TUM School of Computation, Information and Technology
Technische Universität München

Exploring Composable Network Stacks from Isolated Components with WebAssembly and QUIC

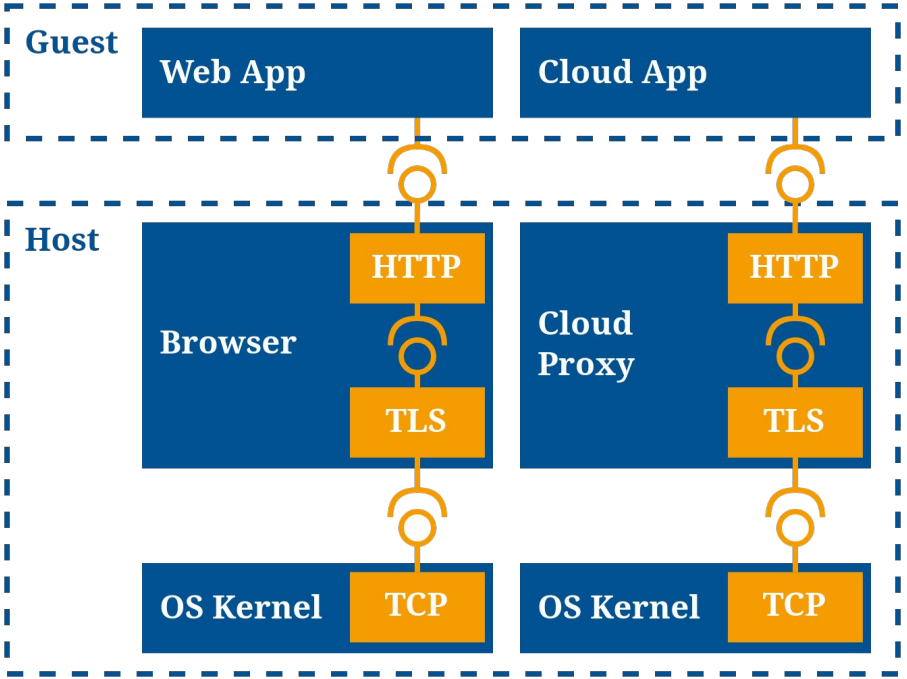
Benedikt Spies, Christian Obermaier, Jörg Ott

TUM School of Computation, Information and Technology
Department of Computer Engineering
Chair of Connected Mobility

NOMS 2025 WACE/Manage-IoT Workshop

State of the Art

- Modern web apps demand tailored network protocols
  - Real-time & Multimedia
- Apps use generic protocol implementations provided by the underlying platform
  - Browser & Cloud Host
- Host runs many guests
  - Browser tabs
  - Containers
  - Isolated guests
  - Efficient host protocols

