



How to build post-quantum cryptographic protocols
and why wall clocks are not to be trusted.

Karolin Varner, Benjamin Lipp, and **Lisa Schmidt**
with support from Alice Bowman, and Marei Peischl
<https://rosenpass.eu>



This is the Plan

1. **Introducing Rosenpass**, briefly.
2. **The Design of Rosenpass** and basics about post-quantum protocols.
3. **Hybrid Security** – how it can be done and how we do it.
4. **ChronoTrigger Attack** and not trusting wall clocks.
5. **Protocol Proofs** – big old rant!
6. **Q&A** – and probably “more of a comment”.



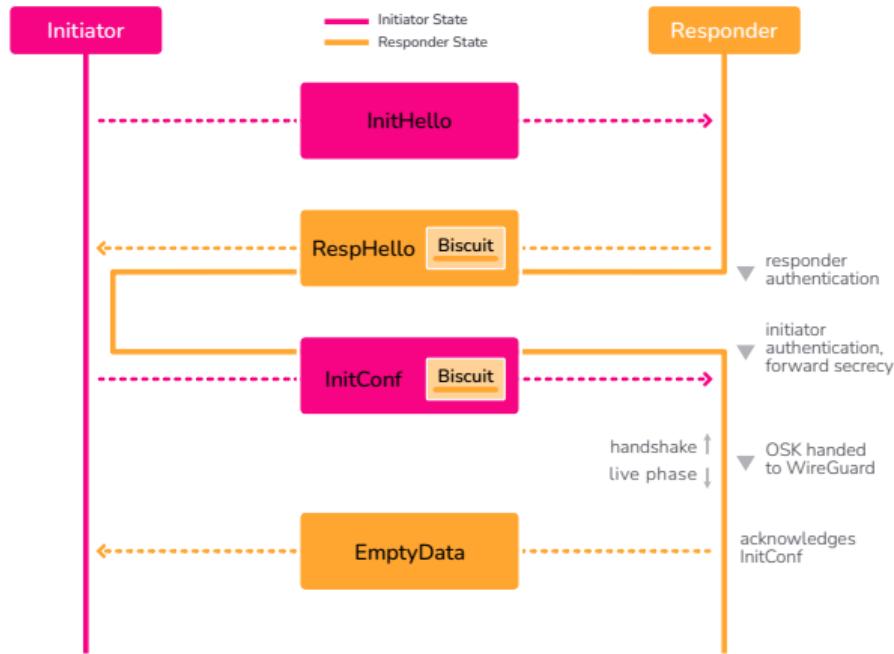
Follow the talk at:

rosenpass.eu/docs/presentations/hackmas-2024/



Introducing Rosenpass, briefly

- A post-quantum secure key exchange **protocol** based on the paper Post-Quantum WireGuard [PQWG]
- An open source Rust **implementation** of that protocol, already in use
- A way to secure WireGuard VPN setups against quantum attacks
- A **post-quantum secure VPN**
- A governance **organization** to facilitate development, maintenance, and adoption of said protocol



The Design of Rosenpass

and how to build post-quantum protocols





In the following slides you will learn ...

... that, to a crypto protocol designer, post-quantum cryptography is not much more than a subtle difference in function interface.



Glossary: Post-Quantum Security

Pre-quantum
cryptography is ...

Post-quantum
cryptography is ...

Hybrid cryptography
combines ...

... susceptible to attacks from
quantum computers.

... not susceptible to attacks
from quantum computers.

... the combination of the
previous two. It is ...

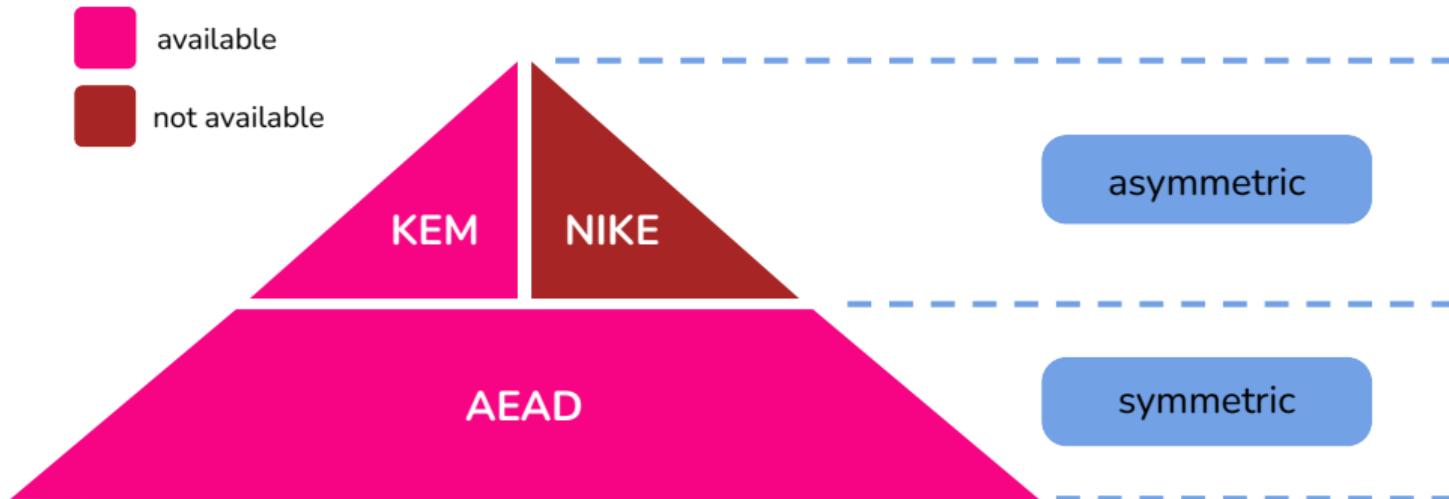
- specifically, to
Shor's Algorithm
- quite fast
- widely trusted

- generally less efficient.
- much bigger ciphertexts.
- less analyzed.

- about as inefficient as
post-quantum
cryptography.
- not widely adopted, which
is a major problem.



What Post-Quantum got





KEMs and NIKEs

Key Encapsulation Method

```
fn Kem::encaps(Pk) -> (Shk, Ct);
```

```
fn Kem::decaps(Pk, Ct) -> Shk;
```

```
(shk, ct) = encaps(pk);
```

```
assert!(decaps(sk, ct) = shk)
```

Think of it as encrypting a key and sending it to the partner.

- secrecy
- implicit authentication of recipient
(assuming they have the shared key, they must also have their secret key)

Non-Interactive Key Exchange

```
fn nike(sk: Sk, pk: Pk) -> Shk;
```

```
assert!(nike(sk1, pk2) =  
       nike(sk2, pk1));
```

Aka. Diffie-Hellman. Note how the keypairs are crossing over to each other.

- secrecy
- implicit mutual authentication (for each party: assuming they have the shared key, they must also have their secret key)



Protocol Security Properties

Implicit authentication

“If you have access to this shared symmetric key then you must have a particular asymmetric secret key.”

