Stream: Internet Engineering Task Force (IETF)
RFC: 9369
Category: Standards Track
Published: May 2023
ISSN: 2070-1721
Author: M. Duke
Google LLC

RFC 9369
QUIC Version 2

Abstract

This document specifies QUIC version 2, which is identical to QUIC version 1 except for some trivial details. Its purpose is to combat various ossification vectors and exercise the version negotiation framework. It also serves as a template for the minimum changes in any future version of QUIC.

Note that "version 2" is an informal name for this proposal that indicates it is the second version of QUIC to be published as a Standards Track document. The protocol specified here uses a version number other than 2 in the wire image, in order to minimize ossification risks.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9369.

Copyright Notice

Table of Contents

Author's Address

1. Introduction

QUIC version 1 [QUIC] has numerous extension points, including the version number that occupies the second through fifth bytes of every long header (see [QUIC-INVARIANTS]). If experimental versions are rare, and QUIC version 1 constitutes the vast majority of QUIC traffic, there is the potential for middleboxes to ossify on the version bytes that are usually 0x00000001.

In QUIC version 1, Initial packets are encrypted with the version-specific salt, as described in Section 5.2 of [QUIC-TLS]. Protecting Initial packets in this way allows observers to inspect their contents, which includes the TLS Client Hello or Server Hello messages. Again, there is the potential for middleboxes to ossify on the version 1 key derivation and packet formats.

Finally, [QUIC-VN] describes two mechanisms endpoints can use to negotiate which QUIC version to select. The "incompatible" version negotiation method can support switching from any QUIC version to any other version with full generality, at the cost of an additional round trip at the start of the connection. "Compatible" version negotiation eliminates the round-trip penalty but levies some restrictions on how much the two versions can differ semantically.

QUIC version 2 is meant to mitigate ossification concerns and exercise the version negotiation mechanisms. The changes provide an example of the minimum set of changes necessary to specify a new QUIC version. However, note that the choice of the version number on the wire is randomly chosen instead of "2", and the two bits that identify each Long Header packet type are different from version 1; both of these properties are meant to combat ossification and are not strictly required of a new QUIC version.

Any endpoint that supports two versions needs to implement version negotiation to protect against downgrade attacks.

2. Conventions

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Differences with QUIC Version 1

Except for a few differences, QUIC version 2 endpoints **MUST** implement the QUIC version 1 specification as described in [QUIC], [QUIC-TLS], and [QUIC-RECOVERY]. The remainder of this section lists the differences.

3.1. Version Field

The Version field of long headers is 0x6b3343cf. This was generated by taking the first four bytes of the sha256sum of "QUICv2 version number".

3.2. Long Header Packet Types

All version 2 Long Header packet types are different. The Type field values are:

- Initial: 0b01
- 0-RTT: 0b10
- Handshake: 0b11
- Retry: 0b00

3.3. Cryptography Changes

3.3.1. Initial Salt

The salt used to derive Initial keys in Section 5.2 of [QUIC-TLS] changes to:

```
initial_salt = 0x0dede3def700a6db819381be6e269dcbf9bd2ed9
```

This is the first 20 bytes of the sha256sum of "QUICv2 salt".

3.3.2. HMAC-based Key Derivation Function (HKDF) Labels

The labels used in [QUIC-TLS] to derive packet protection keys (Section 5.1), header protection keys (Section 5.4), Retry Integrity Tag keys (Section 5.8), and key updates (Section 6.1) change from "quic key" to "quicv2 key", from "quic iv" to "quicv2 iv", from "quic hp" to "quicv2 hp", and from "quic ku" to "quicv2 ku" to meet the guidance for new versions in Section 9.6 of that document.

3.3.3. Retry Integrity Tag

The key and nonce used for the Retry Integrity Tag (Section 5.8 of [QUIC-TLS]) change to:

```
secret =
  0xc4dd2484d681aefa4ff4d69c2c20299984a765a5d3c31982f38fc74162155e9f
key = 0x8fb4b01b56ac48e260fbcbcead7ccc92
nonce = 0xd86969bc2d7c6d9990efb04a
```

The secret is the sha256sum of "QUICv2 retry secret". The key and nonce are derived from this secret with the labels "quicv2 key" and "quicv2 iv", respectively.

