Formal Analysis of Encrypted SNI

No Name

October 26, 2019

Contents

1 Introduction **1**

2 Encrypted SNI Overview **2**
2.1 Security and Privacy Goals 3
2.2 Draft-04 Design Overview and Known Attacks 3

3 Core Protocol **5**

4 Symbolic Model **8**

5 Standard Model **11**
5.1 Adversarial Model 12
5.1.1 Active Adversary 13

6 Summary of Proposals **14**

1 Introduction

The TLS Server Name Indication extension [1] is an increasingly important part of the TLS protocol. Modern TLS server deployments often offer multiple certificates behind a single IP address. The SNI helps servers choose which certificate to choose for a given connection. However, this extension also leaks the server name to any on-path observer. With recent pushes to encrypt DNS and protect names from such adversaries, the SNI extension is a privacy problem for clients. The Encrypted SNI extension [8] attempts to address this privacy problem by encrypting the SNI in transit. However, to date, asserting correctness and privacy properties of the protocol proved difficult.

Formal Analysis. We develop and present a ProVerif model of the core ESNI protocol to show that it achieves the desired secrecy and privacy properties. ProVerif analyzes symbolic protocol models using processes to represent entities which communicate using messages sent over public channels. Processes can trigger security events representing attacks or critical steps of the target protocol, e.g., TLS connection establishment. Moreover, processes can save

messages in lookup tables for use later on. This is useful for storing long-term keying material, such as ESNI and certificate private keys.

Extensions to account for server reaction attacks (see Section 2.1) are described, though not modelled, as there are multiple ways to build this property into the protocol.

All models and code can be found online at `https://github.com/chris-wood/reftls`.

2 Encrypted SNI Overview

Encrypted SNI is a tool for hiding server names from network connections. There are several operational goals for Encrypted SNI [**?**], described below:

- Avoid widely-deployed shared secrets: One approach to the problem would be for all clients and servers to share a secret that encrypts (and decrypts) the SNI. However, any client in this set of trusted peers could then decrypt the SNI of others. Moreover, compromise of any node in possession of the secret puts all members at risk. Thus, ESNI requires public key encryption.
- Work with non-ESNI servers to avoid fallback: Without the need for fallback, ESNI is a simple ECIES-like protocol, wherein the SNI is encrypted under a public key of the service provider.
- Do not introduce extra round trips: Encrypting the SNI must not come at the cost of extra round trips. For example, one possible approach is to SNI-based certificate authentication at the application protocol layer, e.g., using HTTP/2 Secondary Certificates [5], after the TLS connection finishes and is authenticated with a "public name." While this may work, the extra latency cost may be prohibitively expensive for certain clients.
- Forward secrecy: Ideally, SNI encryption would have some amount of forward secrecy. However, as SNI encryption cannot introduce additional round trips, forward secrecy is not possible using public key encryption primitives such as ECIES [9] or HPKE [2]
- Prevent SNI-based DoS attacks: A consequence of using public key encryption is that servers must perform a public key operation without having validated the client. HelloRetryRequests may help dampen the effects of DoS attacks, though these come at the cost of introducing additional complexity into the protocol. See Section 2.1 for more details.
- Mitigate replay attacks: Encrypted SNI values must not be replayable from one ClientHello to another, otherwise an attacker could use an ESNI value from a victim client message in its own ClientHello.
- Support shared and split mode: Client-facing servers which use the SNI to determine the target service may not be the entity which terminates the TLS connection. ESNI should therefore support proxies which route TLS connections to backend or origin services. This suggests two possible deployment models for ESNI, referred to as shared and split mode, as described in [8].

2.1 Security and Privacy Goals

ESNI assumes a standard active and on-path Dolev-Yao attacker that can arbitrarily drop, tamper, replay, and forward messages from clients. Fundamentally, a TLS handshake that negotiates ESNI should leak no more information than one which did not negotiate ESNI in the presence of this adversary. This means there are at least two necessary requirements for ESNI:

1. SNI agreement: A successful TLS handshake implies agreement on the SNI transmitted. This means, among other things, that the client authenticated the server's certificate using the SNI, and that both client and server share the same view of the SNI negotiated.

