Formal Analysis of Encrypted SNI

Karthikeyan Bhargavan, Eric Rescorla, Christopher A. Wood

November 5, 2019

Contents

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.

Core Protocol. We capture the essence of ESNI in a *core protocol*, which is a simplified variant of TLS that models TLS up to the receipt of the certificate message.

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 modeled, 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 Existing Design Overview and Known Attacks

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

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.

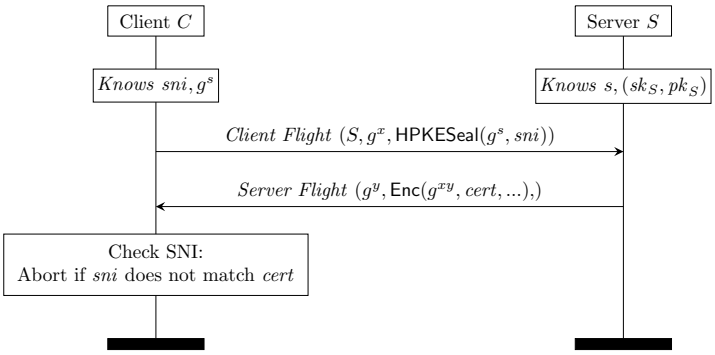


Figure 1: Client reaction attack.

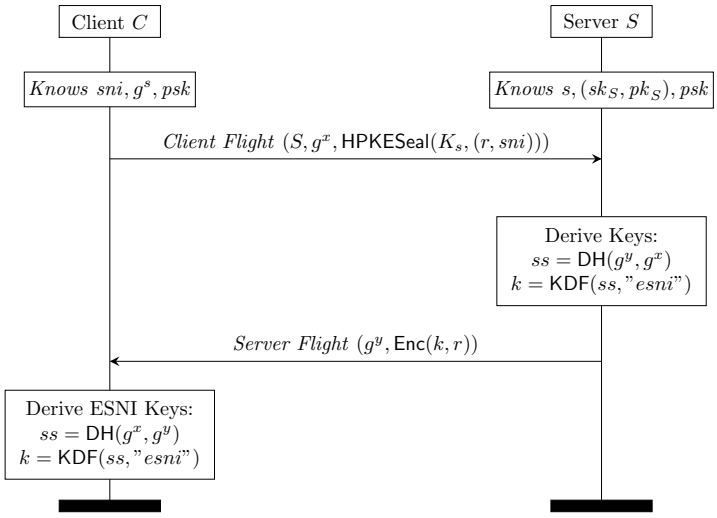


Figure 2: draft-ietf-tls-esni-04 protocol (without HRR).

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 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

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

shares, and then successfully decrypt the server flight to learn the certificate. This attack flow is shown in Figure 3.

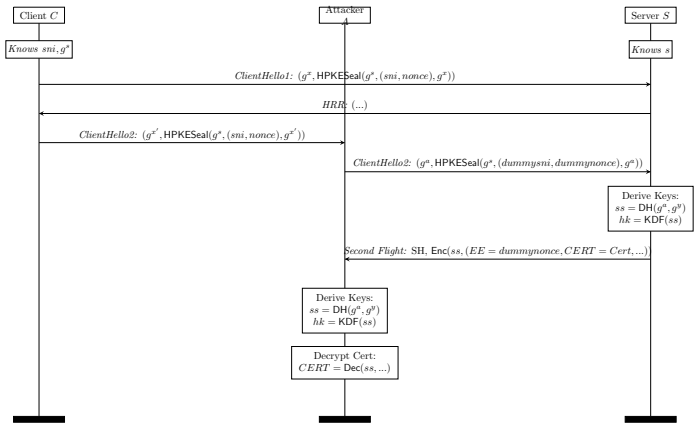
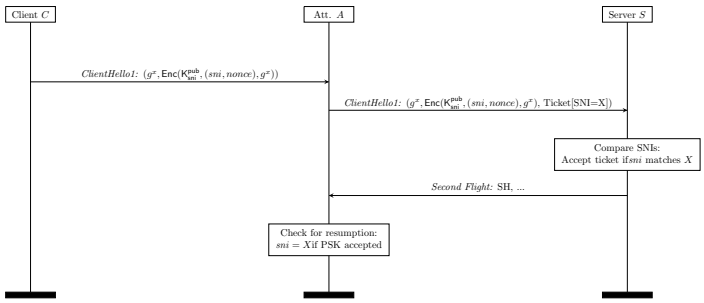