4. Version Negotiation Considerations

QUIC version 2 is not intended to deprecate version 1. Endpoints that support version 2 might continue support for version 1 to maximize compatibility with other endpoints. In particular, HTTP clients often use Alt-Svc [RFC7838] to discover QUIC support. As this mechanism does not currently distinguish between QUIC versions, HTTP servers **SHOULD** support multiple versions to reduce the probability of incompatibility and the cost associated with QUIC version negotiation or TCP fallback. For example, an origin advertising support for "h3" in Alt-Svc should support QUIC version 1, as it was the original QUIC version used by HTTP/3; therefore, some clients will only support that version.

Any QUIC endpoint that supports QUIC version 2 **MUST** send, process, and validate the version_information transport parameter specified in [QUIC-VN] to prevent version downgrade attacks.

Note that version 2 meets the definition in [QUIC-VN] of a compatible version with version 1, and version 1 is compatible with version 2. Therefore, servers can use compatible negotiation to switch a connection between the two versions. Endpoints that support both versions **SHOULD** support compatible version negotiation to avoid a round trip.

4.1. Compatible Negotiation Requirements

Compatible version negotiation between versions 1 and 2 follows the same requirements in either direction. This section uses the terms "original version" and "negotiated version" from [QUIC-VN].

If the server sends a Retry packet, it **MUST** use the original version. The client ignores Retry packets using other versions. The client **MUST NOT** use a different version in the subsequent Initial packet that contains the Retry token. The server **MAY** encode the QUIC version in its Retry token to validate that the client did not switch versions, and drop the packet if it switched, to enforce client compliance.

QUIC version 2 uses the same transport parameters to authenticate the Retry as QUIC version 1. After switching to a negotiated version after a Retry, the server **MUST** include the relevant transport parameters to validate that the server sent the Retry and the connection IDs used in the exchange, as described in Section 7.3 of [QUIC].

The server cannot send CRYPTO frames until it has processed the client's transport parameters. The server **MUST** send all CRYPTO frames using the negotiated version.

The client learns the negotiated version by observing the first long header Version field that differs from the original version. If the client receives a CRYPTO frame from the server in the original version, it indicates that the negotiated version is equal to the original version.

Before the server is able to process transport parameters from the client, it might need to respond to Initial packets from the client. For these packets, the server uses the original version.

Once the client has learned the negotiated version, it **SHOULD** send subsequent Initial packets using that version. The server **MUST NOT** discard its original version Initial receive keys until it successfully processes a Handshake packet with the negotiated version.

Both endpoints **MUST** send Handshake and 1-RTT packets using the negotiated version. An endpoint **MUST** drop packets using any other version. Endpoints have no need to generate the keying material that would allow them to decrypt or authenticate such packets.

The client **MUST NOT** send 0-RTT packets using the negotiated version, even after processing a packet of that version from the server. Servers can accept 0-RTT and then process 0-RTT packets from the original version.

5. TLS Resumption and NEW_TOKEN Tokens

TLS session tickets and NEW_TOKEN tokens are specific to the QUIC version of the connection that provided them. Clients **MUST NOT** use a session ticket or token from a QUIC version 1 connection to initiate a QUIC version 2 connection, and vice versa. When a connection includes compatible version negotiation, any issued server tokens are considered to originate from the negotiated version, not the original one.

Servers **MUST** validate the originating version of any session ticket or token and not accept one issued from a different version. A rejected ticket results in falling back to a full TLS handshake, without 0-RTT. A rejected token results in the client address remaining unverified, which limits the amount of data the server can send.

After compatible version negotiation, any resulting session ticket maps to the negotiated version rather than the original one.