2. SNI privacy: A successful TLS handshake that negotiates ESNI does so without leaking any information about the underlying SNI. Moreover, the SNI is known only to the client and server (or any recipient of the private ESNI key). We do not require forward secrecy for the SNI encryption.

We may optionally want to hide the fact that ESNI was negotiated, as per the "do not stick out" goal. However, this is primarily only deployment concern. Furthermore, we may also want to hide the fact that a client offered ESNI in its handshake. This may be useful for clients that wish to GREASE [4] the extension.

2.2 Draft-04 Design Overview and Known Attacks

Figure 2 shows the ESNI design in draft-04 of the protocol.

Early versions of ESNI did not achieve the desired security and privacy goals. For example, the first version was vulnerable to a certificate-based client reaction attack shown in Figure **??**. The SNI leak occurs if clients processed the certificate message before verifying the CertificateVerify signature. Specifically, if clients abort upon SNI mismatch between what they sent and what was received in the certificate, an attacker attempt to MITM an ESNI connection with a certificate of its choosing and try to learn the client's SNI. The core problem was that servers did not signal to clients whether or not they processed the ESNI extension. Adding a nonce to the server's response implicitly authenticates that the server processed the SNI.

Despite this fix, draft-04 of ESNI does not achieve the necessary requirements stated above. We describe some more attacks on ESNI deployment configurations and the protocol below.

Probing Attacks. If an operator partitions its servers based on SNI-specific values observable on the wire, such as supported ciphersuites, key exchange algorithms, or application-layer protocols, an adversary can use these differences to learn information about the client's SNI.

HelloRetryRequest Mix and Match. In the event of a HelloRetryRequest, clients send a fresh key share and ESNI extension in the second ClientHello. Servers are expected to use the ESNI value from the same ClientHello

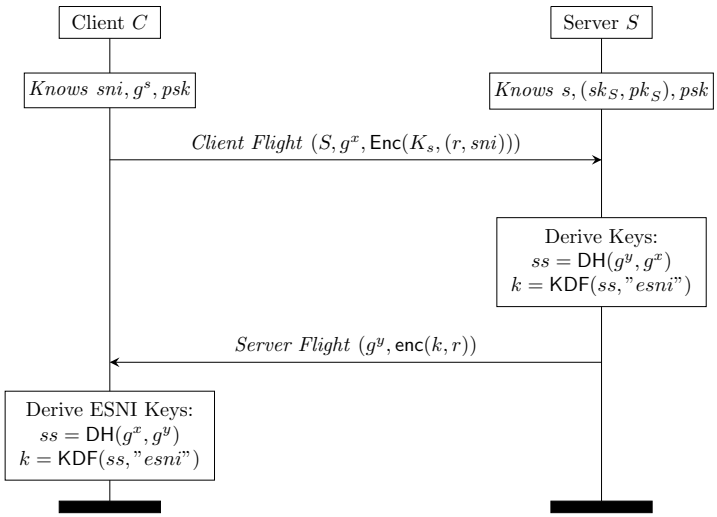

Figure 1: Simple ESNI Protocol without resumption or HelloRetryRequest support.

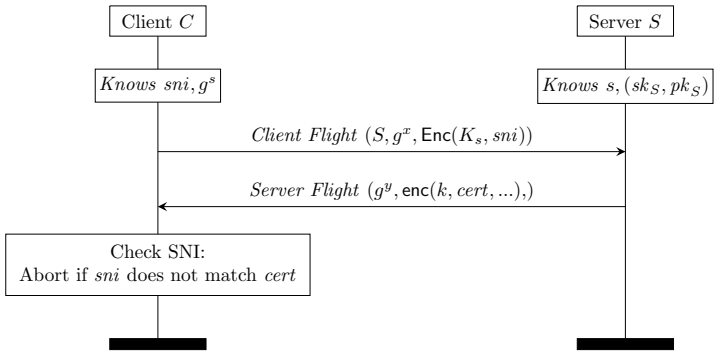

Figure 2: Simple ESNI Protocol without resumption or HelloRetryRequest support.

from which it received the client key shares. However, a server may use mix and match these values, e.g., by selecting the ESNI value from the first ClientHello and the key shares from the second ClientHello.[1] This allows an on-path adversary to hijack a HelloRetryRequest, send a second ClientHello with its own key shares, and then successfully decrypt the server flight to learn the certificate.

