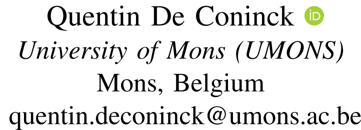
Core QUIC: Enabling Dynamic, Implementation-Agnostic Protocol Extensions

Quentin De Coninck

University of Mons (UMONS)
Mons, Belgium
quentin.deconinck@umons.ac.be

***Abstract*—While applications quickly evolve, Internet protocols do not follow the same pace. There are two root causes for this. First, extending protocol with cleartext control plane is usually hindered by various network devices such as middleboxes. Second, such extensions usually require support from all participating entities, but often these run different implementations, leading to the chicken-and-egg deployment issue. The recently standardized QUIC protocol paved the way for dealing with the first concern by embedding encryption by design. However, it attracted so much interest that there is now a large heterogeneity in QUIC implementations, hence amplifying the second problem.**

To get rid of these deployment issues and to enable interoperable, implementation-independent innovation at transport layer, we propose a paradigm shift called Core QUIC. While Core QUIC keeps compliant with the standardized QUIC protocol, it enforces implementation architecture such that any Core QUIC-supporting participant can be extended with the same, generic bytecode. To achieve this, Core QUIC defines a standardized representation format of common QUIC structures on which plugins running in a controlled environment can operate to extend the underlying host implementation. We demonstrate the feasibility of our approach by making two implementations Core QUIC-compliant. Then, we show that we can extend both with the same plugin code over several use cases.

I. Introduction

Deploying new features in end-to-end protocols over the Internet is difficult. Two main elements affect this. First, there are many middleboxes in the network affecting the behavior of communicating protocols. In-network devices often interact with the traffic based on its public metadata, such as packet headers. Extending TCP was shown to be difficult [1] since its options are observable by in-network third-parties such as firewalls that may block them. Legacy devices also prevented protocols such as SCTP [2] from being broadly deployed.

Second, participating devices are strongly heterogeneous, coming from different vendors and having diverse hardware and software. The Internet built its interoperability success thanks to open standards defining protocol messages and actions to take. Still, it enables very different implementation designs. Defining protocol extensions therefore requires willingness, cooperation and efforts from the various implementations' maintainers, a process that usually takes years, when it succeeds [3].

Recently, the network community tacked the first middlebox issue by introducing QUIC [4], a UDP-based transport protocol with built-in encryption of not only the carried data, but also its protocol metadata. Thanks to this encryption, all control information, including extension negotiation, cannot be altered by middleboxes, making QUIC resilient to their interference. QUIC attracted a lot of interest, and there is now more than twenty publicly known implementations. This success reintroduces the previously discussed heterogeneity issue. To reach wide deployment, most of these implementations need to agree on the proposed feature, and then integrate it and deploy to devices. Only a few large actors can sustain such effort, constraining innovation into their hands.

To enable inter-operable, implementation-independent innovation at transport layer, we propose a paradigm shift called *Core QUIC*. While Core QUIC keeps compliant with the QUIC standard [4], it enforces implementation architecture such that **any Core QUIC-supporting participant can be extended with the same plugin** consisting in architecture-independent bytecode. In addition, Core QUIC addresses deployment and implementation concerns by enabling partial support and does not require major changes in the implementation's internals. To achieve this, Core QUIC defines a standardized representation format of common QUIC structures on which plugins running in a controlled environment can operate to extend the underlying host implementation. By making **two different implementations Core QUIC-compliant**, we demonstrate the feasibility of our approach.

The paper is organized as follow. First, Section II introduces background on QUIC. Then, Section III details the architecture of Core QUIC. We describe our Core QUIC library implementation in Section IV. We demonstrate how plugins can extend our two Core QUIC implementations with several use cases in Section V. Finally, Section VI discusses related works and Section VII concludes the paper. Further details and experiments are available in our technical report [5].

II. Background on QUIC

Our work focuses on QUIC, a transport protocol providing reliable and encrypted services atop UDP. To setup the encryption, QUIC relies on TLS 1.3 [6]. During the TLS handshake, QUIC endpoints advertise *transport parameters* to communicate flow-control values as well as their support for specific QUIC features and extensions. Unlike TCP, except a connection identifier in the header that remains in clear-text for routing purposes, the whole QUIC packet is

ISBN 978-3-903176-63-8 © 2024 IFIP

encrypted and authenticated. The encrypted payload contains *frames*. These form the root QUIC messages, following a type-value format. There are 2^{62} possible frame type values [4]. While the STREAM frames carry the application data, most of the frames are QUIC control information. The ACK frame acknowledges to the peer the reception of QUIC packets. The MAX_DATA frame advertises the flow-control limits. An endpoint can probe a network path using the PATH_CHALLENGE frame, expecting its peer to send back a PATH_RESPONSE. The PADDING frame increases the packet's size without containing any information and is usually used to perform path MTU discovery.

