DTLS 1.3... In the Kernel! Really?

Anthony Hu; Senior Software Developer

OpenSSL Conference
Prague
October 2026

What We’ll Cover

01 **The Demo**

02 **The Fail**

03 **The Analysis**

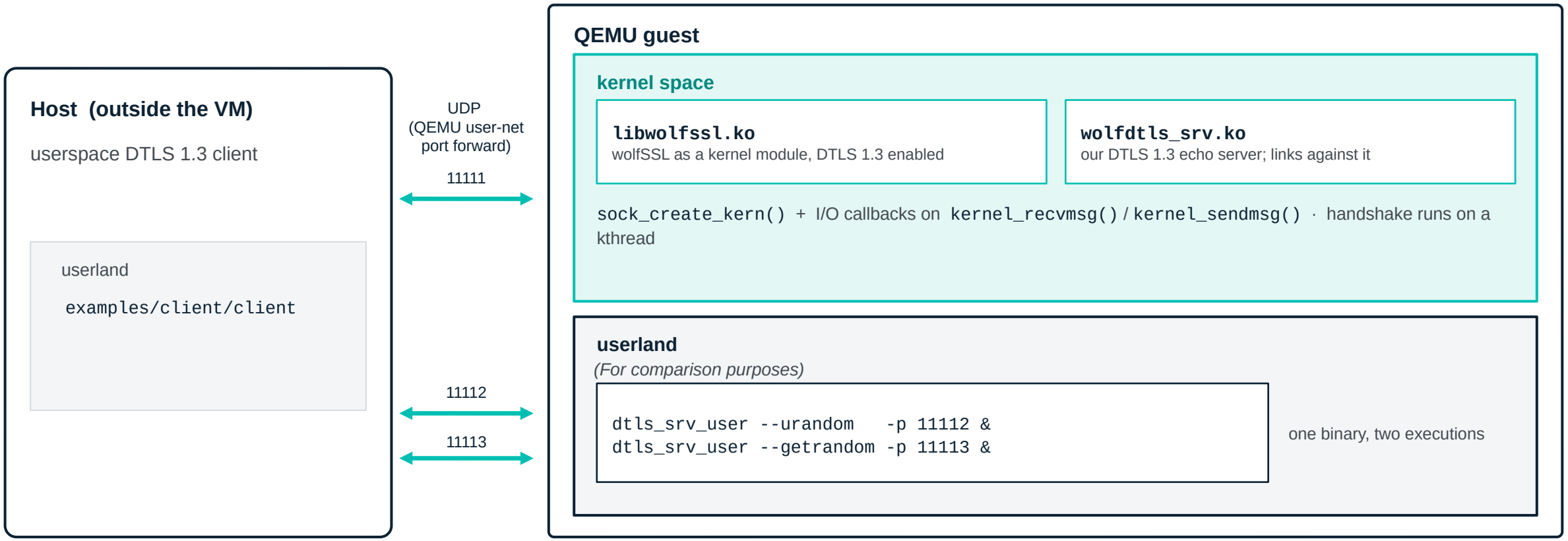
04 **Q&A**

SECTION 01

The Demo

A DTLS 1.3 server running in ring 0; and how to rebuild it from scratch.

What we built



`linuxkm/` ships no in-kernel DTLS sample; `libwolfssl.ko` only exports the API.
The wolfdtls_srv.ko server is ~300 lines we wrote.

Reproducing this: environment and ground rules

What it was built on

Host kernel	`6.17.0-41-generic (Ubuntu questing)`
Guest kernel	**the same image**
QEMU	10.1.0 + KVM
wolfSSL	`252c3b1a (2026-09-10), v5.9.2`

Tools needed, all stock: `gcc`, `make`, `busybox`, `cpio`, `zstd`, `xxd`, `qemu-system-x86_64`, and the distro's kernel headers.

No sudo required at any point.

Two decisions that make this cheap

1. **QEMU, not Docker.** A container shares the host kernel; `insmod` there loads into *your* kernel and a bug panics the machine. QEMU boots a genuinely separate kernel, so a fault kills only the VM.

2. **The guest boots the host's own kernel image.** So the installed headers are an exact match and **no kernel compile is needed**; which is what turns this from a day into coffee break build.

Reference slide; the next four exist so this is rebuildable from the PDF alone.