[1]This is not entirely farfetched, as some operators must process the SNI upon the first ClientHello in order to determine HelloRetryRequest parameters such as supported ciphersuites.

Server Ticket Reaction Attacks. The ESNI contents are not fully bound to the entire ClientHello in draft-04. This means a ClientHello with ESNI *and* no resumption PSK can be intercepted by an on-path adversary, who then attaches a ticket and PSK binder of its choosing, and forwards the result to the original target server. If servers check whether or not the SNI obtained from the ESNI value and the ticket match and change behavior accordingly, e.g., by aborting the connection, this introduces an oracle for adversaries to learn information about client SNIs.

In general, the problems above seem to stem from the same problems: (1) ESNI contents are not fully bound to the ClientHello, and (2) ESNI contents are not fully bound to the rest of the TLS handshake. In the following section, we describe an ideal ESNI protocol that addresses these shortcomings.

3 Core Protocol

At its core, ESNI is a protocol between a client and server that works as described in Figure 3. It aims to provide the following guarantees:

- **Client Secrecy**. TLS handshake secret known by entity which has the private handshake key share (y), corresponding PSK, private ESNI decryption key, and ENSI nonce.
- **Server Secrecy**. TLS handshake secret known by entity which has the private handshake key share (x), corresponding PSK, and ENSI nonce.
- **Agreement**. Client and server both agree on the same TLS handshake secret and transcript.

Thus, the core protocol only models the handshake *up to the point of certificate receipt*. It does not capture the full TLS handshake. Moreover, the core protocol does not account for server reaction attacks, such as the one described in Section 2.1. We treat cryptographic mitigation of this to an extension of the core protocol.

For presentation purposes, the core protocol uses the following helper routine:

$\mathsf{DeriveESNIKeys}(Zx)$

$ek = \mathsf{KDF}(zx, \texttt{'esnikey'})$
$eiv = \mathsf{KDF}(zx, \texttt{'esniiv'})$
return ek, eiv, r

To support HelloRetryRequest flows, we also introduce a function GenerateHRR which produces a HelloRetryRequest message HRR given a ClientHello CH.

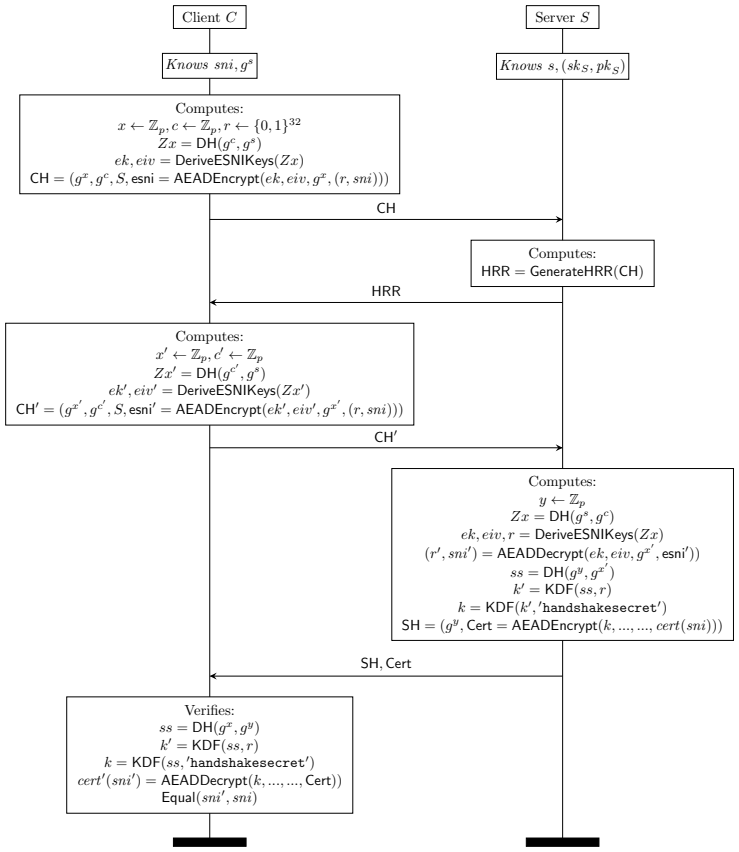

Figure 3: Minimal Core ESNI Protocol without server reaction attack prevention.