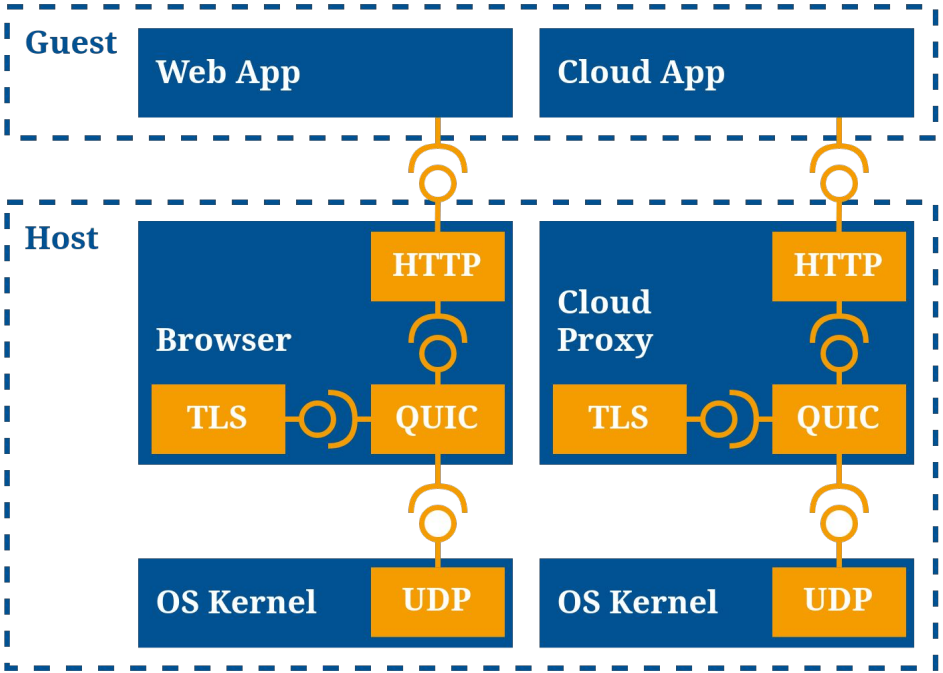


☐ **Lack of protocol adaptability by guest**

QUIC

- Standardized by the IETF in 2021
- General purpose
- Extensible
- Foundation for HTTP/3, RoQ, MoQ
- Userspace implementation maintains rapid development

☐ **Lack of protocol adaptability by guest**



Related Work

2019 De Coninck et al. - Pluginizing QUIC
adapt QUIC via eBPF

2020 Tran et al. - Beyond socket options: Towards fully extensible Linux transport stacks
adapt Linux MPTCP with eBPF

2024 De Coninck - Core QUIC: Enabling Dynamic, Implementation-Agnostic Protocol Extensions
adapt QUIC via WASM

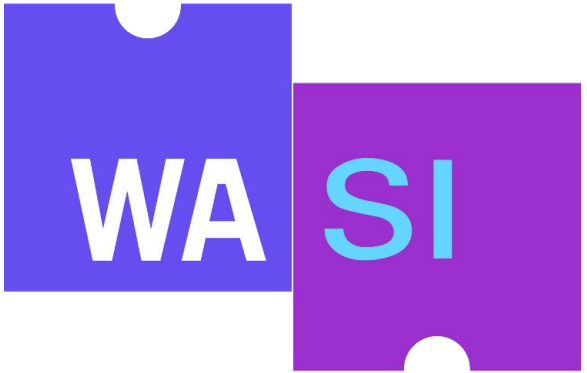
- **Hooks into provided implementation**
- **No extensive performance analysis**

WebAssembly (WASM)

- W3C standard since 2019
- Low-level bytecode format
- Compilation target
- Secure
  - Designed to run untrusted code
- No system calls
- Portable
  - Runs anywhere
  - Requires runtime environment (similar to JVM)

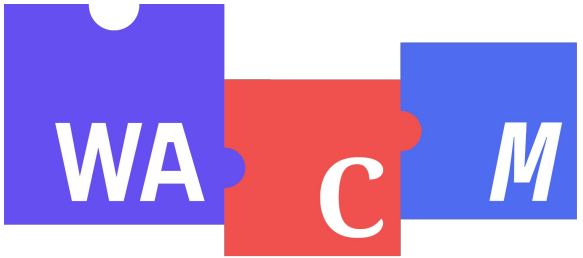
WebAssembly System Interface (WASI)

- In development; Preview 2 in 2024
- Collection of common APIs for system interaction
- Secure & Portable
- Import of required functions into WASM module



WebAssembly Component Model

- In development
- Secure composition of separately compiled WASM modules
- host-to-WASM and WASM-to-WASM linking
- shared-nothing-linking
  - each WASM module runs in isolated memory