Step 1–2: pinned source, then libwolfssl.ko

Step 1; pin the commit

```
mkdir wolfssl && cd wolfssl
git init -q
git remote add origin https://github.com/wolfssl/wolfssl.git
git fetch --depth 1 origin 252c3b1aa3c8cbb5f5c052c9199f27a8a7ea2c29
git checkout FETCH_HEAD
cd .. && git clone -q wolfssl wolfssl-user   # 2nd tree, inherits the pin
```

That is a **31 MB** shallow fetch of exactly one commit. Two trees are needed because wolfSSL refuses a second `configure` in the same directory, and the kernel and userspace builds need different configurations.

Step 2; build the kernel module

```
cd wolfssl && ./autogen.sh
./configure --enable-linuxkm \
            --enable-tls13 --enable-dtls --enable-dtls13 \
            --with-linux-source=/lib/modules/$(uname -r)/build
make -j$(nproc) module
```


Reference slide.

Step 3–4: userspace tree, and a kernel image QEMU can read

Step 3; the userspace wolfSSL tree

```
cd ../wolfssl-user && ./autogen.sh
./configure --enable-dtls13 --enable-static --disable-shared
make -j$(nproc)
```

Step 4; get a readable vmlinuz

The one non-obvious step: `/boot/vmlinuz-$(uname -r)` is mode `0600 root:root`, so QEMU cannot read it as a normal user. Rather than reach for `sudo`, pull the identical image from the distro archive:

```
mkdir -p demo/kernel && cd demo/kernel
apt-get download linux-image-unsigned-6.17.0-41-generic
dpkg-deb -x linux-image-unsigned-*.deb extracted
cp extracted/boot/vmlinuz-6.17.0-41-generic ./vmlinuz
file vmlinuz   # same #41-Ubuntu SMP build as the running host kernel
```

This is what keeps the whole build sudo-free, and it is why the guest and host kernels are byte-identical.

Reference slide.

Step 5–6: build the demo, then run it

What had to be written

file	what	includes
`demo/kmod/wolfdtls_main.c`	DTLS 1.3 server, I/O callbacks, kthread	wolfSSL headers
`demo/kmod/kern_relay.c`	/dev/wolfdtls_relay, timing helpers	kernel headers

Step 5; one script builds everything

```
./demo/build-all.sh
modinfo demo/kmod/wolfdtls_srv.ko | grep -E 'depends|import_ns|vermagic'
#  depends: libwolfssl  import_ns: WOLFSSL  vermagic: 6.17.0-41-generic SMP preempt
```

Step 6; run any of these

Each boots its own guest and tears it down.

```
./demo/demo.sh       # the handshake: host client -> in-kernel server
./demo/bench.sh      # entropy: kernel vs /dev/urandom vs vDSO getrandom
./demo/isolation.sh  # key-extraction attack, lockdown off then on
./demo/relay.sh      # per-message handoff cost + decomposition
```

What comes up inside the guest

All three servers share one cert and key, so comparisons isolate a single variable.

port	space	entropy source
11111	**kernel**	`get_random_bytes()`
11112	user	`read(/dev/urandom)`
11113	user	`getrandom()  (vDSO)`

Request access to scripts and sources by emailing anthony@wolfssl.com

Reference slide.

It works

Host; userspace client

```
connecting to 127.0.0.1:11111
SSL version is DTLSv1.3
SSL cipher suite is TLS_AES_256_GCM_SHA384
SSL curve name is SECP256R1
hello wolfssl!
```

Guest; kernel ring buffer

```
wolfdtls_srv: listening on UDP 0.0.0.0:11111
wolfdtls_srv: client connected from 10.0.2.2:41415
wolfdtls_srv: HANDSHAKE OK:
              DTLSv1.3 / TLS_AES_256_GCM_SHA384
wolfdtls_srv: recv 14 bytes: hello wolfssl!
```

Same session, both sides. The server half is executing in kernel space.

How we made the comparison fair

1. **All servers in the same guest.** No servers on the host.

2. **Identical everything else.** Same DTLS 1.3 method, same embedded ECC cert and key, same request protocol.

3. **Timed at the server, not over the wire.** A tight loop timed with local_clock() in-kernel and CLOCK_MONOTONIC in userspace.

