Pluginizing QUIC*

Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, Olivier Bonaventure
UCLouvain, Belgium
firstname.lastname@uclouvain.be
https://pquic.org

ABSTRACT

Application requirements evolve over time and the underlying protocols need to adapt. Most transport protocols evolve by negotiating protocol extensions during the handshake. Experience with TCP shows that this leads to delays of several years or more to widely deploy standardized extensions. In this paper, we revisit the extensibility paradigm of transport protocols.

We base our work on QUIC, a new transport protocol that encrypts most of the header and all the payload of packets, which makes it almost immune to middlebox interference. We propose Pluginized QUIC (PQUIC), a framework that enables QUIC clients and servers to dynamically exchange protocol plugins that extend the protocol on a per-connection basis. These plugins can be transparently reviewed by external verifiers and hosts can refuse non-certified plugins. Furthermore, the protocol plugins run inside an environment that monitors their execution and stops malicious plugins. We demonstrate the modularity of our proposal by implementing and evaluating very different plugins ranging from connection monitoring to multipath or Forward Erasure Correction. Our results show that plugins achieve expected behavior with acceptable overhead. We also show that these plugins can be combined to add their functionalities to a PQUIC connection.

CCS CONCEPTS

• **Networks → Transport protocols**; *Network protocol design*;

KEYWORDS

PQUIC, QUIC, Transport protocol, Network architecture, Plugin, Protocol operation, eBPF

ACM Reference Format:
Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, Olivier Bonaventure. 2019. Pluginizing QUIC. In *SIGCOMM '19: 2019 Conference of the ACM Special Interest Group on Data Communication, August 19–23, 2019, Beijing, China.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3341302.3342078

*This work does not raise any ethical issues. Quentin De Coninck and François Michel are both F.R.S.-FNRS Research Fellows. Axel Legay is also affiliated with Aalborg University.

SIGCOMM '19, August 19–23, 2019, Beijing, China

ACM ISBN 978-1-4503-5956-6/19/08…$15.00
https://doi.org/10.1145/3341302.3342078

1 INTRODUCTION

Transport protocols including TCP [79], RTP [34], SCTP [91] or QUIC [51, 58] play a key role in today's Internet. They extend the packet forwarding service provided by the network layer and include a variety of end-to-end services. A transport protocol is usually designed to support a set of requirements. Protocol designers know that these requirements evolve over time and all these protocols allow clients to propose the utilization of extensions during the handshake.

These negotiation schemes have enabled TCP and other transport protocols to evolve over recent decades [24]. A modern TCP stack supports a long list of TCP extensions (Windows scale [52], timestamps [52], selective acknowledgments [65], Explicit Congestion Notification [81] or multipath extensions [32]). However, measurements indicate that it remains difficult to deploy TCP extensions [35, 45]. The Windows scale and selective acknowledgment options took more than a decade to be widely deployed [35]. The timestamp option is still not supported by a major desktop OS [3]. Multipath TCP is only available on one major mobile OS [2]. This slow deployment of TCP extensions is caused by three main factors. First, popular stacks rarely implement TCP extensions unless they have been approved by the IETF. Second, TCP is still part of the operating system and client and servers implementations are not upgraded at the same speed. Often, maintainers of client (resp. server) implementations wait until server (resp. client) implementations support a new extension before implementing it. This results in a chicken-and-egg deployment problem. Third, some middleboxes interfere with the deployment of new protocol extensions [42, 46].

Initially proposed by Google to replace HTTP2/TLS/TCP, QUIC [58] addresses some of these deployment issues. A QUIC connection starts with a handshake during which transport parameters are exchanged and TLS keys are negotiated. QUIC then encrypts all the user data and most of the packet headers, this prevents most of the interferences from middleboxes [58]. QUIC includes a flexible framing mechanism to encode user data and control information. QUIC frames are exchanged inside packets that are encrypted and authenticated using the keys derived by TLS. Like SCTP, QUIC supports the reliable delivery of data over multiple streams and includes congestion control schemes and retransmission techniques to recover from packet losses.

Google's version of QUIC was proprietary and did not require IETF consensus to be updated. As QUIC runs above UDP it is possible to ship it as a library which can be updated as often as applications. Measurements indicate that Google updated its version of QUIC at the same pace as its Chrome browser [87].

We believe that the openness of the Internet is a key element of its success, and ultimately anyone should be able to tune or extend Internet protocols to best fit their needs. Traditional transport protocols like TCP are tuned by using configuration variables or socket options [25] and more recently with eBPF code [8]. Although recent works enable an application to add new TCP options [97], it remains impossible to precisely configure the underlying TCP stack.

In this paper, we completely revisit the extensibility of transport protocols. We consider that transport protocols should provide a set of basic functions which can be tuned, combined and dynamically extended to support new use cases on a per-connection basis. Such an approach could enable QUIC applications to adapt the underlying transport layer to their specific needs, e.g., using specialized retransmission algorithms or taking advantage of non-standard extensions. This would bring innovation back in the transport layer with researchers and software developers being able to easily implement, test and deploy new protocol features. For this, we leverage the extensibility and security features of QUIC. We make four main contributions in this paper.

- We design a technique where an extension to the QUIC protocol is broken down in a *protocol plugin* which can be dynamically attached to an existing implementation. These plugins interact with this implementation through code which is dynamically inserted at specific locations called *protocol operations*.
- We propose a safe and scalable technique that enables the on-demand exchange of protocol plugins over QUIC connections. This solves the deployment problem of existing protocol extensions.
- We implement a prototype of Pluginized QUIC (PQUIC) by extending `picoquic` [48], one of the most complete implementations of IETF QUIC [76]. We add to `picoquic` a virtual machine that allows executing the bytecode of protocol plugins in a platform independent manner while monitoring their behavior.
- We demonstrate the benefits of PQUIC through plugins that add new monitoring capabilities, support for Multipath QUIC [19], the Unreliable Datagram Extension [75] and several Forward Erasure Correction techniques.

This paper is organized as follows. Section 2 overviews the design of PQUIC and how a PQUIC implementation supports plugins. Section 3 discusses the security models, attacks and how PQUIC solves them when exchanging plugins over QUIC connections. Section 4 presents several use cases of protocol extension with plugins. Section 5 describes how plugin properties can be verified. Section 6 contrasts with related works. Section 7 discusses the broader implications and questions of this work. Section 8 concludes and considers future work.

2 LOCAL PLUGIN INSERTION

Our design for **Pluginized QUIC** (**PQUIC**) builds upon the QUIC protocol whose specification is being finalized within the IETF[1] [51]. This IETF specification is based on an initial design from Google that is used by Chrome and other applications [58]. From a protocol viewpoint, there are few differences between PQUIC and QUIC. We defer the explanation of these differences until Section 3.

From the implementation viewpoint, the main difference between a PQUIC implementation[2] and a QUIC one is that PQUIC is easily customizable on a per-connection basis. This customization relies upon a modular, extensible design that allows adding and modifying behaviors for the target flows. A PQUIC implementation can be extended by dynamically loading one or more *protocol plugins*. A *protocol plugin* consists of platform-independent bytecode which can be executed within the PQUIC implementation.

A PQUIC implementation provides an API to protocol plugins. Most protocol implementations are designed as black-boxes that expose a small external API to applications. For example, a TCP implementation exposes the socket API. A PQUIC implementation can be represented as a gray-box containing a set of functions that are exposed to protocol plugins. In PQUIC, we call these functions *protocol operations*. These are common routines being part of any implementation, and the workflow of PQUIC can be expressed as a succession of such protocol operations. As in a C API, each protocol operation has a specification and a set of conditions under which it should be called. Sample protocol operations in PQUIC include the parsing and the processing of frames, setting the retransmission timer, updating the RTT, removing acknowledged frames from the sending buffer, etc.

On-the-fly protocol plugin insertion. A *pluglet* consists of bytecode instructions implementing a function, e.g., computing an RTT estimate. A *manifest* contains the globally unique plugin name and indicates how to link several pluglets to a connection, i.e., to which protocol operations they should be attached. The combination of pluglets and the manifest forms a *protocol plugin*. Once a PQUIC connection is established, PQUIC can potentially load plugins at any time.

Isolation between connections and between plugins. Each plugin is instantiated to operate on a given connection. Our framework ensures that each instance has its own memory which is only shared among pluglets of this plugin. The plugin memory is isolated from access or sharing with any other plugin or connection. This yields strong memory safety guarantees for the plugins and the sharing of information. Interactions are still possible through the protocol operation interface or by calling the functions exposed by PQUIC. However, these are clearly defined information flows that ease reasoning about the behavior and the safety of the plugins.

The rest of this section details the core elements of PQUIC. We first describe the environment executing pluglets. Then, we elaborate on the concept of protocol operations. We finally describe how PQUIC interfaces with pluglets.

2.1 Pluglet Runtime Environment (PRE)

Pluglets are the building blocks of the protocol plugins. These pieces of bytecode are independent of the PQUIC implementation itself. Therefore, we need to provide an environment to execute them. This environment has to solve two major concerns. First, it has to provide an abstraction where plugins can run regardless of

[1] We base this work on the version 14 of the IETF QUIC drafts [51].

[2] We applied the principles described in this section to two very different QUIC implementations. We first implemented an early prototype based on `quic-go` [13] written in Go. We then based our work on `picoquic` [48] written in C. This paper only discusses `picoquic`, the `quic-go` version is described in a technical report [21].

the underlying hardware and operating system. Second, given the untrusted nature of the plugins, the environment should keep each pluglet under control.

To address these two issues, PQUIC executes plugins inside a lightweight virtual machine (VM). Various VMs have been proposed for different purposes [4, 30, 36, 39, 61, 101]. In this paper, our *Pluglet Runtime Environment* (PRE) relies on a user-space implementation [49] of the eBPF VM [30]. Although being present in the Linux kernel since 2014 where it has been used to support various services [8, 26, 37], the eBPF VM can be too restrictive to implement some legitimate behaviors. The kernel-space eBPF VM includes a verifier that is very conservative, as it puts hard limits on the size and complexity of an acceptable eBPF program.

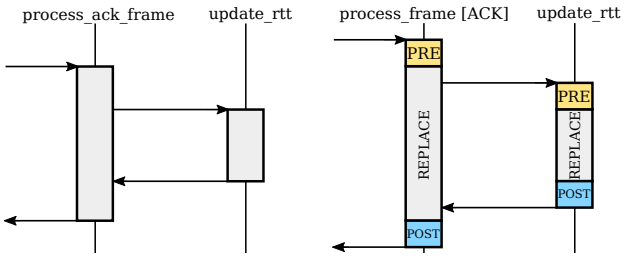
Our implementation extends a relaxed version of the eBPF verifier with additional monitoring capabilities. Those are similar to works in Software-Based Fault Isolation [100, 106]. First, our PRE checks simple properties of the bytecode to ensure its (apparent) validity. This includes checking that: (*i*) the bytecode contains an exit instruction, (*ii*) all instructions are valid (known opcodes and values), (*iii*) the bytecode does not contain trivially wrong operations (e.g., dividing by zero), (*iv*) all jumps are valid, and (*v*) the bytecode never writes to read-only registers. Furthermore, our PRE statically verifies the validity of stack accesses. A plugin is rejected if any of the above checks fails for one of its pluglets.