III. Designing Core QUIC

The design of Core QUIC focuses on the easiness to pluginize the implementation. Specifically, the effort required by the QUIC maintainer to include all the pluginization mechanisms should be as minimal as possible. The easier the integration is, the higher the potential adoption of the system. Specifically, Core QUIC has the following design objectives.

Core QUIC can dynamically load features on a per-connection basis, regardless of the QUIC host implementation. The fundamental idea behind Core QUIC is to have implementations that can be tuned or extended without requiring binary change. While PQUIC [7] shares a similar idea, the development of its plugins is strongly tight to the PQUIC implementation. Instead, Core QUIC provides a common layer where any compliant implementation can be safely extended through the same plugin. Such a layer defines *routines* and *fields* that any QUIC implementation needs to expose.

Implementations can incrementally support Core QUIC. The story of the Internet taught us that successful solutions need to be easily deployable. Given the paradigm shift of Core QUIC, such a solution should be simple to integrate. Furthermore, QUIC implementations have very different internal architectures and some QUIC features may be harder to expose. To tackle this, initial Core QUIC support requires minimal changes to the QUIC implementation. The common layer exposition can then be incrementally implemented, and Core QUIC checks at plugin load time that all the requested elements are actually provided by the host implementation.

Core QUIC plugins operate in a safe architecture-independent environment. Plugins consist in bytecode that may have been written by anyone. They may contain unintentional mistakes or malicious content. To mitigate this, plugins operate in an isolated environment and cannot access memory outside of its scope.

Core QUIC supports the combination of non-overlapping plugins. In both TCP and QUIC, many extensions can be simultaneously enabled on a given session. Indeed, these usually provide orthogonal features and, when applicable, define extension-specific messages. Core QUIC keeps this property at the plugin level, as long as plugins do not override the exact same QUIC routine. Furthermore, these plugins can collaborate through a well-defined interface.

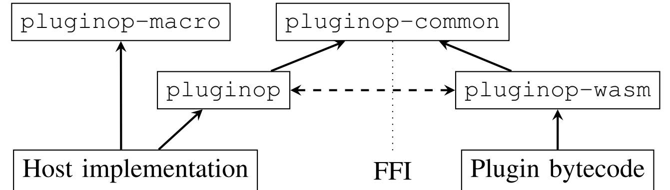


Fig. 1. Architectural view of the different components of `pluginop`. Plain arrows mean ”uses”, dashed arrows mean ”interacts”.

To achieve this, this paper introduces `pluginop`, a library enabling QUIC implementations to be dynamically extended through plugins. Figure 1 shows the four architectural components. The core `pluginop` library is the one a QUIC implementation needs to integrate to become Core QUIC-compliant. The remaining of this Section describes how these building blocks work together to define the protocol routines (Section III-A), to set up a common representation layer (Section III-B), to run plugins in a safe environment (Section III-C) and to integrate Core QUIC in an existing QUIC implementation (Section III-D).

A. Defining Protocol Routines

Core QUIC defines hooks inside QUIC implementations, called *protocol routines* (PRs), that correspond to operations that any QUIC agent must provide. Each PR define input parameters and, optionally, output values. The definition of such PRs introduces a trade-off. On the one hand, having numerous PRs enables fine-grained protocol tuning and extensibility. On the other hand, requiring support for a large number of PRs makes it harder for implementations, having different software architectures, to fully support Core QUIC. Core QUIC PRs should be common to all implementations.

To use an extension, QUIC first needs to negotiate it. QUIC endpoints advertise parameters and support for extensions through QUIC transport parameters, exchanged during the handshake. Such transport parameters follow a well-known type-length-value scheme. If both participants advertise the value, the extension is then enabled.

To interact with transport parameters, we introduce two PRs: ***WriteTransportParameter*** and ***DecodeTransportParameter***. Each of these take as parameter the type of the transport parameter, i.e., an unsigned integer up to $2^{62} - 1$. Different plugins can then concurrently define new transport parameters having different types. Plugins then *register* these new types to make the host implementation aware of them.