SECTION 02

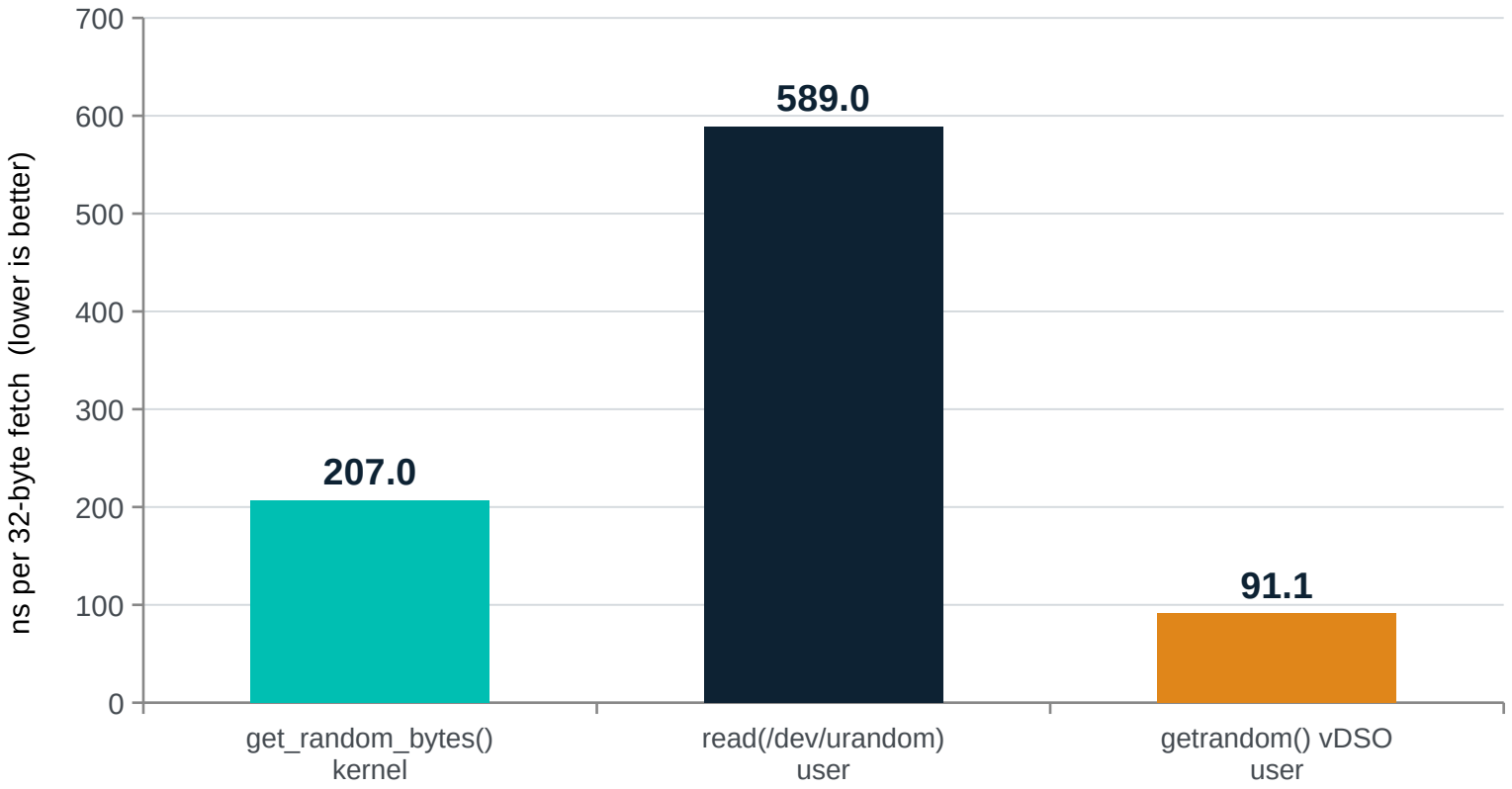
The Fail

OUR HYPOTHESIS

“Entropy is faster in the kernel; no syscall, no `copy_to_user`.”

Makes sense, right?

Entropy: three sources on the same guest



3 trials × 200,000 fetches

source	t1	t2	t3	mean
kernel	191.00	215.56	214.55	**207.0**
/dev/urandom	612.84	540.37	613.85	**589.0**
vDSO	89.28	81.18	102.96	**91.1**

kernel vs /dev/urandom

2.85× faster

Yay!

kernel vs vDSO getrandom()

2.27× SLOWER

Huh?