Second, our PRE monitors the correct operation of the pluglets by injecting specific instructions when their bytecode is JITed. These monitoring instructions check that the memory accesses operate within the allowed bounds. To achieve this, we add a register to the VM that cannot be used by pluglets. This register is used to check that the memory accesses performed by a pluglet remain within either the plugin dedicated memory or the pluglet stack. Any violation of memory safety results in the removal of the plugin and the termination of the connection. The LLVM Clang compiler supports the compilation of C code into eBPF. This allows us to abstract the development of pluglets from eBPF bytecode and propose a convenient C API for writing pluglets.

2.2 Protocol Operations

In order to attach pluglets to PQUIC, we define an API revealing the insertion points and the interface between PQUIC and the pluglets. We break down the protocol execution flow into generic subroutines. These specified procedures are called *protocol operations*. Each has its human-readable identifier, inputs, outputs and specifications. A protocol operation with a parameter has a high-level goal, but its actual behavior, and therefore its function, changes depending on the given parameter. This provides a generic entry point allowing the definition of new behaviors without changing the caller for, e.g., the serialization of new QUIC frames. Our PQUIC implementation currently includes 72 protocol operations. Four of them take a parameter.[3] We can split these operations into several categories. A first category concerns the handling of the QUIC frames. This includes their parsing, processing and writing. A second category groups all the internal processing of QUIC. It contains the logic for retransmissions, updating the RTT, deciding which stream to send next, etc. A third category involves QUIC packet management. It includes setting the *Spin Bit* [96], retrieving the connection IDs, etc. A fourth category contains several events in the connection workflow, whose protocol operations have empty anchor points, i.e., no default behavior. For instance, a protocol operation exists after decoding all frames of an incoming packet or after a packet loss.

[3]Please refer to https://pquic.org for the latest numbers.



(a) Monolithic design. **(b) PQUIC design.**

Figure 1: Turning a monolithic design into protocol operations with the ACK processing example. It also illustrates the different anchors for the pluglets.

To illustrate how an implementation can be split into protocol operations, consider the example shown in Figure 1a. The processing of an ACK frame would likely be performed in its dedicated function. One of its sub tasks is the computation of the RTT estimation, which is implemented in its own function too. PQUIC keeps the same programming flow. As shown in Figure 1b, PQUIC functions are wrapped by a protocol operation whose human-readable string identifier describes its goal. While the name of the protocol operation and the original function are similar, the processing of ACK frames is linked to a more generic `process_frame` operation taking `ACK` as parameter. As illustrated, a given protocol operation can call other operations. Furthermore, protocol operations are split into three *anchors*, each of which is a possible insertion point for a pluglet. Protocol operations with parameters propose a specific set of anchors for each parameter value. The first anchor, called *replace*, consists of the actual implementation of the operation. This part is usually provided by the original PQUIC function. This mode enables a pluglet to override the default behavior. Because it may modify the connection context, at most one pluglet can *replace* a given protocol operation. If a second one tries to replace the same operation, it will be rejected and the plugin it belongs to will be rolled back. The two other anchors, *pre* and *post*, attach the plugin just before (resp. just after) the protocol operation invocation. These modes are similar to the eBPF kprobes in the Linux kernel [55]. By default, those are no-ops in PQUIC. Unlike the *replace* anchor, any number of *pre* and *post* pluglets can be inserted for a given protocol operation. However, they only have read access to the connection context and the operation arguments and outputs. The only write accesses they have is to their pluglet stack and their plugin-specific memory. In the rest of the paper, unless explicitly stated, we discuss pluglet insertions in *replace* mode, and refer to pluglet inserted in *pre* and *post* as passive pluglets.

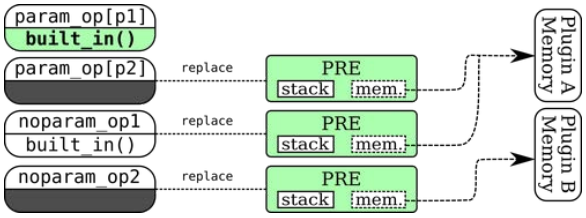


Figure 2: Attaching PREs in *replace* mode to protocol operations.

Functions	Usage
get/set	Access/modify connection fields.
pl_malloc/pl_free	Management of the plugin memory.
get_opaque_data	Retrieve a memory area shared by pluglets.
pl_memcpy/pl_memset	Access/modify data outside the PRE
plugin_run_protoop	Execute protocol operations.
reserve_frames	Book the sending of QUIC frames.

Table 1: PQUIC API exposed to pluglet bytecode.

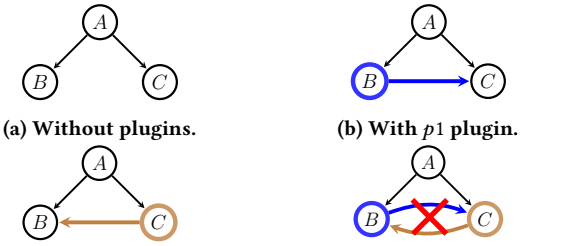
2.3 Attaching Protocol Plugins

Implementing protocol extensions may require a combination of several pluglets forming a plugin. The PRE provides a limited instruction set and isolates the bytecode from the host implementation. Therefore, plugins require an interface with which they can operate on their connection. Moreover, a plugin might need to share some state among its pluglets.

To address these needs, PQUIC is organized as illustrated in Figure 2. As explained in the previous section, the behavior of a protocol operation is either provided by a built-in function (e.g. `param_op [p1]`) or overridden by a pluglet (e.g. `noparam_op1`). Observe that plugins can also provide new protocol operations absent from the original PQUIC implementation. This can be done either by hooking a new parameter value for an existing protocol operation (e.g. like `param_op [p2]`) or by adding a new protocol operation (e.g. `noparam_op2`). PQUIC is thus extensible by design.

A PRE is created for each inserted pluglet. Each PRE contains its own registers and stack. The PRE heap memory points to an area common to all pluglets of a plugin, as illustrated in Figure 2. This link, ensured by the PQUIC implementation, provides a communication channel between pluglets. In addition to the isolation benefits, this architecture ensures that aggressive or ill memory management only affects the plugin itself. Thanks to our PRE, pointer dereferencing is restricted only to the pluglet stack or its plugin memory. In addition, the pluglet also needs to communicate with the host implementation to interact with its connection. As in a related work [1], PQUIC exposes some functions to the PRE. These functions form an API that pluglets can use (Table 1). We detail its six major operations below.

Exposing connection fields through getters and setters. Letting plugins directly access the fields of PQUIC structures makes the injected code very dependent on PQUIC internals. Consider the case of two hosts with different PQUIC versions. If the newest version added a new field to a structure being used by a pluglet, the offset contained in its bytecode would point to a possibly different field, leading to undefined behavior. Therefore, this interface abstracts the implementation internals from the pluglets, making them compatible with different PQUIC versions or implementations. In addition, it allows the PQUIC host to monitor which fields



(a) Without plugins. **(b) With $p1$ plugin.**

(c) With $p2$ plugin. **(d) With both $p1$ and $p2$ plugins.**

Figure 3: Combining plugins requires protocol operation monitoring. (a), (b) and (c) are valid calls graphs while (d) is not since it creates a loop between B and C.

are accessed by the injected code. A host could thus reject plugins based on the fields that it wishes to access. For example, a client could refuse plugins that modify the *Spin Bit*, as it is not encrypted. Similarly, depending on its local user policies, a host could accept or deny a plugin accessing the TLS state.[4]

Managing plugin memory. Pluglets might need to keep persistent data across calls. Therefore, we provide functions to allocate and free memory in the plugin dedicated area. Our framework dedicates a fixed-size memory area split into constant size blocks [56]. Such approach provides algorithmic $\Theta(1)$ time memory allocation while limiting fragmentation.

Retrieving data shared by pluglets. Pluglets from the same plugin might need to access a common data structure. Pluglets can assign an identifier to a given plugin memory area enabling them to retrieve and modify it consistently.

Modifying connection memory area. Plugins might need to modify memory outside the PRE. For instance, a pluglet might need to write a new frame inside a buffer. The API keeps control on the plugin operations by checking the accessed memory areas.

Calling other protocol operations. This is required when a protocol operation depends on another one. However, such capability raises potential safety issues. As plugins can call any protocol operation, a PQUIC implementation needs to take care of possible loops due to these calls. To prevent such loops, the call graph of the protocol operations must always remain loop-free. However, ensuring this property for any combination of loop-free plugins is not practical to assess before executing them due to the combinatorial state explosion. Consider the example shown in Fig. 3. There are three protocol operations A, B and C, all guaranteed to terminate. Even if both $p1$ and $p2$ plugins are legitimate, their combination might introduce an infinite loop, as shown in Figure 3d. To avoid this situation, a PQUIC implementation keeps track of all the currently running protocol operations in the call stack. If a call is requested for an operation that is already running, PQUIC stops the connection and raises an error.

Scheduling the transmission of QUIC frames. PQUIC provides a way for pluglets to reserve a slot for sending frames, whether they define a new frame or use an existing type. However, it should enforce two rules. First, plugins must not prevent PQUIC from sending application data. Therefore, as long as there is payload data to be sent, standard QUIC frames such as `STREAM`, `ACK` and `MAX_DATA`

[4] We do not currently expose TLS keys to plugins.

should have a guaranteed fraction of the available congestion window. Second, a plugin sending many large frames should not be able to starve other plugins. Concurrently active plugins should have a potentially fair share of the sending congestion window. To achieve this, PQUIC includes a frame scheduler which is a combination of class-based queuing [31] and deficit round robin [89]. Frames are classified based on their origin, either from the core implementation itself or from plugins. When both classes are pushing frames, the scheduler ensures that the core ones get at least $x\%$ of the available congestion window. A deficit round robin then distributes the remaining budget between the plugin frames.

2.4 Interacting with Applications

We showed how plugins can interact within PQUIC. Plugins can also interact with the application using PQUIC. This allows them to extend the application-facing interface of PQUIC to bring new functionalities. For example, a plugin could implement a message mode for QUIC to supplement the standardized ordered byte-stream abstraction [51]. This communication is established in a per-plugin bidirectional manner. First, an application can call *external* protocol operations. These are new anchor points that can be defined when injecting pluglets. The *external* mode is similar to the *replace* mode, but it makes the protocol operation only executable by the application. This allows it to directly invoke new methods, e.g. queuing a message to be sent. Second, a plugin can asynchronously push messages back to the application, so that it remains independent of the application control flow.

2.5 Reusing Plugins across Connections

The plugin injection involves the creation and the insertion of PREs at their anchor and the instantiation of the plugin heap. These remain dedicated to a given connection for its entire lifetime. Once the connection completes, the plugin resources may be freed. Nevertheless, it is likely that plugins get reused on subsequent connections. Furthermore, PREs only depend on the pluglets and are isolated from the connections on which they operate. Therefore, to limit the injection overhead, we introduce a cache storing the plugin associated PREs and memory. When a new connection injects the same plugin, it can reuse the cached PREs as is, without verifying or compiling the pluglets again. The plugin heap must be reinitialized to avoid leaking information between unrelated connections.