Explicit authentication

“I know you have access to this shared key because I checked by making you use it, therefore you also have a particular asymmetric secret key.”

Secrecy

“The data we exchange cannot be decrypted unless someone gets their hands on some of our static keys!”

Forward secrecy

“Even if our static keys are exposed, the data we exchanged cannot be retroactively decrypted!”*

* Terms and conditions apply:

We are using an extra key that we do not call a static key. This key is generated on the fly, not written to disk and immediately erased after use, so it is more secure than our static keys. Engaging in cryptography is a magical experience but technological constructs can – at best – be asymptotically indistinguishable from miracles.



KEMs and NIKEs: Key Exchange

Key Encapsulation Method

Responder Authentication: Initiator encapsulates key under the responder public key.

Initiator Authentication: Responder encapsulates key under the initiator public key.

Forward Secrecy: In case the secret keys get stolen, either party generates a temporary keypair and has the other party encapsulate a secret under that keypair.

How to do this properly? See Rosenpass.

Non-Interactive Key Exchange

Responder Authentication: Static-static NIKE since NIKE gives mutual authentication.

Initiator Authentication: Static-static NIKE since NIKE gives mutual authentication.

Forward secrecy: Another NIKE, involving a temporary keypair.

How to do this properly? See the Noise Protocol Framework. [NOISE]



KEMs and NIKEs

Key Encapsulation Method

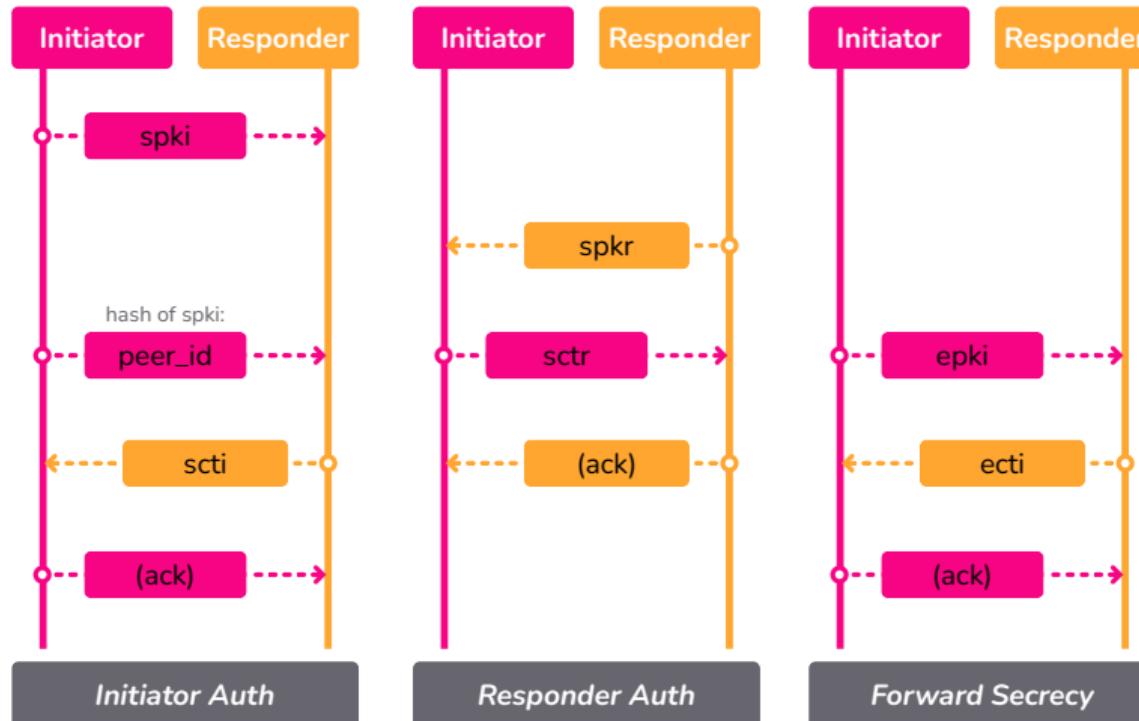
```
trait Kem {
    // Secret, Public, Symmetric, Ciphertext
    type Sk; type Pk; type Shk; type Ct;
    fn genkey() -> (Sk, Pk);
    fn encaps(pk: Pk) -> (Shk, Ct);
    fn decaps(sk: Pk, ct: Ct) -> Shk;
}
#[test]
fn test<K: Kem>() {
    let (sk, pk) = K::genkey();
    let (shk1, ct) = K::encaps(pk);
    let shk2 = K::decaps(sk, ct);
    assert_eq!(shk1, shk2);
}
```

Non-Interactive Key Exchange

```
trait Nike {
    // Secret, Public, Symmetric
    type Sk; type Pk; type Shk;
    fn genkey() -> (Sk, Pk);
    fn nike(sk: Sk, pk: Pk) -> Shk;
}
#[test]
fn test<N: Nike>() {
    let (sk1, pk1) = N::genkey();
    let (sk2, pk2) = N::genkey();
    let ct1 = N::nike(sk1, pk2);
    let ct2 = N::nike(sk2, pk1);
    assert_eq!(ct1, ct2);
}
```

