Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich

Accelerated Site-to-Site VPN

Intermediate Talk

Maximilian Pudelko, M. Sc.

September 18, 2018

Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich

Why VPNs are Important

- Todays business is multi-national and international
- Many distributed sites that need to be interconnected
- Provide a secure channel for communication over insecure medium
- Other usecases: VM interconnects, cell tower backbones, firm intra-nets

=> Need for high throughput solutions

Focus: Site-to-Site VPN

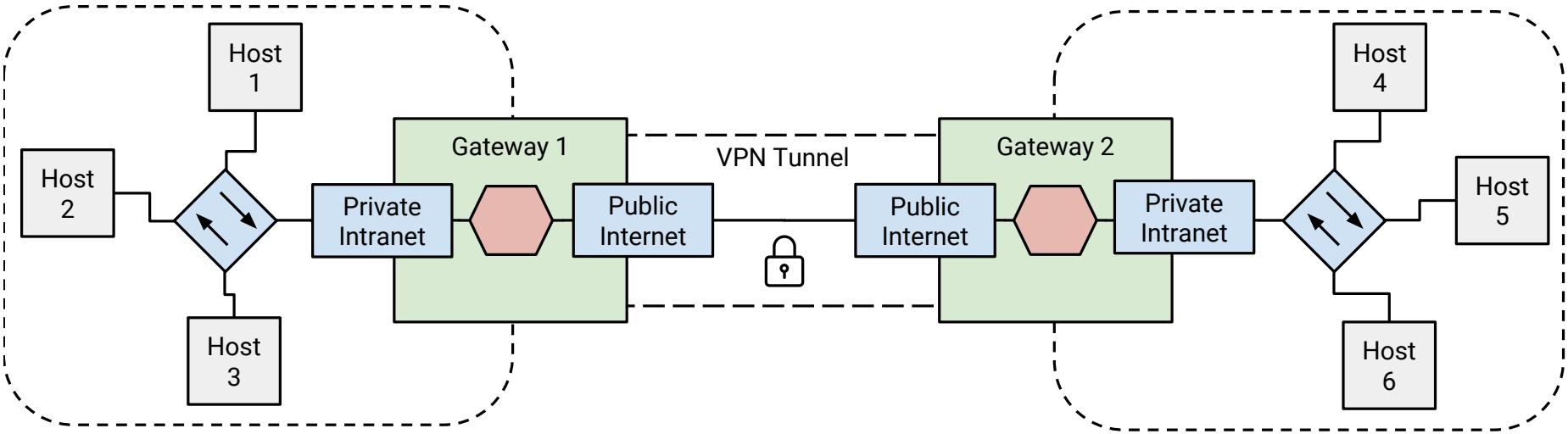

Figure 1: Overview of example Site-to-Site VPN setup

- Only two (or similar few) endpoints connecting many hosts
- Very high bandwidth between the gateways

Goals of this Thesis

- Create benchmark criteria for Site-to-Site setups
- Evaluate performance of common implementations
- Develop a general performance model for VPNs
- Explore different approaches for performance improvements

Overview of Common Implementations

OpenVPN

- Pure Userspace Sockets
- TLS & X.509
- L2 and L3
- Platform independent
- Single-threaded

IPsec (on Linux)

- Very complex protocol & code
- Build into Kernel
- L3 only (without L2TP)

WireGuard

- Very new with radical approaches
- State-of-the-Art cryptography
- Kernel module (inclusion ongoing)
- L3 only

Overview of Common Implementations

OpenVPN

- Pure Userspace Sockets
- TLS & X.509
- L2 and L3
- Platform independent
- Single-threaded

IPsec (on Linux)

- Very complex protocol & code
- Build into Kernel
- L3 only (without L2TP)

WireGuard

- Very new with radical approaches
- State-of-the-Art cryptography
- Kernel module (inclusion ongoing)
- L3 only

Shared problem:

Different degrees of slow under high load and in general

OpenVPN, TUN, UDP, AES-256-CBC: **0.06 - 0.14 Mpps**

IPsec, AES-GCM, 12 cores, 100% Load: **2.45 Mpps**

WireGuard under Load

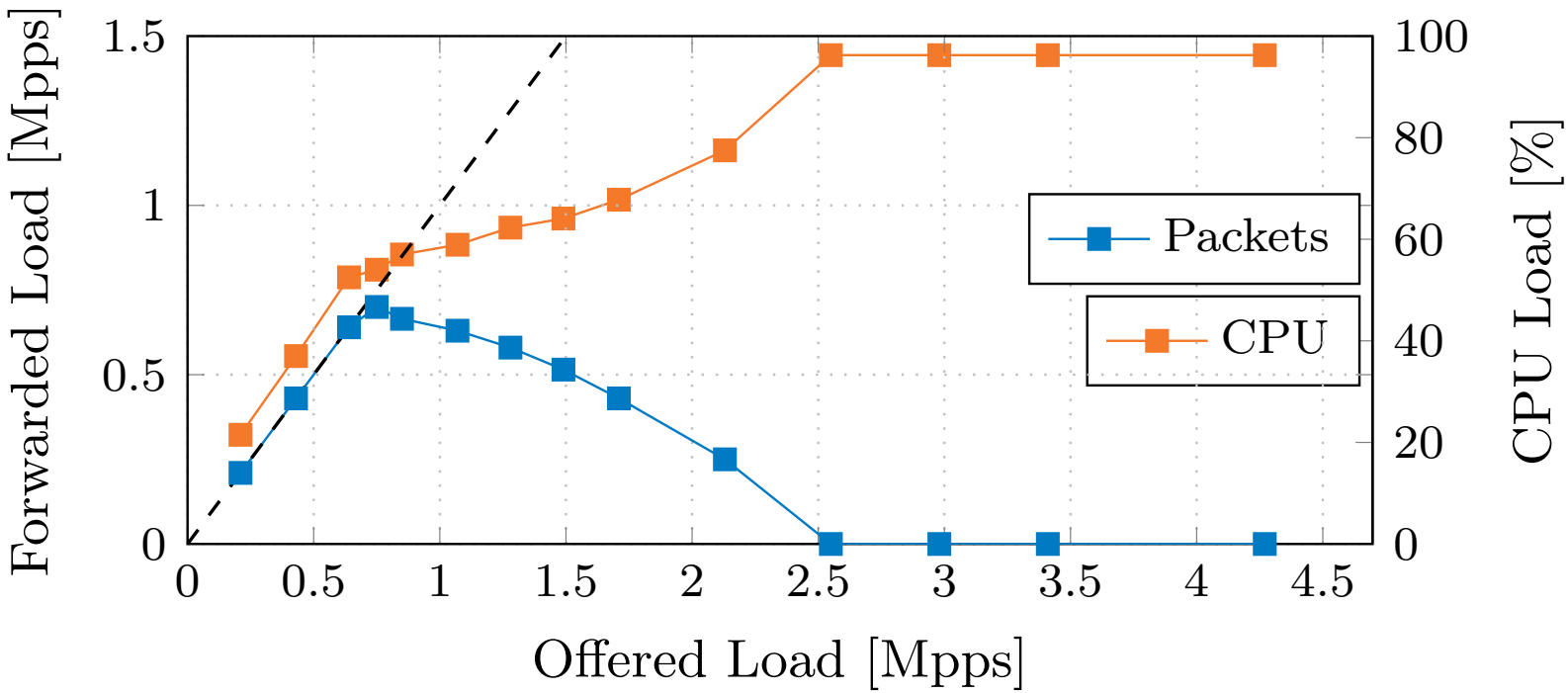

Figure 2: WireGuard v0.0.20180625 dropping packets under load

Benchmarking: Traffic Shapes

Exact traffic pattern and distributions vary depending on setup/use-case. **Influences performance**.

Common metric is **number of flows**. Identifies a group of packets to a connection/subnet/host. Usually 3-tuple (L2 proto, L3 src, L3 dst) or 5-tuple (+ L4 ports).

Packets-per-second (Mpps) is more interesting than **bits-per-second** (Gbit/s). 64 byte packets (minimum Ethernet frame size), but more realistic distributions are possible.

- Single flow, high bandwidth
  Worst case, models single client-server setup
- Multiple flows, equal bandwidth
  Best case, fits site-to-site setups, easy to model
- "Elephant" flows (few number of flows dominate bandwidth-wise)
  Realistic case (Netflix, Youtube, ...)