3 EXCHANGING PLUGINS

The previous section has described the system aspects of PQUIC and the possibility of extending a PQUIC implementation through protocol plugins that are injected on the local host. As TLS 1.3 is an integral part of QUIC, third parties such as middleboxes or attackers cannot modify the data exchanged over such a connection. Protocol plugins could, therefore, be exchanged over an existing QUIC connection. Accepting remote protocol plugins poses the challenge of establishing the trust in their validity, e.g. their termination. We first propose an open system solving this challenge.

Our system bears a similarity to Certificate Transparency [59]. Both are using a Merkle tree as the base for the system log and allow each party to contribute to their global protection only by checking their own safety. Our design enables independent developers to publish their own plugins for which the PQUIC peers' trust in their validity is established by independent plugin validators (PV). Yet, we have fundamental differences from Certificate Transparency in the various roles of the distributed system, and within the construction of the system log itself. As opposed to Certificate Transparency, no party has to track the entire log to keep the users safe. We directly offer to plugin developers an efficient mean, i.e., logarithmic in the number of plugins published, to check whether any spurious plugin has been published on their behalf. This prevents the need for third party monitor to emerge. A second important design choice make PQUIC peers able to formulate their safety requirements by combining the PVs they trust. This process allows end-users to pin security requirements as a logic expression. The validity of plugins with respect to this expression can be efficiently checked.

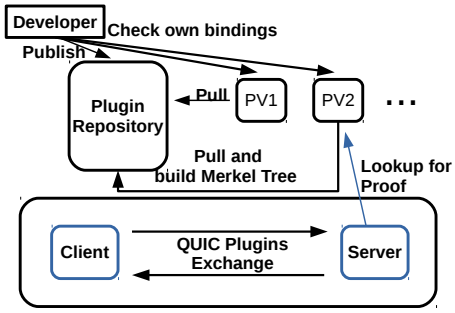


Figure 4: Secure Plugin Management System.

In the following sections, we explain some of the additional security properties our system offers. Finally, we describe our extension to the QUIC protocol to support the exchange of protocol plugins over a QUIC connection.

3.1 Distributing Trust

A simple approach for establishing trust in plugins would be for an application developed by `foo.com` and using PQUIC to only accept plugins from authenticated `foo.com` servers. We go beyond this restrictive approach and propose the open system illustrated in Figure 4. It includes four types of participants: (*i*) the plugin developers, (*ii*) the Plugin Repository (PR) that hosts protocol plugins, (*iii*) the plugin validators that vouch for plugins validity and (*iv*) the PQUIC peers.

Plugin developers may be independent of the PQUIC implementers. They write plugins and publish them on the PR. Publishing a plugin forms a *binding*, which we define as the concatenation of the globally unique plugin name with the bytecodes of all its pluglets and the associated manifest, i.e.

$$binding = pluginname||plugincode$$

The PR holds all protocol plugins from all developers and centralizes the secure communication between all participants.

A plugin validator (PV) validates the correct functioning of a plugin. The validation itself depends on the PV capabilities as described in Section 5. PVs can obtain the source code from developers willing to ease their validation, but must first check that they are able to reproduce the submitted code. PVs can serve the bytecodes of plugins they validated. The state of our system, i.e. the plugins

hosted on the PR and their validation by PVs, progresses on a discrete time scale defined by the *epoch* value. At each epoch, plugins can be added or updated, and each PV can update their plugins validation.

Each PV builds a Merkle Prefix Tree [68] containing the plugins it successfully validated and digitally signs its root, forming a Signed Tree Root (STR). The STR is sent to the PR. A PV can build at most one tree per epoch. More details on building the Merkle Prefix Tree by PVs are provided in Section 3.3. When the PR is offline, the PVs can serve their own STRs. The absence of a plugin in a tree can be due to two reasons. Either the validation failed or no validation took place at that epoch. The failure cause is communicated to the PR. Plugin developers monitor the validations published by PVs to ensure the tested plugins match the submitted code.

Before exchanging plugins, PQUIC peers must provide evidence of plugin validity. Our system allows expressing requirements in terms of PV approbation. More precisely, if PV_i is the identifier of a PV, a PQUIC implementation can send a logical formula that expresses its required validation, e.g., $PV_1 \wedge (PV_2 \vee PV_3)$. This design allows the PQUIC peers to precisely express their required safety guarantees.

Our system is distributed. This makes PQUIC peers tolerant to participant failures. In the previous example, if both the PR and PV_3 are offline, then the peer can rely on both PV_1 and PV_2 to validate the provided plugin.

3.2 Threat Model and Security Goals

Our distributed system addresses the following threat model. Any participant can act maliciously. Plugin developers may publish malicious code. A PQUIC peer may want to inject illegitimate code. PVs may give false assertions on the validity of a plugin. The PR may equivocate on the STRs received from PVs. Both PR and PVs may modify the code served, or impersonate the developers.

Our system offers the guarantee that some aforementioned problems are immediately detected, and the others are eventually detected. It also ensures that a plugin name securely summarizes its code. Furthermore, a PQUIC peer is always able to identify which party faulted. As a result, given that the PR and PVs can be freely selected by PQUIC peers, we assume that they are willing to protect their reputations, which could be degraded upon discovering problems. In summary, our system covers the following security goals.

Central identities, distributed validation. The PR centralizes the identities of both developers and PVs. A PV can publish its current STR necessary for the *proof of consistency*, and notify developers about plugins that failed its validation. A developer can publish plugins, report PVs equivocations and inconsistencies of its bindings at all PVs.

Non-equivocation. A PV cannot equivocate by presenting different STRs to different PQUIC peers. If it does, participants eventually detect this with the help of others. An STR acts as a tamper-resistant log for all the bindings validated by a PV. We assume that regular checks for non-equivocation of STRs are performed by other participants and reported on the PR. For instance, other PVs could query the STR of a particular PV and compare it with the STRs maintained by the PR. We provide more details on how

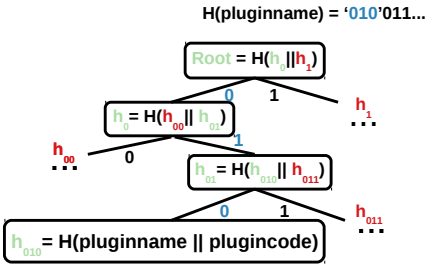


Figure 5: Proof of consistency. Red values forms the *authentication path*, and green values are re-computed to verify that it matches the root.

the STRs are stored by the PR in Appendix B.1. Any PQUIC peer may eventually learn about an equivocation on any received STR.

Secure human-readable names for plugins. When a PQUIC peer wants to use a plugin, it does not need to reason about developers identity or plugin validity. The name is globally unique, human-readable and unequivocally matches a plugin.

Detection of spurious plugins. If a PV injects a spurious binding, the developer owning the plugin name will be able to detect this and alert PQUIC peers through the PR. A peer may be abused before the detection happens. However, the end-user will eventually know which PV faulted.

3.3 System Overview

PVs retrieve plugins from the PR. They build a Merkle Prefix Tree at each epoch containing all successfully validated plugins. Each path to the leaf of this binary tree represents a prefix, and *bindings* are placed in leaves depending on the truncated bits of $H(pluginname)$. For instance, in a 3-depth tree, the prefix of the left-most leaf is '000'. Empty leaves are replaced by a large constant value c chosen by the PV. Interior nodes are hashed as $H(h_l||h_r)$ with h_l (resp. h_r) being the hash value of the left (resp. right) child subtree.

Leaf nodes contain one or more *bindings*. Several bindings may be located at a same leaf node when the hash prefix of different names ($H(pluginname)$) collides. In this case, the leaf node contains a linked list of bindings. Under the assumption of uniform hashing, we can engineer the depth of the tree such that a collision happens with low probability, depending on the number of plugins within the PR. Without collision, the leaf node value is defined as:

$$h_{leaf} = H(binding)$$

If there is any collision, the leaf node value concatenates the bindings $i, j, \ldots$ as follows:

$$h_{leaf} = H(H(binding_i)||H(binding_j)||\ldots)$$

After having updated its Merkle tree, the PV digitally signs its root and publishes the STR to the PR where its public-key information is available for all participants (handling STRs is further described in Appendix B.1).

The tree computation is inspired by CONIKS [67], but our construction differs so that bindings are located in the tree depending on the hash value of their plugin names, which makes it impossible for a PV to put two bindings for the same plugin and to stealthily populate one with a malicious code. We provide a detailed security analysis in Appendix B.2.

A PQUIC peer willing to send a plugin over a QUIC connection needs to provide the *authentication paths* from PVs that fulfill this peer's required validation, e.g., $PV_1 \wedge (PV_2 \vee PV_3)$. Obtaining a path only requires sending the name of the plugin to a PV. The PV then computes the *authentication path* in $\Theta(log(n) + \alpha)$, with $\alpha = n/m$ being the load factor defined from n the number of plugins and m the number of available leaves.

The PV then sends back to the PQUIC peer the authentication path for the binding corresponding to the requested plugin name (Figure 5). The hash values of any other bindings that may be part of this leaf are also sent back by the PV. The PQUIC peer then sends the plugin alongside the corresponding authentication paths obtained from a set of validators matching the other peer's required validation. PQUIC peers can preemptively fetch authentication paths for the plugins they intend to use at each epoch.

Finally, the peer receiving the plugin recomputes the root value from the binding and the authentication path to match the STR cached for the current epoch as illustrated on Figure 5. If the computed root matches the STR, then the plugin is accepted. We provide a more detailed efficiency analysis of this scheme in Appendix B.3.

When verifying a binding at a PV, a developer sends the name of the corresponding plugin to this PV. If only a single hashed binding is present in the tree, the developer checks that it match theirs. If multiple hashed bindings are present, i.e. because of a collision, the developer receives their clear text. This allows them to discern whether the collision was due to a prefix collision or the PV added a spurious binding.

If the PV tree does not contain the hashed binding of a given plugin, its developer obtains a proof of absence, i.e., an authentication path to the linked-list without the developer's binding or an authentication path to the constant value c indicating an empty leaf such that it matches the truncated bits of $H(pluginname)$.