Once negotiated, the plugin can then define frames with new types. Like transport parameters, the plugin advertises its *frame registrations* to make Core QUIC aware of these types. Core QUIC defines nine frame-related PRs, all taking as parameter the frame type. Algorithm 1 shows how these PRs operate when sending plugin-defined frames. When preparing a packet to be sent, the Core QUIC implementation collects all the frame registrations made by the loaded plugins. Then, for each registration, Core QUIC checks whether such frames should be part of the next packet thanks to the five PRs

Algorithm 1 Protocol routines when sending frames.

```
for all frame reservations r do
  if ShouldSendFrame(r.frame_type()) then
    f ← PrepareFrame(r.frame_type())
    if FrameWireLen(r.frame_type()), f) ≤ buf_len then
      if WriteFrame(r.frame_type()), f) = OK then
        OnFrameReserved(r.frame_type()), f)
      end if
    end if
  end if
end for
```

described in Algorithm 1. Note that PRs may also alter the sending processing, e.g., to stop sending packets for some reason (see, e.g., Sect. V-B). Once the packet sent, the plugin can later be notified whether the frame was acknowledged or declared as lost thanks to the ***NotifyFrame*** routine.

The receiving process, on its side, consists of two PRs. The first, ***ParseFrame***, takes the raw bytes of the buffer and converts it into an internal processable structure. The second, ***ProcessFrame***, uses the internal structure to adapt the state of the running connection. Finally, the optional ***LogFrame*** enables the plugin to provide a textual description of the frame.

One could note that some PRs may have been merged. ***PrepareFrame***, ***FrameWireLen***, and ***WriteFrame*** could have been combined into a single PR, as some QUIC implementations do. However, other ones have different internals where the frame scheduling and the on-wire conversion are performed by distinct modules. To ease its adoption, and because splitting a function into several smaller ones is usually a small effort, Core QUIC adopts this decomposed scheme.

In addition to the previous ones, Core QUIC defines three additional PRs. ***Init*** is used to initialize the plugin, e.g., to make registrations. ***OnPluginTimeout*** provides timer-based callbacks and are further described in Section III-C. ***PluginControl*** serves as an external API, similar to `ioctl`, that can be called by other plugins or by the application.

While extensions usually define new behaviors with new PRs, they may also want to monitor routines without directly altering them, e.g., for logging purposes. Each PR has three anchors. The **DEFINE** anchor sets the behavior of the PR, with at most one attached plugin. The **BEFORE** and **AFTER** anchors defines hooks when calling and returning from the PR, respectively. Multiple plugins can attach on the same **BEFORE** and **AFTER** anchors. However, unlike the **DEFINE** one, code loaded with these anchors cannot modify the Core QUIC host implementation state and does not output any value.

Finally, note that a plugin should not support a new frame if the extension is not supported by the peer. To achieve this, Core QUIC splits the plugin loading into two steps. Initially, only ***Init***, ***WriteTransportParameter*** and ***DecodeTransportParameter*** are loaded. If the negotiation succeeds, or if no negotiation is needed, the plugin notifies Core QUIC that all the remaining routines can be loaded.

B. *Specifying a Common Representation*

Since plugins extend or tune running connections, they need to access their state. To interact with Core QUIC implementations, plugins need a stable interface. This not only covers the routines (as described in Sec. III-A), but also the exposed data structures. For instance, from Alg. 1, ***PrepareFrame*** outputs a frame structure that is then taken as input of ***FrameWireLen***, ***WriteFrame*** and ***OnFrameReserved***. This frame structure may have different internal representations on the Core QUIC host implementation side, but the plugin must have a stable layout.

To define such interface, Core QUIC introduces two elements in the `pluginop-common` library shared by both the host implementation and the plugin. The first element consists in a list of all the QUIC fields that a plugin may access or modify. Such fields derive from the core specification of QUIC [4]. Depending on the host implementation internals, some fields can be harder to collect. Instead of requiring a complete support, Core QUIC implementations can only provide access to a part of these fields. When loading a plugin, Core QUIC checks that the host implementation supports all the requested fields. If it is not the case, the plugin is denied loading. Such an approach enables incremental Core QUIC compliance, hence easing its adoption.

The exposed fields expose values corresponding to specific types whose plugins and the host implementation must agree on. To address this, the second element of the Core QUIC interface consists in a list of data structure types that can be exchanged between the Core QUIC implementation and the plugin. Besides primitive types (boolean, integer,...), it also provides time-related structures, socket address ones and QUIC-specific fields such as frames. From the implementer perspective, the main effort is to write the conversion from the QUIC host internal data structure to the Core QUIC one. Core QUIC follows the mapping of the QUIC specification [4] for the exposed structures, so such effort is in practice limited.

Plugins may need to exchange raw bytes with the Core QUIC host implementation. For such interactions, Core QUIC adopts a capability-based approach by introducing a `Bytes` type acting as an unforgeable token to the exchanged data.

C. *Running Plugins*

To dynamically extend the behavior of an implementation, Core QUIC introduces an environment in which the extension bytecode can be safely executed. Such an environment must cope with two main concerns: i) abstracting the plugin from the actual computer system, and ii) providing isolation mechanisms and monitoring capabilities on the external bytecode.

