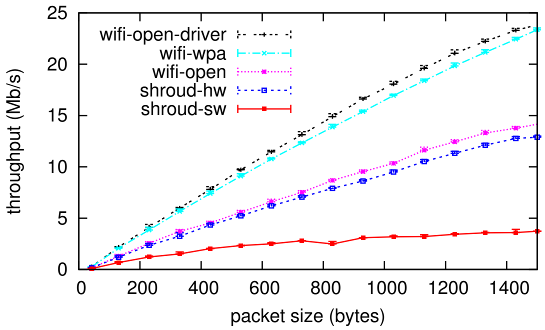

Figure 10—Throughput comparison of UDP packets when transmitting at 54 Mbps for 30 seconds. Each point is the average of 50 runs.

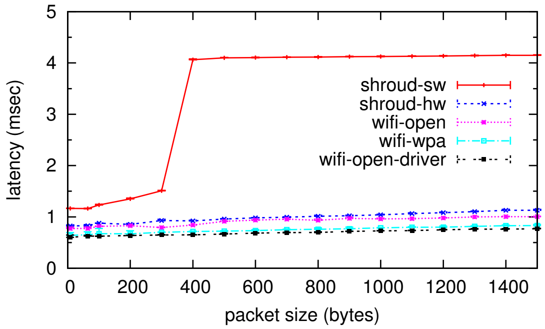

Figure 11—Comparison of round trip times of ICMP ping messages for variously sized packets. Each point is the average of 1000 pings; pings that experienced link-layer packet loss, or re-keying delays (in the case of wifi-wpa), were removed.

hw would take 14 usec to encrypt a pair of addresses and 32 usec to encrypt and MAC packet payloads. To simulate these times, we modify our code to idle-wait for these times instead of computing the cryptographic operations in software. We note that shroud-hw still includes the actual software processing time of all non-AES operations.

Micro-benchmarks. Table 5 breaks down the time to send, filter, and receive Shroud messages. On a busy network, a packet received by a client is often intended for someone else, so filtering packets quickly is imperative. The filter column shows that shroud-hw's filtering time (32 usecs) is much faster than the theoretical minimum packet transmission time in 802.11a for a 1500 byte packet (∼225 usecs), suggesting that a receiver could filter packets faster than the medium could supply them.

Sender-side processing of a 1500 byte packet, shown in the send column (215 usec), also edges out the time to transmit it, and thus, shroud-hw should be capable of supporting 802.11a's line speed. Receiver-side processing (the receive column) from the radio takes 407 usec, which is greater than the theoretical time to transmit, but still reasonable, since 802.11 rates in practice are much slower (e.g., see Figure 10). When packets are lost, additional address computations must be performed after a reception. A reception following the maximum 49 packet burst loss (for $w = 50$) requires Shroud to compute and update 50 new addresses. This takes 2003 usec compared to 117 usec for a single address update (the case with no loss).

The cryptographic operations are much slower when implemented in software than they are in hardware, and thus the performance of shroud-sw is significantly below line speed. Regardless, we present these results to characterize our proof-of-concept implementation that can be used today to protect privacy. Obviously, an engineering effort is required to make use of hardware cryptography.

Throughput and latency. Figure 10 shows achievable throughput for shroud-hw, shroud-sw, wifi-open, and wifi-open-driver, measured using `iperf`. Shroud is implemented in Click, so wifi-open provides a baseline against which to evaluate its performance. While wifi-open (802.11 implemented in Click) performs worse than wifi-open-driver (the native driver implementation), the throughput degradation of shroud-hw is comparable to wifi-wpa relative to their respective baselines. Optimizing wifi-open is a subject for future work. When sending 1500 byte packets, shroud-hw degrades wifi-open performance by only 1.44 Mbps compared to the 0.71 Mbps degradation from running wifi-wpa. Since both Shroud and wifi-wpa use some non-parallelizable cryptographic operations, the relative performance degradation increases with packet size. shroud-sw experiences a much larger drop in throughput, but still provides a functional link (3.73 Mbps).