6. Ossification Considerations

QUIC version 2 provides protection against some forms of ossification. Devices that assume that all long headers will encode version 1, or that the version 1 Initial key derivation formula will remain version-invariant, will not correctly process version 2 packets.

However, many middleboxes, such as firewalls, focus on the first packet in a connection, which will often remain in the version 1 format due to the considerations above.

Clients interested in combating middlebox ossification can initiate a connection using version 2 if they are reasonably certain the server supports it and if they are willing to suffer a round-trip penalty if they are incorrect. In particular, a server that issues a session ticket for version 2 indicates an intent to maintain version 2 support while the ticket remains valid, even if support cannot be guaranteed.

7. Applicability

QUIC version 2 provides no change from QUIC version 1 for the capabilities available to applications. Therefore, all Application-Layer Protocol Negotiation (ALPN) [RFC7301] codepoints specified to operate over QUIC version 1 can also operate over this version of QUIC. In particular, both the "h3" [HTTP/3] and "doq" [RFC9250] ALPNs can operate over QUIC version 2.

Unless otherwise stated, all QUIC extensions defined to work with version 1 also work with version 2.

8. Security Considerations

QUIC version 2 introduces no changes to the security or privacy properties of QUIC version 1.

The mandatory version negotiation mechanism guards against downgrade attacks, but downgrades have no security implications, as the version properties are identical.

Support for QUIC version 2 can help an observer to fingerprint both client and server devices.

9. IANA Considerations

IANA has added the following entries to the "QUIC Versions" registry maintained at <https://www.iana.org/assignments/quic>.

Value: 0x6b3343cf
Status: permanent
Specification: RFC 9369
Change Controller: IETF
Contact: QUIC WG

Value: 0x709a50c4
Status: provisional
Specification: RFC 9369
Change Controller: IETF
Contact: QUIC WG
Notes: QUIC v2 draft codepoint

10. References

10.1. Normative References

[QUIC]

Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

[QUIC-RECOVERY] Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", RFC 9002, DOI 10.17487/RFC9002, May 2021, <https://www.rfc-editor.org/info/rfc9002>.

[QUIC-TLS] Thomson, M., Ed. and S. Turner, Ed., "Using TLS to Secure QUIC", RFC 9001, DOI 10.17487/RFC9001, May 2021, <https://www.rfc-editor.org/info/rfc9001>.

[QUIC-VN] Schinazi, D. and E. Rescorla, "Compatible Version Negotiation for QUIC", RFC 9368, DOI 10.17487/RFC9368, May 2023, <https://www.rfc-editor.org/info/rfc9368>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

10.2. Informative References

[HTTP/3] Bishop, M., Ed., "HTTP/3", RFC 9114, DOI 10.17487/RFC9114, June 2022, <https://www.rfc-editor.org/info/rfc9114>.

[QUIC-INVARIANTS] Thomson, M., "Version-Independent Properties of QUIC", RFC 8999, DOI 10.17487/RFC8999, May 2021, <https://www.rfc-editor.org/info/rfc8999>.

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[RFC7838] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <https://www.rfc-editor.org/info/rfc7838>.

[RFC9250] Huitema, C., Dickinson, S., and A. Mankin, "DNS over Dedicated QUIC Connections", RFC 9250, DOI 10.17487/RFC9250, May 2022, <https://www.rfc-editor.org/info/rfc9250>.

Appendix A. Sample Packet Protection

This section shows examples of packet protection so that implementations can be verified incrementally. Samples of Initial packets from both the client and server plus a Retry packet are defined. These packets use an 8-byte client-chosen Destination Connection ID of 0x8394c8f03e515708. Some intermediate values are included. All values are shown in hexadecimal.