Figure 3: Active HRR attack.

This reveals the basic source of the problem: in order to prevent mix-and-match attacks on the client's key share, the ESNI encryption binds in g^x. However, if the server uses the SNI from first ClientHello and then the key share from the second ClientHello, which is attacker-controlled, then the attacker is free to generate their own ESNI extension containing a bogus ESNI and nonce. The server correctly verifies that these correspond to g^a but because these are all supplied by the attacker, this check succeeds and the server encrypts the certificate to the attacker with a key known to the attacker.

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. This attack, which is shown in Figure **??**, is also a mix-and-match attack.

This is a failure of binding: currently, ESNI is not bound enough of the ClientHello. (All of the proposals described in Section 4 prevent this attack in the same way: they bind the ESNI to the entire ClientHello. If the attacker attaches their own PSK, then this invalidates the binding, thus preventing the mix-and-match. None of this depends on the confidentiality of the ClientHello.)



3 Core Protocol

This is still work in progress undergoing refinements.

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 this section, we describe an ideal ESNI protocol that addresses these shortcomings. At its core, ESNI is a protocol between a client and server that 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. To support HelloRetryRequest flows, we also introduce a function GenerateHRR which produces a HelloRetryRequest message HRR given a ClientHello CH.

Beyond these core protocol goals, 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*.

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

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.

There are two variants of the core protocol. One keeps the *key schedule* secret, and another keeps the *transcript* secret. We model both in Section **??** and show they achieve the desired goals. We describe them in the following subsections. The ProVerif ProVerif [6] model used to assert their correctness is also included. Note that our models do not account for message indistinguishability, as this is something largely determined by configuration.

3.1 Key Schedule Secrecy Variant

This core protocol variant achieves what we call *forward binding* of the handshake to the ESNI shared secret, even in the event of HRR. Forward binding ensures the client and server secrecy properties previously described. Note that the core protocol as described achieves this by mixing the ESNI nonce into the TLS key schedule. Alternative designs that have the same effect exist, and are discussed later in Section 4.

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.)

This core protocol variant 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.