For this, Core QUIC relies on WebAssembly (Wasm) [8]. While initially scoped for the web browsers, it is also empowering embedded devices [9] and blockchain [10] usecases. A Core QUIC plugin, consisting in a Wasm module, defines the different hooks it wants to attach thanks to its exported functions. Specifically, the functions follow a naming convention to determine to which PRs they are attached. When loading the plugin, Core QUIC checks the different plugin's exported functions and maps their name to the associated PR. Except

TABLE I
SUMMARY OF THE CORE QUIC API AVAILABLE TO PLUGINS.

Category	Purpose
Getters/Setters	Interact with Core QUIC session
`get_input/save_output`	Parameter communication
`Bytes` API	Raw bytes reading/writing
File system API	Read and write persistent files (logs,...)
Registration API	Notify new types to host implementation
`enable_plugin`	Fully enable all routines (negotiated)
Time API	Provide timer-based callbacks

their name, all exported functions follow the same signature by having a single integer input and a single integer output. This output value notifies to the Core QUIC host implementation whether the plugin correctly performed its function or failed.

To enable interactions between the isolated Wasm plugin and the host implementation, Core QUIC makes available to the Wasm VM the plugin's API summarized in Table I. It includes getting and setting Core QUIC session's state, fetching and saving protocol routine-specific inputs and outputs, and reading/writing raw bytes or files. It also provides functions to register new types (transport parameters and frames) and to fully activate the plugin (see Sec. III-A). Finally, the Core QUIC API enables plugin to register timers along with callback functions, providing event-based programmability.

Yet, Core QUIC implementations may want to restrict parts of the API made available. For instance, a host implementation may prevent untrusted plugins from accessing sensitive session's fields, such as cryptographic keys. Core QUIC adopts a capability-based permission system whose rights are determined by the host implementation. Different permissions can be provided on a per-plugin basis, enabling different levels of trust in the loaded plugins. If a plugin tries to access a forbidden field or API, the running environment will deny access by returning an error code to the plugin.

D. Making Implementations Core QUIC-Compliant

To support Core QUIC, a QUIC host implementation needs to integrate the three previously discussed elements: i) turning internal functions into PRs, ii) adding conversion functions from internal data structures to Core QUIC ones, and iii) embedding the plugin environment runtime. The `pluginop` library eases this process by fully taking care of iii), remaining for the implementer to focus on i) and ii).

To turn an internal function into a PR, the implementer first wraps the content of the exposed function in an internal one. Then, the original function needs to check if a plugin is attached to the related PR with the given parameter. If so, the inputs are converted to the Core QUIC types. The plugin code can then be called at the related anchors thanks to the `pluginop` library. If the PR outputs values, these are then converted back to the host implementation types. When there is no plugin for the PR at the **DEFINE** anchor, the function then calls the original, wrapped internal function.

While the code performing data structure conversions is typically a purely additive change, the remaining changes are not.

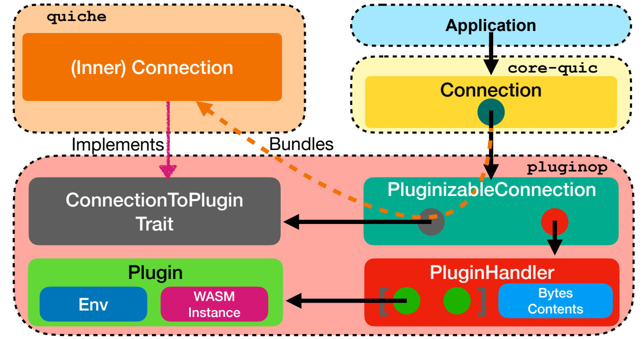


Fig. 2. The architecture of `pluginop`. Black arrows mean "contains".

To limit the codebase changes, and indirectly human mistakes, Core QUIC comes with a `pluginop-macro` library defining macros that automate such code changes at compilation time. Turning a function into a PR is then made with a single line of code, making Core QUIC adoption easier. Note that the implementer may need to add new code, e.g., to support sending new frames as depicted by Algorithm 1, but without requiring modification of the remaining code base.

IV. IMPLEMENTATION

To make Core QUIC concrete, we implemented the different architectural components depicted in Figure 1. We rely on the Rust language [11] as it enforces safety in performance-critical software. Each block in Figure 1 corresponds to a Rust library. To make an implementation Core QUIC-compliant, it needs to integrate in its code base the `pluginop` library. Figure 2 depicts its internal architecture. To make a session pluginizable, the implementer needs to wrap the connection structure into a `pluginop`'s `PluginizableConnection`. This structure contains two fields. The first is a `PluginHandler` that handles all the processing required by the dynamic pluginization. The second is the original connection structure itself, implementing the `ConnectionToPlugin` trait. This trait contains two methods that establishes a common interaction between the pluginization engine and the QUIC implementation, while abstracting the plugin complexity from the original connection.