A.1. Keys

The labels generated during the execution of the HKDF-Expand-Label function (that is, HkdfLabel.label) and part of the value given to the HKDF-Expand function in order to produce its output are:

client in: 00200f746c73313320636c69656e7420696e00

server in: 00200f746c7331332073657276657220696e00

quicv2 key: 001010746c7331332071756963632206b657900

quicv2 iv: 000c0f746c733133207175696376322069760 0

quicv2 hp: 00100f746c7331332071756963763220687000

The initial secret is common:

```
initial_secret = HKDF-Extract(initial_salt, cid)
    = 2062e8b3cd8d52092614b8071d0aa1fb
      7c2e3ac193f78b280e72d8f5751f6aba
```

The secrets for protecting client packets are:

```
client_initial_secret
    = HKDF-Expand-Label(initial_secret, "client in", "", 32)
    = 14ec9d6eb9fd7af83bf5a668bc17a7e2
      83766aade7ecd0891f70f9ff7f4bf47b

key = HKDF-Expand-Label(client_initial_secret, "quicv2 key", "", 16)
    = 8b1a0bc121284290a29e0971b5cd045d

iv  = HKDF-Expand-Label(client_initial_secret, "quicv2 iv", "", 12)
    = 91f73e2351d8fa91660e909f

hp  = HKDF-Expand-Label(client_initial_secret, "quicv2 hp", "", 16)
    = 45b95e15235d6f45a6b19cbcb0294ba9
```

The secrets for protecting server packets are:

```
server_initial_secret
    = HKDF-Expand-Label(initial_secret, "server in", "", 32)
    = 0263db1782731bf4588e7e4d93b74639
      07cb8cd8200b5da55a8bd488eafc37c1

key = HKDF-Expand-Label(server_initial_secret, "quicv2 key", "", 16)
    = 82db637861d55e1d011f19ea71d5d2a7

iv  = HKDF-Expand-Label(server_initial_secret, "quicv2 iv", "", 12)
    = dd13c276499c0249d3310652

hp  = HKDF-Expand-Label(server_initial_secret, "quicv2 hp", "", 16)
    = edf6d05c83121201b436e16877593c3a
```

A.2. Client Initial

The client sends an Initial packet. The unprotected payload of this packet contains the following CRYPTO frame, plus enough PADDING frames to make a 1162-byte payload:

```
060040f1010000ed0303ebf8fa56f129 39b9584a3896472ec40bb863cfd3e868
04fe3a47f06a2b69484c000004130113 02010000c000000010000e00000b6578
616d706c652e636f6dff01000100000a 00080006001d0017001800100070005
04616c706e0005000501000000000033 00260024001d00209370b2c9caa47fba
baf4559fedba753de171fa71f50f1ce1 5d43e994ec74d748002b000302030400
0d0010000e0403050306030203080408 050806002d00020101001c0002400100
3900320408ffffffffffffffff050480 00ffff07048000ffff08011001048000
75300901100f088394c8f03e51570806 048000ffff
```

The unprotected header indicates a length of 1182 bytes: the 4-byte packet number, 1162 bytes of frames, and the 16-byte authentication tag. The header includes the connection ID and a packet number of 2:

```
d36b3343cf088394c8f03e5157080000449e00000002
```

Protecting the payload produces an output that is sampled for header protection. Because the header uses a 4-byte packet number encoding, the first 16 bytes of the protected payload is sampled and then applied to the header as follows:

```
sample = ffe67b6abcdb4298b485dd04de806071

mask = AES-ECB(hp, sample)[0..4]
     = 94a0c95e80

header[0] ^= mask[0] & 0x0f
     = d7
header[18..21] ^= mask[1..4]
     = a0c95e82
header = d76b3343cf088394c8f03e5157080000449ea0c95e82
```