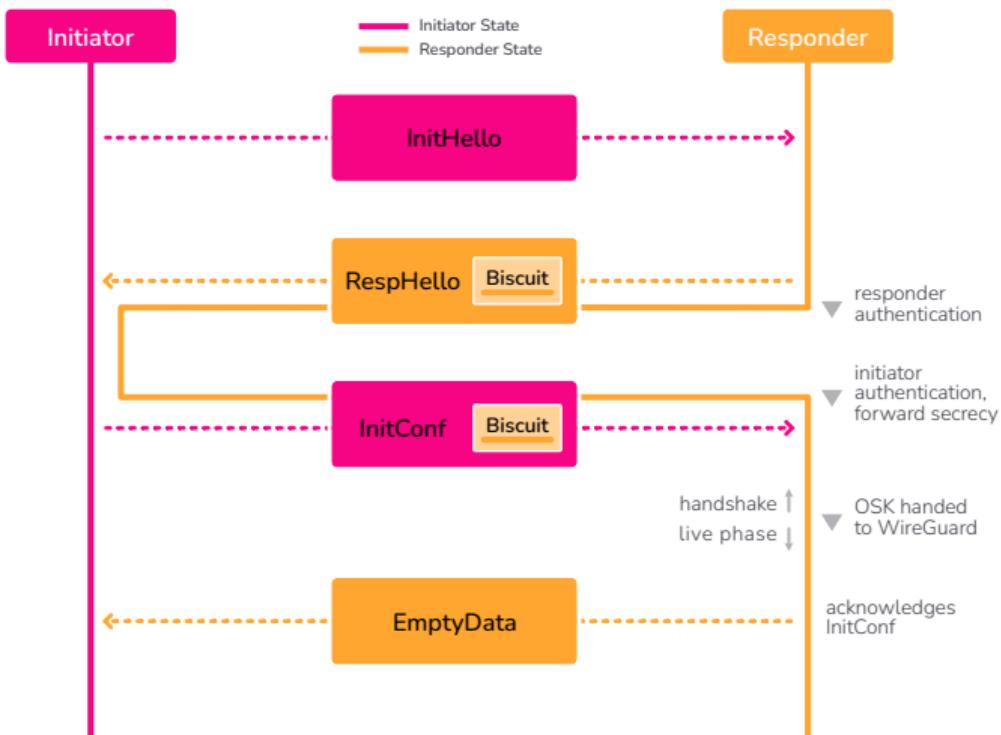


Rosenpass Key Exchange Parts





Rosenpass Protocol Features

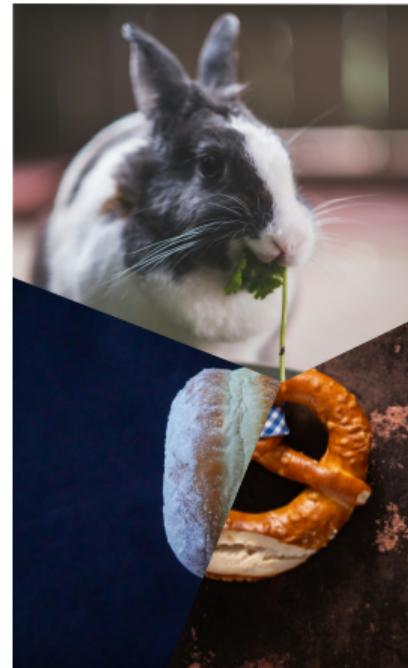


- authenticated key exchange
- three KEM operations interleaved to achieve mutual authentication and forward secrecy
- no use of signatures
- first package (**InitHello**) is unauthenticated
- stateless responder to avoid disruption attacks

Hybridization

In the following slides you will learn ...

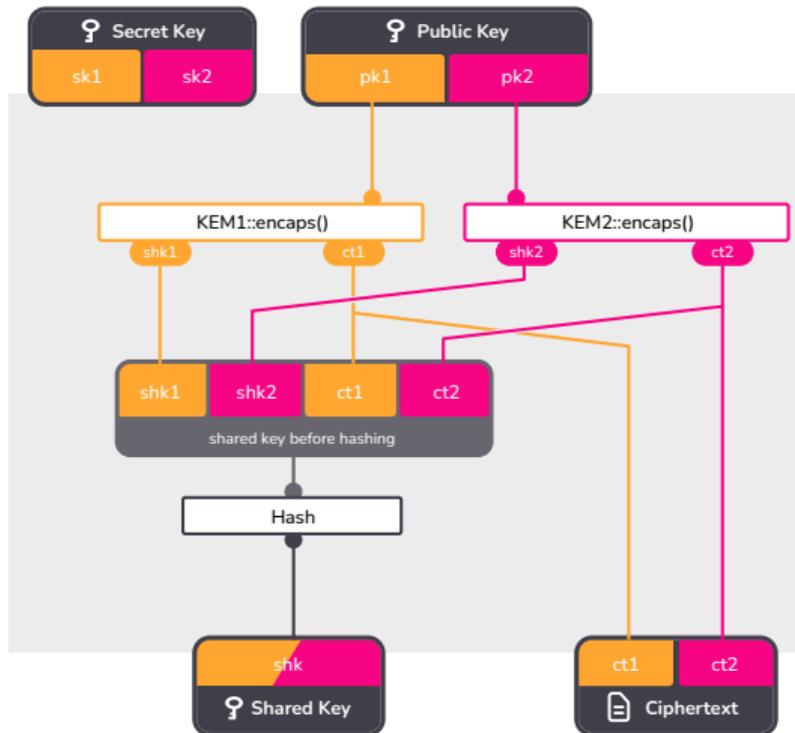
... that hybrid security can be achieved by building hybrid primitives and that it is not always wise to do so.





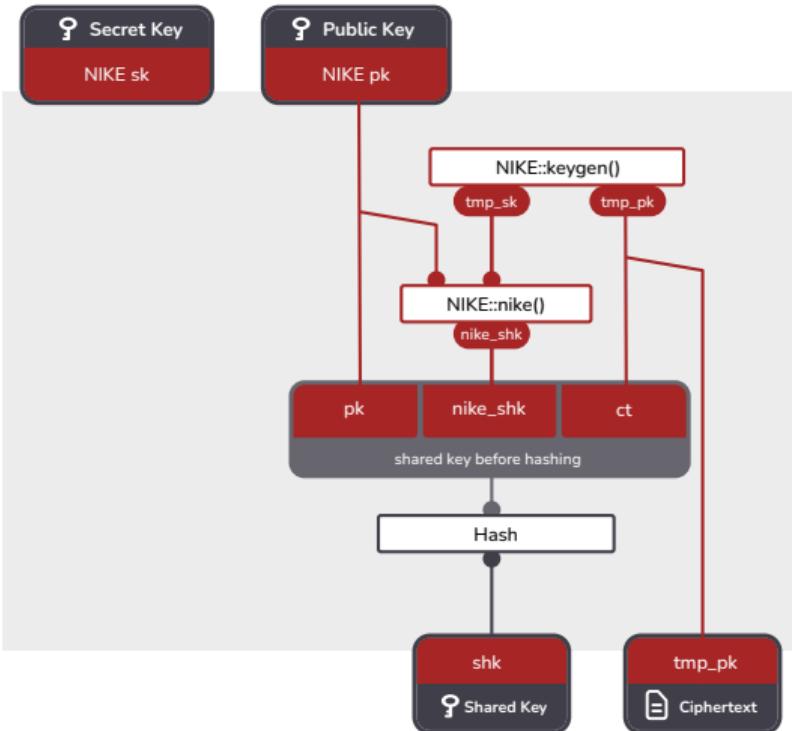
Combining two KEMs with the GHP Combiner

- “Giacon-Heuer-Poettering” [GHP]
- running both KEMs in parallel
- secret keys, public keys, and ciphertexts are concatenated
- shared keys are hashed together
- ciphertexts included in hash for proof-related reasons