The `PluginHandler` structure takes care of the pluginization process. As it operates on a trait, it is agnostic to the exact QUIC implementation. When a plugin wants to augment a Core QUIC session, it is inserted in the `PluginHandler`. A `Plugin` structure is created and contains the Wasm instance along with a plugin-specific environment (e.g., the plugin's API permissions, its inputs/outputs,...). Our `pluginop` library relies on the `wasmer` Wasm engine [12] to execute the plugin bytecode. When calling a PR, `PluginHandler` checks if a `Plugin` provides the requested routine in **DEFINE** anchor. If so, the related bytecode is then executed, and its result given back to the caller. Furthermore, the `pluginop` library provides the API required by the plugins to be run.

While plugins correspond to Wasm modules, their source code may be in any language. However, relying on unsafe

languages such as C may lead to exploitable Wasm bytecodes [13]. Because the Rust language prevents a whole range of safety issues by design, we offer to developers to write their plugins in safe Rust. Yet, the Wasm external functions rely on the Foreign Function Interface (FFI) which is, by essence, unsafe to use. To mitigate this, the `pluginop-wasm` Rust library handles all the serialization concerns related to the FFI and provides a safe abstraction on which plugins can be built. In addition, the `pluginop-wasm` exposes the Core QUIC data structures defined by the `pluginop-common` library. Overall, the generic libraries represent about 3250 lines of Rust code.

A. Integration into QUIC Implementations

To demonstrate the generality of Core QUIC's PRs and `pluginop`, we make two QUIC implementations Core QUIC-compliant: Cloudflare's `quiche` and `quinn`. Due to the QUIC complexity, many implementations abstract the I/O interactions from the protocol logic. This is the case for `quiche`, and `quinn` delegates the protocol logic handling in the `quinn-proto` library. Similar approaches are also taken by other QUIC implementations, such as `s2n-quic` and `neqo`.

Ideally, the integration effort of Core QUIC into existing implementations should be low. This translates into two concrete properties. First, the public API offered by these stacks should not introduce any breaking changes, such as structure or function renaming. Second, the code diff should be as low as possible, and with only purely additive changes when possible. Figure 2 shows how `pluginop` integrates with the `quiche` implementation. To make it pluginizable, its `Connection` structure needs to implement the `ConnectionToPlugin` trait and some of its internal functions should be converted to PRs thanks to the `pluginop-macro` library. Overall, this introduces a code diff of +1043, -46 over a code base of 38000 lines of Rust code. More than half of the additive changes are related to conversion code between internal data structures and Core QUIC's ones. To let applications benefit from the Core QUIC-compliant `quiche` version, we wrote a 100-line `core-quic` library exposing public `quiche` API along with functions to insert plugins on a per-session basis. These applications then simply need to rely on the `core-quic` library instead of the `quiche` without further code modification. Such results suggest reasonable efforts to deploy Core QUIC in `quiche`.

Making the `quinn` implementation Core QUIC-compliant only requires changes in the internal `quinn-proto` library, leading to a code diff of +1184, -253. These are still mostly additive changes, and the large negative change is due to the wrapping of 200 lines of code into a dedicated function. Yet, such changes remain reasonable for Core QUIC integration.

V. Exploring Use Cases

This section now provides some examples of use cases that Core QUIC can currently provide. Each of the discussed use cases and the corresponding plugins have been tested on our two Core QUIC-compliant implementations, namely `quiche` and `quinn`. We rely on the example client and server provided by each implementation.

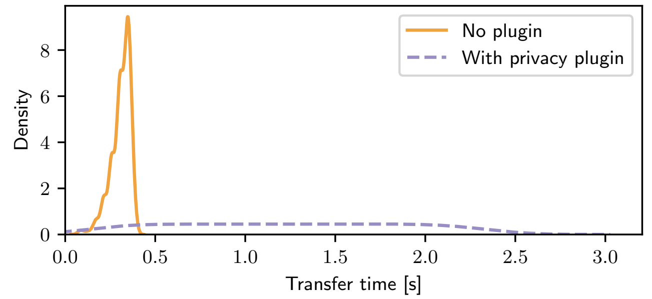


Fig. 3. Time distribution of client's packets of a 500 KB download.

A. Frame Logging

Analyzing QUIC network trace is not possible without having access to the encryption keys. Usually, QUIC implementations can output logs [14] to let operators debug their sessions. Core QUIC can also help in this process by embedding the logging process in a plugin. Such a plugin relies on the **BEFORE** and **AFTER** anchors of PRs to log the different events in a file. We implemented a prototype reporting ACK frame events, showing that plugins can provide monitoring capabilities without altering the base QUIC behavior.