3.4 Exchanging QUIC plugins

Given the flexibility of QUIC, it is relatively simple to modify it to support the exchange of plugins. The QUIC connection establishment packets contain QUIC transport parameters such as the maximum number of streams, the maximum packet size or the idle timeout [51]. To enable plugin injection, PQUIC proposes two new QUIC transport parameters: `supported_plugins` and `plugins_to_inject`, both containing an ordered list of protocol plugins identifiers. The first announces the plugins a PQUIC peer can inject locally. These plugins are stored inside its local cache. The second announces the plugins that a PQUIC peer would like to communicate to the other PQUIC peer. Once the QUIC handshake has completed, both peers have a complete view of the available and requested plugins. Then, there can be two outcomes: (*a*) all plugins requested for injection are already available. In this case, they are injected as local plugins, as explained in Section 2, in the order described by the `plugins_to_inject` transport parameter. Otherwise, (*b*) one or more plugins are unavailable locally, they are then transferred as illustrated in Figure 6. In this example, the client announces the support of a monitoring plugin while the server would like to inject a FEC plugin into the client. First, the client announces its required validation formula for missing plugins, here FEC, with the `PLUGIN_VALIDATE` frame. Second, the server responds with authentication paths from PVs that fulfill this formula in a `PLUGIN_PROOF` frame. The requested plugin is then transferred over the plugin stream in `PLUGIN` frames, akin to the QUIC cryptographic stream. When receiving the remote plugin, the client performs the check of the proof of consistency. Upon success, it stores the plugin in its local cache. Remote plugins are not activated for the current connection, but rather offered in subsequent connections as part of the locally available plugins. While PQUIC is capable of injecting plugins at any time (see Section 2), synchronizing their injection between two hosts raises issues that are prevented by this conservative choice.

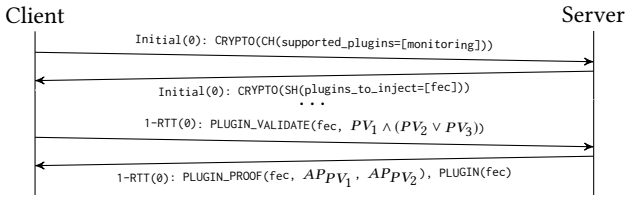


Figure 6: Example flow for the exchange of the FEC plugin.

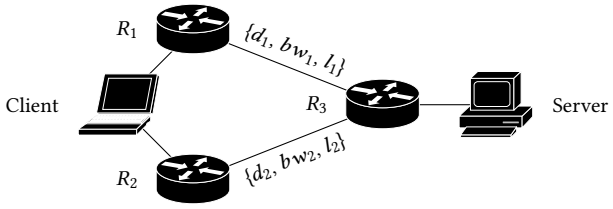


Figure 7: Network topology used for experiments.

Plugin	LoC	Pluglets	Proven terminating	ELF Size	Compressed size
Monitoring	500	14	13	86 kB	27 kB
Datagram	500	11	8	28 kB	25 kB
Multipath	2600	32	29	138 kB	40 kB
FEC	2500	51	37	238 kB	61 kB

Table 2: Statistics for each implemented plugin.

While the exchange mechanisms introduce some overhead, we believe it remains acceptable. The fixed cost of exchanging plugin bytecodes is only present during the first connection, as subsequent ones will take advantage of the PQUIC caching system described in Section 2.5. Furthermore, if the plugin is not mandatory for the use of the application (e.g., adding FEC to protect the data transfer), the plugin exchange does not prevent data from being transmitted over the connection. Indeed, data and plugin streams can be concurrently used thanks to the QUIC frame multiplexing.

4 USE CASES

Protocol plugins can be used to implement various extensions to PQUIC. With less than 100 lines of C code a PQUIC plugin can add the equivalent of Tail Loss Probe in TCP [29], or support for Explicit Congestion Notification [102]. Due to space limitations, this section focuses on more complex use cases that demonstrate the extensibility of PQUIC with four very different protocol plugins.

Our first plugin (Section 4.1) adds new monitoring capabilities to PQUIC. This is a relatively simple plugin that collects statistics by reading state variables. It provides similar features as Web100 [64], the MIB-2 or the `TCP_INFO` socket options for TCP. Our second plugin (Section 4.2) extends PQUIC to support unreliable messages [75]. We use it to implement a VPN service that is similar to the ones using DTLS [70]. This demonstrates the possibility of extending the PQUIC interface proposed to the application. Our third plugin (Section 4.3) adds multipath capabilities [20] to PQUIC. This plugin demonstrates the possibility of using PQUIC over several network paths. Our fourth plugin (Section 4.4) provides a flexible framework that adds support for Forward Erasure Correction to PQUIC. This demonstrates the possibility of building modular plugins including complex computations. Table 2 summarizes the complexity of these plugins by lines of C code, number of pluglets, and size of bytecode.

We evaluate the performance of these four plugins in a lab equipped with Intel Xeon X3440 processors, 16GB of RAM and 1 Gbps NIC, running Linux kernel 4.19 and configured as shown in Figure 7. The links between R_1, R_3 and R_2, R_3 are configured using *NetEm* [41] to add transmission delays and using *HTB* to limit their bandwidth. Losses are generated using a seeded random loss generator attached to the routers. This allows fair performance comparisons as the same loss pattern is applied when an experiment is replayed. One-way delay d is expressed in milliseconds, bandwidth bw in Mbits and uniform losses l as a percentage of packets transmitted.

To evaluate the plugins in a wide range of environments, we use the experimental design approach [28]. We define ranges on the possible values for the parameters presented and use the WSP algorithm [88] to broadly sample this parameter space into 139 points. Each parameter combination is run 9 times and the median run is reported. This mitigates a possible bias in parameter selection and gives a general confidence in the experiment results. Unless otherwise noted, we use the parameters range $\{d_1 \in [2.5, 25]\text{msec}, bw_1 \in [5, 50]\text{Mbps}, l_1 = 0, d_2 = d_1, bw_2 = bw_1, l_2 = l_1\}$, and assume that both links have similar bandwidth, delay and loss characteristics. Note that when links are lossless, congestion-induced losses can still be observed due to the limited bandwidth and router buffers. All plugins are cached on both client and server.

4.1 Monitoring PQUIC

QUIC is an encrypted protocol that leaves only a small fraction of its headers in cleartext, preventing third-party flow performance analysis for events such as losses and retransmissions [22, 90]. The QUIC working group has reached consensus on disclosing a single bit in the header, called the *Spin Bit* [96], to enable third parties to measure the RTT of a connection, but no solution is proposed for other metrics such as retransmissions, losses, etc.

Design. Our monitoring plugin adds passive pluglets, i.e. pluglets that hook to *pre* and *post* anchors, to several protocol operations in PQUIC to record the performance indicators (PI) such as the bytes/packets sent/received, lost, received out-of-order, etc. A set of PIs are recorded during the handshake and a second are updated while the connection is active. Our plugin exports these PIs to a local daemon that sends them over UDP to a collector. A similar approach could be used to feed an SNMP agent that maintains a QUIC MIB.

Implementation. Each pluglet is hooked to a particular step of the connection. Upon the arrival or transmission of a packet, the relevant PIs are updated. Once a set of PI is complete, either because the connection was established or terminated, it is sent to the local daemon.

4.2 QUIC VPN

Given the security of QUIC, it could be interesting to use it as a replacement for TLS-based VPNs. Experience with such VPNs indicate that DTLS [70] provides a better user experience that using TLS over TCP because of the well-known TCP-over-TCP performance problems [47, 63].

We leverage the flexibility of PQUIC with a plugin that supports unreliable messages in addition to the reliable QUIC bytestreams. This plugin supports a new `DATAGRAM` frame [75] that only maintains the transported data boundaries but not transmission order nor reliable delivery.

Design. We implement a simple VPN that captures raw IP packets and passes them to PQUIC. We chose to encapsulate the user traffic at the IP level because this allows handling of all types of packets.

We modified the PQUIC client and server to carry IP datagrams between Linux tunnel interfaces. This was done by adding 70 lines, modifying 10 lines and removing 200 lines in the `picoquic` sample applications. This VPN application leverages a key point of our design presented in Section 2.4, i.e. a plugin is also able to extend the API exposed to the application. Our VPN application reads IP datagrams from the tunnel interface and writes them to the message socket exposed by the Datagram plugin. It reads the received IP datagrams from the message socket and writes them to the tunnel interface. This simplistic approach and the default PQUIC parameters lead to an overhead of 44 bytes per conveyed IP datagram when used over IPv4. This overhead could be minimized by using techniques such as header compression, but these are outside the scope of this paper.

Evaluation. We evaluate the overhead introduced by our VPN in terms of Download Completion Time (DCT) for a single file transfer using $\text{TCP}_{\text{Cubic}}$. We compare the file transfer times inside and outside the VPN tunnel for different file sizes. We only use the top path of Figure 7 and explore the default parameters range. We used a 1400-byte MTU inside the tunnel and 1500 outside. The VPN overhead of 44 bytes per conveyed packet translates into a bound on the DCT ratio of 1.031. Figure 8 illustrates the CDF of the DCT ratio obtained. For short files, the DCT is mostly below the bound computed. When transferring small amounts of data, the network path buffers can accommodate the VPN encapsulation overhead and only the VPN processing time affects the DCT. For longer files, the DCT ratio is more stable and mainly caused by the per-packet overhead.

4.3 Multipath QUIC

A key design point of QUIC is that it includes connection IDs in the packet's public header [51]. Unlike TCP or UDP, a QUIC connection is not bound to a given 4-tuple (IP_{src}, IP_{dst}, port_{src},

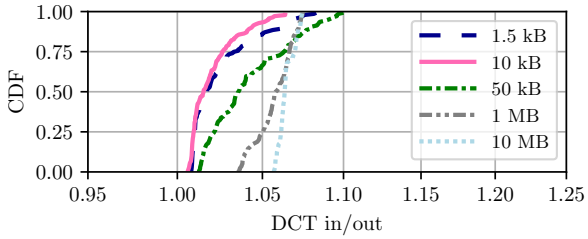


Figure 8: DCT ratio of TCP in and outside a client-server PQUIC tunnel.

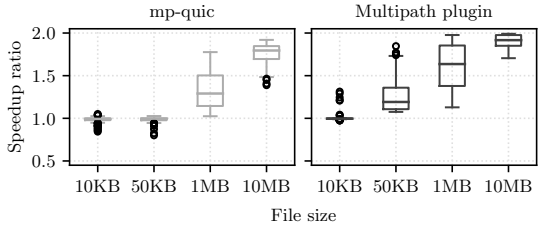


Figure 9: Over two symmetric network paths, multipath tends to complete transfers twice faster than single-path.

$port_{dst}$) but to these IDs. This makes QUIC resilient to events such as NAT rebinding. This also opens the possibility of letting a single QUIC connection use multiple paths [19, 99]. Adding multipath capabilities to a transport protocol creates challenges as shown by Multipath TCP [32, 80].

Design & Implementation. We implement a PQUIC plugin that provides basic multipath capabilities similar to proposed Multipath QUIC extensions [20]. Our plugin supports the exchange of path connection IDs and host addresses. It then associates a path ID between each pair of host addresses. Once the connection has been established, packets are scheduled in a round-robin manner between available paths and it uses a new ACK frame to acknowledge received packets with path-specific packet numbers. We also implement a packet scheduler sending packets on the path having the lowest RTT to mimic Multipath TCP [80], but do not evaluate it due to space constraints.