Turning a NIKE into a KEM

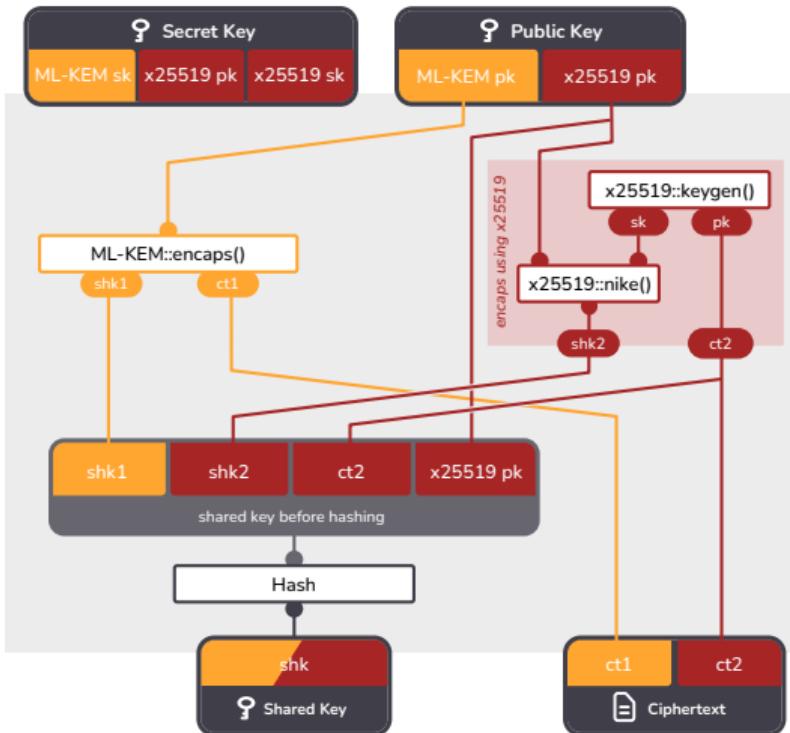


- from the HPKE RFC [HPKE]
- *remote* keypair is static keypair
- *local* keypair is temporary keypair
- local keypair public key is treated as ciphertext
- for proof-related reasons, ciphertext and public key are included in hash
- RFC work by Barnes, Bhargavan, Lipp, Wood supported by analysis work by Alwen, Blanchet, Hauck, Kiltz, Lipp, Riepel [HPKE]



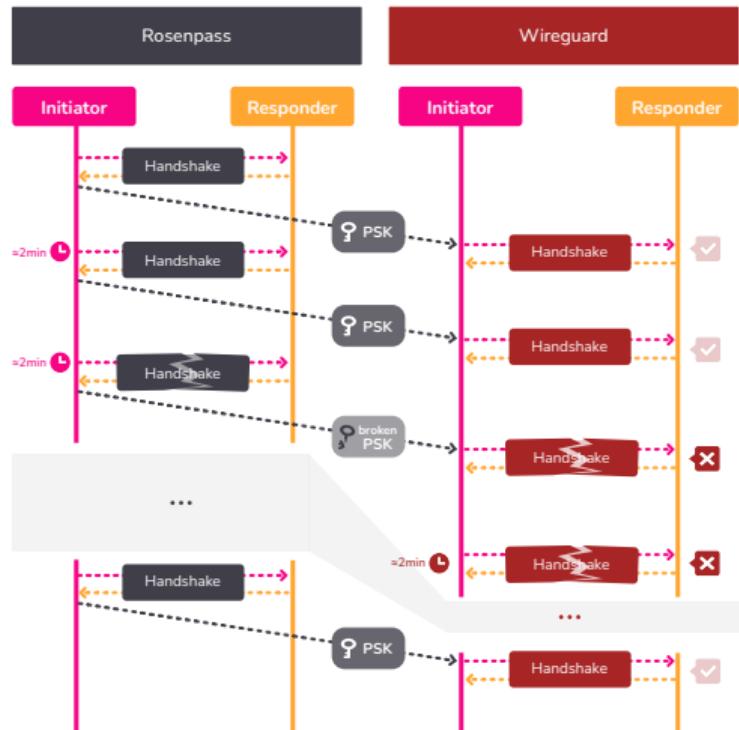
X-Wing [XWING]

- combines ML-KEM and X25519
- techniques from DHKEM to turn X25519 into a KEM
- techniques from GHP to combine the two
- optimizations applied to make hashing more efficient
- bespoke proof of security
- work by Barbosa, Connolly, Duarte, Kaiser, Schwabe, Varner, Westerbaan [XWING]





Rosenpass & WireGuard Hybridization



- Rosenpass and WireGuard are hybridized on the protocol level
- preserving efficiency of and trust in WireGuard
- straightforward transition path; existing WireGuard implementation remains in use
- key from Rosenpass used as PSK in WireGuard

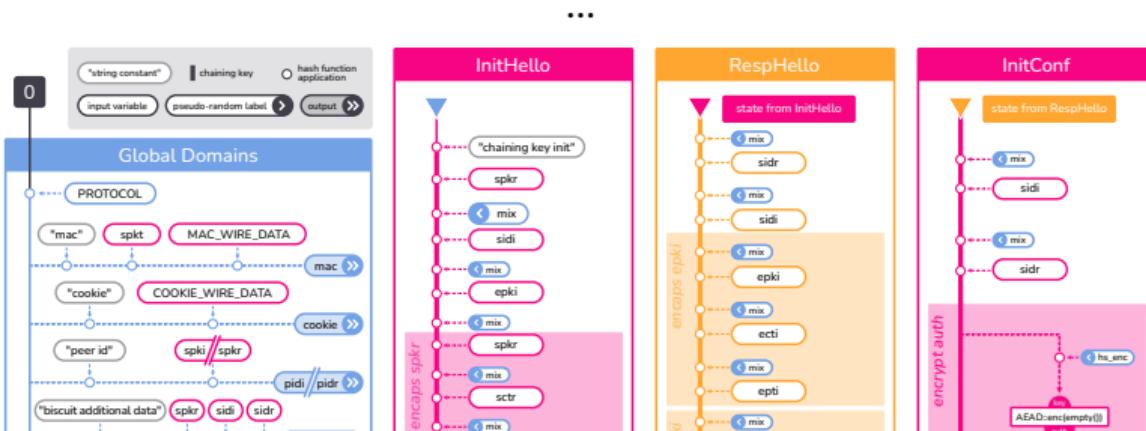


Full Protocol Reference in the Whitepaper

Initiator Code	Responder Code	Comments
1	InitHello { sidi, epki, sctr, pidiC, auth }	2
Line Variables \leftarrow Action	Variables \leftarrow Action	Line
IH1 ck \leftarrow lhash("chaining key init", spkr)	ck \leftarrow lhash("chaining key init", spkr)	IHR1
IH2 sidi \leftarrow random_session_id();		
IH3 eski, epki \leftarrow EKEM:keygen();		
IH4 mix(sidi, epki);	mix(sidi, epki)	IHR4
IH5 sctr \leftarrow encaps_and_mix<SKEM>(spkr);	decaps_and_mix<SKEM>(spkr, skr, ct1)	IHR5
IH6 pidiC \leftarrow encrypt_and_mix(pidi);	spki, psk \leftarrow lookup_peer(decrypt_and_mix(pidiC))	IHR6
IH7 mix(spki, psk);	mix(spki, psk);	IHR7
IH8 auth \leftarrow encrypt_and_mix(empty())	decrypt_and_mix(auth)	IHR8



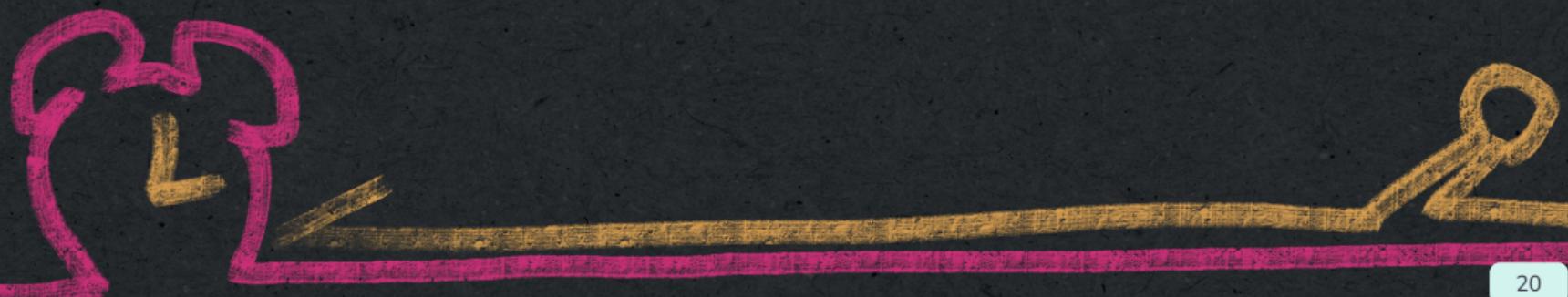
rosenpass.eu/docs



rosenpass.eu/whitepaper.pdf

Trials ~ Attacks found

ChronoTrigger



In the following slides you will learn ...

