



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

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



This is the Plan

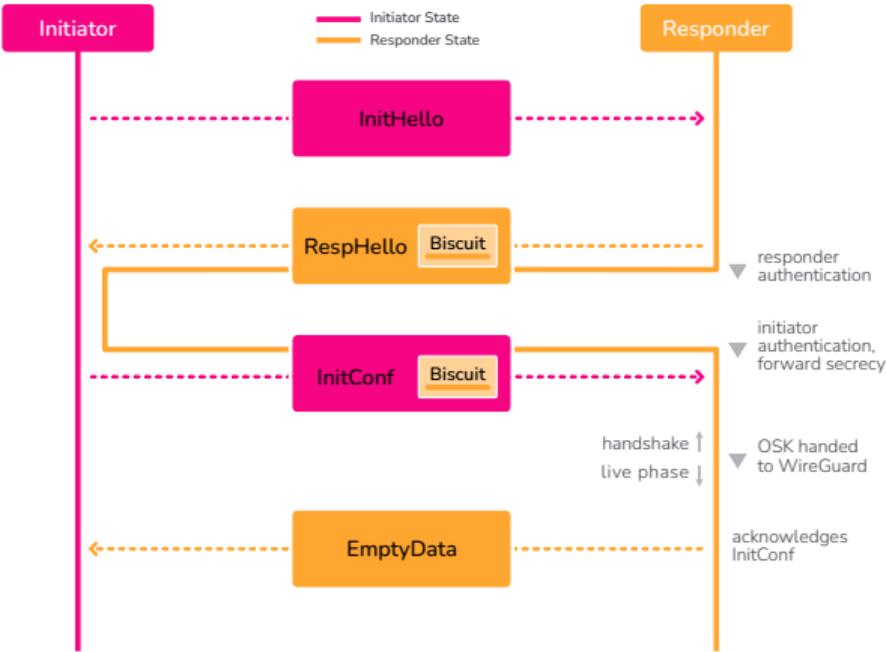
1. **Introducing Rosenpass**, briefly.
2. **The design of Rosenpass** and basics about post-quantum protocols.
3. **Hybrid Security** how it can be done and how we do it.
4. **ChronoTrigger Attack** and not trusting wall clocks.
5. **Protocol proofs** – big old rant!





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 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...

...susceptible to attacks from
quantum computers.

Post-quantum
cryptography is...

...not susceptible to attacks
from quantum computers.

Hybrid cryptography
combines...

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

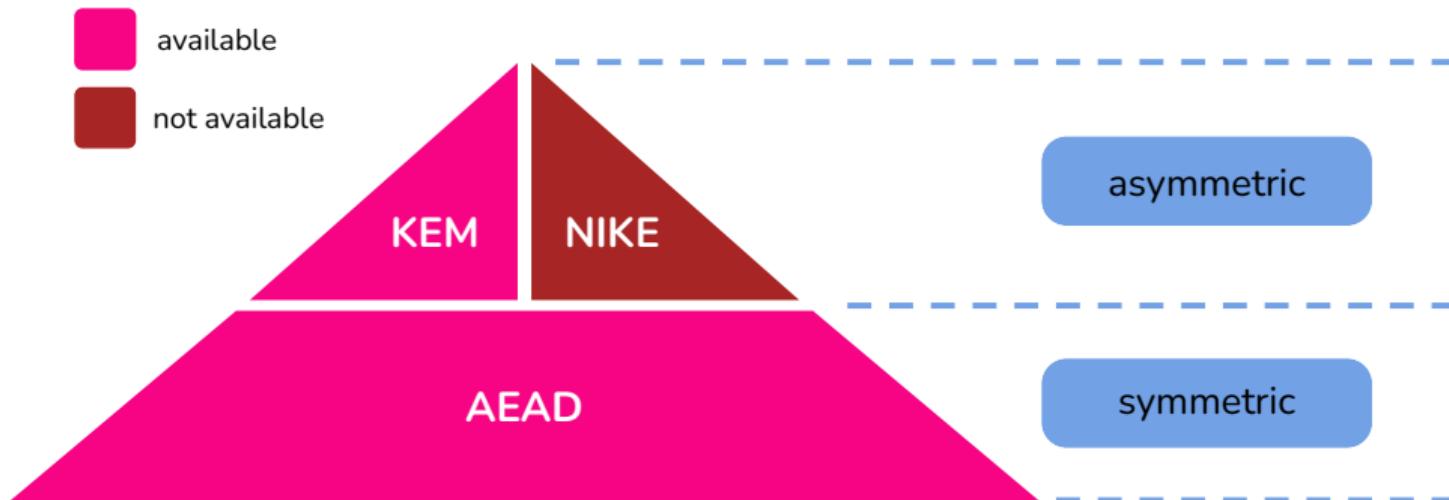
- Specifically, to *Shor's Algorithm*
- Quite fast
- Widely widely trusted

- 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) = nake(sk2, pk1));
```

Aka. Diffie-Hellman. I don't know a good analogy, but 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 checked by making you use it, therefor you also have a particular asymmetric secret key."

Secrecy

"The data we exchange can not 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 can not 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 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.