Example: Underutilization with single flows

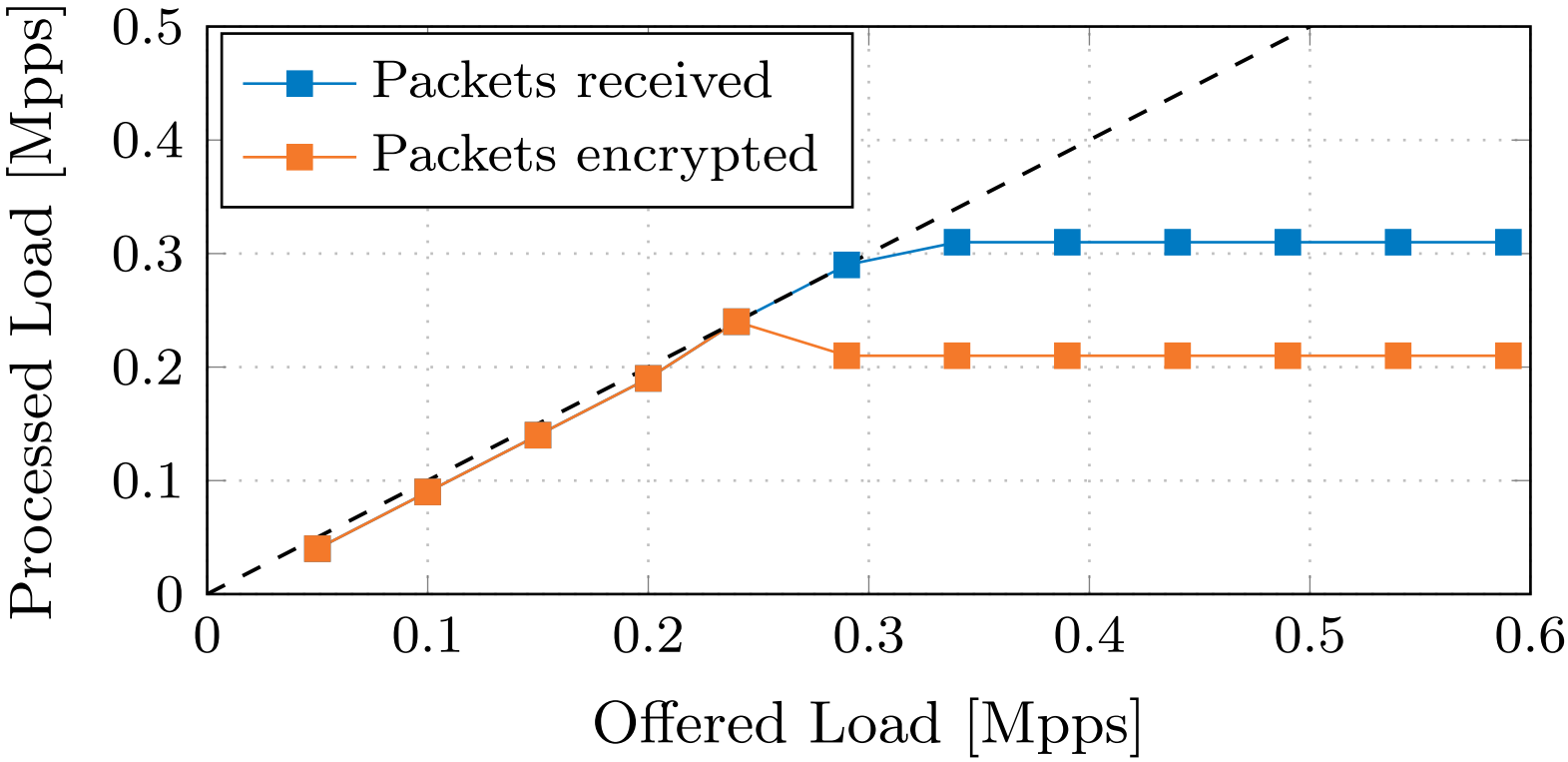

Figure 3: WireGuard forwarding rate of 64 byte packets, single flow, X540-AT2

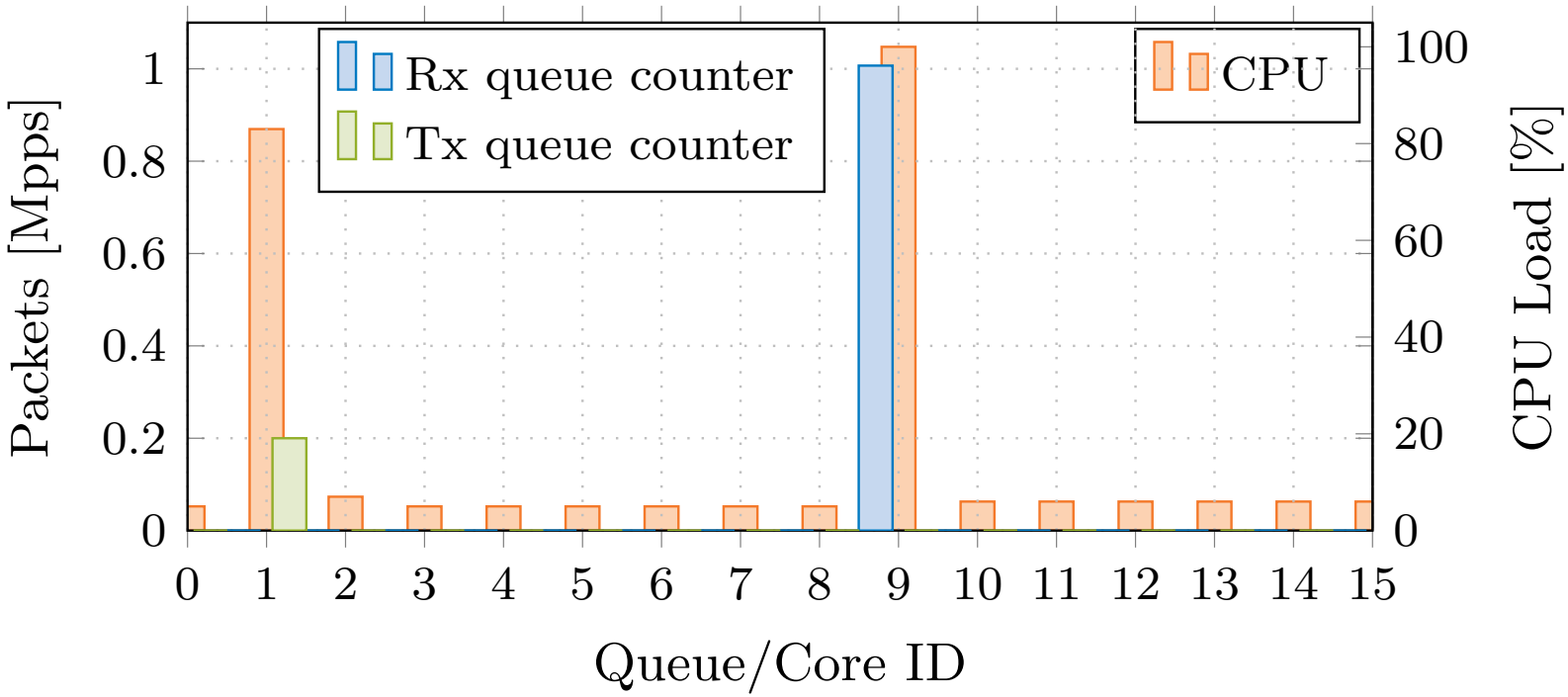

Figure 4: CPU/Queues utilization under single flow traffic

- NIC distributes packets to queues by L3 addresses
- One flow => everything in one queue
- Can be configured to include ports, not every NIC supports this

MoonWire

- DPDK network stack to bypass slow kernel
- Lua for fast prototyping and interfacing with libraries (crypto)
- Aims for protocol compatibility with WireGuard
- Allows experimenting with different data structures, algorithms, ...

MoonWire

- Ideally load is 100% encryption, real world 0% - 75%
- Cryptographic operations can become the bottleneck
- Hard to improve, correctness is more important

Solution: distribute work to multiple cores

- WireGuard utilizes Kernel worker tasks and a queue
- Lots of possible implementations
  Naive MoonWire version with per worker queues:
  Distributor: ≈8 Mpps
  Worker: ≈1.5 Mpps per core

Symmetric Encryption in a Nutshell

```
encrypt(shared_key, nonce, message) = ciphertext
```

Should be easy to scale up?

Symmetric Encryption in a Nutshell

```
encrypt(shared_key, nonce, message) = ciphertext
```

- Correct nonce generation/handling is critical. Nonce reuse (under the same key) breaks scheme and allows key recovery
- Nonce generation depends on length
  IETF ChaCha20 & AES256 GCM: 96 bit (12 byte)
  Too short to be random (birthday problem)