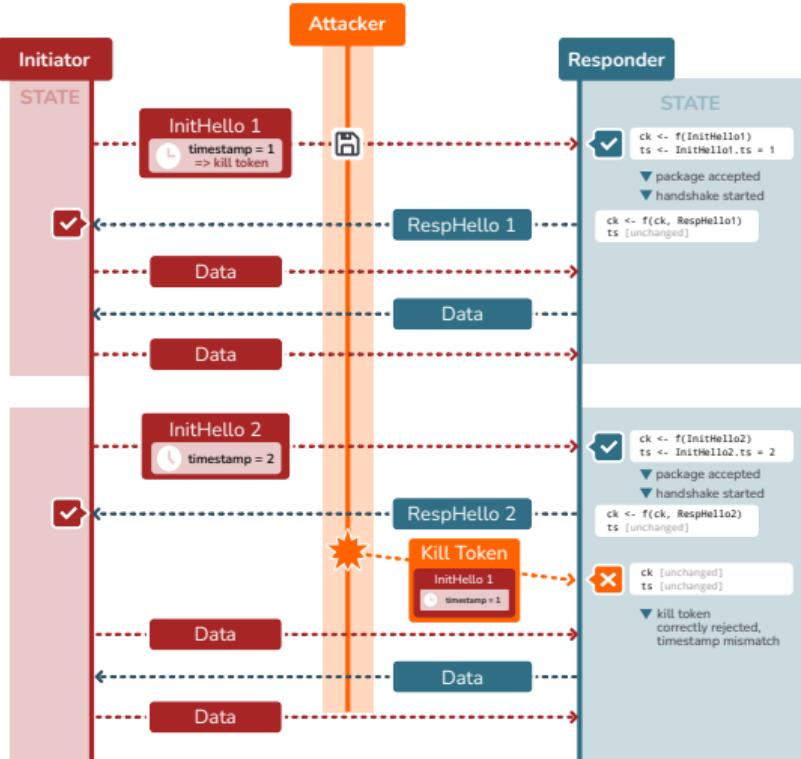
how to perform a protocol-level DoS attack against WireGuard, why wall clocks are not to be trusted and how to face replay attacks without fear.





Retransmission Protection in WireGuard

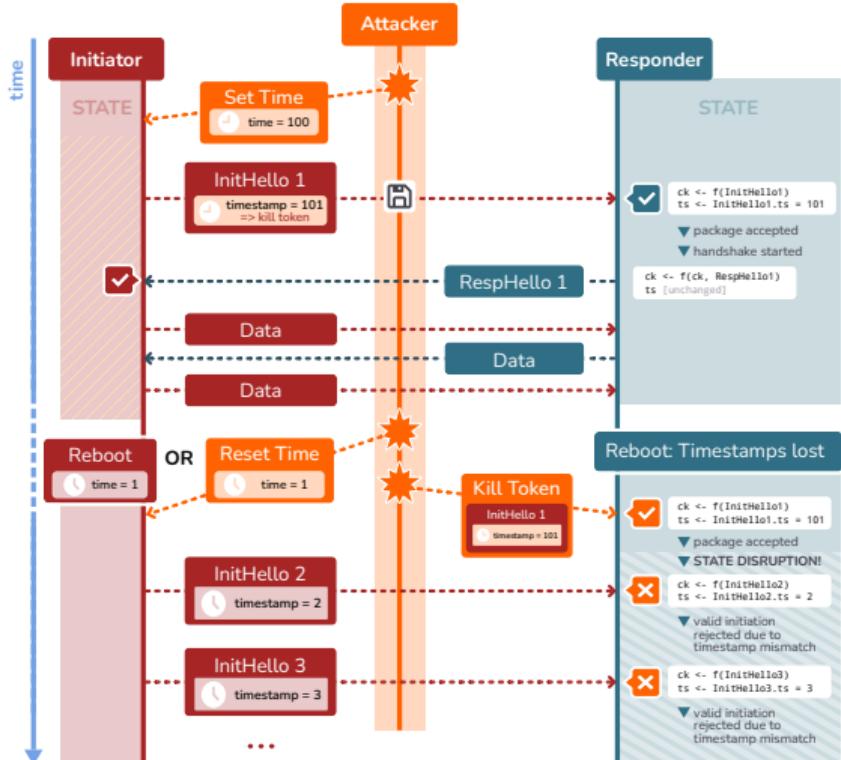
time



- replay attacks thwarted by counter
- counter is based on real-time clock
- responder is semi-stateful (one retransmission at program start may be accepted, but this does not affect protocol security)
 - ⇒ WG requires either reliable real-time clock or stateful initiator
 - ⇒ adversary can attempt replay, but this cannot interrupt a valid handshake by the initiator
- ! Assumption of reliable system time is invalid in practice!



ChronoTrigger Attack



A. Preparation phase:

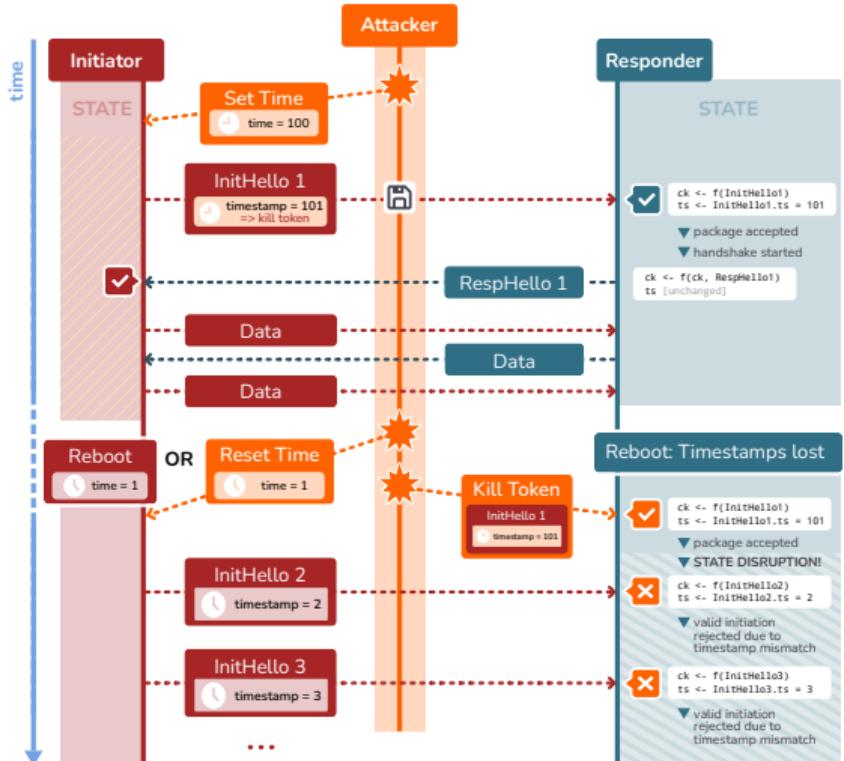
1. **Attacker** sets *initiator system time* to a future value
2. **Attacker** records *InitHello* as *KillToken* while both peers are performing a valid handshake
... both peers are being reset ...