Evaluation. To assess the performance of our plugin, we consider the network scenario shown in Figure 7 and use the two paths with the default parameter range. For both single path and multipath settings, we record the time between a GET request issued by the client and the reception of the last byte of the server response. We then compute the ratio of the single path completion time over the multipath one to obtain the speedup ratio. We observe its evolution with the size of the requested file and compare it with the ratio obtained using the original implementation of `mp-quic` [19]. Figure 9 shows that with small files, there is little gain in using two paths. This is not surprising since each path is constrained by its initial congestion window. Notice that the initial path window of `mp-quic` (32 kB), inherited from `quic-go` [13], is twice the default one of PQUIC (16 kB). This explains the small speedup gain of our plugin on 50 KB files. With larger files, both `mp-quic` and our plugin efficiently use the two available paths. The speedup ratio of both the native `mp-quic` implementation and our multipath plugin tends to reach 2 with 10 MB files.

4.4 Forward Erasure Correction QUIC

The QUIC protocol provides reliable delivery by using classical retransmission-based mechanisms. These techniques are coupled with congestion control and usually assume that losses are mainly caused by congestion. This assumption is not always true and there are situations (e.g. wireless networks) where losses are caused by other factors than congestion [85]. Researchers have proposed a variety of recovery techniques that transmit redundant data to enable the receiver to recover from packet losses without needing retransmissions [11, 82, 92]. These are computationally intensive as packets need to be encoded by the sender and decoded by the receiver.

We leverage the flexibility of PQUIC to implement a flexible framework inspired by QUIC-FEC, a recent design and implementation of FEC in QUIC [69]. Our plugin sends redundancy (Repair Symbols) to enable PQUIC receivers to recover lost QUIC packets without waiting for retransmissions and therefore meeting the delay constraints. This design goes beyond the naive XOR-based solution that was tested in Google-QUIC [40] before being removed from the protocol since it led to negative experimental results [54].

Design & Implementation. Our FEC Plugin allows plugging different FEC Frameworks, currently providing both block and sliding-window-based codes. It adds several protocol operations and can be extended to change the FEC Framework (block or sliding-window-based), the erasure-correcting code (ECC) used and which part of the data should be protected. Due to space limitations we only consider the sliding-window-based encoding in this section. Our frameworks attach passive pluglets to protocol operations that send and receive packets. Each packet containing QUIC stream frames that needs to be sent will be protected by sending Repair Symbols (RS) later. On the receiver-side, the FEC-protected packets are added to their respective FEC encoding window. The missing packets of a window are recovered upon reception of RS.

Our FEC plugin adds two new types of frames, the *FEC RS* frame and the *FEC ID* frame. The first contains a RS while the latter identifies the packets that are FEC-protected and their corresponding window.

Using the new protocol operations added by our framework, we provide two pluglets implementing different ECCs. The first one implements a XOR code similar to the one proposed by Google [40]. A XOR RS is generated by doing an XOR operation between all the packets in the encoding window. It is thus simple to compute and can be used on lightweight clients. It can however only recover from the loss of a single packet, as only one Repair Symbol can be generated from the same encoding window. The second code is a Random Linear Code (RLC) [33]. A RLC RS is generated by computing a linear combination with randomly chosen coefficients between all the packets of the encoding window. Conversely to the XOR code, this code can generate multiple Repair Symbols for the same set of Source Symbols. It can thus handle the loss of more than one packet per FEC window. However, its recovery process, i.e. solving a system of linear equations whose unknowns are the lost packets, is more computationally intensive. Other erasure-correcting codes could easily be added by implementing new pluglets.

Our FEC plugin also includes other protocol operations to customize its behavior. One can choose between protecting the entire

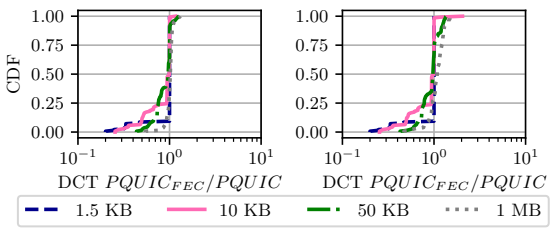


Figure 10: DCT ratio between PQUIC with and without the FEC plugin. Left: only the end of stream is protected. Right: The whole stream is protected.

data transfer or only the last packets of a stream by using different pluglets. The first mode can reduce the data delivery delay of real-time applications while the second mode can reduce the Download Completion Time (DCT) of a bulk data transfer by recovering from tail losses without spending too much bandwidth for sending the Repair Symbols. The number of FEC pluglets shown in Table 2 sums the number of pluglets of the window-based FEC Framework with both XOR and RLC ECCs and the two transmission modes described above.

Evaluation. In this section, we evaluate FEC in the In-Flight Communications use-case, where the delay and losses are important and FEC might improve the performances. This scenario explores the parameters range $\{d_1 \in [100, 400], bw_1 \in [0.3, 10], l_1 \in [1, 8]\}$, based on the experimental results of Rula et al. [86]. Figure 10 compares the performance of downloading a regular HTTP object with and without the FEC plugin. On the left graph only the end of the data stream is protected (i.e., some Repair Symbols are only sent at the end of the connection), while on the right graph the whole data stream is protected, by sending 5 Repair Symbols every 25 Source Symbols. As we can see, there is a benefit in only protecting the end of the stream for larger file transfers. Protecting the whole transfer requires more bandwidth which negatively impacts the DCT. Packets lost in the middle of the transfer can be easily recovered through retransmissions without a significant impact on the DCT.

Only sending redundancy at the end of the transfer considerably lowers the negative impact on the bandwidth needed during the connection and still reduces the DCT ratio when losses occur on the last packets of the stream.

4.5 QUIC Multipath VPN

As demonstrated in the previous sections, our PQUIC plugins provide a range of extensions to the protocol. Furthermore, given the isolation provided by PQUIC, it is possible to load different plugins on a given PQUIC implementation provided that they do not replace the same protocol operation. All the plugins discussed in this section have orthogonal features. They can be combined to provide more advanced services. As an example of the flexibility of our approach, we demonstrate how the multipath plugin (see Section 4.3) can be injected together with the Datagram plugin (see Section 4.2).

We evaluate the performance of combining these two plugins in a setting akin to the evaluation of the Datagram plugin but with the two paths of Figure 7 and by exploring the default parameters range. We measure the ratio of Download Completion Times (DCT)

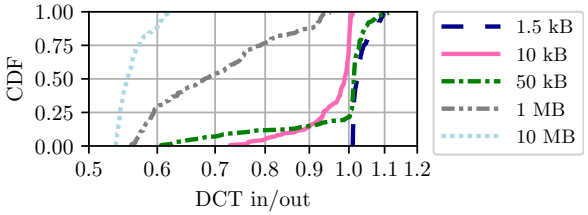


Figure 11: DCT ratio of TCP in and outside a multipath client-server tunnel.

Plugin	$\tilde{x}$ Goodput	$\sigma/\tilde{x}$	$\tilde{x}$ Load Time
PQUIC, no plugin	1104.2 Mbps	3.8%	0.0 ms
Monitoring (a)	1037.3 Mbps	4.6%	6.35 ms
Multipath 1-path (b)	756.6 Mbps	3.1%	8.28 ms
a and b	714.2 Mbps	4.4%	13.00 ms
FEC XOR EOS	661.5 Mbps	3.2%	11.70 ms
FEC RLC EOS	648.0 Mbps	4.5%	11.22 ms
FEC XOR	516.6 Mbps	3.2%	11.71 ms
FEC RLC	187.4 Mbps	1.1%	11.21 ms

Table 3: Benchmarking plugins over 10Gbps links (20 runs).

when running a single $\text{TCP}_{\text{Cubic}}$ file transfer inside and outside the multipath VPN tunnel for different file sizes.

As expected, we do not observe any benefit in using multipath for tunneling a short TCP transfer. However, as file size grows the benefits of multipath become clear. By spreading the traffic over the two symmetric paths, our combined plugins reach a DCT ratio that tends to 0.55 as illustrated in Figure 11.

4.6 Plugin Overhead

The balance between flexibility and performance is a classical trade-off. Google and the IETF have decided to run QUIC over UDP despite the fact that Google measured that "*QUIC's server CPU-utilization was about 3.5 times higher than TLS/TCP*" [50]. The performance gap between TCP and UDP has since been slightly reduced, but UDP remains slower [18]. As PQUIC delegates the execution of the plugins to the PRE, there is a processing overhead due to the JIT compilation, the runtime verifications performed by our monitor at each memory access, and the utilization of the API to safely access PQUIC state variables.

Executing code in the PRE is less efficient than running native code. With computationally expensive micro-benchmarks that are not shown due to space limitations, we determined that the PRE is two times slower than native code. Such overhead could probably be mitigated by using a more optimized VM. Finally, our get/set API is five times slower compared to direct memory accesses in micro benchmarks.

To observe this performance impact in more bandwidth-intensive environments, we benchmark our PQUIC implementation by measuring the completion time of a 1 GB download between two servers with 10Gbps NICs running two Intel Xeon E5-2640 v3 CPUs. We do not run the benchmarks on the lab setup used in Section 4 to benefit from newer instructions sets, e.g, AES-NI, and to focus on the plugin execution overhead. Note that PQUIC is single-threaded and thus does not benefit from the additional CPU cores.

Table 3 reports the median achieved goodput, its relative variance and the plugin loading time for each plugins. PQUIC achieves

a median goodput of 1104.2 Mbps. This low performance compared to TCP is partly due to the fact that `picoquic`, the implementation on which PQUIC is based, has not been optimized for performance yet. Indeed, as the QUIC specification is still evolving, there is limited interest in deeply optimizing a QUIC implementation at this stage. Comparing PQUIC to the version of `picoquic` we based our work on could allow measuring the impact of adding our approach to a QUIC implementation. However, this is technically challenging, as PQUIC contains also performance improvements and bug fixes to `picoquic`. There is thus no `picoquic` version matching the current state of PQUIC. We could also compare the latest version of `picoquic` with PQUIC. However, `picoquic` has substantially evolved during our work, adding many degrees of freedom that would confuse the performance evaluation.

When evaluating our monitoring plugin, we observe a goodput reduction of 7% which matches the additional 8% CPU instructions executed. We also observed a 10% increase of TLB load misses caused by the context switches between PQUIC code and the PREs. Given that the monitoring code complexity is low, these results illustrate the overhead of adding several pluglets within the critical path.

Benchmarking the multipath plugin over a single path achieves a goodput of 756.6 Mbps. Several factors explain this reduction. First, the acknowledgments are created by the plugin using `MP_ACK` frames, which constitutes a significant part of the client execution time. Second, the multipath plugin provides a path manager and a path-aware frame scheduler. This adds several new protocol operations into the packet processing and creation loop.

Combining both the monitoring and multipath plugin results in a 9% reduction compared to multipath only. This demonstrate that plugins with orthogonal features are efficiently combined using PQUIC.

The FEC plugin achieves lower rates, but two major factors affect this result. First, the specific FEC scheme used impacts the computational cost, as RLC is more expensive than XOR. RLC complexity is proportional to the size of the sending window. Second, FEC introduces a bandwidth cost by generating repair symbols over the network with the code rate 5/6, limiting the achieved goodput. Restricting their generation to the end of the stream (EOS) reduces this cost. Finally, because protocol plugins do not change the extensions specifications they implement, a peer could include them natively to improve performance.