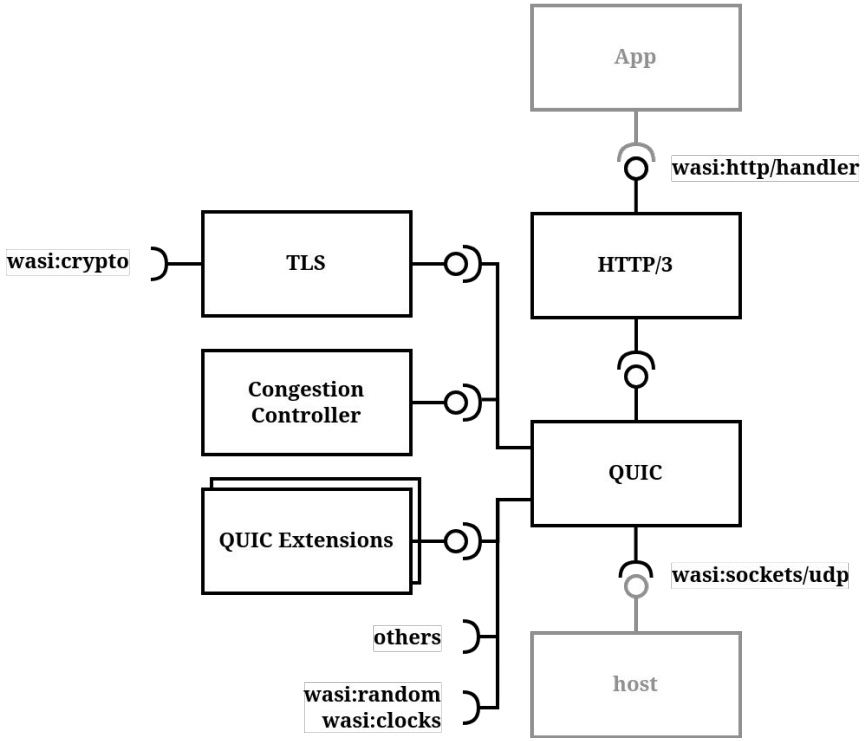
non-official logo

Design - QUIC Components

- QUIC stacks consist of functional units
  - demuxer, tls, stream frame sorter, cc, flow controller, frame scheduler, h3
- tight-coupling & in same memory region

⇒

- define interfaces by WIT IDL
- call-by-value
- resource handles

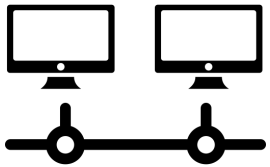


Component Sources

- Host system
  - OS, browser, or cloud runtime

- Third-party repository
  - warg

- Guest application
  - e.g. cloud tenant



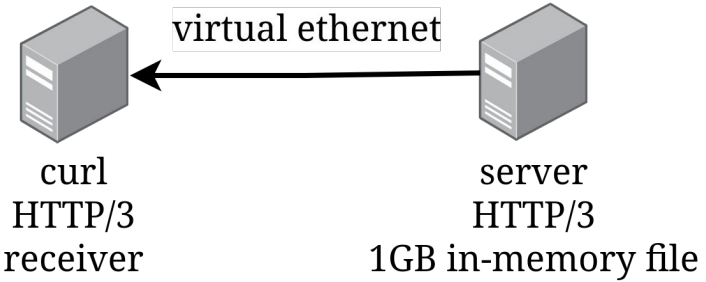
- Peer
  - during handshake or connection

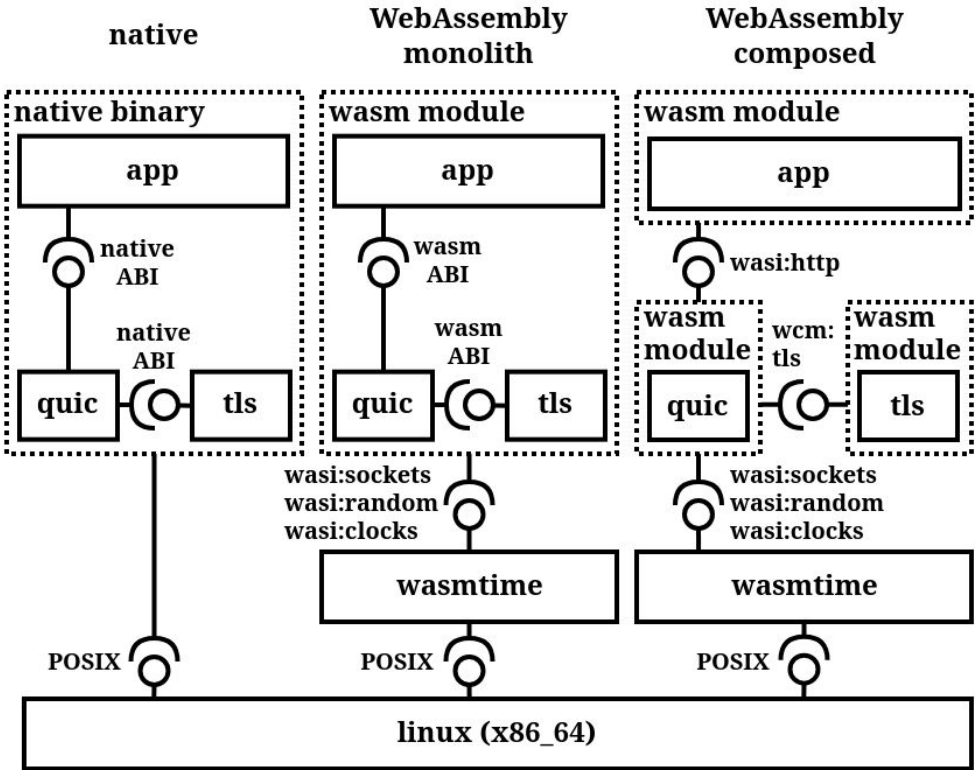
⇒ **strong isolation is key when running untrusted network components**

