How Secure and Quick is QUIC?

Provable Security and Performance Analyses

Robert Lychev[*], Samuel Jero[+],

Alexandra Boldyreva[*], and Cristina Nita-Rotaru[++]

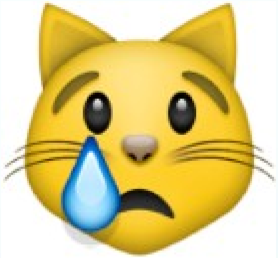
[*]Georgia Institute of Technology

[+]Purdue University

[++]Northeastern University

Minimizing Latency

- Proliferation of mobile and web applications has made latency a very important issue for online businesses
  - users might visit a web site less often if it is slower than a competitor by over 250ms, S. Lohler NY Times 2012
  - 100ms latency costs Amazon 1% in sales, G. Linden, 2006
- Bandwidth is cheap and will continue to grow, but information cannot travel faster than the speed of light



my internets are so slow

Challenge: minimize number of RTT's required to establish a connection, without sacrificing security

What is QUIC?

- Google's answer to the latency challenge
- Stands for **Q**uick **U**DP **I**nternet **C**onnections
- Communication protocol developed by Google and implemented as part of Chrome browser in 2013
- Was designed to
  - produce security protection comparable to TLS
  - reduce connection latency

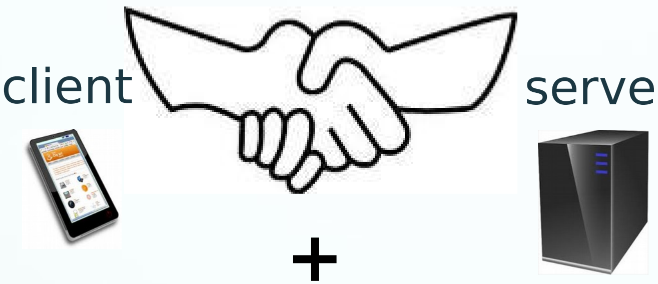
Can QUIC do this in presence of attackers?