B. Delayed execution phase:

1. **Attacker** sends *KillToken* to responder, setting their timestamp to a future value
⇒ Initiation now fails again due to timestamp mismatch



ChronoTrigger Attack

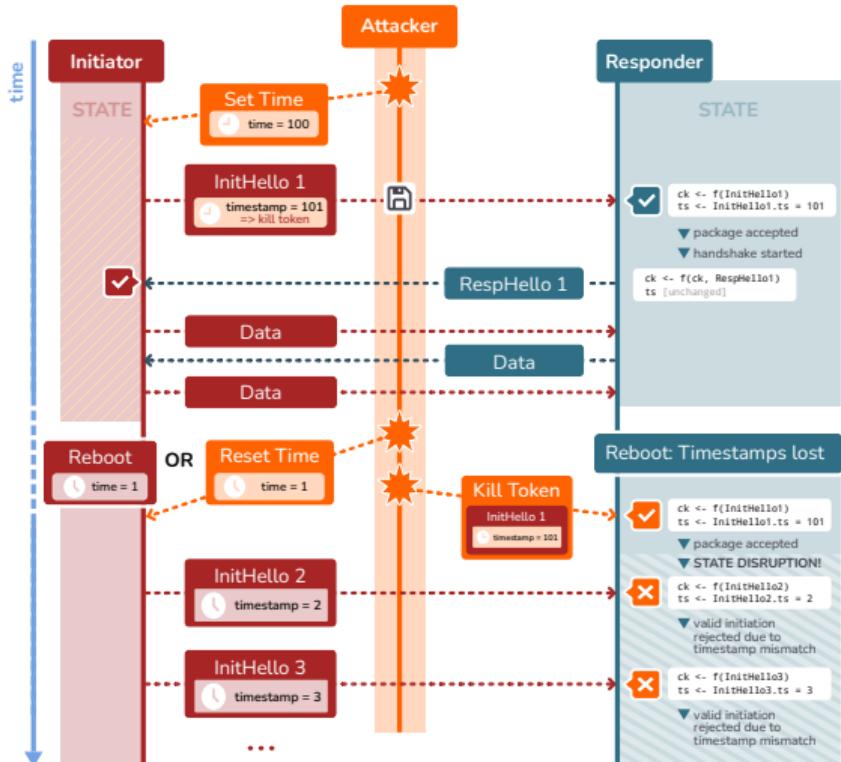


Gaining access to system time:

- Network Time Protocol is insecure, mitigations are of limited use
- ⇒ break NTP once; kill token lasts forever



ChronoTrigger Attack



Attacker gains

- extremely cheap protocol-level DoS

Preparation phase, attacker needs:

- eavesdropping of initiator packets
- access to system time

Delayed execution, attacker needs:

- no access beyond message transmission to responder



What are State Disruption Attacks?