Figure 11 presents round trip time measurements using `ping`. For 1500 byte packets, two packet payload encryptions and decryptions take ∼60 usec in wifi-wpa and ∼130 usec in shroud-hw; the extra time is due to address encryption. We believe the sudden marked increase in shroud-sw between 300 and 400 byte packets is due to an inefficiency in the Click runtime.[9]

Background traffic. Shroud's design is motivated by the requirement that background traffic must be filtered efficiently. To study how well Shroud filters packets, we run an experiment in which a client, C_1, sends packets as fast as possible to an access point, AP_1. Nearby, we generate background traffic by having another client, C_2 send traffic to another AP, AP_2. We measure the throughput at AP_1. Since the number of keys AP_1 manages (i.e., number of associations) and the amount of background traffic both affect throughput, we vary both independently.

Figure 12 shows throughput measured at C_1 for both the software implementation and hardware simulation of Shroud

[9] Short packets get through the Click data path without a context switch from the OS, while longer packets do not.

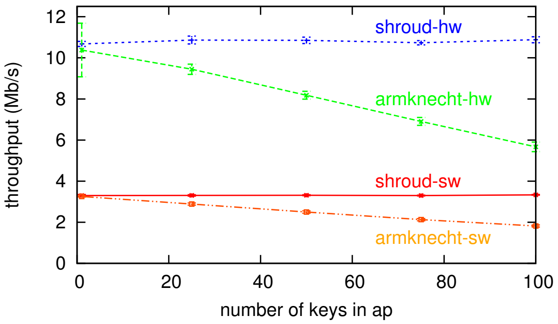

Figure 12—Effect of association set size on achievable throughput when exposed to 5 Mbps of background traffic for Shroud's and armknecht's software implementation and hardware simulation. Each run is 30 seconds; each point is the average of 50 runs.

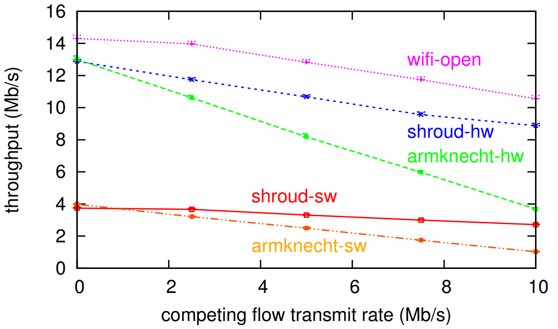

Figure 13—Effect of background traffic on achievable throughput from a client to an AP. The APs association set includes 50 keys. Each run is 30 seconds; each point is the average of 50 runs.

and armknecht, as the number of keys at AP_1 is varied. Since Shroud can filter background packets with just a hash table lookup, its achievable throughput is independent of the number of keys. However, armknecht's performance gets progressively worse as the number of keys increases. This is because clients and APs running armknecht must try every key they have before discarding background packets.

Figure 13, which shows throughput achieved as a function of the competing flow rate, depicts a similar effect. As the amount of background traffic increases, throughput decreases for both Shroud and armknecht, but considerably more so for armknecht. E.g., with 10 Mbps of background traffic, throughput is 31% lower for shroud-hw than it is with no competing traffic, but it is 72% lower for armknecht-hw. This reduction results from a combination of two effects: First, background traffic reduces the availability of the channel, as is evident in the throughput reduction (26%) of our baseline, wifi-open, which performs no cryptographic operations. This affects Shroud and armknecht similarly. Second, background traffic requires work to filter, which is much more expensive in armknecht.

6. RELATED WORK

Private Discovery. We presented the case for private service discovery in a previous workshop paper [22]. Although we sketched some of the challenges in designing Tryst in [22], this paper is the first to present its cryptographic design and implementation.