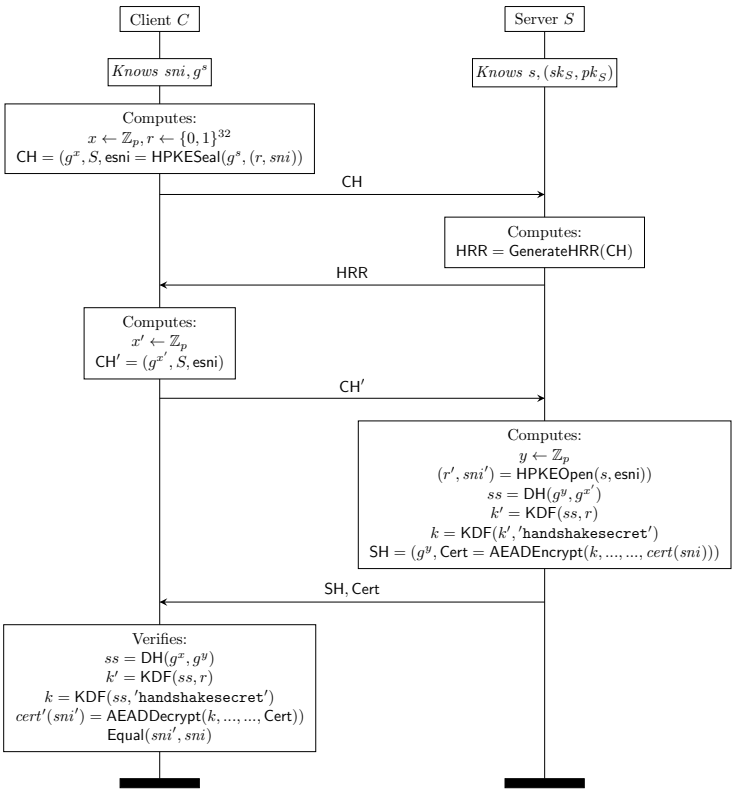


Figure 4: Minimal Core ESNI Protocol with key schedule secrecy.

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.

The ProVerif model for this variant is below.

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

let ClientESNICoreHRR() =
    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 handshake keying material *)
  new cr:random;
  let (x:bitstring, gx:element) = dh_keygen(g) in

  (* Compute HPKE(g^s, (sni_nonce, sni)) *)
  new sni_nonce:random;
  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, zero) in
  let aeiv = hkdf_expand_label(kcs, tls13_esni_iv, zero) in

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

  (* The SNI extension carries the "public name", which is modeled
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));

  (* Generate a new CH with a fresh key share but same ESNI contents. *)
  let (x:bitstring, gx:element) = dh_keygen(g) in
  let ch' = (cr, gx, sni_ext, esni_ext) in
  out(io, ch');

  (* Read the server flight *)
  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

    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 ServerESNICoreHRR() =
    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

    (* And send a HRR in response *)
    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. *)
    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, zero) in
    let aeiv = hkdf_expand_label(kcs, tls13_esni_iv, zero) in
    let (sni_nonce:random, sni:bitstring) =
```

```
        aead_dec(StrongAE, b2ae(aek), aeiv, zero, 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)).

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

3.2 Transcript Secrecy Variant

Note that this diagram does not show the flow where the server chooses the outer "public" ClientHello. As in the prior core protocol variant, some signalling

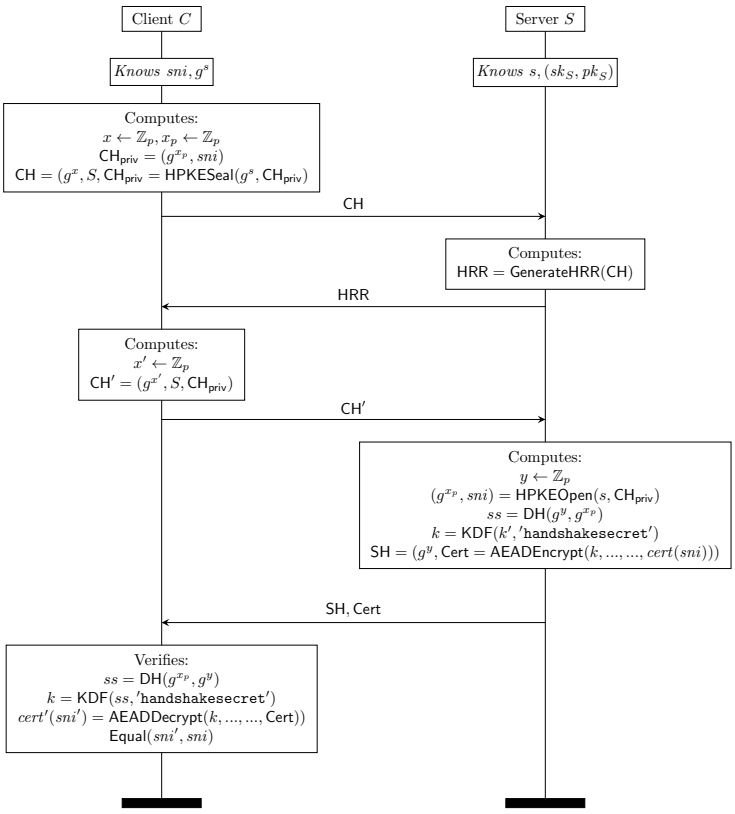


Figure 5: Minimal Core ESNI Protocol with transcript secrecy.

mechanism is needed for clients to determine which ClientHello was used.