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

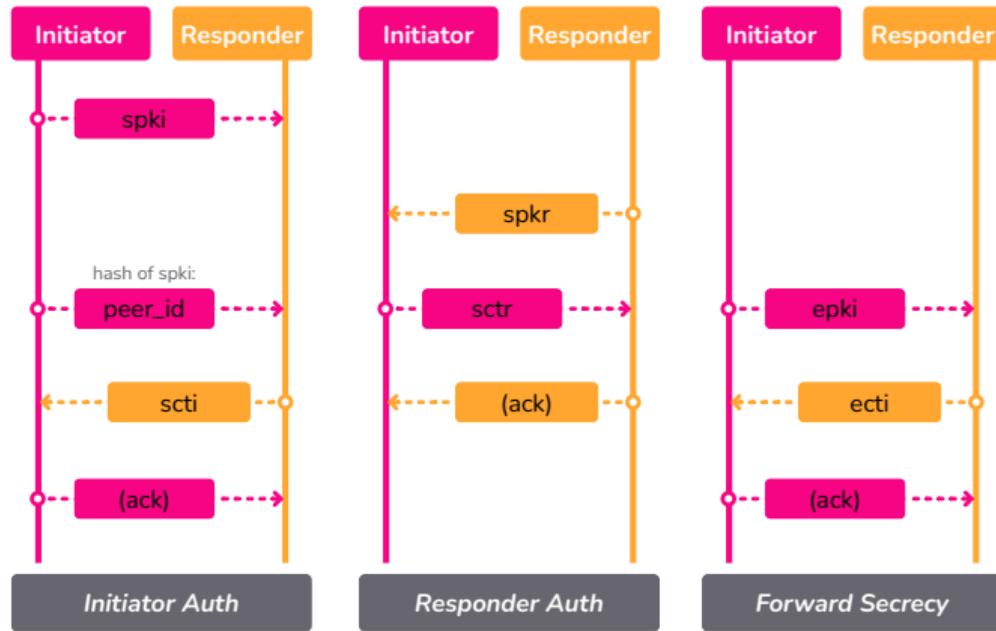
```

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);
}

```

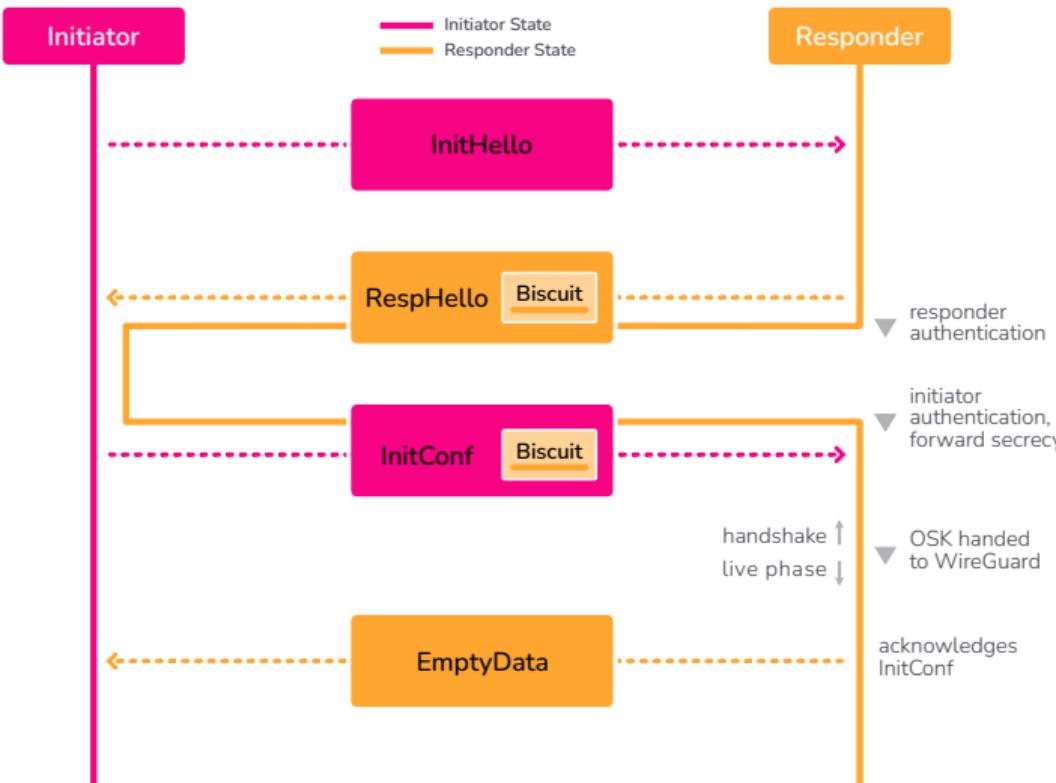


Rosenpass Kex Exchange Parts





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

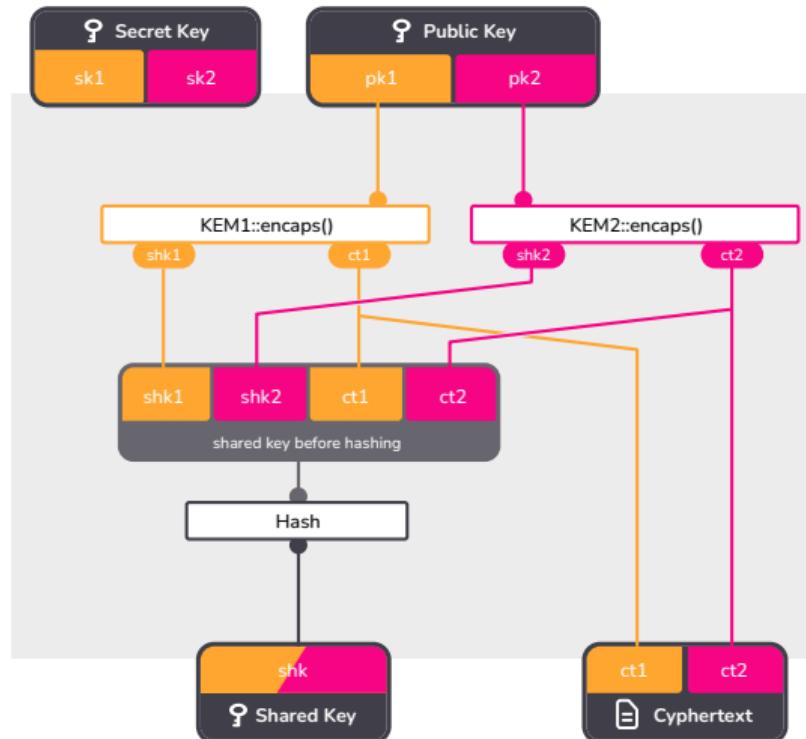
In the following slides you will learn...

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



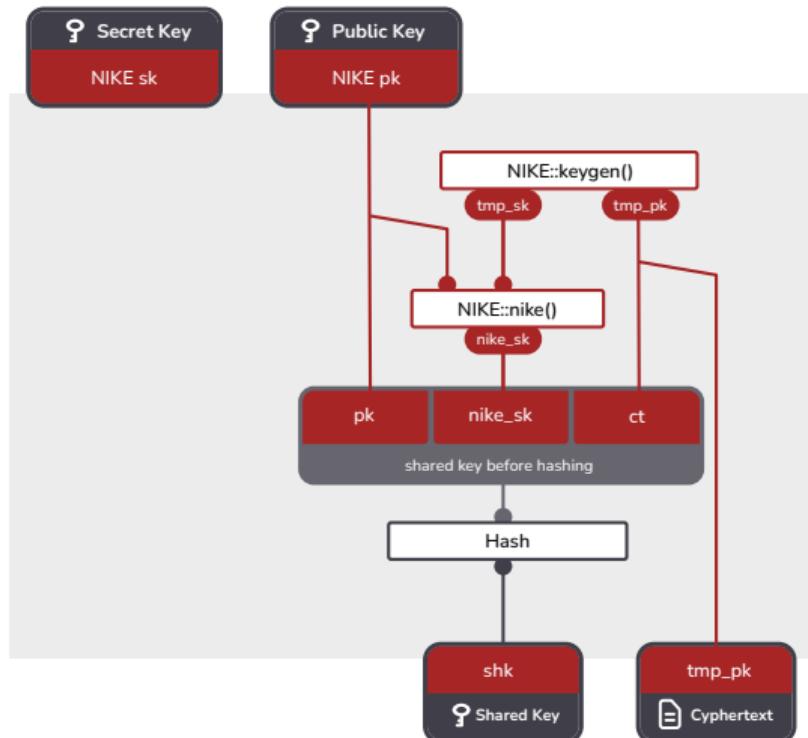


Combining two KEMs with the GHP-Combiner



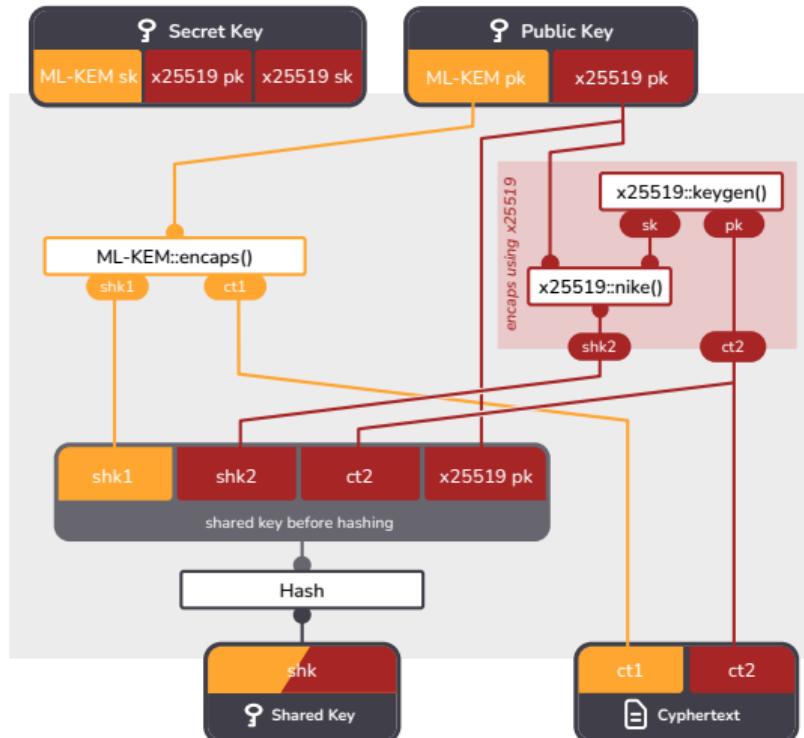


Turning a NIKE into a KEM



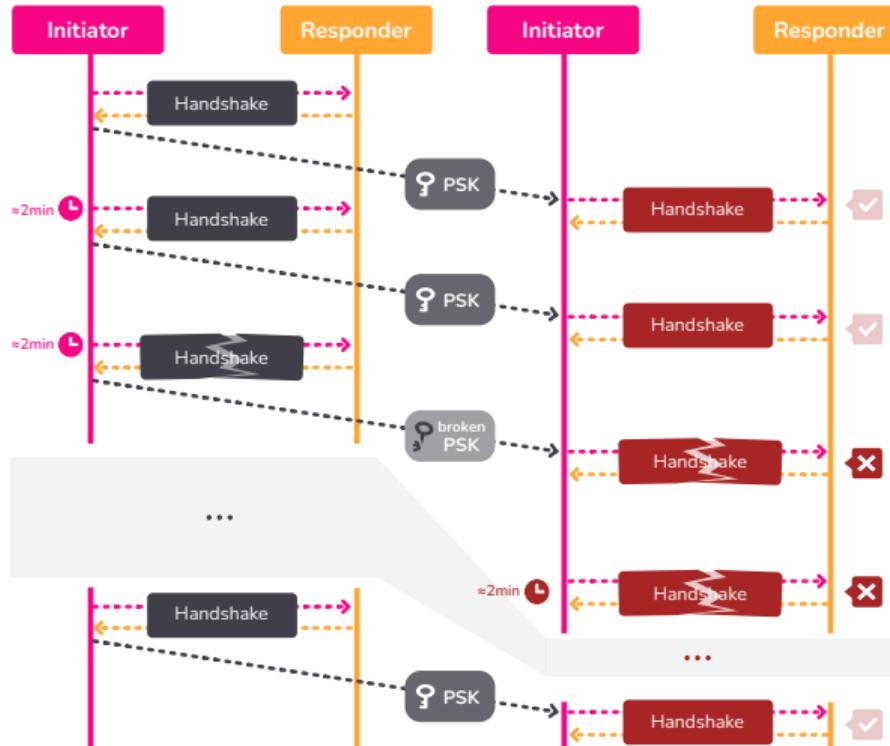


X-Wing





Rosenpass & WireGuard Hybridization





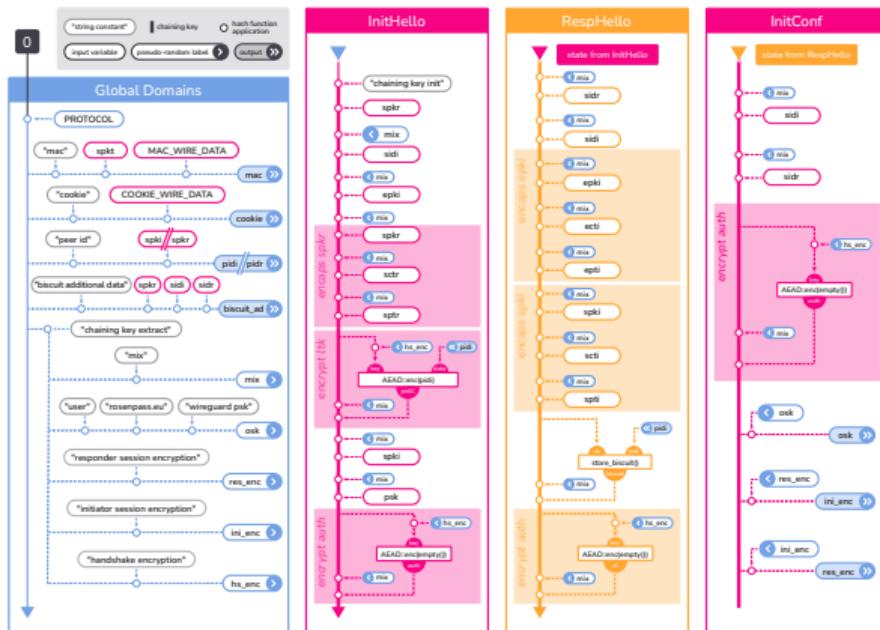
Full Protocol Reference in the Whitepaper

Initiator Code	Responder Code	Comments
<p>1 InitHello { sidi, epki, sctr, pidIC, auth }</p> <pre> Line Variables -- Action R#1 ck ← (hash("chaining key init", spk) R#2 sid ← random_session_id(); R#3 eskl, epki ← EKEM-keygen(); R#4 mix(sid, epki); R#5 sctr ← encaps_and_mix<SKEM>(spkr); R#6 pidIC ← encrypt_and_mix(pid); R#7 mix(epki, pid); R#8 auth ← encrypt_and_mix(empty()); </pre>	<p>2 ck ← (hash("chaining key init", spk)</p> <pre> Variables -- Action Line R#1 ck ← (hash("chaining key init", spk) R#2 mix(sid, epki); R#3 decaps_and_mix<SKEM>(jskr, spkr, ct1) R#4 spki, pk ← lookup_peer_decrypt_and_mix(pidC) R#5 mix(epki, pk); R#6 decrypt_and_mix(auth) </pre>	<p>Comment</p> <p>Initialize the chaining key, and bind to the responder's public key.</p> <p>The session ID is used to associate packets with the handshake state.</p> <p>Generate fresh ephemeral keys, for forward secrecy.</p> <p>InitHello includes sid and epki as part of the protocol transcript, and so we mix them into the chaining key to prevent tampering.</p> <p>Key encapsulation using the responder's public key. Mixes public key, shared static key, and session ID to prevent replay attacks and identify the responder.</p> <p>Tell the responder who the initiator is by transmitting the peer ID.</p> <p>Ensure the responder has the correct view on spki. Mix in the PSK as optional static symmetric key, with epki and spkr serving as nonces.</p> <p>Add a message authentication code to ensure both participants agree on the session state and protocol transcript at this point.</p>
<p>4 RespHello [sidr, sidi, ecti, scti, biscuit, auth]</p> <pre> Line Variables -- Action R#1 sidr ← random_session_id(); R#2 ck ← lookup_session(sid); R#3 mix(sidr, sid); R#4 decaps_and_mix<SKEM>(eski, epki, ecti); R#5 decaps_and_mix<SKEM>(jski, spki, scti); R#6 mix(biscuit); R#7 decrypt_and_mix(auth) </pre>	<p>3 sidi ← random_session_id()</p> <pre> Variables -- Action Line R#1 sidr ← random_session_id() R#2 mix(sidr, sid); R#3 ecti ← encaps_and_mix<SKEM>(spkr); R#4 scti ← encaps_and_mix<SKEM>(jskr); R#5 biscuit ← store_biscuit(); R#6 auth ← encrypt_and_mix(empty()); </pre>	<p>Comment</p> <p>Responder generates a session ID.</p> <p>Initiator looks up their session state using the session ID it generated.</p> <p>Mix both session IDs as part of the protocol transcript.</p> <p>Key encapsulation using the ephemeral key, to provide forward secrecy.</p> <p>Key encapsulation using the initiator's static key, to authenticate the initiator, and non-forward-secret confidentiality.</p> <p>The responder transmits their state to the initiator in an encrypted container to avoid having to store state.</p> <p>Add a message authentication code for the same reason as above.</p>