SmokeScreen [13] is a system that privately announces your presence to your friends. Its protocol also uses symmetric key cryptography to compute temporary addresses. However, it is not a general mechanism for packet delivery and is not authenticated like Tryst. Furthermore, since SmokeScreen uses a hash-chain to compute subsequent addresses, a device that is asleep for a while would have to compute every intermediate address before obtaining the current address to use. In contrast, Tryst simply enciphers a unique counter value based on the time and thus requires only a single cryptographic operation to compute the current address. SmokeScreen tolerates this extra expense because its addresses change very infrequently.

As discussed in §3.1, [7] presented a public key protocol for private authenticated discovery. Since symmetric keys are already often established between wireless clients and services, we argue that a more efficient protocol based on symmetric cryptography would suffice.

Encrypted 802.11 Headers. Armknecht *et al.* [8] propose a way to encrypt the 802.11 header that tries to address many of the same goals as Shroud. Although they compute per-packet addresses like Shroud, there are four key differences that contrast our work. First, to deal with message loss their proposal requires a receiver to try every key it has to decode packets with no matching address. Shroud instead maintains the w subsequent addresses so such a scenario is extremely unlikely. As our evaluation demonstrated, Shroud is substantially more efficient in the face of competing background traffic. Second, unlike *SlyFi*, their proposal is not a complete link layer, as it does not address service discovery, broadcast, higher layer bindings, etc. Finally, unlike their proposal, we have demonstrated *SlyFi* with a real implementation.

Singelée and Preneel [27] also propose an addressing scheme similar to Shroud using a hash-chain instead of an AES counter. Unlike *SlyFi*, this proposal is not a complete link layer and it did not include an implementation.

Pseudonyms. Gruteser et al. [16] and Jiang et al. [18] present 802.11 pseudonym schemes where users change MAC addresses each session or when idle. *SlyFi* enables address changes per packet, which are often desired to mitigate profiling of packet sequences. In addition, unlike *SlyFi*, neither scheme obscures names in discovery or link setup messages.

7. CONCLUSION

We presented the design and evaluation of *SlyFi*, an identifier-free 802.11 link layer that obfuscates all transmitted bits, including addresses. We showed how a link layer could use two mechanisms, Tryst and Shroud, to perform this obfuscation while still achieving efficient discovery, link establishment, and data transport, and without sacrificing other crucial link layer functions such as higher layer name binding. Our evaluation showed that *SlyFi* performs comparably to WPA and performs substantially better than previous techniques.

8. ACKNOWLEDGMENTS

We thank our shepherd Ramón Cáceres and the anonymous reviewers for their comments and suggestions. This work is funded by the National Science Foundation through grant numbers NSF-0721857 and NSF-0722004, and by the Army Research Office through grant number DAAD19-02-1-0389.

9. REFERENCES

[1] Hostap driver. http://hostap.epitest.fi/.

[2] libgcrypt. http://directory.fsf.org/project/libgcrypt/.

[3] Madwifi driver. http://madwifi.org/.

[4] roofnet. http://pdos.csail.mit.edu/roofnet/doku.php.

[5] Soekris engineering. http://www.soekris.com/net4801.htm.

[6] Ieee 802.11i-2004 amendment to ieee std 802.11, 2004. standards.ieee.org/getieee802/download/802.11i-2004.pdf.

[7] Abadi, M., and Fournet, C. Private authentication. *Theor. Comput. Sci. 322*, 3 (2004), 427–476.

[8] Armknecht, F., Girão, J., Matos, A., and Aguiar, R. L. Who said that? privacy at link layer. In *INFOCOM* (2007), IEEE.

[9] Bellare, M., Boldyreva, A., Desai, A., and Pointcheval, D. Key-privacy in public-key encryption. In *ASIACRYPT* (2001).

[10] Bellare, M., and Yee, B. Forward-security in private-key cryptography. *Topics in Cryptology - CT-RSA'03, LNCS 2612* (2003).

[11] Chandra, R., Mahajan, R., Padmanabhan, V., and Zhang, M. CRAWDAD data set microsoft/osdi2006 (v. 2007-05-23). http://crawdad.cs.dartmouth.edu.