Why the kernel loses: fixed per-call overhead

Fixed overhead

~197 ns per call, not slower crypto

`get_random_bytes()` must be callable from any context; per-CPU state under `local_lock_irqsave`, plus a reseed check every call. The vDSO owns per-thread state: no lock, no IRQ save/restore, no syscall.

SECTION 03

The Analysis

Advantage 1; key isolation: the attack

1. **Both servers load the same P-256 private key.**

2. **The scanner searches structurally**; for the SEC1 `ECPrivateKey` DER header; so it never needs to know the key in advance. That is what an attacker with a memory-disclosure primitive actually does.

Run twice

```
lockdown off
    ↓
lockdown=confidentiality
```

The signature being hunted

```
30 77 02 01 01 04 20   <32-byte private scalar>
```

Userspace target
`walk every /proc/<pid>/mem`

Kernel target
`walk module memory via /proc/kcore`

Advantage 1; the result

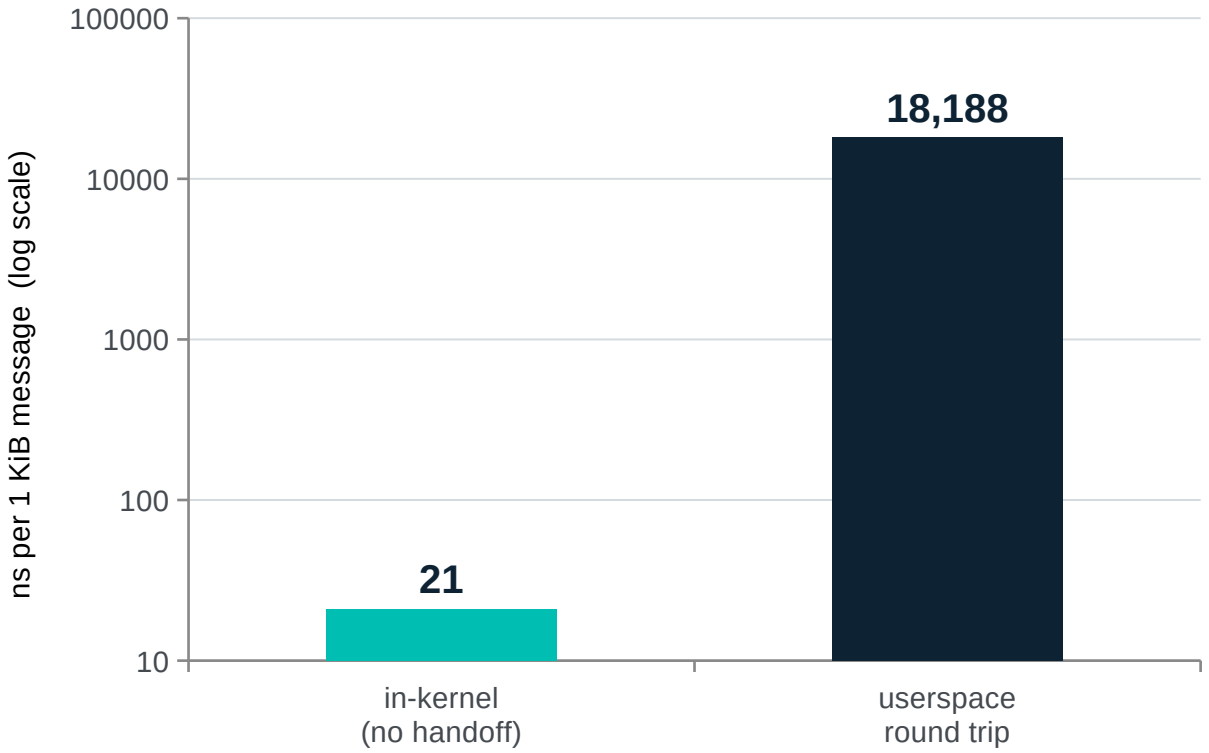
	lockdown = none	lockdown = confidentiality
userspace server key	STOLEN	STOLEN
kernel module key	STOLEN via /proc/kcore	/proc/kcore: EPERM; 0 hits

But wait! Can’t you just scan the kernel module binary? YES!

But in a real production environment, the private key would not be hard-coded into the kernel module binary.