Unlike the prior variant, this design *does* achieve forward and backward binding by design, since the private ClientHello is authenticated via encryption. (Note that the interaction with session resumption is also left out. In practice, this variant might include ESNI resumption PSK binders only in the private ClientHello.) However, this design comes at the cost of building handshake secrecy on secrecy of the transcript. We discuss this property in Section 4.

The ProVerif model for this variant is below.

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

let *ClientESNICoreHRR*() =

```
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 handshake keying material *)
new cr:random;
let (x:bitstring, gx:element) = dh_keygen(g) in

(* Compute HPKE(g^s, (g^x, ch)) = HPKE(g^s, (g^x', sni))) *)
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, zero) in
let aeiv = hkdf_expand_label(kcs, tls13_esni_iv, zero) in

(* Derive the private ClientHello *)
let (x2:bitstring, gx2:element) = dh_keygen(g) in
let private_ch = (gx2, sni) in
let tunnel_ch = aead_enc(StrongAE, b2ae(aek), aeiv, zero, private_ch) in

(* The SNI extension carries the "public name", which is modeled
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 tunnel_ext = (g, gc, tunnel_ch) in

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

in (io, (hrr:bitstring));

(* Generate a new CH with a fresh key share but same ESNI contents. *)
let (x:bitstring, gx:element) = dh_keygen(g) in
let ch' = (cr, gx, sni_ext, tunnel_ext) in
out(io, ch');

(* Read the server flight *)
in (io, (gy:element, msg:bitstring));

if gy ≠ BadElement then

(* Derive the handshake secret using the private CH *)
```

```
    let log = (private_ch, gy) in
    let gxy = e2b(dh_exp(g, gy, x2)) in
    let kxy = hkdf_extract(zero, gxy) in
    let aek = hkdf_expand_label(kxy, tls13_key, log) in
    let aeiv = hkdf_expand_label(kxy, tls13_iv, log) in

    let ((recv_sni:bitstring, p:pubkey), sig:bitstring) = aead_dec(StrongAE, b2ae(aek), aeiv, zero, msg) in
    if recv_sni = sni then (
      let log = (log, (recv_sni, p)) in
      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)))
   else (
      (* Derive the handshake secret using the public CH *)
      let log = (ch, gy) in
      let gxy = e2b(dh_exp(g, gy, x)) in
      let kxy' = hkdf_extract(zero, gxy) in
      let aek = hkdf_expand_label(kxy, tls13_key, log) in
      let aeiv = hkdf_expand_label(kxy, tls13_iv, log) in
      let ((recv_sni:bitstring, p:pubkey), sig:bitstring) = aead_dec(StrongAE, b2ae(aek), aeiv, zero, msg) in
      let log = (log, (recv_sni, p)) in
       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 ServerESNICoreHRR() =
    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, tunnel_ext:bitstring) = ch in

    (* And send a HRR in response *)
    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 private CH. *)
    let (g:group, gc:element, tunnel_ch:bitstring) = tunnel_ext 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, zero) in
    let aeiv = hkdf_expand_label(kcs, tls13_esni_iv, zero) in

    let (gx2:element, sni:bitstring) =
        aead_dec(StrongAE, b2ae(aek), aeiv, zero, tunnel_ch) in
    let private_ch = (gx2, sni) in

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

    let (y:bitstring, gy:element) = dh_keygen(g) in
    let log = (private_ch, gy) in
    let gxy = e2b(dh_exp(g, gx2, y)) in

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

let ServerNoESNICoreHRR() =
    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, tunnel_ext:bitstring) = ch in
```

```
    let SNI(e2b(= gs)) = sni_ext in

    let (y:bitstring, gy:element) = dh_keygen(g) in
    let log = (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

    (* 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, (gs, p)) in
    let sig = sign(sk, log) in
    let enc_sflt = aead_enc(StrongAE, b2ae(aek), aeiv, zero, ((e2b(gs), p), sig)) in
    out(io, (gy, enc_sflt));
    event ESNINonESNIServerDone(e2b(gs), gs, p).

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

const C:prin.
process
    new keyA:ae_key;
    new keyB:ae_key;
    !ClientESNICoreHRR() |!ServerESNICoreHRR() |!ServerNoESNICoreHRR() |!LongTermKeys()
```