Evaluation

- 3 server variants
- Cloudflare quiche: replace mio
- BoringSSL: remove ASM; WASI randomness
- Dotted boxes show isolated memory

Testbed:

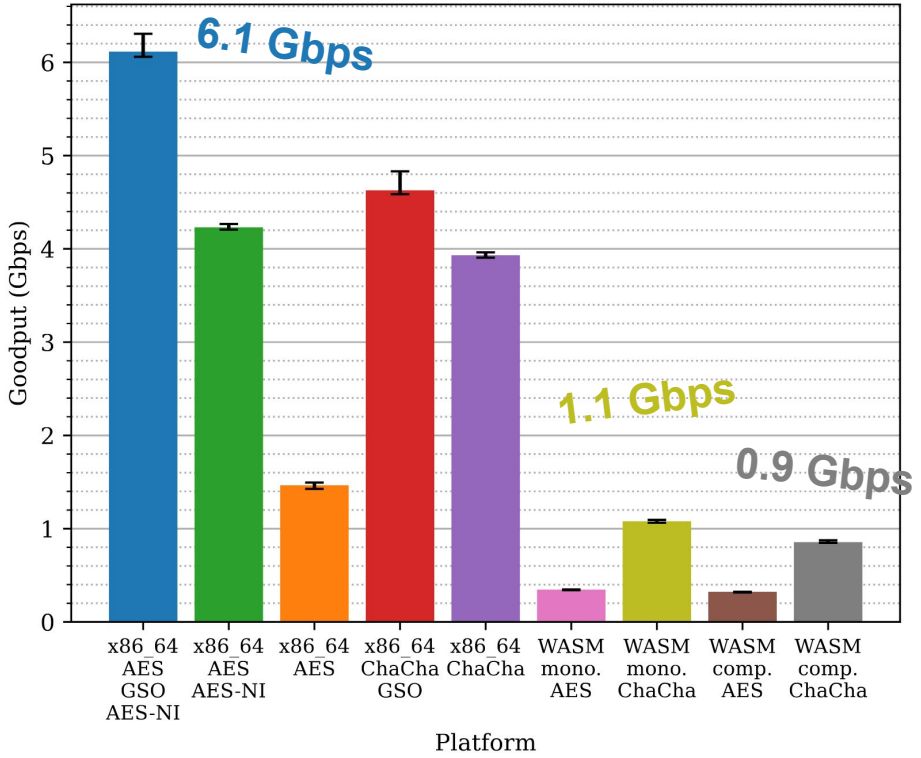




Evaluation - Goodput

- understand cost of isolation
- meet application demands
  - high bandwidth
  - low power consumption
- optimizations
  - GSO
  - AES-NI