B. QUIC-based Privacy Solution

Although QUIC encrypts all the control and application data, it is still exposed to privacy concerns. Packet sizes and timestamps are side-channel information that can be leveraged to build, e.g., website fingerprinting tools [15], [16]. The observer can map a website by translating the network trace into a feature vector. A classification algorithm then determines if the feature vector is similar to a website's one.

To counter such fingerprinting, endpoints could adopt defenses consisting in padding and delaying packets sent by the participating entities. To show how Core QUIC can contribute to this field, we built a plugin that pads all packets to the MTU size (using PADDING frames) and adds random delay between sent packets. To achieve such packet delaying, Core QUIC relies on the return value of the ***PrepareFrame*** routine to halt the batch of packet sending. The delay is then controlled by the Time API provided to the plugin (see Table I). When the plugin timer fires, Core QUIC invokes the ***OnPluginTimeout*** behavior provided by the plugin, which then enables further packet sending. Note that this plugin, while altering the sending logic, does not introduce any extension on the wire. Therefore, such a feature does not require negotiation and can be deployed on only one participating endpoint.

To evaluate its impact, we perform a Mininet [17] evaluation between a `quinn` client downloading a 500 KB file on a `quiche` server connected through a 50 Mbps, 40 ms RTT network path. We consider two setups: $i)$ the regular behavior of the unmodified implementations, and $ii)$ both the `quinn` client and the `quiche` server are tuned by the same plugin that pads and delays sent packets. For each variant, we perform 30 runs. Figure 3 shows that the unmodified time distribution

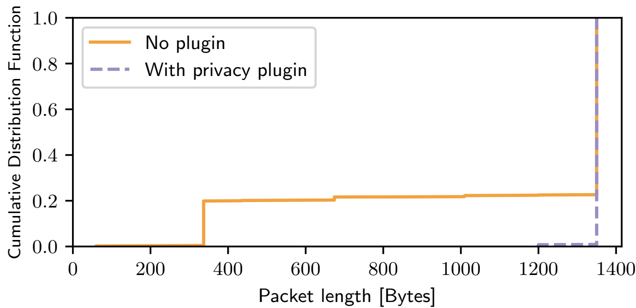


Fig. 4. Size distribution of server's packets of a 500 KB download.

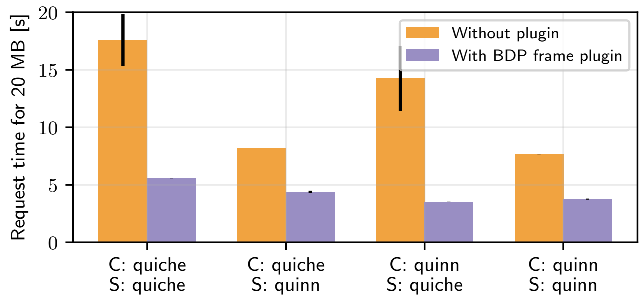


Fig. 5. The BDP frame plugin enables the server to directly operate to a suitable operating state. Each distinct case is run 20 times.

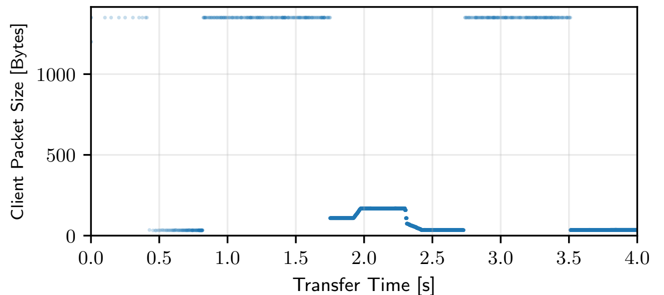


Fig. 6. Occurrences of client's packets when combining the DUMMY frame extension with the privacy padding one.

of the client's sent packets is driven by the received server's packets following the typical Cubic's slow start increase. The random delay introduced by the plugin breaks this pattern by flattening and expanding the probability density function curve. A similar result is observed for the time distribution of server's sent packets. Besides, Figure 4 shows that server's packet sizes reflect the congestion control limitation. The injected plugin hides such information by padding all the packets (except the QUIC Initial packets being 1200-byte long) to the MTU size of the network path, i.e., 1350 bytes.