Inserting plugins takes time, as described in the last column of Table 3. This time is proportional to the number of inserted pluglets and their complexity. The instantiation of PREs (between 4 and 7 ms) is the major contributor to this loading time. Note that this overhead is only present when there is no cached plugin available. If the host previously loaded the plugin in a completed connection, it can reuse its PREs as described in Section 2.5 to load the plugin in less than 30 μs.

Plugins can be exchanged between PQUIC peers over the network as described in Section 3.4. To limit the exchange overhead, we rely on a ZIP compression scheme to transfer plugins. We take advantage from the fact that pluglets from a given plugin can contain duplicate code from common functions. Table 2 shows that compressing the plugin reduces the exchanged overhead compared to a plain exchange of the ELF files. Assuming the plugin exchange starts at the beginning of the connection with an initial congestion window of 16 KB, it lasts 2-3 RTTs on average.

5 VALIDATING PLUGINS

As explained in Section 3, PQUIC peers can request proofs of the validity of protocol plugins when receiving them over a QUIC connection. This validation is carried out by the Plugin Validators described in Section 3. These validators can apply a range of techniques, from manual inspection, privacy checks [27, 60], to fuzzing [73] or using formal methods to validate the plugins submitted by developers. Formal methods are attractive because they enable validators to provide strong proofs for network protocols [5, 6, 12].

A very important property for any code is its (correct) termination. If a protocol plugin would be stuck in an infinite loop with some specific input, then it would obviously be unsafe to use it in a PQUIC implementation. To demonstrate the possibility of using formal techniques to validate protocol plugins, we have used the state-of-the-art T2 [9, 16] automated termination checker. This tool checks termination of programs written in the T2 language implementing a counter example-guided abstraction refinement procedure. This procedure builds on the seminal works on transition invariants (used to characterize termination) [77] and predicate abstraction (used to simplify the representation) [78] to build a proof of termination, or to disprove it. It is a counter-example based approach starting from an abstracted version of the system, and refining it until either no counter example to termination can be found, or there is a clear proof that the system does not terminate. The procedure largely depends on the abstraction built by the tool and may not terminate. T2 has been extended to handle a large fragment of Computational Tree Logic (CTL) [14, 15]. T2 concepts are used in most existing verification tools for termination such as Ultimate [98]. Verifying CTL reduces to an extended termination proof of the program combined with CTL information in states. Therefore, we focus here on the termination property.

Using the appropriate tools [57, 62], we checked the termination of our pluglets by compiling their C source code to T2 programs. We proved the termination of nearly all the pluglets of the monitoring plugin. The T2 prover assumes the termination of external functions, i.e., functions of the PQUIC implementation available through the PRE. We also proved the termination of most of the pluglets of our complex plugins, such as FEC (including the XOR ECC) and multipath, as reported in Table 2. To obtain those proofs, we had to slightly modify the source code of some pluglets to ease the proof process. For example, we added an explicit size to null-terminated linked lists and used it to bound the loops iterating over the lists. Three of the multipath pluglets could not be proven due to their complexity. Since T2 can export its termination proofs in files, these could be attached to the plugins to be the proof-carrying code proposed by Necula [72]. However, given the size and complexity of these proofs, it is unreasonable to expect a PQUIC implementation to download and process them when loading plugins.

6 RELATED WORKS

Improving the flexibility of networks is a topic widely studied in the literature. In the late nineties, active networks [93, 94] were proposed as a solution to bring innovation inside the network.

Various techniques were proposed to place bytecode inside packets so that routers could execute it while forwarding them. PLAN [43], ANTS [103] and router plugins [23] are examples of such active techniques. Interest in active networks slowly decreased in the early 2000s [10], but Software Defined Networks [66] and P4 [7] can be considered as some of their successors.

Most of the work on active networks focused on the network layer and only a few researchers addressed the extensibility of the transport protocols. CTP [104] is transport protocol that is composed of various micro-protocols that can be dynamically combined at runtime through the Cactus [44] system. STP [74] enables the utilization of code written in Cyclone [53] to extend a TCP implementation. icTCP [38] exposes TCP state information and control to user-space applications to enable them to implement various extensions. To our knowledge, these techniques have not been deployed. We believe our approach can be deployed at a large scale as it relies on QUIC which prevents middlebox interference and enables a safe exchange of protocol plugins. Our plugins go beyond the extensions proposed for STP and icTCP. CCP [71] provides a framework to write congestion control schemes in transport protocols in a generic way. Although we did not describe it in this paper, a new congestion controller could easily be implemented as a protocol plugin.

To deploy protocol plugins, we design a secure plugin management system that bares similarities to Certificate Transparency [59]. Our construction has the major difference that plugin developers do not have to scan the entire tree in order to detect spurious plugins linked to their owned name, but only the branches in which their plugins lie.

Our Plugin Runtime Environment is based on a simple user-space implementation of the eBPF VM [30, 49]. Other execution environments that provide built-in memory checks such as CHERI-MIPS [105] or WebAssembly [39] could be a valid alternative to our PRE. Evaluating their relevance in the protocol plugin context is part of our future work.

7 DISCUSSION

Extending the behavior of client-side protocol implementations is difficult. First, deploying client updates can take several months or even years. Second, it remains unpractical to tune a protocol implementation when connections require very different services. Currently, experimental QUIC extensions such as MP-QUIC [19] and QUIC-FEC [69] are implemented as code-source forks. Updating the base QUIC protocol and combining these extensions impose a significant engineering and maintenance burden, which is currently only affordable by large Internet companies. We envision PQUIC implementations to be both simple and stable, providing connection-specific extensions through plugins. PQUIC could enable developer to focus on one implementation interface while still supporting very different implementation internal architectures, such as zero-copy and partial hardware offload.

Through its plugins, PQUIC provides quick deployments of extension prototypes. Still, the benefits of plugins for complex standardized extensions do not fade. For instance, while the FEC specification [83] describes its wire format, the algorithms defining the correction scheme mainly depend on the application. Whereas server implementations would likely provide built-in support of the FEC extension for better performances, PQUIC provides more flexible tuning at client-side than a simple on/off extension switch. Similarly, for Multipath QUIC, plugins remain desirable to let the server push algorithms to the client that are tailored to the application needs, such as a specific path scheduler.

8 CONCLUSIONS AND FUTURE WORK

Extensibility is a key requirement for many protocol designs. We leverage the unique features of QUIC to propose a new extensibility model that we call Pluginized QUIC (PQUIC). A PQUIC implementation is composed of a set of *protocol operations* which can be enriched or replaced by *protocol plugins*. These plugins are bytecodes executed by a Protocol Runtime Environment that ensures their safety and portability. The plugins can be dynamically loaded by an application that uses PQUIC or received from the remote host thanks to our secure plugin management system. We demonstrate the benefits of this approach by implementing very different protocol plugins that add monitoring, multipath, VPN and Forward Erasure Correction capabilities to QUIC.

This new protocol extensibility model opens several directions for future work. First, a similar approach could be used for other networking protocols in both the data plane and the control plane. Second, new techniques ensuring the implementation conformance to protocol specifications could be explored. These could leverage the PQUIC interface to assess that a plugin composition is correct. A third direction would be to revisit how we design protocols robustness [84].

Another direction would be to develop new verification techniques adapted to pluginized protocols. While our PRE detects and prevents the execution of a range of incorrect and malicious programs, there remain areas of improvements in the domain of static eBPF verification techniques.

Finally, PQUIC could be the starting point for the next version of QUIC. This would require the IETF to also specify protocol operations to ensure the inter-operability of plugins among different implementations. To achieve this, one must identify the minimal core protocol operations required. This set should be simple enough to allow very different implementations, having possibly very specific internal architectures such as zero-copy support, to inter-operate. Instead of adding more and more features to a monolithic implementation, developers could leverage the inherent extensibility of a pluginized protocol to develop a simple set of kernel features that are easy to extend.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Ankit Singla, for their valuable comments. We also thank Christian Huitema, as well as the IO Visor, clang and T2 teams and all the developers of the open-source softwares without which this work wouldn't be possible. This work is partially supported by funding from the Walloon Government (DGO6) within the MQUIC project, and the Digitrans project (convention number 7618).

REFERENCES

[1] Nadav Amit and Michael Wei. 2018. The design and implementation of hyperupcalls. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 97–112.

[2] Apple. 2018. Improving Network Reliability Using Multipath TCP. (2018). https://developer.apple.com/documentation/foundation/urlsessionconfiguration/improving_network_reliability_using_multipath_tcp.

[3] Praveen Balasubramanian. 2018. Usage for timestamp options in the wild. (Sept. 2018). https://mailarchive.ietf.org/arch/legacy/msg/tcpm/11522.

[4] Andrew Begel, Steven McCanne, and Susan L. Graham. 1999. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. *ACM SIGCOMM Computer Communication Review* 29, 4 (1999), 123–134.

[5] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 2017. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 483–502.

[6] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. 2005. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *ACM SIGCOMM Computer Communication Review*, Vol. 35. ACM, 265–276.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[8] Lawrence Brakmo. 2017. TCP-BPF: Programmatically tuning TCP behavior through BPF. *NetDev 2.2* (2017).

[9] Marc Brockschmidt and Heidy Khlaaf. 2019. *T2 Temporal Prover*. http://mmjb.github.io/T2/.

[10] Ken Calvert. 2006. Reflections on network architecture: an active networking perspective. *ACM SIGCOMM Computer Communication Review* 36, 2 (2006), 27–30.

[11] Georg Carle and Ernst W Biersack. 1997. Survey of error recovery techniques for IP-based audio-visual multicast applications. *IEEE Network* 11, 6 (1997), 24–36.

[12] Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Tasiran Serdar, Aaron Tomb, and Eddy Westbrook. 2018. Continuous formal verification of Amazon s2n. In *International Conference on Computer Aided Verification*. Springer, 430–446.

[13] Lucas Clemente and Marten Seemann. 2018. *quic-go*. Source code. https://github.com/lucas-clemente/quic-go.

[14] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. 2007. Proving that programs eventually do something good. *ACM SIGPLAN Notices* 42, 1 (2007), 265–276.

[15] Byron Cook, Eric Koskinen, and Moshe Vardi. 2011. Temporal property verification as a program analysis task. In *International Conference on Computer Aided Verification*. Springer, 333–348.

[16] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. TERMINATOR: beyond safety. In *International Conference on Computer Aided Verification*. Springer, 415–418.

[17] Scott A. Crosby and Dan S. Wallach. 2009. Efficient Data Structures For Tamper-Evident Logging. In *USENIX Security Symposium*. 317–334.

[18] Willem de Bruijn and Eric Dumazet. 2018. Optimizing UDP for content delivery: GSO, pacing and zerocopy. In *Linux Plumbers Conference*.

[19] Quentin De Coninck and Olivier Bonaventure. 2017. Multipath QUIC: Design and Evaluation. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. ACM, 160–166.

[20] Quentin De Coninck and Olivier Bonaventure. 2018. *Multipath Extension for QUIC*. Internet-Draft draft-deconinck-quic-multipath-01. Internet Engineering Task Force.