The resulting protected packet is:

```
d76b3343cf088394c8f03e5157080000 449ea0c95e82ffe67b6abcdb4298b485
dd04de806071bf03dceebfa162e75d6c 96058bdbfb127cdfcbf903388e99ad04
9f9a3dd4425ae4d0992cfff18ecf0fdb 5a842d09747052f17ac2053d21f57c5d
250f2c4f0e0202b70785b7946e992e58 a59ac52dea6774d4f03b55545243cf1a
12834e3f249a78d395e0d18f4d766004 f1a2674802a747eaa901c3f10cda5500
cb9122faa9f1df66c392079a1b40f0de 1c6054196a11cbea40afb6ef5253cd68
18f6625efce3b6def6ba7e4b37a40f77 32e093daa7d52190935b8da58976ff33
12ae50b187c1433c0f028edcc4c2838b 6a9bfc226ca4b4530e7a4ccee1bfa2a3
d396ae5a3fb512384b2fdd851f784a65 e03f2c4fbe11a53c7777c023462239dd
6f7521a3f6c7d5dd3ec9b3f233773d4b 46d23cc375eb198c63301c21801f6520
bcfb7966fc49b393f0061d974a2706df 8c4a9449f11d7f3d2dcbb90c6b877045
636e7c0c0fe4eb0f697545460c806910 d2c355f1d253bc9d2452aaa549e27a1f
ac7cf4ed77f322e8fa894b6a83810a34 b361901751a6f5eb65a0326e07de7c12
16ccce2d0193f958bb3850a833f7ae43 2b65bc5a53975c155aa4bcb4f7b2c4e5
4df16efaf6ddea94e2c50b4cd1dfe060 17e0e9d02900cffe1935e0491d77ffb4
fdf85290fdd893d577b1131a610ef6a5 c32b2ee0293617a37cbb08b847741c3b
8017c25ca9052ca1079d8b78aebd4787 6d330a30f6a8c6d61dd1ab5589329de7
14d19d61370f8149748c72f132f0fc99 f34d766c6938597040d8f9e2bb522ff9
9c63a344d6a2ae8aa8e51b7b90a4a806 105fcbca31506c446151adfeceb51b91
abfe43960977c87471cf9ad4074d30e1 0d6a7f03c63bd5d4317f68ff325ba3bd
80bf4dc8b52a0ba031758022eb025cdd 770b44d6d6cf0670f4e990b22347a7db
848265e3e5eb72dfe8299ad7481a4083 22cac55786e52f633b2fb6b614eaed18
d703dd84045a274ae8bfa73379661388 d6991fe39b0d93debb41700b41f90a15
c4d526250235ddcd6776fc77bc97e7a4 17ebcb31600d01e57f32162a8560cacc
7e27a096d37a1a86952ec71bd89a3e9a 30a2a26162984d7740f81193e8238e61
f6b5b984d4d3dfa033c1bb7e4f0037fe bf406d91c0dccf32acf423cfa1e70710
10d3f270121b493ce85054ef58bada42 310138fe081adb04e2bd901f2f13458b
3d6758158197107c14ebb193230cd115 7380aa79cae1374a7c1e5bbcb80ee23e
06ebfde206bfb0fcbc0edc4ebec30966 1bdd908d532eb0c6adc38b7ca7331dce
8dfce39ab71e7c32d318d136b6100671 a1ae6a6600e3899f31f0eed19e3417d1
34b90c9058f8632c798d4490da498730 7cba922d61c39805d072b589bd52fdf1
e86215c2d54e6670e07383a27bbffb5a ddf47d66aa85a0c6f9f32e59d85a44dd
5d3b22dc2be80919b490437ae4f36a0a e55edf1d0b5cb4e9a3ecabee93dfc6e3
8d209d0fa6536d27a5d6fbb17641cde2 7525d61093f1b28072d111b2b4ae5f89
d5974ee12e5cf7d5da4d6a31123041f3 3e61407e76cffcdcfd7e19ba58cf4b53
6f4c4938ae79324dc402894b44faf8af bab35282ab659d13c93f70412e85cb19
9a37ddec600545473cfb5a05e08d0b20 9973b2172b4d21fb69745a262ccde96b
a18b2faa745b6fe189cf772a9f84cbfc
```

A.3. Server Initial

The server sends the following payload in response, including an ACK frame, a CRYPTO frame, and no PADDING frames:

```
0200000000600405a020000560303ee fce7f7b37ba1d1632e96677825ddf739
88cfc79825df566dc5430b9a045a1200 130100002e00330024001d00209d3c94
0d89690b84d08a60993c144eca684d10 81287c834d5311bcf32bb9da1a002b00
020304
```