4 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. This proposal implements the transcript secrecy core protocol variant. A high level summary of this proposal is as follows:

- Bind the ESNI contents to the entire ClientHello with an explicit trans-

formation 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 modeled 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 Tunnel Transformation. This proposal implements the transcript secrecy core protocol variant. A high level summary of this proposal is as follows:

- Encrypt a fully formed "private" ClientHello under the server's public ESNI key. Then, include this encryption as an extension in an outer "public" ClientHello.

- Bind the ESNI contents to the handshake by choosing either the "public" or reconstructed "private" ClientHello to mix into the transcript.

This proposal is similar to the proxy transformation change, with the following notable differences:

+ The private ClientHello is encrypted and authenticated, rather than simply authenticated.

– The public ClientHello nearly doubles in size to store the encrypted private ClientHello.

ESNI PSK Binders and Key Schedule Injection. This proposal implements the key schedule secrecy core protocol variant. 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."

A Attack Flow Discussion

A.1 HRR Attack

With the HRR attack from , we now examine why each of the variants described above prevents the attack. In each case, the attacker is unable to determine the handshake keys and therefore is not able to decrypt the certificate.

- The "proxy transformation" approach *assumes* that the server is going to use the CH1 SNI but then mixes in the nonce from CH1. The result is that although the attacker knows ss, he will not be able to determine the handshake keys.
- The "tunnel transformation" approach is agnostic about whether the server is going to use the CH1 SNI or the CH2 SNI. However, as with the proxy transformation approach, the attacker does not know the nonce and therefore is unable to determine the handshake keys.

- The "key schedule injection" approach is also agnostic about whether the server uses the CH1 SNI or the CH2 SNI, but because it injects secret information from the ESNI encryption into the key schedule, the attacker is unable to determine the handshake keys.

In the first two variants, the secrecy of the handshake keys is provided by having the transcript be secret. In the last variant, the secrecy of the handshake keys is provided by explicitly including secret information from the ESNI encryption into the key schedule. The diagram below shows this, The key step is that hk is derived from ss plus some secret, which is unknown to the attacker, and therefore the attacker is unable to re-derive hk.

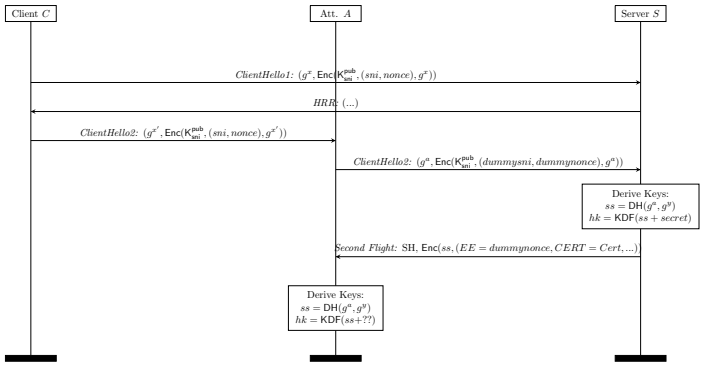


Figure 6: Modified HRR Attack.

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.