[21] Quentin De Coninck and Olivier Bonaventure. 2019. *The Case for Protocol Plugins*. Technical Report. https://hdl.handle.net/2078.1/216493

[22] Piet De Vaere, Tobias Bühler, Mirja Kühlewind, and Brian Trammell. 2018. Three Bits Suffice: Explicit Support for Passive Measurement of Internet Latency in QUIC and TCP. In *Proceedings of the Internet Measurement Conference 2018*. ACM, 22–28.

[23] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. 1998. Router plugins: A software architecture for next generation routers. *ACM SIGCOMM Computer Communication Review* 28, 4 (1998), 229–240.

[24] Martin Duke, Robert Braden, Wesley M. Eddy, Ethan Blanton, and Alexander Zimmermann. 2015. A Roadmap for Transmission Control Protocol (TCP) Specification Documents. RFC7414. (Feb. 2015), 57 pages.

[25] Tom Dunigan, Matt Mathis, and Brian Tierney. 2002. A TCP tuning daemon. In *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE, 1–16.

[26] Jake Edge. 2015. A seccomp overview. *Linux Weekly News* (September 2015). https://old.lwn.net/Articles/656307/.

[27] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting Privacy Leaks in iOS Applications. In *Network and Distributed System Security Symposium (NDSS'11)*. 177–183.

[28] Ronald Aylmer Fisher. 1935. *The design of experiments*. Oliver & Boyd.

[29] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. 2013. Reducing web latency: the virtue of gentle aggression. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 159–170.

[30] Matt Fleming. 2017. A thorough introduction to eBPF. *Linux Weekly News* (December 2017). https://old.lwn.net/Articles/740157/.

[31] Sally Floyd and Van Jacobson. 1995. Link-sharing and resource management models for packet networks. *IEEE/ACM transactions on Networking* 3, 4 (1995), 365–386.

[32] Alan Ford, Costin Raiciu, Mark Handley, and Olivier Bonaventure. 2013. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824. (Jan. 2013), 64 pages. https://www.rfc-editor.org/rfc/rfc6824.txt

[33] Christina Fragouli, Jean-Yves Le Boudec, and Jörg Widmer. 2006. Network coding: an instant primer. *ACM SIGCOMM Computer Communication Review* 36, 1 (2006), 63–68.

[34] Ron Frederick, Stephen L. Casner, Van Jacobson, and Henning Schulzrinne. 1996. RTP: A Transport Protocol for Real-Time Applications. RFC 1889. (Jan. 1996). https://doi.org/10.17487/RFC1889

[35] Kensuke Fukuda. 2011. An analysis of longitudinal TCP passive measurements (short paper). In *International Workshop on Traffic Monitoring and Analysis*. Springer, 29–36.

[36] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. 2010. VMKit: a substrate for managed runtime environments. *ACM Sigplan Notices* 45, 7 (2010), 51–62.

[37] Brendan Gregg. 2015. eBPF: One Small Step. (May 2015). http://www.brendangregg.com/ blog/2015-05-15/ebpf-one-small-step.html.

[38] Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2004. Deploying Safe User-Level Network Services with icTCP. In *OSDI*. 317–332.

[39] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. *ACM SIGPLAN Notices* 52, 6 (2017), 185–200.

[40] Ryan Hamilton, Janardhan Iyengar, Ian Swett, and Alyssa Wilk. 2016. *QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2*. Internet-Draft draft-hamilton-early-deployment-quic-00.

[41] Stephen Hemminger. 2005. Network emulation with NetEm. In *Australia's National Linux Conference*. 18–23.

[42] Benjamin Hesmans, Fabien Duchene, Christoph Paasch, Gregory Detal, and Olivier Bonaventure. 2013. Are TCP extensions middlebox-proof?. In *Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization*. ACM, 37–42.

[43] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. 1998. PLAN: A packet language for active networks. *ACM SIGPLAN Notices* 34, 1 (1998), 86–93.

[44] Matti A. Hiltunen, Richard D. Schlichting, Xiaonan Han, Melvin M. Cardozo, and Rajsekhar Das. 1999. Real-time dependable channels: Customizing QoS attributes for distributed systems. *IEEE Transactions on Parallel and Distributed Systems* 10, 6 (1999), 600–612.

[45] Michio Honda, Felipe Huici, Costin Raiciu, Joao Araujo, and Luigi Rizzo. 2014. Rekindling network protocol innovation with user-level stacks. *ACM SIGCOMM Computer Communication Review* 44, 2 (2014), 52–58.

[46] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. 2011. Is it still possible to extend TCP?. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, 181–194.

[47] Osamu Honda, Hiroyuki Ohsaki, Makoto Imase, Mika Ishizuka, and Junichi Murayama. 2005. Understanding TCP over TCP: effects of TCP tunneling on end-to-end throughput and latency. In *Performance, Quality of Service, and Control of Next-Generation Communication and Sensor Networks III*, Vol. 6011. International Society for Optics and Photonics, 60110H.

[48] Christian Huitema. 2018. *picoquic*. Source code. https://github.com/private-octopus/picoquic.

[49] IO Visor Project. 2018. *Userspace eBPF VM*. Source code. https://github.com/iovisor/ubpf.

[50] Janardhan Iyengar and Ian Swett. 2018. QUIC: Developing and Deploying a TCP Replacement for the Web. In *Netdev 0x12*.

[51] Jana Iyengar and Martin Thomson. 2018. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-14. Work in Progress.

[52] Van Jacobson, Robert Braden, and Dave Borman. 1992. TCP Extensions for High Performance. RFC1323. (May 1992), 37 pages.

[53] Trevor Jim, Gregory Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference, General Track*. 275–288.

[54] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. 2017. Taking a long look at QUIC: an approach for rigorous evaluation of rapidly evolving transport protocols. In *Proceedings of the 2017*

Internet Measurement Conference. ACM, 290–303.
[55] Jim Keniston, Prasanna S. Panchamukhi, and Masami Hiramatsu. 2016. Kernel probes (kprobes). (2016). Documentation provided with the Linux kernel sources.
[56] Ben Kenwright. 2012. Fast Efficient Fixed-Size Memory Pool: No Loops and No Overhead. In *The Third International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking*.
[57] Heidy Khlaaf, Marc Brockschmidt, Stephan Falke, Deepak Kapur, and Carsten Sinz. 2015. *llvm2KITTeL tailored for T2*. Source code. https://github.com/hkhlaaf/llvm2kittel.
[58] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Kulik, Joanna, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC transport protocol: Design and Internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 183–196.
[59] Ben Laurie, Adam Langley, and Emilia Kasper. 2013. Certificate Transparency. RFC 6962. (June 2013), 27 pages. https://www.rfc-editor.org/rfc/rfc6962.txt
[60] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE Press, 280–291.
[61] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java virtual machine specification*. Pearson Education.
[62] LLVM Team. 2019. Clang: a C language family frontend for LLVM. (2019). https://clang.llvm.org/.
[63] Daniel Lukaszewski and Geoffrey Xie. 2017. Multipath transport for virtual private networks. In *10th USENIX Workshop on Cyber Security Experimentation and Test (CSET 17)*. USENIX.
[64] Matt Mathis, John Heffner, and Raghu Reddy. 2003. Web100: extended TCP instrumentation for research, education and diagnosis. *ACM SIGCOMM Computer Communication Review* 33, 3 (2003), 69–79.
[65] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. 1996. TCP Selective Acknowledgment Options. RFC2018. (Oct. 1996), 12 pages.
[66] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* 38, 2 (2008), 69–74.
[67] Marcela Melara, Aaron Blankstein, Joseph Bonneau, Edward Felten, and Michael Freedman. 2015. CONIKS: Bringing Key Transparency to End Users. In *USENIX Security Symposium*, Vol. 2015. 383–398.
[68] Ralph C. Merkle. 1987. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*. Springer, 369–378.
[69] François Michel, Quentin De Coninck, and Olivier Bonaventure. 2019. QUIC-FEC: Bringing the benefits of Forward Erasure Correction to QUIC. *IFIP Networking* (2019).
[70] Nagendra Modadugu and Eric Rescorla. 2004. The Design and Implementation of Datagram TLS. In *Network and Distributed System Security Symposium (NDSS'04)*.
[71] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 30–43.
[72] George C. Necula. 2002. Proof-carrying code. Design and implementation. In *Proof and system-reliability*. Springer, 261–288.
[73] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium (USENIX Security 18)*. 729–743.
[74] Parveen Patel, Andrew Whitaker, David Wetherall, Jay Lepreau, and Tim Stack. 2003. Upgrading transport protocols using untrusted mobile code. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 1–14.
[75] Tommy Pauly, Eric Kinnear, and David Schinazi. 2018. *An Unreliable Datagram Extension to QUIC*. Internet-Draft draft-pauly-quic-datagram-01.
[76] Maxime Piraux, Quentin De Coninck, and Olivier Bonaventure. 2018. Observing the Evolution of QUIC Implementations. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. ACM, 8–14.
[77] Andreas Podelski and Andrey Rybalchenko. 2004. Transition invariants. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004*. IEEE, 32–41.
[78] Andreas Podelski and Andrey Rybalchenko. 2005. Transition predicate abstraction and fair termination. *ACM SIGPLAN Notices* 40, 1 (2005), 132–144.
[79] Jon Postel. 1981. Transmission Control Protocol. RFC793. (Sept. 1981), 91 pages.
[80] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. 2012. How hard can it be? Designing and implementing a deployable multipath TCP. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 399–412.
[81] K. K. Ramakrishnan, Sally Floyd, and David L. Black. 2001. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168. (Sept. 2001), 63 pages. Updated by RFCs 4301, 6040, 8311.
[82] Luigi Rizzo. 1997. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM computer communication review* 27, 2 (1997), 24–36.
[83] Vincent Roca, Ian Swett, and Marie-Jose Montpetit. 2019. *Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC) Schemes for QUIC*. Internet-Draft draft-roca-nwcrg-rlc-fec-scheme-for-quic-01. IETF Secretariat.
[84] Florentin Rochet, Olivier Bonaventure, and Olivier Pereira. 2019. Flexible Anonymous Network. In *12th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2019)*.
[85] John P. Rula, Fabián E. Bustamante, and David R. Choffnes. 2016. When IPs Fly: A Case for Redefining Airline Communication. In *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications*. ACM, 9–14.
[86] John P. Rula, James Newman, Fabián E. Bustamante, Arash Molavi Kakhki, and David Choffnes. 2018. Mile High WiFi: A First Look At In-Flight Internet Connectivity. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1449–1458.
[87] Jan Rüth, Ingmar Poese, Christoph Dietzel, and Oliver Hohlfeld. 2018. A First Look at QUIC in the Wild. In *International Conference on Passive and Active Network Measurement*. Springer, 255–268.
[88] Jenny Santiago, Magalie Claeys-Bruno, and Michelle Sergent. 2012. Construction of space-filling designs using WSP algorithm for high dimensional spaces. *Chemometrics and Intelligent Laboratory Systems* 113 (2012), 26–31.
[89] Madhavapeddi Shreedhar and George Varghese. 1996. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking* 4, 3 (1996), 375–385.
[90] Emile Stephan, Mathilde Cayla, Arnaud Braud, and Fred Fieau. 2017. QUIC Interdomain Troubleshooting. (July 2017). Internet draft, draft-stephan-quic-interdomain-troubleshooting-00.txt, work in progress.
[91] Randall R. Stewart, Qiaobing Xie, Ken Morneault, Chip Sharp, Hanns Juergen Schwarzbauer, Tom Taylor, Ian Rytina, Malleswar Kalla, Lixia Zhang, and Vern Paxson. 2000. Stream Control Transmission Protocol. RFC2960. (Oct. 2000), 134 pages.
[92] Jay Kumar Sundararajan, Devavrat Shah, Muriel Médard, Szymon Jakubczak, Michael Mitzenmacher, and João Barros. 2011. Network coding meets TCP: Theory and implementation. *Proc. IEEE* 99, 3 (2011), 490–512.
[93] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. 1997. A survey of active network research. *IEEE communications Magazine* 35, 1 (1997), 80–86.
[94] David L. Tennenhouse and David J. Wetherall. 1996. Towards an active network architecture. *ACM SIGCOMM Computer Communication Review* 26, 2 (1996), 5–17.
[95] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. 2018. Transparency Logs via Append-only Authenticated Dictionaries. Cryptology ePrint Archive, Report 2018/721. (2018). https://eprint.iacr.org/2018/721.
[96] Brian Trammell and Mirja Kuehlewind. 2018. *The QUIC Latency Spin Bit*. Internet-Draft draft-ietf-quic-spin-exp-01.
[97] Viet Hoang Tran and Olivier Bonaventure. 2019. Beyond socket options: making the Linux TCP stack truly extensible. *IFIP Networking* (2019). http://hdl.handle.net/2078.1/214176
[98] Ultimate Team. 2018. *Ultimate*. Source code. https://github.com/ultimate-pa/ultimate.
[99] Tobias Viernickel, Alexander Froemmgen, Amr Rizk, Boris Koldehofe, and Ralf Steinmetz. 2018. Multipath QUIC: A deployable multipath transport protocol. In *2018 IEEE International Conference on Communications (ICC)*. IEEE, 1–7.
[100] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1994. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review* 27, 5 (1994), 203–216.
[101] Kunshan Wang, Yi Lin, Stephen M. Blackburn, Michael Norrish, and Antony L. Hosking. 2015. Draining the swamp: Micro virtual machines as solid foundation for language development. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[102] Magnus Westerlund. 2018. Proposal for adding ECN support to QUIC. (2018). https://github.com/quicwg/base-drafts/pull/1372.
[103] David J. Wetherall, John V. Guttag, and David L. Tennenhouse. 1998. ANTS: A toolkit for building and dynamically deploying network protocols. In *Open Architectures and Network Programming, 1998 IEEE*. IEEE, 117–129.
[104] Gary Wong, Matti Hiltunen, and Richard Schlichting. 2001. A configurable and extensible transport protocol. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society*. IEEE, 319–328.