The HRR path generates a fresh key share rather than re-using the value from the first ClientHello. The key share can only be re-used if the key derivation changes, e.g., with a different label to `HKDF-Expand`, since this otherwise leads to encryption key and nonce re-use. The core protocol only requires that the client key share is bound to the ESNI contents. There are multiple ways to achieve this property. The ideal solution is one in which *ESNI contents do not change across ClientHello messages*.

The core protocol achieves what we call *forward binding* of the handshake to the ESNI shared secret, even in the event of HRR. This means that knowledge of both ESNI secret(s) and one of the TLS key shares is needed to derive the handshake secrets.

As a consequence of forward binding and the need to interoperate with ESNI-incapable servers, we also require a signalling mechanism for clients to determine

whether or not the ESNI contents were used to protect the handshake secret. (Trial decryption is always an option, though other more practical solutions exist.)

The core protocol described above is *not* secure against server reaction attacks. Specifically, it allows an on-path adversary to tamper with a ClientHello with no binder in flight. Recall that the ticket-based server reaction attack relies on servers behaving differently if the ESNI contents do not match that of the ticket contents. In general, though, this type of reaction attack is possible in all cases where a server performs SNI-specific behavior on unauthenticated fields in the ClientHello.

There are at least two ways to mitigate this problem:

- Require that servers not implement any SNI-specific check on unauthenticated fields in the ClientHello. This requires no protocol change, yet may be fragile in practice.
- Use ESNI to authenticate the entire ClientHello. This requires a protocol change.

For safety reasons, the second option seems prudent. Thus, to mitigate server reaction attacks, we require the ESNI contents to authenticate the ClientHello, or to be backward bound to the handshake. ESNI contents are *backward bound* to a ClientHello if it is not possible to modify a ClientHello in any way without causing an ESNI check to fail. This modification includes, among other things, the addition of a PSK binder where one was previously not present.

Beyond backward and forward binding, it must also be the case that observable information, which includes messages sent on the wire and the *actions* of clients and servers using ESNI, does not leak information about the SNI. Indeed, the reaction attack described above was due in part to such an information leak.[2] We must capture this notion of information indistinguishability for completeness. We do so via *message indistinguishability* and *action indistinguishability*. Informally, message indistinguishability means that all messages written on the wire do not vary based on SNI. Similarly, action indistinguishability means that all node behavior as observed by Adv does not vary based on SNI (or any SNI-influenced value used in the protocol).

We do not currently model message indistinguishability. This is because it is mainly determined by configuration. For example, if a service provider partitions its names based on support ciphersuites, then ciphersuite selection may leak information about the SNI. (This also applies to clients, which may, in theory, use different ciphersuites, parameters, or extensions per domains.) Essentially, message indistinguishability requires that all parameter selection *within* the same anonymity set is SNI-agnostic.

[2]Fortunately, backward binding resolves this problem by removing a branch in the way servers handle ClientHello messages.