<p>5 InitConf { sidi, sidr, biscuit, auth }</p> <pre> Line Variables -- Action R#1 biscuit_no ← load_biscuit(biscuit); R#2 encrypt_and_mix(empty()); R#3 mix(sidr, sid); R#4 auth ← encrypt_and_mix(empty()); R#5 assert(biscuit_no > biscuit_used); R#6 biscuit_used ← biscuit_no; R#7 enter_livel; </pre>	<p>6 biscuit_no ← load_biscuit(biscuit);</p> <pre> Variables -- Action Line R#1 biscuit_no ← load_biscuit(biscuit); R#2 encrypt_and_mix(empty()); R#3 mix(sidr, sid); R#4 decrypt_and_mix(auth); R#5 assert(biscuit_no > biscuit_used); R#6 biscuit_used ← biscuit_no; R#7 enter_livel; </pre>	<p>Comment</p> <p>Responder loads their biscuit. This restores the state from after R#R7.</p> <p>Responder recomputes R#R7, since this step was performed after biscuit encoding.</p> <p>Mix both session IDs as part of the protocol transcript.</p> <p>Message authentication code for the same reason as above, which in particular ensures that both participants agree on the final chaining key.</p> <p>Biscuit replay detection.</p> <p>Biscuit replay detection.</p> <p>Derive the transmission key, and the output shared key for use as WireGuard's PSK.</p>

rosenpass.eu/docs
rosenpass.eu/whitepaper.pdf

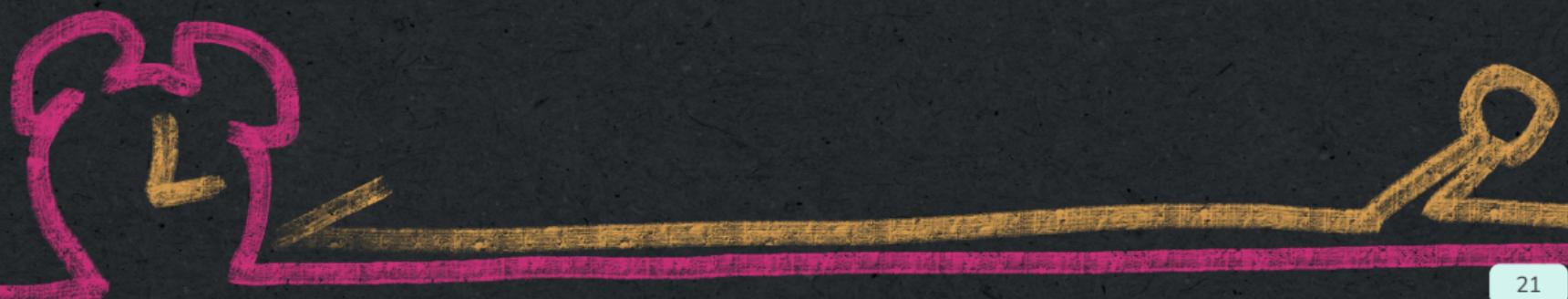


Rosenpass Key Derivation Chain



Trials ~ Attacks found

ChronoTrigger





In the following slides, you will learn...