[105] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 457–468.

[106] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy*. IEEE, 79–93.

Appendices are supporting materials that have not been peer reviewed.

A RESEARCH REPRODUCIBILITY

PQUIC and all the plugins presented in the paper are available at https://pquic.org. The scripts required to reproduce the results presented are also made available.

B DETAILS ON THE PLUGIN MANAGEMENT SYSTEM

B.1 Storing STRs

Each PV signs a root at each epoch and publishes the root to all participants through the PR which would act as a provider for those roots, or through direct connections. STRs from different epochs should be stored in an append-only log structure, preventing any tampering from the PR and PVs. CONIKS [67] suggests using a hashchain, which makes verification of the data structure to audit to be of linear complexity with the number of epochs. Other solutions with various advantages and inconveniences are possible too, such as a History Tree [17] or append-only authenticated dictionaries [95]. Fundamentally, the performance of the data structure to handle the history of roots does not matter much since an epoch, yet undefined in this work, is assumed to last several hours, which should not make the data structure's scaling an issue. The real issue is to notice the equivocation in a reasonable time. Noticing equivocation requires secure communications with other participants or auditors of the system, which might not be automated.

B.2 Security Analysis

Our threat model assumes that any participant to the distributed system of trust for plugin management may act maliciously. A developer may publish malicious code; a PQUIC peer may want to inject illegitimate code. PVs may give false assertions on the validity of a plugin. The PR may equivocate on the STRs received from PVs. Both PR and PVs may modify the code served, or impersonate the developers. We claim to have designed a transparent system such that part of those issues are efficiently and immediately detected, and the others are eventually detected, leading to a loss of reputation for the culprit. In this security analysis, we focus on explaining the security role of developer and PQUIC lookups, how we obtain human-readable names for plugins and non-equivocation.

B.2.1 The role of the lookups. Lookups to the bindings handled by PVs are performed from different parties for different security guarantees. On the one hand, each developer performs a *developer lookup* once for each of his plugin they own and at each epoch to verify the presence of spurious plugins. On the other hand, PQUIC peers may do a *PQUIC user lookup* to the validator to obtain proofs of consistency in order to inject a plugin to the peer.

For *developer lookup*, the developer sends the name of a plugin to a PV and receives back an authentication path as in Figure 5, plus the clear text for all other bindings from the linked-list located at the same leaf (i.e., the elements pluginname || plugincode). Then, to detect a spurious plugin, the developer performs the following checks: 1) verify in the linked-list of plugins whether any other name is different from his plugin; 2) compute the leaf hash value, as described in Section 3.3 and check that it matches the one received in the authentication path; 3) compute the root of the tree and checks that the root matches the STR of the current epoch. If any of these checks fails, the developer must report an alert. Each developer performing their developer lookup is a necessary condition to obtain the "secure human-readable name for plugins" property that we prove below.

For efficiency reasons, the amount of information in the output of the validator is not the same for PQUIC user lookup. For *PQUIC user lookup* (which returns a proof of consistency), the difference with a *developer lookup* only lays down in the data sent back from the PV. The PV sends back the requested binding in clear text but does not for any other binding within the linked-list. For any other binding located at the same leaf, the PV sends back its hash value to reduce the bandwidth consumption. Consequently of this bandwidth optimization, the secure human-readable property requires the developer to correctly perform his check and report its validity at the PR (if multiple bindings are located at the leaf). Yet, one remains to prove that there exist only one valid authentication path in the tree for a given plugin (see below).

B.2.2 Human-readable names for plugins. The following theorems help us to prove that our system offers secure human readable names for plugins.

Theorem B.1. *Under the assumption that* H *is a uniformly random function, there exists only one valid authentication path in the tree for a given plugin.*

Proof. Let $RO : \{0,1\}^* \rightarrow \{0,1\}^n$ an oracle access to an uniformly random function. Let $\mathcal{A}$ an adversary trying to compute an authentication path for the binding $m' = pluginname||plugincode^{\mathcal{A}}$. To provide a valid but fake authentication path to the leaf located at the truncated bits of $\mathsf{H}(pluginname)$, $\mathcal{A}$ has to find m' given m such that:

$$\mathsf{H}(m||m') = \mathsf{H}(m'||m) \tag{1}$$

Assuming $\mathsf{H}(x) := RO(x)$, $\mathcal{A}$ has to perform q queries, each of probability $\frac{1}{2^n}$ to verify equation 1. By the union bound, the adversary succeeds with probability $\frac{q}{2^n} = \mathsf{negl}(n)$. □

Theorem B.2. *Under the assumption that* H *is a collision-resistant hash function, c a constant value representing an empty leaf different for each PV and $n < m$ with n the number of plugins and m the number of leaves, then $root_i$ is unique for PV i.*

Proof. Let TR a function computing the Tree Root from successive applications of H. If $\mathsf{H} : \{0,1\}^* \rightarrow \{0,1\}^n$, then the probability for $root_i$ and $root_j$ to collide is $\frac{1}{2^n}$. By the union bound, we have

the probability that any collision happens for the root of PVs to be $\frac{q^2}{2^n}$ for q attempts.

Consequently, we can claim that the success for an adversary $\mathcal{A}$ having access to a random oracle such that $TR(x) := RO(x)$ is bounded by:

$$\Pr\left[\mathcal{A}^{RO(\cdot)} \rightarrow (x, x'), RO(x) = RO(x')\right] \leq \frac{q^2}{2^n} = \mathsf{negl}(n)$$

□

Under the assumption that developers perform lookups, and because it exists only one valid authentication path in the tree for a given plugin, and because each root is unique, a plugin name matches unequivocally a legitimate bytecode with the security properties offered by a PV.

B.2.3 Non-Equivocation. Consider an attack where a PV wants a particular PQUIC user to plug a malicious plugin. To achieve that, the PV needs to add this plugin to his tree. However, doing so exposes the malicious act to the developer checking the validity of the plugins linked to his namespace. To avoid this, the PV may build a second Merkle Tree and expose the correct one when the developer performs a lookup and the malicious one for the PQUIC user. However, it is computationally infeasible to build two different trees that hash to the same root. Therefore, under the assumption that PQUIC users can eventually compare their root value with one of the other participants or independent auditors, our system provides non-equivocation. That is, PVs are not able to maintain different Merkle Trees (hence different STRs) without being eventually pilloried.

B.3 Efficiency Analysis

Apart from offering transparency and achieving the security goals described in Section 3.2, our plugin management system has to provide good performance for PQUIC users as we want the plugins to be loaded as fast as possible.

When a plugin is not in the PQUIC cache, we have to verify the proof of consistency received from the peer. This verification has a complexity of $\Theta(log(n) + \alpha)$ to recompute the root value and compare it to the known STR, with n the number of plugins handled by the PV and α the load factor. To have a better intuition of the CPU cost, we may compare the efficiency of the proof of consistency check with traditional signature schemes that are usually considered for certification. Let $Vrfy$ be the signature verification of a signature scheme. In theory, the computational cost of checking the proof of consistency ($O(d)$ applications of a hash function with d the depth of the tree) is faster by a factor of

$$\frac{time(Vrfy(H(m)))}{time(H(m)) + d * time(H(h_0||h_1))}$$

than the verification of an authoritative signature on message m with h_0, h_1 as the hash output of both subtrees, for a similar security level (e.g., using SHA256 and ECDSA over $\mathbb{F}_{256}$). Given that $Vrfy$ must hash the message m, which is in our context the whole binding $m = pluginname||plugincode$, and given that the time to compute H scales linearly with the size of the input, we have

$$time(H(m)) >> d * time(H(h_0||h_1))$$

since $size(m) >> d * size(h_0||h_1)$. The improvement factor on traditional signature schemes can then be written as:

$$\approx \frac{time(H(m)) + time(Vrfy)}{time(H(m))}$$

The communication cost between participants scales logarithmically with the number of plugins handled by a PV, and the information can be cached. More precisely, the bandwidth consumption for the proof of consistency has a complexity of $\Theta(\lambda(log(n) + \alpha))$, with λ the size of H's output. At the PV, the binary tree can be computed within a few seconds for millions of entries [67]. Knowing that the Merkle Tree must be recomputed at each epoch (which can be hours or days, and must be decided by the system), the time required by a PV to build its Merkle Tree is negligible.