4 Symbolic Model

We developed a model for the core ESNI protocol based on ProVerif [6]. The core model achieves the forward binding property. It does not account for backwards binding, nor does it account for message indistinguishability, as this is something largely determined by configuration. Each process begins by installing or obtaining long-term secrets used each TLS connection. These secrets include public and private certificate signing keys, as well as ESNI keying material.

```
event ESNIClientDone(bitstring, element, pubkey).
event ESNIClientLeak(bitstring, element, pubkey).
event ESNIServerLeak(bitstring, element, bitstring).
event ESNIServerDone(bitstring, element, pubkey).
(*event Reachable(). *)

let ClientESNICore() =
    in(io, (host:prin, origin_host:prin));
    get esniKeys(= host, = StrongDH, xxx, gs) in
    let g = StrongDH in
    let sni = secret_sni(origin_host) in

    (* Derive ESNI keying material *)
    new cr:random;
    let (x:bitstring, gx:element) = dh_keygen(g) in
    let (c:bitstring, gc:element) = dh_keygen(g) in
    new sni_nonce:random;
    let gcs = e2b(dh_exp(g, gs, c)) in
    let kcs = hkdf_extract(zero, gcs) in
    let aek = hkdf_expand_label(kcs, tls13_esni_key, random2b(cr)) in
    let aeiv = hkdf_expand_label(kcs, tls13_esni_iv, random2b(cr)) in

    (* Derive ESNI keying material *)
    let esni_payload = (sni_nonce, sni) in
    let enc_sni = aead_enc(StrongAE, b2ae(aek), aeiv, e2b(gx), esni_payload) in

    (* The SNI extension carries the "public name", which is modelled
  here as the public key share of the public name *)
    let sni_ext = SNI(e2b(gs)) in

    (* The ESNI extension carries the the client's ESNI key share
  and the encrypted SNI payload *)
    let esni_ext = ESNI(g, gc, enc_sni) in

    (* Write out the CH and get a HRR in response *)
    let ch = (cr, gx, sni_ext, esni_ext) in
    out(io, ch);
```

```
in (io, (hrr:bitstring));

(* Upon HRR, the ESNI payload does not change. Only the client's key shares change. *)
new cr:random;
let (x:bitstring, gx:element) = dh_keygen(g) in
let (c:bitstring, gc:element) = dh_keygen(g) in
let gcs = e2b(dh_exp(g, gs, c)) in
let kcs = hkdf_extract(zero, gcs) in
let aek = hkdf_expand_label(kcs, tls13_esni_key, random2b(cr)) in
let aeiv = hkdf_expand_label(kcs, tls13_esni_iv, random2b(cr)) in
let enc_sni = aead_enc(StrongAE, b2ae(aek), aeiv, e2b(gx), (sni_nonce, sni)) in
let sni_ext = SNI(e2b(gs)) in
let esni_ext = ESNI(g, gc, enc_sni) in

(* Write out the CH and get a server flight in response *)
let ch' = (cr, gx, sni_ext, esni_ext) in
out(io, ch');
in (io, (gy:element, msg:bitstring));

if gy ≠ BadElement then
let log = (ch, hrr, ch', gy) in
let gxy = e2b(dh_exp(g, gy, x)) in

(* Derive the handshake secret as per normal. *)
let kxy' = hkdf_extract(zero, gxy) in

(* Mix in the ESNI nonce. *)
let kxy = hkdf_extract(kxy', random2b(sni_nonce)) in

(* Derive remaining key material. *)
let aek = hkdf_expand_label(kxy, tls13_key, log) in
let aeiv = hkdf_expand_label(kxy, tls13_iv, log) in

(* Decrypt the server flight *)
let ((recv_sni:bitstring, p:pubkey), sig:bitstring) = aead_dec(StrongAE, b2ae(aek), aeiv, zero, msg) in
let log = (log, (recv_sni, p)) in

event Reachable(log);

get longTermKeys(= origin_host, xxxx, p) in
if recv_sni = sni then (
 if verify(p, log, sig) = true then
    event ESNIClientDone(sni, gs, p))
else (
     event ESNIClientLeak(sni, gs, p);
 out(io, sni)).
```

```
let ServerESNICore() =
    in(io, host:prin);
    get esniKeys(= host, = StrongDH, s, gs) in
    get longTermKeys(= host, sk, p) in
    let g = StrongDH in

    (* Read in a CH. *)
    in(io, (ch:bitstring));
    let (cr:random, gx:element, sni_ext:sni, esni:sni) = ch in

    (* Decrypt the ESNI extension. *)
    let ESNI(g, gc, enc_sni) = esni in
    if gx ≠ BadElement then
    if gc ≠ BadElement then
    let gcs = e2b(dh_exp(g, gc, s)) in
    let kcs = hkdf_extract(zero, gcs) in
    let aek = hkdf_expand_label(kcs, tls13_esni_key, random2b(cr)) in
    let aeiv = hkdf_expand_label(kcs, tls13_esni_iv, random2b(cr)) in
    let (sni_nonce:random, sni:bitstring) =
        aead_dec(StrongAE, b2ae(aek), aeiv, e2b(gx), enc_sni) in

    let origin_host = get_host(global_sni_secret(), sni) in

    (* Produce a HRR, and then read another CH. *)
    let hrr = zero in
    out (io, hrr);
    in (io, (ch':bitstring));
    let (cr':random, gx':element, sni_ext':sni, esni':sni) = ch' in

    (* Decrypt the ESNI contents again. *)
    let ESNI(g', gc', enc_sni') = esni' in
    if gx' ≠ BadElement then
    if gc' ≠ BadElement then
    let gcs = e2b(dh_exp(g', gc', s)) in
    let kcs = hkdf_extract(zero, gcs) in
    let aek = hkdf_expand_label(kcs, tls13_esni_key, random2b(cr')) in
    let aeiv = hkdf_expand_label(kcs, tls13_esni_iv, random2b(cr')) in
    let (sni_nonce':random, sni':bitstring) =
        aead_dec(StrongAE, b2ae(aek), aeiv, e2b(gx'), enc_sni') in

    let origin_host = get_host(global_sni_secret(), sni') in

    let (y:bitstring, gy:element) = dh_keygen(g) in
    let log = (ch, hrr, ch', gy) in
```

```
    let gxy = e2b(dh_exp(g, gx', y)) in

    (* Derive the handshake secret as per normal. *)
    let kxy' = hkdf_extract(zero, gxy) in

    (* Mix in the SNI nonce. *)
    let kxy = hkdf_extract(kxy', random2b(sni_nonce')) in

    (* Derive remaining key material. *)
    let aek = hkdf_expand_label(kxy, tls13_key, log) in
    let aeiv = hkdf_expand_label(kxy, tls13_iv, log) in

    (* Encrypt the server flight, which includes the SNI (or would-be certificate)
 and signature over the transcript. *)
    let log = (log, (sni', p)) in
    let sig = sign(sk, log) in
    let enc_sflt = aead_enc(StrongAE, b2ae(aek), aeiv, zero, ((sni', p), sig)) in
    out(io, (gy, enc_sflt));
    event ESNIServerDone(sni, gs, p).

query h:bitstring, gs:element, p:pubkey;
      event(ESNIClientDone(h, gs, p));
      event(ESNIServerDone(h, gs, p)).

query origin_host:prin, h:bitstring, gs:element, p:pubkey, tkt:bitstring;
      attacker(secret_sni(origin_host));
      event(ESNIClientLeak(h, gs, p));
      event(ESNIServerLeak(h, gs, tkt)).

query log:bitstring; event(Reachable(log)).

const C:prin.
process
    new keyA:ae_key;
    new keyB:ae_key;
    !ClientESNICore() |!ServerESNICore() |!LongTermKeys()
```