Setup Time: QUIC vs TLS

TLS over TCP

TCP session establishment

client



server

+

TLS key establishment

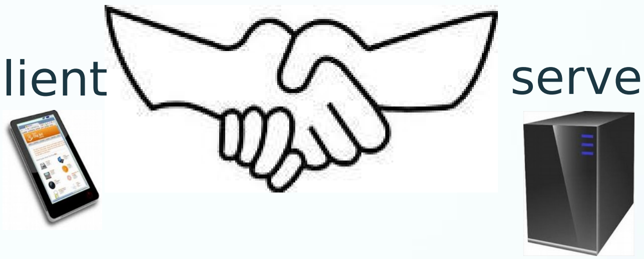
exchange data

setup
latency

QUIC

connection establishment
and key agreement

client



server

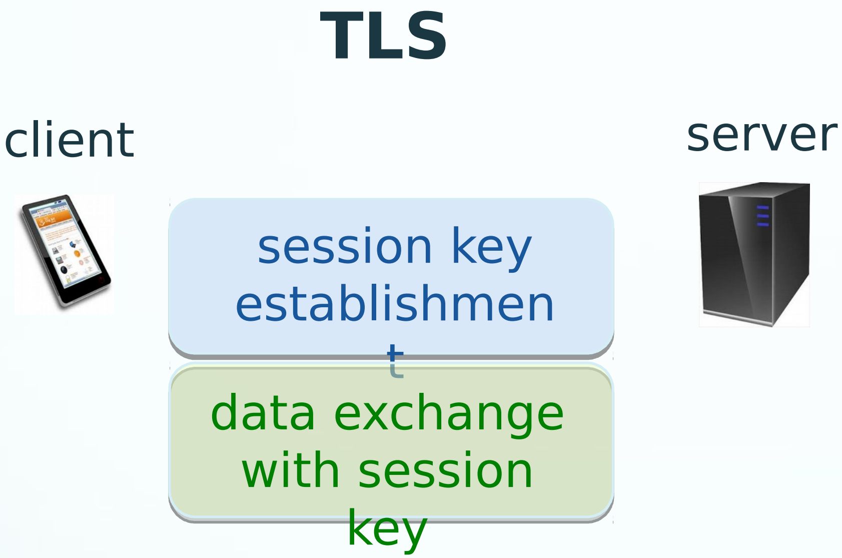
exchange data

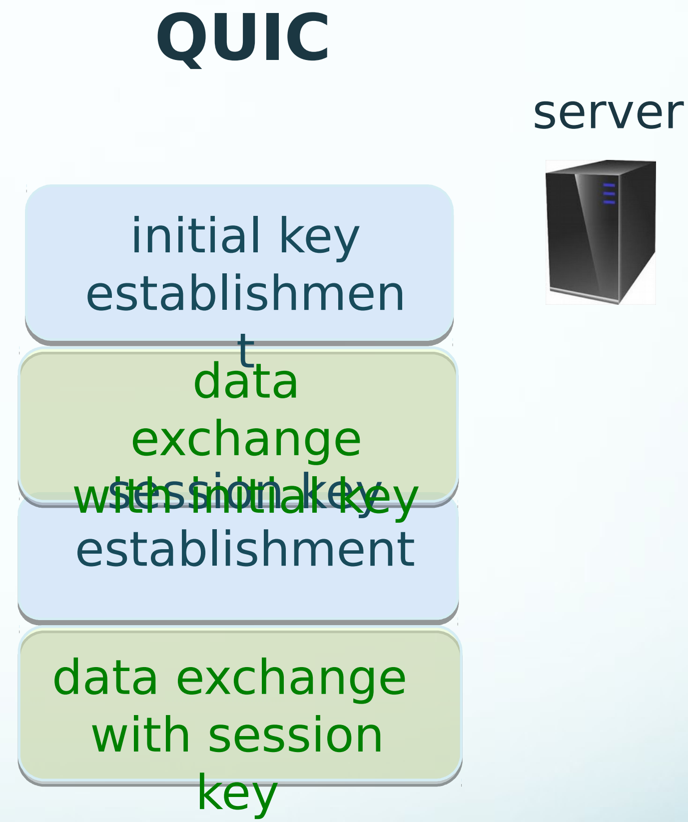
TCP guarantees ordered delivery, provides protection against connection-spoofing, but

- adds latency
- suffers from subtle performance attacks, e.g., TCP reset, Clayton et al, 2006

What about QUIC?

Starting Data Exchange: QUIC vs TLS





- Parties can often avoid 1 RTT in initial key establishment of QUIC by caching some parameters (achieving 0-RTT connections)
- What implications does this have on security?

Previous Work on QUIC

- Fischlin & Günther, *ACM CCS* 2014
  - develop a security definition for multi-stage key agreement and show that QUIC's key exchange meets this definition
  - show how to modify QUIC so that it can compose with any secure data exchange protocol
  - prove QUIC's key exchange with a modification is secure

What about security of the whole protocol as is?

What about its latency in presence of attackers?

Main Questions We Address

1. What provable security guarantees does QUIC provide, and under which assumptions?
2. How effective is QUIC at minimizing latency in presence of attackers?

Summary of Our Results

1. What provable security guarantees does QUIC provide, and under which assumptions?
   - we develop a security definition suitable for performance driven protocols and show that QUIC satisfies it
   - QUIC does not satisfy the traditional notion of forward secrecy, provided by some TLS modes, e.g., TLS-DHE

2. How effective is QUIC at minimizing latency in presence of attackers?
   - with simple attacks on some parameters, it is easy to prevent QUIC from achieving its minimal latency goals
   - we have implemented these attacks and demonstrated that they are practical

Outline

1. Provable Security Analysis of QUIC
    a. how QUIC works
    b. new protocol and security models
    c. security of QUIC
2. QUIC Performance-degradation attacks
3. Recent Related Work
4. Summary

QUIC Protocol

client

server

cid $\xleftarrow{\$} \{0,1\}^{64}$

c_i_hello: (*cid*) →

-generate *stk* based
on client's IP

s_reject: (*cid, scfg*, *stk*) ←

-verify *scfg* signature
-generate DH values
(sec_c, pub_c)
-establish initial
key using *scfg*

c_hello: (*cid, stk, scfg,* pub_c) →

-verify *stk*
-establish initial
key using pub_c

initial data exchange ↔

-generate session
DH values
(sec_s,pub_s)

s_hello: (*cid,* pub_s) ←

-establish session
key using pub_c

-establish session
key using pub_s

data exchange ↔

- *cid*: connection id picked by the client
- *stk*: source-address token used to prevent spoofing
- *scfg*: server config contains server's public Diffie-Hellman (DH) values

can be reused

QUIC Protocol

Connection Resumption

client

server

1 RTT

-cid $\xleftarrow{\$} \{0,1\}^{64}$
-generate DH values
 (sec_c, pub_c)
-establish initial
 key using *scfg*

c_hello: (*cid*, *stk*, *scfg*, pub_c)

-verify *stk*
-establish initial
 key using pub_c

initial data exchange

s_hello: (*cid*, pub_s)

-generate session
 DH values
(sec_s,pub_s)
-establish session
 key using pub_c

-establish session key
 using pub_s

data exchange

- *cid* is the new connection id picked by the client
- *stk* can be reused before expiration
- *scfg* can be reused before expiration

QUIC Protocol

Connection Resumption

client

server

-can achieve 0-RTT connections!