[12] Cheng, Y.-C., Bellardo, J., Benko;, P., Snoeren, A. C., Voelker, G. M., and Savage, S. Jigsaw: solving the puzzle of enterprise 802.11 analysis. *SIGCOMM CCR* (2006).

[13] Cox, L. P., Dalton, A., and Marupadi, V. SmokeScreen: Flexible privacy controls for presence-sharing. In *MobiSys* (2007).

[14] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The second-generation onion router. In *USENIX Security* (2004).

[15] Greenstein, B., Gummadi, R., Pang, J., Chen, M. Y., Kohno, T., Seshan, S., and Wetherall, D. Can Ferris Bueller Still Have His Day Off? Protecting Privacy in an Era of Wireless Devices. In *HotOS XI* (2007).

[16] Gruteser, M., and Grunwald, D. Enhancing location privacy in wireless LAN through disposable interface identifiers: A quantitative analysis. *ACM MONET 10* (2005).

[17] Hu, Y.-C., and Wang, H. J. A framework for location privacy in wireless networks. In *SIGCOMM Asia Workshop* (April 2005).

[18] Jiang, T., Wang, H., and Hu, Y.-C. Preserving location privacy in wireless LANs. In *MobiSys* (2007).

[19] Juels, A. RFID security and privacy: A research survey. *IEEE Journal on Selected Areas in Communication 24*, 2 (Feb. 2006).

[20] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The click modular router. *ACM Transactions on Computer Systems 18*, 3 (August 2000), 263–297.

[21] Pang, J., Greenstein, B., Gummadi, R., Seshan, S., and Wetherall, D. 802.11 user fingerprinting. In *MobiCom* (Sept. 2007).

[22] Pang, J., Greenstein, B., McCoy, D., Seshan, S., and Wetherall, D. Tryst: The case for confidential service discovery. In *HotNets* (2007).

[23] Patwari, N., and Kasera, S. K. Robust location distinction using temporal link signatures. In *MobiCom* (2007).

[24] Reis, C., Mahajan, R., Rodrig, M., Wetherall, D., and Zahorjan, J. Measurement-based models of delivery and interference in static wireless networks. *SIGCOMM CCR 36*, 4 (2006).

[25] Rodrig, M., Reis, C., Mahajan, R., Wetherall, D., Zahorjan, J., and Lazowska, E. CRAWDAD data set uw/sigcomm2004 (v. 2006-10-17). http://crawdad.cs.dartmouth.edu.

[26] Saponas, T. S., Lester, J., Hartung, C., Agarwal, S., and Kohno, T. Devices that tell on you: Privacy trends in consumer ubiquitous computing. In *USENIX Security* (2007).

[27] Singelée, D., and Preneel, B. Location privacy in wireless personal area networks. In *WiSe* (2006).

[28] Song, D. X., Wagner, D., and Tian, X. Timing analysis of keystrokes and timing attacks on ssh. In *USENIX Security* (2001).

[29] Song, J., Poovendran, R., Lee, J., and Iwata, T. The AES-CMAC algorithm. RFC 4493, June 2006.

[30] Sun, Q., Simon, D. R., Wang, Y.-M., Russell, W., Padmanabhan, V. N., and Qiu, L. Statistical identification of encrypted web browsing traffic. In *IEEE Security and Privacy* (2002).

[31] Suomalainen, J., Valkonen, J., and Asokan, N. Security associations in personal networks: A comparative analysis. Tech. Rep. NRC-TR-2007-004, Nokia Research Center, Jan. 2007.

[32] Tao, P., Rudys, A., Ladd, A., and Wallach, D. Wireless LAN location sensing for security application. In *WISE* (2003).

[33] Wright, C., Ballard, L., Monrose, F., and Masson, G. Language identification of encrypted VoIP traffic: Alejandra y Roberto or Alice and Bob? In *USENIX Security* (Aug. 2007).

[34] Wright, C., Monrose, F., and Masson, G. On inferring application protocol behaviors in encrypted network traffic. *Journal of Machine Learning Research* (Aug. 2006).