5 Standard Model

Let $N \in \{0,1\}^{2^{16}-1}$ be a name, and let $\mathbf{N}$ be the set of possible names. ClientHello messages typically carry names as one of their parameters. Let ClientConfig be a set of parameters that determine a client's configuration, including supported ciphersuites, named groups, extensions, etc. Let $\mathbf{CC}$ be the set of all client configuration parameters. Similarly, let ServerConfig be a set

of parameters that determine a server's configuration, and let **SC** be the set of all client configuration parameters. Clients produce CH messages to a given server, identified by its name N, and configuration ClientConfig, using a function GenerateCH : $\mathbf{N} \times \mathbf{CC}$. Servers produce SH messages (and do other parameter selection) based on a CH message and server configuration, using a function GenerateSH : $\mathbf{CH} \times \mathbf{SC}$.

The function Metadata takes as input a CH or SH and returns the set of metadata, or parameters, associated with the message, including the server name N, ciphersuite list, named groups, and other extensions that are visible on the wire. The function PublicMetadata returns the same output as Metadata, except that the server name is omitted.

Using the notation above, every handshake is a probabilistic function of some server name N, ClientConfig, and ServerConfig. In particular:

- A client generates a CH message using N and ClientConfig.
- The recipient server generates a SH message using the input CH and ServerConfig.
- The remainder of the handshake contains encrypted handshake messages.

Thus, let Handshake be a function that takes as input N, cc = ClientConfig, and sc = ServerConfig and produces a trace t of messages $m_1, m_2, \ldots$, where m_1 is a CH and m_2 is a SH. Let PublicHandshake be a similar function that computes $t = m_1, m_2, \cdots$ = Handshake(t, cc, sc) and returns t^v = PublicMetadata(m_1), PublicMetadata$(m_2), \ldots$. That is, it returns publicly visible metadata associated from a handshake trace.