C. Application-controlled Network Probing

Existing QUIC implementations provide different APIs. An application may want a feature that its serving implementation does not give access. As an example, an application facing an idling connection may need to probe its network path. A possible way to achieve this in QUIC is to send a PATH_CHALLENGE frame and the peer replies with a PATH_RESPONSE one. Our prototype plugin provides a ***PluginControl*** API where the application can request the sending of the probe. Another ***PluginControl*** entry point enables the application to get the experienced latency between the PATH_CHALLENGE and the PATH_RESPONSE.

D. Optimizing QUIC in Large RTT Scenarios

QUIC brings interests to operate it in challenging environment such as satellite or interplanetary communications, where there is a large bandwidth with a consequent latency due to the distance between endpoints. The congestion control's slow-start process usually takes several RTTs before operating on the available bandwidth, affecting short transfers' experience. A recent proposal [18] suggests defining a frame where the server advertises to the client its congestion control state. In future connections to the same server, the client sends back the content of that frame to the server and allow it to resume its congestion control state to directly operate at the target rate.

We implement the proposed BDP frame idea in a plugin. To evaluate its potential benefits, we consider a Mininet scenario where endpoints are connected to a 50 Mbps, 500 ms RTT network path. We consider the four possible setups where the `quinn` and `quiche` implementations operate as the client and/or the server. For each case, we compare the default behavior of the implementations with the one provided by our plugin. Figure 5 shows the median and the standard deviation of the transfer time to download a 20 MB file. In such a large RTT scenario, a lot of time is wasted in probing the network capacity and the transfer completes when the `quinn` server finally reaches its target rate. We note that the Cubic implementation in the `quiche` server badly behaves in such cases, exiting slow start without reaching the optimal value. With the plugin's BDP frame, the server can directly reuse the optimal congestion control state for the scenario and the transfer time considerably decreases in all studied cases.

E. Combining Extensions

The design of Core QUIC supports the inclusion of multiple plugins providing orthogonal extensions. All the previously discussed plugins can be injected over a single connection to merge their functionalities. But plugins can also cooperate between each other's. To illustrate this, we setup the following toy case. The privacy padding plugin has a ***PluginControl*** routine letting the caller to dynamically enable the packet padding and delaying features over the session. We then define a new plugin introducing a dummy frame. This plugin negotiates it and then sends this new frame when there is no other dummy frame in-flight. This implements a once-per-perceived-RTT sending logic. Every four received dummy frame, the endpoint calls the ***PluginControl*** routine provided by the privacy padding plugin to toggle its activation.

To visualize this feature, we consider a scenario where the `quiche` client loads both the privacy padding and the dummy frame plugins, while the `quinn` server only inserts the dummy frame one. We consider a scenario where endpoints are connected by a 50 Mbps, 100 ms RTT network path. Figure 6 shows the time and the size of packets sent by the client for a given run. Initially, the privacy plugin pads the client's packets to the MTU size. After about 400 ms, the client received four dummy frames from the server and disabled the privacy plugin, leading to much smaller packets. Then, about 400 ms later, the privacy plugin is enabled again. This toggling continues until the end of the transfer, but this later tends to happen every second. This is due to the Cubic's server congestion control that saturates the path, and the network buffer introduces queuing delay, increasing the perceived RTT. The client packets between 1.75 s and 2.5 s contains large ACK frames with up to 70 ACK blocks. These illustrate packet losses due to the saturation of the network buffer.

VI. Related Works

While there is a large research community around programmable networks using Software Defined Networks [19] and P4 [20], these target network layer solutions and usually require specific hardware. At the transport layer, there were previous attempts in making configurable and extensible transport protocols [21], [22]. These were not widely deployed for several reasons (unencrypted protocols, very specific implementation architecture,...). With the rise of the QUIC protocol, researchers [7] proposed PQUIC to dynamically extend a QUIC implementation with plugins executed by a virtual machine. For this, PQUIC relies on a user-space version of the eBPF VM thanks to its popularity in the Linux kernel [23]. However, plugins are tight to the researchers' implementation and PQUIC relies on a core hash-table holding all exposed operations, requiring large implementation changes. More recently, Wirtgen et al. [3] proposed xBGP, a solution to dynamically extend BGP implementations with a same plugin. While xBGP has similar ideas, Core QUIC focuses on the QUIC protocol and relies on different building blocks, e.g., to run plugins. Both PQUIC and xBGP relies on eBPF bytecode. While the in-kernel eBPF has now a mature support, it bases on features provided by the Linux kernel (verifier, BPF Type Format,...). The integration for other applications requires consequent effort that is error-prone such as memory management and isolation [24]. Unlike eBPF, Wasm, on which Core QUIC is based, was designed with security in mind [25]. There are also orthogonal works applying dynamic programmability in other encrypted protocols [26], [27].