-cid $\xleftarrow{\$} \{0,1\}^{64}$
-generate DH values
(sec_c, pub_c)
-establish initial key using *scfg*

c_hello: (*cid, stk, scfg,* pub_c)

-verify *stk*
-establish initial key using pub_c

initial data exchange

-generate session DH values (sec_s,pub_s)

s_hello: (*cid,* pub_s)

-establish session key using pub_s

data exchange

-establish session key using pub_c

-client cannot initially check *stk* authenticity, so this can lead to inconsistent view of the handshake

-compromising the server before *scfg* expires can reveal data encrypted with initial key

Provable Security Methodology

- Protocol and/or Environment Definition
  - who are the entities and how they are able to communicate
- Security Model
  - what the attacker is allowed to do (e.g. peek on communication, corrupt entities, collude)
  - when the attacker is considered successful
- Proof by Reduction
  - attacker can succeed with only negligible probability under reasonable assumptions on the security of the building blocks (e.g. digital signatures, block cipher, etc)

Security Analysis Main Challenges

- Previous analyses of TLS are not suitable (Jager et al, Krawzcyk et al, Bhargavan et al, Crypto 2012, 2013, 2014)
  - data in QUIC can be exchanged using initial key before the session key is set
- Parties can set distinct initial keys
  - notion of having a ‘*matching conversation*’ is not sufficient
  - need new notion of ‘*setting a key with’* to capture data privacy
- *scfg* is public and can be reused before it expires
  - need weaker notion for forward secrecy for initial keys
  - use traditional notion of forward secrecy for session keys
- UDP does not address unordered delivery and spoofing
  - need to capture attacks involving misordering, selectively delaying or dropping packets, and connection spoofing

Security Analysis Main Challenges

- To address these challenges we developed
  - protocol model that captures data exchanges under initial key before session key is set: Quick Communications (QC)
  - security notion: Quick Authenticated and Confidential Channel Establishment (QACCE)

How Secure is QUIC?

QUIC meets our notion of QACCE-security if

- The underlying signature scheme is *suf-cma*
  - QUIC supports ECDSA-SHA256 and RSA-PSS-SHA256
- The underlying AEAD scheme is *ind-cpa* and *auth-secure*
  - QUIC uses AES Galois-Counter Mode (GCM), McGrew et al, *INDOCRYPT* 2004
- SCDH Problem is hard
- In the random oracle (RO) model
  - model HMAC with RO in the key derivation

Outline

1. Provable Security Analysis of QUIC
2. QUIC Performance-degradation attacks
   a. types of performance-degradation attacks on QUIC
   b. performance-degradation attacks we have implemented
   c. similarities with existing attacks and mitigations
3. Recent Related Work
4. Summary

Performance Attack Overview

- Replaying public, cacheable content, e.g., *scfg* and *stk*
  - results in fooling client and/or server parties into trying to achieve a connection and maintain state
- Manipulating unprotected packet fields, e.g., *cid* & *stk*
  - leads clients and server to have a distinct view of the key exchange resulting in a failure to establish a session key
- The attacks we have studied
  - cause servers and clients to waste time and resources
  - stem from parameters whose purpose was to minimize latency, e.g., *scfg* and *stk*
  - do not concern data authenticity and confidentiality

Attacks We Have Implemented

targeted QUIC Chromium implementation from October 1, 2014
used Python scapy library (http://www.secdev.org/projects/scapy/)

Attack Name	**Attack Type**	**Impact**
cid Manipulation Attack	Manipulation	Connection Failure, Server Load
stk Manipulation Attack	Manipulation	Connection Failure, Server Load
scfg Replay Attack	Replay	Connection Failure
stk Replay Attack	Replay	Server DoS
Crypto Stream Offset Attack	Other	Connection Failure

Attacks can be used to deny clients access to any application of choice and cause servers to waste resources!

stk Manipulation Attack

client

server

cid $\xleftarrow{\$}$ $\{0,1\}^{64}$

c_i_hello: (*cid*) →

*stk** ≠ *stk*

-generate *stk* based
on client's IP

← (*cid, scfg, stk*)

← (*cid, scfg,* ***stk****)