Finally, we define a function SNI which takes as input a handshake trace t and returns the name N which was used to generate t.

5.1 Adversarial Model

In this section, we describe two variants of an ESNI adversary: passive and active. We derive definitions from the classic "find-and-choose" notion of indistinguishability. Informally, in each game, the adversary Adv is given information about a game and must present two options for the challenger. The challenger chooses one option at random and presents the result of some operation applied to the selected option to Adv, who must then identify which of the two options was selected. (This is typically the encryption of one message.) If the Adv's advantage in making this selection is negligibly later than 0.5, then the options are indistinguishable under the challenger's operation.

In porting this definition to ESNI, we seek to capture indistinguishability of handshake traces derived from names. That is, the challenger's operation is to generate a public handshake trace based on a randomly chosen name. Specifically, Adv chooses two names N_0 and N_1 during a "find" phase, and presents them to the challenger C. C then generates t_b^v = PublicHandshake(N_b), where $b \leftarrow \{0, 1\}$, and presents this to Adv. The adversary then presents its selection b' to the challenger. The output of the game depends on whether or not $b = b'$.

We capture this definition in the following NameGame, which is defined in Algorithm 1.

Algorithm 1 NameGame

1: On input security parameter λ, $\mathbf{N}$, $\mathbf{CC}$, $\mathbf{SC}$, and adversary Adv a challenger C initializes Adv with $\mathbf{CC}$, $\mathbf{SC}$, and N.
2: Adv presents the challenger with two distinct names N_0 and N_1, where $N_0 \neq N_1$.
3: C chooses a random bit b and returns $t_b^v = \mathsf{PublicHandshake}(N_b)$ to Adv.
4: Adv replies with a bit b'.
5: Output 1 if $b = b'$, otherwise output 0.

We say that Adv wins NameGame if it outputs 1, i.e., if Adv was able to determine which name the challenger used to produce the challenge handshake trace. We define the advantage Adv has in this game as $|\Pr[\mathsf{NameGame}(\lambda, \mathbf{N}, \mathbf{CC}, \mathbf{SC}, \mathsf{Adv})] - \frac{1}{2}|$.

We say that a handshake trace is *SNI agnostic* if, for all PPT adversaries Adv, Adv's advantage in winning NameGame is negligibly small in λ.

5.1.1 Active Adversary

An active adversary has the ability to generate handshake traces at will. We assume servers are neither malicious nor compromised. Therefore, adapting our model for privacy requires extending NameGame to give Adv an oracle for producing handshake traces with an SNI of its choosing. We define this oracle as $\mathcal{O}_H$, and it takes as input an SNI $N \in \mathbf{N}$ and returns a handshake trace $H(N) \in \mathbf{H}$. The modified game, called ActiveNameGame, proceeds as follows:

Algorithm 2 ActiveNameGame

1: On input security parameter λ, $\mathbf{N}$, $\mathbf{CC}$, $\mathbf{SC}$, and adversary Adv a challenger C initializes Adv with $\mathbf{CC}$, $\mathbf{SC}$, N, and access to $\mathcal{O}_H$.
2: Adv queries $\mathcal{O}_H$ with names values of its choosing from $\mathbf{N}$ at most $\mathsf{poly}(\lambda)$ times.
3: Adv presents the challenger with two distinct names N_0 and N_1, where $N_0 \neq N_1$.
4: C chooses a random bit b and returns $t_b^v = \mathsf{PublicHandshake}(N_b)$ to Adv.
5: Adv continues querying $\mathcal{O}_H$ at most $\mathsf{poly}(\lambda)$ times.
6: Adv replies with a bit b'.
7: Output 1 if $b = b'$, otherwise output 0.

As before, we say that Adv wins ActiveESNIGame if it outputs 1. We define the advantage Adv has in this game as $|\Pr[\mathsf{ActiveESNIGame}(\lambda, \mathbf{N}, \mathsf{config}, \mathsf{Adv})] - \frac{1}{2}|$. And we say that a handshake trace is *SNI agnostic* with respect to an active attacker if Adv's advantage in winning ActiveESNIGame is negligibly small in λ.

