UCLouvain
Institute of Information and Communication Technologies, Electronics and Applied Mathematics

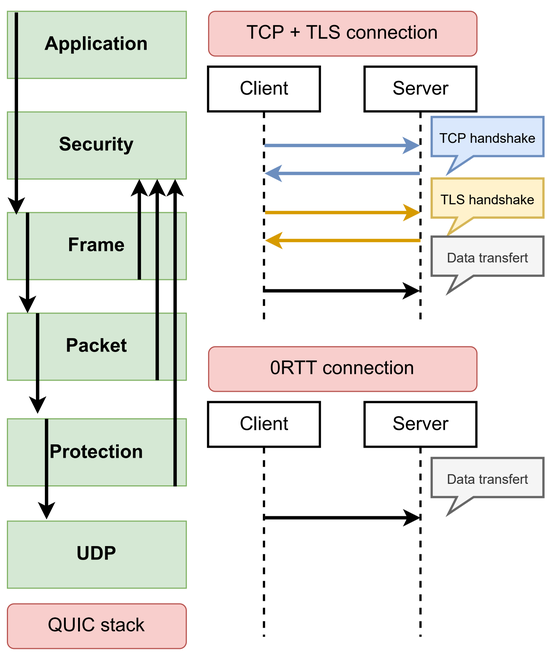
icteam

Software Testing meets Formal Verification for the Good

Tom Rousseaux, Christophe Crochet, Axel Legay

QUIC

QUIC is a new transport protocol intended for widespread use on the Internet. Built over UDP, it is designed to replace the entire TCP/TLS/HTTP stack while combining the benefits of TCP and TLS. It is described in the RFC9000 and specifies some innovative features as 0-RTT connections, multiplexing, connection migration, etc.



Even if QUIC implementations are young, the protocol is used in production.

Implementation	Language	SLOC	Company	Version
picoquic picotls	C	84k	Private Octopus H2O	ad23e6c 47327f8
lsquic boringssl	C	129k	LiteSpeed Tech. Google	v2.29.4 a2278d4
quic-go	Go	73k	-	v0.20.0
quinn	Rust	41k	-	0.7.0
aioquic	Python	19k	-	0.9.3
quiche	Rust	58k	Cloudflare	0.7.0
quant	C	18k	NetApp	29
mvfst	C++	105k	Facebook	36111c1

Verification

To verify implementations, a common approach, called *interoperability testing*, is to manually generate sets of tests and then to compare implementations behaviours. Albeit such approach sounds appealing, it is limited by the capacity to manually produce interesting test suites from the requirements and it compares implementations with each other, not with the specifications.
Another approach is to produce a mathematical model for the protocol and its requirements, and then use formal verification to automatically assess correctness of the implementations.

Formal Methods

Formal methods are sound and precise. But, they suffer from the **state explosion problem** which would occur due to QUIC implementations sizes.

"Verifying QUIC implementations using Ivy"

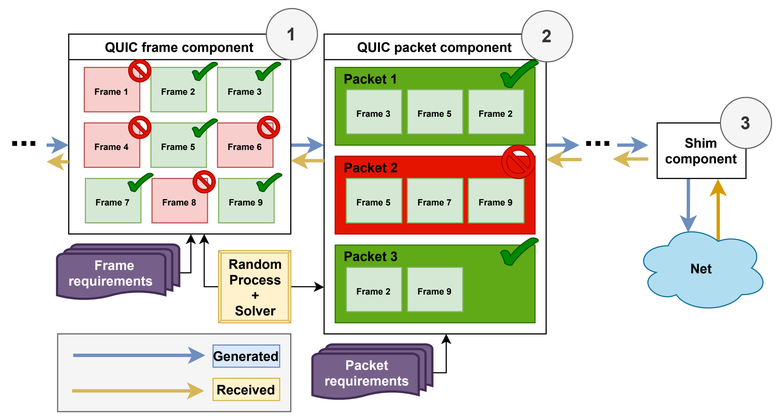
Ivy is a protocol specification, modeling, implementation, and verification tool.

Subsequent packets sent in the same packet number space MUST increase the packet number by at least one.

$\longleftrightarrow$

```
require pkt.seq_num =
                last_pkt_num(scid,pkt.ptype)
```

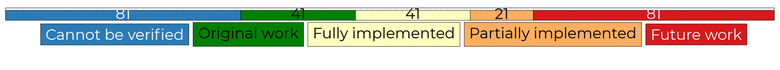
A formal Ivy model of the protocol is defined as a set of components linked by their input/output as the frame layer ① or the packet layer ②. The `shim` component ③ acts as a link with the network. Each component only generates correct items.



Tests are also written in Ivy. They define a specific feature to be tested.

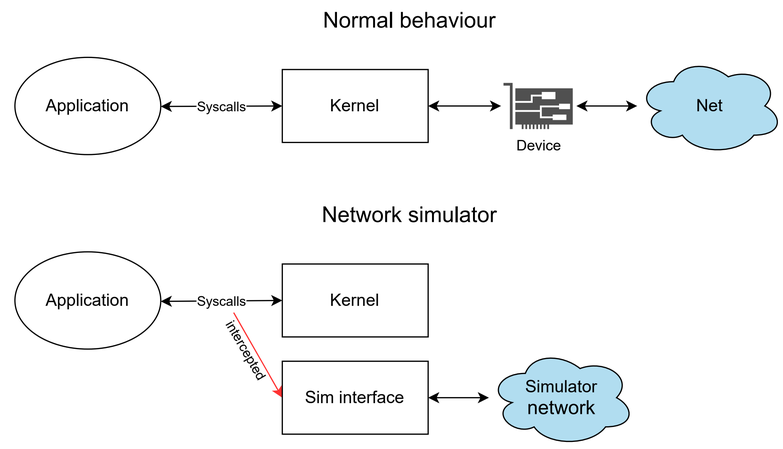
	quinn	mvfst	picoquic	quic-go	aioquic	quant	quiche
stream	79%	6%	56%	95%	18%	12%	97%
max	85%	3%	47%	39%	27%	21%	96%
reset_stream	29%	7%	61%	100%	24%	5%	98%
...	...						

Each test is run several times and different results may be obtained because tests are random and vary due to internal protocol timeouts (**race condition**). Indeed, packet generation is very slow. **Thus an error cannot be reproduced with certainty.**
Some specifications cannot be verified and some of them **apply on timing properties**.



Discrete-Event Network Simulator

Discrete-event network simulators intercept API calls and emulate them. They connect processes through an internal network. They consider code execution to be instantaneous.
This let control over the processes time perception, and thus **let verify timing properties**. By ignoring real execution time, it produces deterministic results, which **solves race conditions**.



The two most promising discrete-event network simulators are ns3-DCE and Shadow. ns3 is powerfull network simulator. ns3-DCE is a framework for ns3 which allows direct code execution. It emulates GLIBC calls and can only be used with C programs. It is not very well maintained.
Shadow is an active project still in development. It simulates directly syscalls allowing to run any program.

Attacker models

Another extension to Ivy is to allow to model attackers.

- *Man in the Middle* (MitM) model for QUIC
- Multiple way to control the connection
- All entities presented in the figure have a model
- To extend with other attacker models and protocols