The header from the server includes a new connection ID and a 2-byte packet number encoding for a packet number of 1:

```
d16b3343cf0008f067a5502a4262b50040750001
```

As a result, after protection, the header protection sample is taken, starting from the third protected byte:

```
sample = 6f05d8a4398c47089698baeea26b91eb
mask   = 4dd92e91ea
header = dc6b3343cf0008f067a5502a4262b5004075d92f
```

The final protected packet is then:

```
dc6b3343cf0008f067a5502a4262b500 4075d92faaf16f05d8a4398c47089698
baeea26b91eb761d9b89237bbf872630 17915358230035f7fd3945d88965cf17
f9af6e16886c61bfc703106fbaf3cb4c fa52382dd16a393e42757507698075b2
c984c707f0a0812d8cd5a6881eaf21ce da98f4bd23f6fe1a3e2c43edd9ce7ca8
4bed8521e2e140
```

A.4. Retry

This shows a Retry packet that might be sent in response to the Initial packet in Appendix A.2. The integrity check includes the client-chosen connection ID value of 0x8394c8f03e515708, but that value is not included in the final Retry packet:

```
cf6b3343cf0008f067a5502a4262b574 6f6b656ec8646ce8bfe33952d9555436
65dcc7b6
```

A.5. ChaCha20-Poly1305 Short Header Packet

This example shows some of the steps required to protect a packet with a short header. It uses AEAD_CHACHA20_POLY1305.

In this example, TLS produces an application write secret from which a server uses HKDF-Expand-Label to produce four values: a key, an Initialization Vector (IV), a header protection key, and the secret that will be used after keys are updated (this last value is not used further in this example).

```
secret
    = 9ac312a7f877468ebe69422748ad00a1
      5443f18203a07d6060f688f30f21632b

key = HKDF-Expand-Label(secret, "quicv2 key", "", 32)
    = 3bfcddd72bcf02541d7fa0dd1f5f9eee
      a817e09a6963a0e6c7df0f9a1bab90f2

iv  = HKDF-Expand-Label(secret, "quicv2 iv", "", 12)
    = a6b5bc6ab7dafce30ffff5dd

hp  = HKDF-Expand-Label(secret, "quicv2 hp", "", 32)
    = d659760d2ba434a226fd37b35c69e2da
      8211d10c4f12538787d65645d5d1b8e2

ku  = HKDF-Expand-Label(secret, "quicv2 ku", "", 32)
    = c69374c49e3d2a9466fa689e49d476db
      5d0dfbc87d32ceeaa6343fd0ae4c7d88
```

The following shows the steps involved in protecting a minimal packet with an empty Destination Connection ID. This packet contains a single PING frame (that is, a payload of just 0x01) and has a packet number of 654360564. In this example, using a packet number of length 3 (that is, 49140 is encoded) avoids having to pad the payload of the packet; PADDING frames would be needed if the packet number is encoded on fewer bytes.

```
pn                 = 654360564 (decimal)
nonce              = a6b5bc6ab7dafce328ff4a29
unprotected header = 4200bff4
payload plaintext  = 01
payload ciphertext = 0ae7b6b932bc27d786f4bc2bb20f2162ba
```

The resulting ciphertext is the minimum size possible. One byte is skipped to produce the sample for header protection.

```
sample = e7b6b932bc27d786f4bc2bb20f2162ba
mask   = 97580e32bf
header = 5558b1c6
```

The protected packet is the smallest possible packet size of 21 bytes.

```
packet = 5558b1c60ae7b6b932bc27d786f4bc2bb20f2162ba
```

Acknowledgments

The author would like to thank Christian Huitema, Lucas Pardue, Kyle Rose, Anthony Rossi, Zahed Sarker, David Schinazi, Tatsuhiro Tsujikawa, and Martin Thomson for their helpful suggestions.

Author's Address

Martin Duke
Google LLC
Email: martin.h.duke@gmail.com