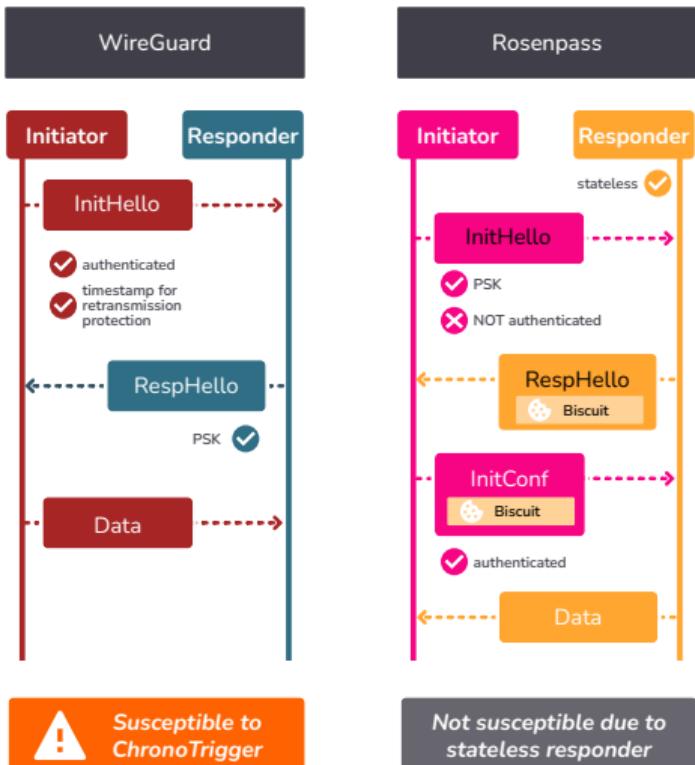


Protocol-level DoS





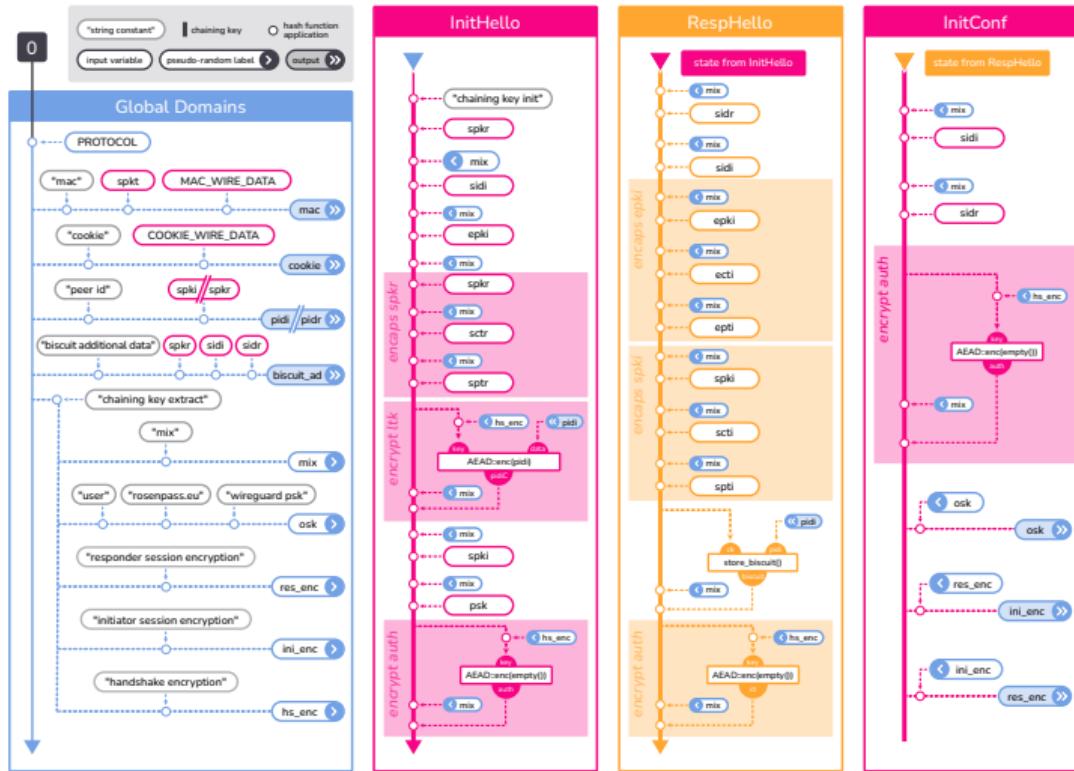
ChronoTrigger: Changes in Rosenpass



- InitHello is unauthenticated because responder still needs to encapsulate secret with initiator key
- since InitHello is unauthenticated, retransmission protection is impossible
- responder state is moved into a cookie called *Biscuit*; this renders the responder stateless
- retransmission of InitHello is now easily possible, but does not lead to a state disruption attack
- ⇒ stateless responder prevents ChronoTrigger attack



Rosenpass Key Derivation Chain: Spot the Biscuit





Rosenpass Protocol Messages: Spot the Biscuit

Envelope		bytes
type	1	
reserved	3	
payload	n	
mac	16	
cookie	16	
<hr/>		envelope n + 36
COOKIE_WIRE_DATA		MAC_WIRE_DATA

InitHello		type=0x81
sidi	4	
epki	800	
sctr	188	
pidiC	$32 + 16 = 48$	
auth	16	
<hr/>		payload 1056 + envelope 1092

RespHello		type=0x82
sidi	4	
sidi	4	
ecti	768	
stci	188	
biscuit	$76 + 24 + 16 = 116$	
auth	16	
<hr/>		payload 1096 + envelope 1132

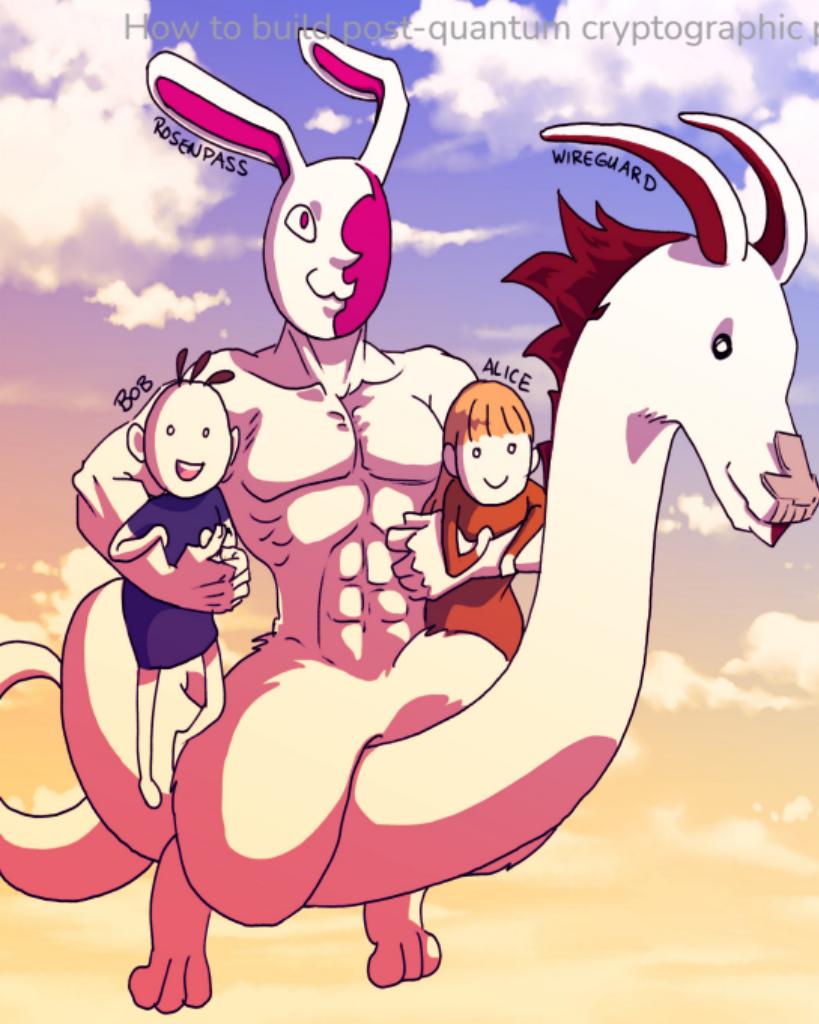
InitConf		type=0x83
sidi	4	
sidi	4	
biscuit	$76 + 24 + 16 = 116$	
auth	16	
<hr/>		payload 140 + envelope 176

EmptyData		type=0x84
sid	4	
ctr	8	
auth	16	
<hr/>		payload 28 + envelope 64

Data		type=0x85
sid	4	
ctr	8	
data	$\text{variable} + 16$	
<hr/>		payload variable + 28 + envelope variable + 64

CookieReply		type=0x86
type(0x86)	1	
reserved	3	
sid	4	
nonce	24	
cookie	$16 + 16 = 32$	
<hr/>		payload 64

biscuit		
pidi	32	
biscuit_no	12	
ck	32	
<hr/>		biscuit 76 + nonce 100 + auth code 116
data nonce auth code		



Tribulations ~ Tooling

Oh These Proof Tools

Vive la Révolution! Against the
Bourgeoisie of Proof Assistants!