6 Summary of Proposals

There are two proposals that conform to the core ESNI protocol described in Section 3. They are summarized below.

ESNI Proxy Transformation. A high level summary of this proposal is as follows:

- Bind the ESNI contents to the entire ClientHello with an explicit transformation function that works as follows. Given a fully-formed "private" ClientHello (with the unencrypted SNI value and ESNI nonce), encrypt the SNI value using the ClientHello as AAD, and output a "public" ClientHelo with the SNI extension replaced with the ESNI extension.
- Bind the ESNI contents to the handshake by choosing either the "public" or reconstructed "private" ClientHello to mix into the transcript.
- Do not send an ESNI extension in the event of HRR.

It has the following properties:

+ ClientHello messages sent in response to HelloRetryRequest messages do not carry an ESNI extension, which means that servers must maintain state about which SNI was chosen from the first ClientHello.

– The transcript used upon ESNI negotiation is not that which is sent over the wire. This is a significant deviation from the TLS 1.3 model and introduces interesting side effects, such as the ability of an unknown third party, e.g., a backend origin server, to complete the handshake without the client being aware. (This is a problem in TLS in general, as secret information can always be exfiltrated to another party to complete the handshake.)

– Treats the handshake transcript, which is fed into the TLS key schedule as the `Info` parameter to `HKDF-Expand`, as secret information. (In contrast, the normal transcript in TLS 1.3 is composed of public information.) Note that HKDF leaks no information about this parameter if modelled as a random oracle or if the key is secret. More generally, PRFs have no guarantees about the secrecy of their inputs if the key is known. Importantly, in this proposal, an attacker does know the key in the event of HRR, yet the input is secret. Bellare and Lysyanskaya [3] proved that HMAC satisfies the notion of a dualPRF, which roughly states that HMAC is a PRF if either the the key or the input is secret. Thus, in practice, this should not affect security, though it may affect the proofs used for TLS 1.3.

ESNI PSK Binders and Key Schedule Injection. A high level summary of this proposal is as follows:

- Bind the ESNI contents to the entire ClientHello with an explicit PSK binder, whose value is derived from the ESNI shared secret.

- Bind the ESNI contents to the handshake by mixing the derived ESNI nonce into the key schedule.
- Always generate and send an ESNI extension. In the event of HRR, the contents of this ESNI extension are identical to that of the first ClientHello.

It has the following properties:

+ Backward binding relies on existing PSK binder properties.
+ HelloRetryRequest contents do not change across ClientHello messages, simplifying server implementations.
– Forward binding requires key schedule modification.
– In the event of session resumption, two binders are added to a ClientHello and both must be verified by the server. This goes against text in the TLS 1.3 specification [7], which states that servers "SHOULD NOT attempt to validate multiple binders."

References

[1] Donald E. Eastlake 3rd. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, January 2011.

[2] Richard Barnes and Karthikeyan Bhargavan. Hybrid Public Key Encryption. Internet-Draft draft-irtf-cfrg-hpke-00, Internet Engineering Task Force, July 2019. Work in Progress.

[3] Mihir Bellare and Anna Lysyanskaya. Symmetric and dual prfs from standard assumptions: A generic validation of an hmac assumption. *IACR Cryptology ePrint Archive*, 2015:1198, 2015.

[4] David Benjamin. Applying GREASE to TLS Extensibility. Internet-Draft draft-ietf-tls-grease-04, Internet Engineering Task Force, August 2019. Work in Progress.

[5] Mike Bishop, Nick Sullivan, and Martin Thomson. Secondary Certificate Authentication in HTTP/2. Internet-Draft draft-ietf-httpbis-http2-secondary-certs-04, Internet Engineering Task Force, April 2019. Work in Progress.

[6] Bruno Blanchet et al. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135, 2016.

[7] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[8] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. Encrypted Server Name Indication for TLS 1.3. Internet-Draft draft-ietf-tls-esni-04, Internet Engineering Task Force, July 2019. Work in Progress.

[9] Victor Shoup. A proposal for an iso standard for public key encryption (version 2.1). *IACR e-Print Archive*, 112, 2001.