? what causes the reduced performance



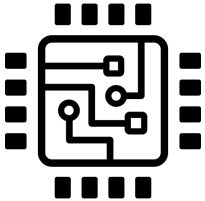
Evaluation - Call Stack Analysis

- Sample and categorize call stacks
- **100x** AES slowdown
- **20x** crypto slowdown using ChaCha
- **17x** cycles spent on memory copies
- **5x** socket IO slowdown

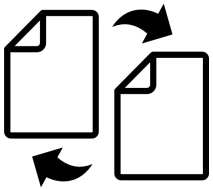
	crypto	copy	socket io	total
x86_64 AES128 GSO AES-NI	1.05 ± 0.21	0.43 ± 0.07	2.05 ± 0.33	7.68 ± 0.16
x86_64 AES128 AES-NI	1.28 ± 0.09	0.51 ± 0.04	7.28 ± 0.08	11.47 ± 0.13
x86_64 ChaCha20 GSO	1.81 ± 0.06	0.44 ± 0.04	2.69 ± 0.11	9.97 ± 0.17
x86_64 ChaCha20	2.00 ± 0.05	0.49 ± 0.05	7.02 ± 0.07	12.16 ± 0.08
x86_64 AES128	11.19 ± 0.09	0.57 ± 0.05	8.79 ± 0.10	32.93 ± 0.08
WASM AES128 monolith	111.83 ± 0.40	2.73 ± 0.08	11.68 ± 0.13	150.31 ± 0.48
WASM ChaCha20 monolith	20.63 ± 0.21	2.92 ± 0.15	11.23 ± 0.17	53.64 ± 0.31
WASM AES128 composed	111.42 ± 0.55	8.79 ± 0.21	11.74 ± 0.22	157.00 ± 0.48
WASM ChaCha20 composed	20.47 ± 0.10	8.66 ± 0.15	11.24 ± 0.24	59.57 ± 0.25

10^9 CPU cycles per transferred GiB. 4000 Hz sample rate. Mean of 20 repetitions.

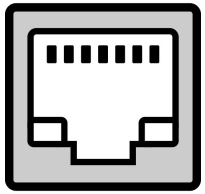
Open Challenges



- Accelerating cryptography
  - ISA specific ASM
  - wasi-crypto



- Reduce memory copies
  - inter-component communication with safe memory sharing



- Access efficient host network APIs
  - batched or asynchronous OS socket APIs

Use Cases

- Serverless and browser plugins
  - e.g. host provides API for guest to register custom multipath schedulers
- TLS isolation
  - cryptographic material stays in TLS component
- Kernel deployment
  - augment high-efficient QUIC implementation in kernel
- Simulation and test environments
  - easy to integrate in existing tools and environments

Conclusion

- Modularizing network stacks is technically feasible
- Overhead limits use for high-performance applications
  - from 6.1 Gbps to 0.9 Gbps
- Comprehensive analysis of WASM component isolation
  - Throughput, and call stack
- Identified major open challenges and possible solutions
  - cryptography, memory-copy, and network IO

⇒ **on the best way to an efficient and modular network stack**

Q&A

Crypto WIT Interface

```
package cm:quichewasm;

interface crypto {
    type c-int = s32;
    type usize = u32;

    resource evp-aead;
    resource evp-aead-ctx { constructor(); }

    resource buffer {
        constructor(buffer: list<u8>);
        take-back-copy: func() → option<list<u8>>;
    }
    resource usize-ptr {
        constructor(number: usize);
        back: func() → usize;
    }

    evp-aead-aes-one28-gcm: func() → evp-aead;
    evp-aead-aes-two56-gcm: func() → evp-aead;
    evp-aead-aes-chacha20-poly1305: func() → evp-aead;

    evp-aead-ctx-init: func(ctx: borrow<evp-aead-ctx>, aead: borrow<evp-aead>, key: borrow<buffer>, key-len: usi
    evp-aead-ctx-open: func(ctx: borrow<evp-aead-ctx>, out: borrow<buffer>, out-len: borrow<usize-ptr>, max-out-
    evp-aead-ctx-seal-scatter: func( ctx: borrow<evp-aead-ctx>, out: borrow<buffer>, out-tag: borrow<buffer>, ou
}

NOR   interface.wit                                                    1 sel  1:23
```