VII. Conclusion

This paper presented Core QUIC, an extension to the QUIC standard that enables its compliant implementations to be dynamically extended by a common, implementation-agnostic plugin. Once deployed, Core QUIC nodes provides protocol tuning and extensibility without requiring any change to the implementation's binary. We demonstrated that such an approach is deployable by making two different QUIC implementations Core QUIC-compliant with limited code base changes. We also showcased a few use cases where a same plugin can extend both Core QUIC-compliant implementations.

We believe Core QUIC addresses the transport protocol extensibity inertia. While it can be deployed at both sides, Core QUIC is well-suited for client implementations, often being out of control, and server can then natively implement extensions they want to support. Our future works include making more implementations Core QUIC-compliant to strengthen the genericity of the API and extending Core QUIC to cover more complex use cases requiring more than operations on frames.

Artifacts available: https://core-quic.github.io.

References

[1] M. Honda *et al.*, "Is It Still Possible to Extend TCP?" in *ACM IMC '11*, 2011, pp. 181–194.

[2] L. Budzisz *et al.*, "A taxonomy and survey of SCTP research," *ACM Computing Surveys (CSUR)*, vol. 44, no. 4, p. 18, 2012.

[3] T. Wirtgen *et al.*, "xBGP: Faster innovation in routing protocols," in *USENIX NSDI'23*, Apr. 2023, pp. 575–592.

[4] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021.

[5] Q. De Coninck, "Core QUIC: Enabling Dynamic, Implementation-Agnostic Protocol Extensions," *arXiv preprint arXiv:2405.01279*, 2024.

[6] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018.

[7] Q. De Coninck *et al.*, "Pluginizing QUIC," in *ACM SIGCOMM '19*. Beijing, China: ACM Press, 2019, pp. 59–74.

[8] A. Haas *et al.*, "Bringing the web up to speed with WebAssembly," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 185–200, 2017.

[9] M. Jacobsson and J. Willén, "Virtual machine execution for wearables based on webassembly," in *EAI BODYNETS*, 2018, pp. 381–389.

[10] eosio, "Eos virtual machine: A high-performance blockchain webassembly interpreter," 2019. [Online]. Available: https://eos.io/news/eos-virtual-machine-a-high-performance-blockchain-webassembly-interpreter/

[11] N. D. Matsakis and F. S. Klock II, "The rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, 2014.

[12] WasmerIO, "Wasmer," https://github.com/wasmerio/wasmer.

[13] D. Lehmann *et al.*, "Everything old is new again: Binary security of WebAssembly," in *USENIX Security '20*, 2020, pp. 217–234.

[14] R. Marx *et al.*, "Debugging quic and http/3 with qlog and qvis," in *ACM ANRW '20*, 2020, pp. 58–66.

[15] M. S. Rahman *et al.*, "Tik-tok: The utility of packet timing in website fingerprinting attacks," *PoPETs '20*, vol. 3, pp. 5–24, 2020.

[16] J.-P. Smith *et al.*, "QCSD: A QUIC client-side website-fingerprinting defence framework," in *USENIX Security '22*, pp. 771–789.

[17] N. Handigol *et al.*, "Reproducible Network Experiments Using Container-Based Emulation," in *ACM CoNEXT '12'*, pp. 253–264.

[18] N. Kuhn *et al.*, "Signalling CC Parameters for Careful Resume using QUIC," IETF Draft draft-kuhn-quic-bdpframe-extension-05, Mar. 2024.

[19] N. McKeown *et al.*, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, 2008.

[20] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM CCR*, vol. 44, no. 3, pp. 87–95, 2014.

[21] P. G. Bridges *et al.*, "A Configurable and Extensible Transport Protocol," *IEEE/ACM ToN*, vol. 15, no. 6, pp. 1254–1265, Dec. 2007.

[22] P. Patel *et al.*, "Upgrading Transport Protocols using Untrusted Mobile Code," *ACM SIGOPS OSR*, vol. 37, no. 5, pp. 1–14, 2003.

[23] M. Fleming, "A thorough introduction to eBPF," *Linux Weekly News*, Dec. 2017, https://old.lwn.net/Articles/740157/,Accessed:2021-02-04,.

[24] Q. De Coninck *et al.*, "On integrating ebpf into pluginized protocols," *ACM SIGCOMM CCR*, vol. 53, no. 3, pp. 2–8, 2024.

[25] J. Dejaeghere *et al.*, "Comparing security in ebpf and webassembly," in *ACM Workshop on eBPF and Kernel Extensions*, 2023, pp. 35–41.

[26] F. Rochet *et al.*, "Tcpls: Modern transport services with tcp and tls," in *ACM CoNEXT'21*, 2021, p. 45–59.

[27] F. Rochet and T. Elahi, "Towards flexible anonymous networks," *arXiv preprint arXiv:2203.03764*, 2022.