-verify *scfg* signature
-generate DH values
(sec_c, pub_c)
-establish initial key *ik** using *scfg*

(*cid*,*stk**,*scfg*,pub_c) →

(*cid*,*stk*,*scfg*,pub_c) →

-verify *stk*
-establish initial key *ik* using pub_c

cannot decrypt exchanged data

stk is an input into the key derivation process, because client uses *stk**, client and server derive different initial keys: *ik** ≠ *ik*

scfg Replay Attack

client

server

cid $\xleftarrow{\$} \{0,1\}^{64}$

c_i_hello: (*cid*) →

← (*cid, scfg*, ***stk***)

-generate *stk*

-verify *scfg* signature
-generate DH values
 (sec_c, pub_c)
-establish initial
 key *ik* using *scfg*

($cid, stk, scfg, pub_c$) →

-verification of
 stk fails

cannot exchange any data

the server is not aware of the client's request,
so it rejects *stk* and any associated client's messages

stk Replay Attack

client

server

c_i_hello: (*cid*) →

cid $\xleftarrow{\$} \{0,1\}^{64}$

← s_reject: (*cid, scfg, stk*)

c_hello: ($cid^*, scfg, stk, pub^*_c$) →

c_hello: ($cid^*, scfg, stk, pub^*_c$) →

c_hello: ($cid^*, scfg, stk, pub^*_c$) →

c_hello: ($cid^*, scfg, stk, pub^*_c$) →

-grab *scfg* and *stk*
-spoofed connections

stk is bound to an IP address and is reusable while not expired.
Server must derive keys, keep state, and send replies for each of these connections.

Similar Attacks on TCP/TLS

- *stk* Replay Attack is similar to TCP SYN Flood
  - both attacks overwhelm a server's resources by starting and then abandoning a connection
  - single use SYN-Cookies are the traditional mitigation
  - *stk* has to be replayable for 0-RTT
- Manipulation Attacks show similarity with SSL Downgrade Attacks
  - downgrade attacks: rewrite handshake to request vulnerable crypto
  - protection in SSL 3+ by including hash of all messages in Finished message, causing failure at end of handshake
  - manipulation attacks in QUIC detected by different keys at end of handshake
  - QUIC fails much more slowly than SSL/TLS

Mitigations

- Mitigating Replay Attacks
  - seems impossible without limiting public, cacheable parameters (e.g., *scfg* and *stk*) to single use, but
  - this would prohibit the possibility of 0-RTT connections
- Mitigating Packet Manipulation Attacks
  - could sign modifiable parameters (e.g., *cid* and *stk*), but
  - this would require additional signature-related computations, introducing other DoS attacks via IP-spoofing

Outline

1. Provable Security Analysis of QUIC
2. QUIC Performance-degradation attacks
   a. types of performance-degradation attacks on QUIC
   b. performance-degradation attacks we have implemented
   c. similarities with existing attacks and mitigations
3. Recent Related Work
   a. TLS 1.3
4. Summary

TLS 1.3

The next performance-optimized secure protocol

1. TLS 1.3 has a number of similarities with QUIC
   - handshake with multiple keys
   - performance optimized
   - 0-RTT mode
2. Currently in the draft stage
3. Provable Security Analyses already being published
   - Very encouraging

Provable Security for TLS 1.3

1. Dowling et al, *ACM CCS* 2015
   - show that TLS1.3 drafts are secure multi-stage key exchange protocols
   - show how to compose with symmetric-key protocols to securely exchange data
2. Cremers et al, *IEEE S&P 2016*
   - formal model of TLS1.3 handshakes in Tamarin
   - show security of all TLS1.3 handshakes
3. Li et al, *IEEE S&P 2016*
   - show that all TLS1.3 handshakes compose securely

Implementation Attacks

1. Jager et al, *ACM CCS 2015*
   - weaknesses of RSA-based PKCS#1 v1.5 encryption can result in attacks against TLS1.3 and QUIC
     a. if they have to coexistence with previous TLS versions
     b. even if they do not support PKCS#1 v1.5

Outline

1. Provable Security Analysis of QUIC
2. QUIC Performance-degradation attacks
   a. types of performance-degradation attacks on QUIC
   b. performance-degradation attacks we have implemented
3. Recent Related Work
   a. TLS 1.3
4. Summary

Summary

- Developed security definition for performance-driven protocols and showed that QUIC meets our definition
- Have implemented five different practical performance degradation attacks against QUIC
- Highlights an example of a tradeoff between performance vs security

Thank You

Please check out the full version

https://eprint.iacr.org/2015/582

1. **Security definitions and proofs**
2. **Attack implementation details**