Pen and Paper



Bellare and Rogaway: [BR06]

many “essentially unverifiable” proofs, “crisis of rigor”

Halevi: [Hal05]

some reasons are social, but “our proofs are truly complex”



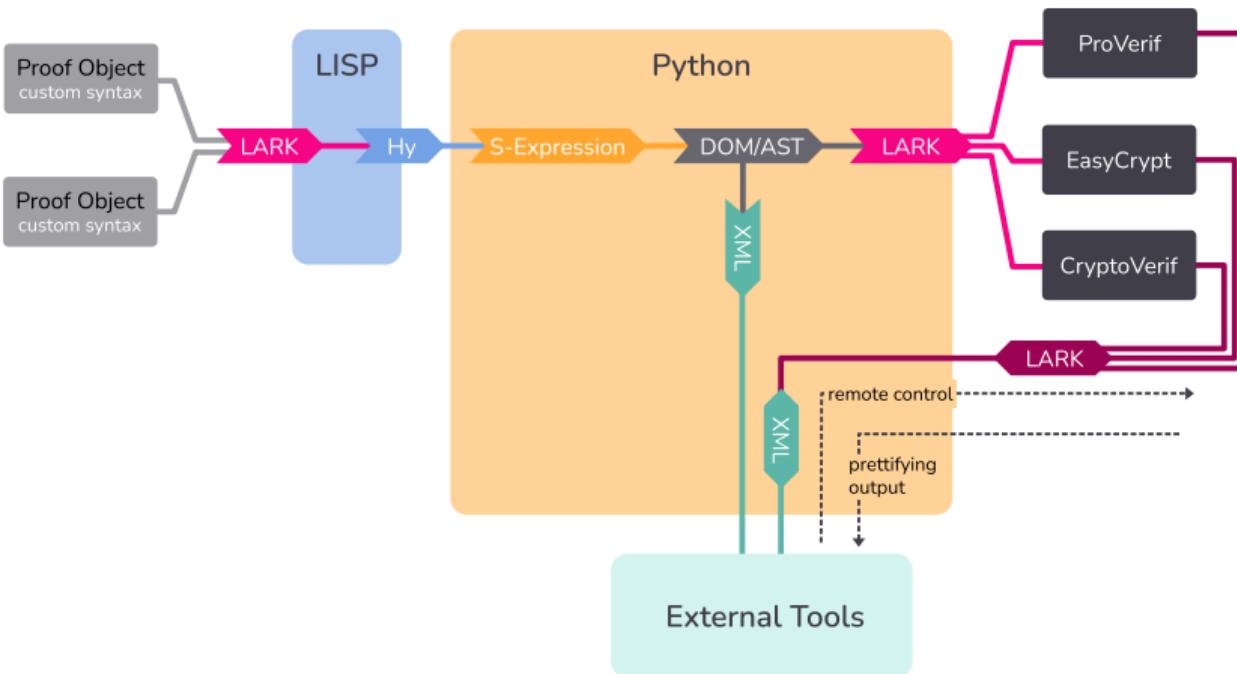
Symbolic Modeling of Rosenpass

```
~/p/rosenpass ➜ dev/karo/rwpqc-slides ? nix build .#packages.x86_64-linux.proof-prov
rosenpass-proverif-proof> unpacking sources
rosenpass-proverif-proof> unpacking source archive /nix/store/cznyv4ibwlzbh257v6lx8r8al4c
rosenpass-proverif-proof> source root is source
rosenpass-proverif-proof> patching sources
rosenpass-proverif-proof> configuring
rosenpass-proverif-proof> no configure script, doing nothing
rosenpass-proverif-proof> building
rosenpass-proverif-proof> no Makefile, doing nothing
rosenpass-proverif-proof> installing
rosenpass-proverif-proof> $ metaverif analysis/01_secrecy.entry.mpv -color -html /nix/stor
-rosenpass-proverif-proof
rosenpass-proverif-proof> $ metaverif analysis/02_availability.entry.mpv -color -html /nix
ym6dv-rosenpass-proverif-proof
rosenpass-proverif-proof> $ wait -f 34
rosenpass-proverif-proof> $ cpp -P -I/build/source/analysis analysis/01_secrecy.entry.mpv
y.i.pv
rosenpass-proverif-proof> $ cpp -P -I/build/source/analysis analysis/02_availability.entry
ility.i.pv
rosenpass-proverif-proof> $ awk -f marzipan/marzipan.awk target/proverif/01_secrecy.entry.
rosenpass-proverif-proof> $ awk -f marzipan/marzipan.awk target/proverif/02_availability.e
rosenpass-proverif-proof> 4s ✓ state coherence, initiator: Initiator accepting a RespHello
ed the associated InitHello message
rosenpass-proverif-proof> 35s ✓ state coherence, responder: Responder accepting an InitCon
ted the associated RespHello message
rosenpass-proverif-proof> 0s ✓ secrecy: Adv can not learn shared secret key
rosenpass-proverif-proof> 0s ✓ secrecy: There is no way for an attacker to learn a trusted
rosenpass-proverif-proof> 0s ✓ secrecy: The adversary can learn a trusted kem pk only by u
rosenpass-proverif-proof> 0s ✓ secrecy: Attacker knowledge of a shared key implies the key
rosenpass-proverif-proof> 31s ✓ secrecy: Attacker knowledge of a kem sk implies the key is
```

- symbolic modeling using ProVerif
- proofs treated as part of the codebase
- uses a model internally that is based on a fairly comprehensive Maximum Exposure Attacks (MEX) variant
- covers non-interruptability (resistance to disruption attacks)
- mechanized proof in the computational model is an open issue



Rosenpass going Rube-Goldberg



We will build a framework around existing tools

Keep expressivity and precision

Generate & Parse their languages

Make these tools available to other ecosystems using Python, Lisp, XML

Epilogue



Epilogue

Rosenpass

About Protocols

Talk To Us

- post-quantum secure AKE
- same security as WireGuard
- improved state disruption resistance
- transfers key to WireGuard for hybrid security
- it is possible to treat NIKEs as KEMs with DHKEM
- the GHP Combiner can be used to combine multiple KEMs
- X-Wing makes this easy
- wall clocks are not to be trusted
- adding syntax rewriting to the tool belt of mechanized verification in cryptography
- using broker architectures to write more secure system applications
- using microvms to write more secure applications
- more use cases for rosenpass

Appendix — Here Be Dragons



Bibliography

[PQWG]: <https://eprint.iacr.org/2020/379>

[GHP]: <https://eprint.iacr.org/2018/024>

[HPKE]: <https://eprint.iacr.org/2020/1499> (analysis) &
<https://www.rfc-editor.org/rfc/rfc9180.html> (RFC)

[XWING]: <https://eprint.iacr.org/2024/039>

[NOISE]: <https://www.noiseprotocol.org/noise.html>

[BR06]: <https://eprint.iacr.org/2004/331>

[Hal05]: <https://eprint.iacr.org/2005/181>



Graphics attribution

- <https://unsplash.com/photos/brown-rabbit-Efj0HGPdPKs>
- <https://unsplash.com/photos/barista-in-apron-with-hands-in-the-pockets-standing-near-the-roaster-machine-Y5qjv6Dj4w4>
- https://unsplash.com/photos/a-small-rabbit-is-sitting-in-the-grass-1_YMm4pVeSg
- <https://unsplash.com/photos/yellow-blue-and-black-coated-wires-iOLHAlaxpDA>
- <https://unsplash.com/photos/gray-rabbit-XG06d9Hd2YA>
- <https://unsplash.com/photos/big-ben-london-MdJq0zFUwrw>
- https://unsplash.com/photos/white-rabbit-on-green-grass-u_kMWN-BWYU
- <https://unsplash.com/photos/3-brown-bread-on-white-and-black-textile-WJDsVFwPjRk>
- <https://unsplash.com/photos/a-pretzel-on-a-bun-with-a-blue-ribbon-ymr0s7z6Ykk>
- <https://unsplash.com/photos/white-and-brown-rabbit-on-white-ceramic-bowl-rcfp7YEnJrA>