Advantage 2; when the data starts in the kernel

If the data being sent originates in the kernel; a userspace DTLS implementation forces every message out to a daemon and back.



20,000 messages of 1 KiB, 3 trials

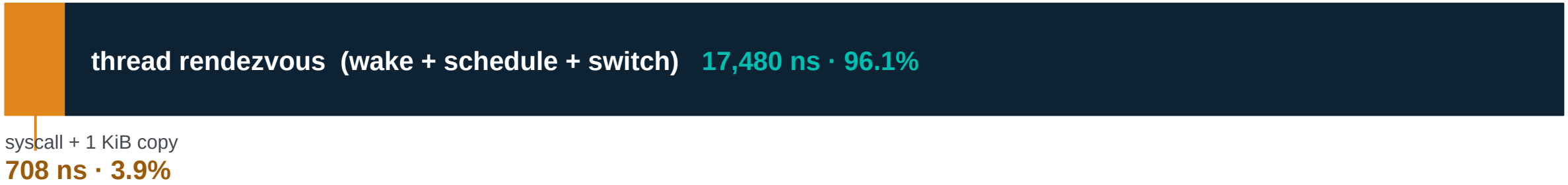
path	t1	t2	t3	mean
in-kernel	17	16	30	**21**
userspace RT	19,181	17,367	18,017	**18,188**

Exactly 1.00 context switch per message, plus a matching one kernel-side.

It is not the syscall

We first attributed that cost to “crossing the user/kernel boundary”. So we measured again with the daemon **never sleeping** (`relay_read()` returns immediately); no wakeup, no context switch, leaving only two syscalls and one 1 KiB `copy_to_user()`.

total round trip 18,188 ns; drawn to scale



component	ns	share
syscall + 1 KiB copy	**708**	3.9%
thread rendezvous (wake + schedule + switch)	**17,480**	**96.1%**
total round trip	**18,188**	100%

The rendezvous costs

25×

the boundary crossing

The syscall was never the problem. The right lesson is *“avoid waking another thread per message”*, not *“avoid syscalls”*.

When is DTLS most appropriate in the kernel?

When there is no userspace process in the data path to move the crypto into.

NFS client / server
`CONFIG_NFS_FS · CONFIG_NFSD`

SMB client / server
`CONFIG_CIFS · CONFIG_SMB_SERVER`

WireGuard, IPsec, MACsec
`CONFIG_WIREGUARD · CONFIG_XFRM_USER · CONFIG_MACSEC`

Others?

Honesty note: these transports run over TCP and therefore want TLS, not DTLS. The datagram members of this class are VPN data planes and UDP-based cluster transports. The principle is the same; the consumer is already in the kernel; but this is not a claim that NFS is asking for DTLS.

The FIPS 140-3 argument: one module, one OE

1. **FIPS validates a cryptographic module bound to a specific Operating Environment.** Note that Userland and Kernelland are different and separate OEs!

2. **If you are doing dm-crypt, ZFS, or wolfGuard with wolfCrypt for FIPS,** you are already running the wolfCrypt Kernel Module variant, on a kernel OE you have already validated.

3. **So if you then need DTLS 1.3 and/or TLS 1.3,** doing it in the kernel makes it one more consumer of a module on an OE you already have. (No need for a userland OE!)

one module, one OE, one integrity check, one audit.

Scoreboard and takeaways

claim	verdict	
faster entropy	**false; vDSO getrandom() beats the kernel 2.27×**	measured
faster crypto	not measured, no reason to expect it	not tested
key isolation	**true; conditional on lockdown**	measured
fewer per-message handoffs for kernel-originated data	**true; no context switch per message**	measured
no userspace side exists for some transports	**true; not a benchmark, an architecture**	ARGUMENT
reuses a FIPS OE you might have already validated	**true; the business case**	ARGUMENT

In-kernel code does not win by computing faster. It wins by **not handing work to another thread**, by putting data **where the other side cannot reach**, or by **not needing a second validated environment at all**.

Closing remarks

- All this basically applies to TLS too!
- E-mail anthony@wolfssl.com for a copy of these slides and a link to demo source and script.
- Find me after the talk to show you a live demo of what I've presented here.

Thank You

Questions?

Anthony Hu Senior Software Developer, wolfSSL

anthony@wolfssl.com | facts@wolfssl.com