  Recommendation: Counting up
- Regular re-keying still recommended

But: Must be global over all threads/cores. Accessed for each packet => highly critical. Synchronization (mutex) and atomics are far too slow.

Nonce Generation Tricks

- Partition nonce space per worker/CPU/thread:
  8 bit worker_id + 94 bit counter = 96 bit
  Worker_0: **0**123, **0**124, **0**125, ...
  Worker_1: **1**123, **1**124, **1**125, ...
  Beware of "overflows" into different worker partition
- Different cipher: XChaCha20 has 192 bit (24 byte) nonce
  Can be randomly generated safely
  Each worker has own PRNG instance (seeded carefully)
  Independent state & no sharing => fast
  Trade-off: messages get larger (by 10 bytes), incompatible with existing protocol

Nonce Generation Tricks

- Partition nonce space per worker/CPU/thread:
  8 bit worker_id + 94 bit counter = 96 bit
  Worker_0: **0**123, **0**124, **0**125, ...
  Worker_1: **1**123, **1**124, **1**125, ...
  Beware of "overflows" into different worker partition
- Different cipher: XChaCha20 has 192 bit (24 byte) nonce
  Can be randomly generated safely
  Each worker has own PRNG instance (seeded carefully)
  Independent state & no sharing => fast
  Trade-off: messages get larger (by 10 bytes), incompatible with existing protocol

Good news: Decryption is much easier. Message contains everything.

Remaining Work

- More benchmarking & measurements
- Try more other performance improvements
  - AVX512 cipher implementations & CPU downclocking
  - NUMA
- Thesis writing