...that denial of service can happen on the level of cryptography protocols!

...that the wall clock is not to be trusted.

...how to accept replay attacks and face them without fear!





What are State Disruption Attacks?



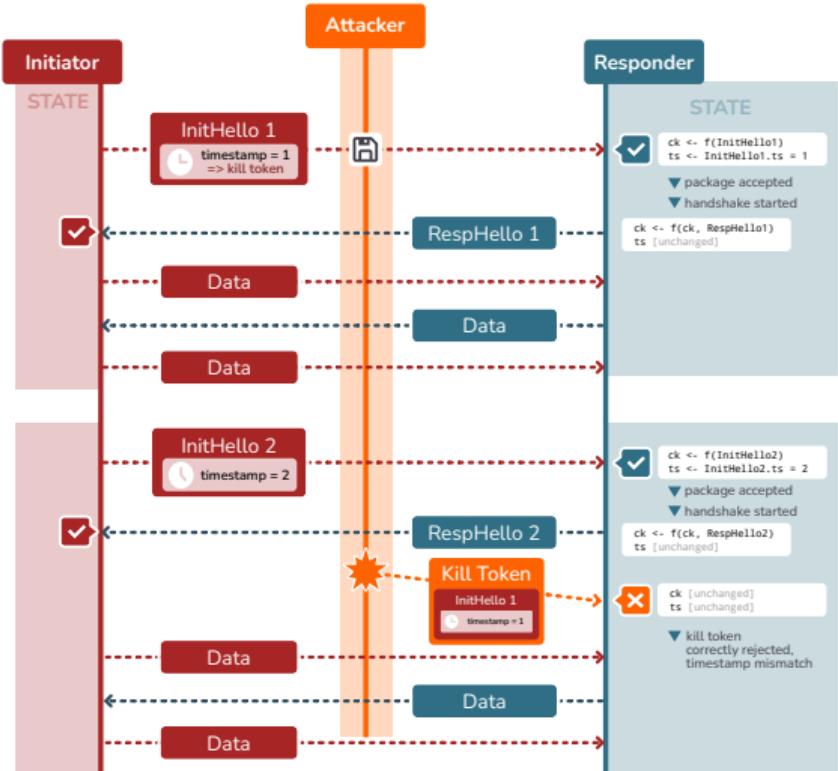
Protocol level DOS





Retransmission Protection in WireGuard

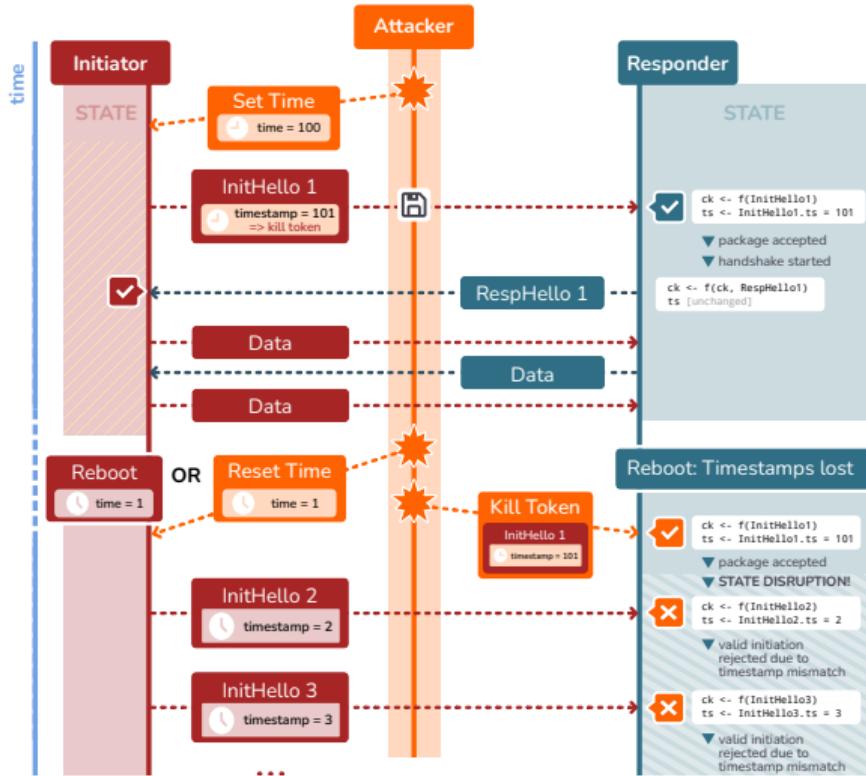
time



- 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 timer



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

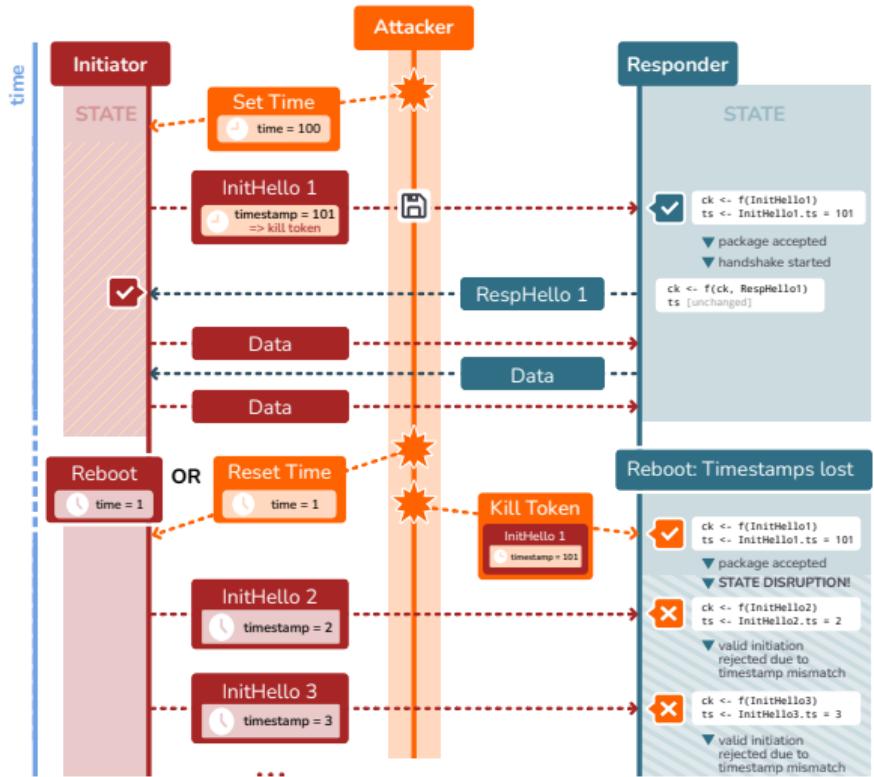
... both peers are being reset ...

B. Delayed execution phase:

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



ChronoTrigger Attack

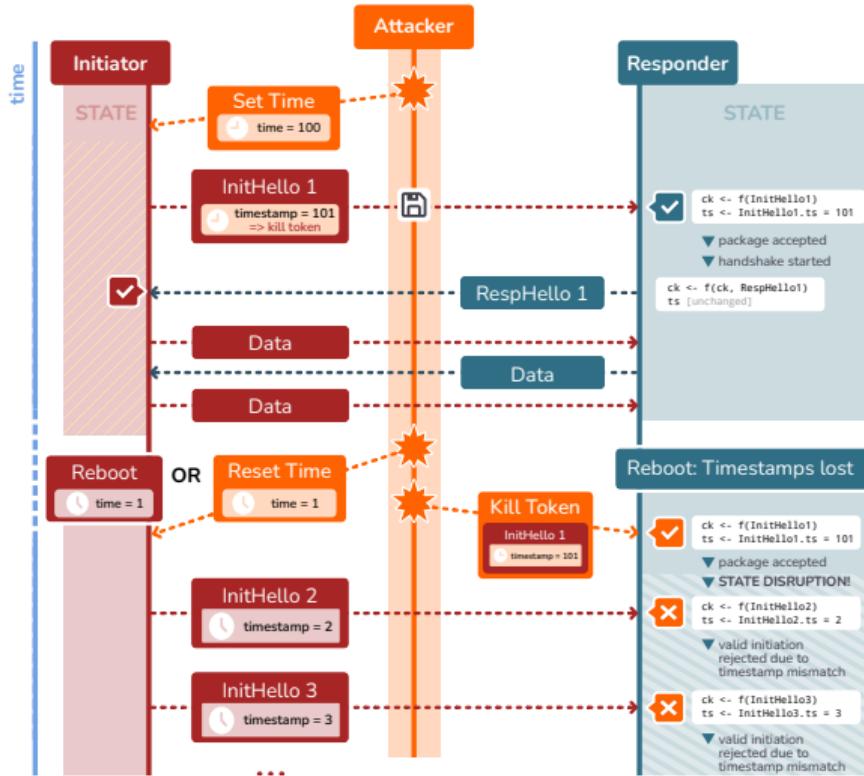


Gaining access to system time:

- 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:

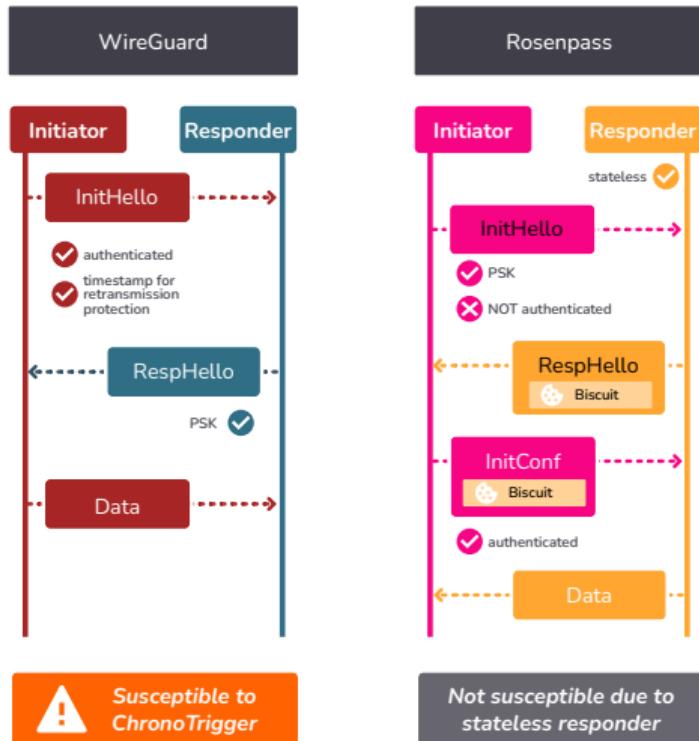
- Eavesdropping of initiator packets
- Access to system time

Delayed execution, attacker needs:

- No access beyond message transmission to responder



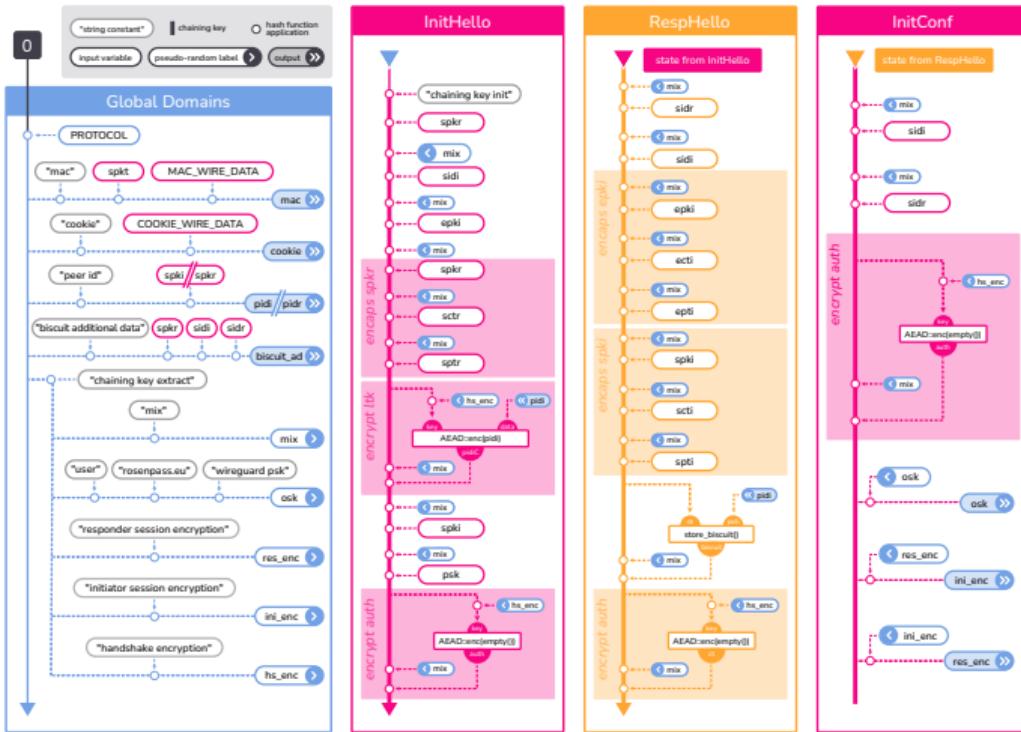
ChronoTrigger: Changes in Rosenpass



- **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
COOKIE_WIRE_DATA		
MAC_WIRE_DATA		
type	1	
reserved	3	
payload	n	
mac	16	
cookie	16	
envelope	n + 36	

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

RespHello		type=0x82
sidr	4	
sidi	4	
ecti	768	
scti	188	
biscuit	76 + 24 + 16 = 116	
auth	16	
		payload 1096
		+ envelope 1132

InitConf		type=0x83
sidi	4	
sidr	4	
biscuit	76 + 24 + 16 = 116	
auth	16	
		payload 140
		+ envelope 176

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

Data		type=0x85
sid	4	
ctr	8	
data	variable + 16	
		payload variable + 28
		+ envelope variable + 64

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

biscuit	
pidi	32
biscuit_no	12
ck	32
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/rwppc-slides ? ➤ nix build .#packages.x86_64-linux.p  
rosenpass-proverif> unpacking sources  
rosenpass-proverif> unpacking source archive /nix/store/cznyv4ibwlzbh257v6l  
rosenpass-proverif> source root is source  
rosenpass-proverif> patching sources  
rosenpass-proverif> configuring  
rosenpass-proverif> no configure script, doing nothing  
rosenpass-proverif> building  
rosenpass-proverif> no Makefile, doing nothing  
rosenpass-proverif> installing  
rosenpass-proverif> $ metaverif analysis/01_secrecy.entry.mpv -color -html  
-rosenpass-proverif  
rosenpass-proverif> $ metaverif analysis/02_availability.entry.mpv -color -  
ym6dv-rosenpass-proverif  
rosenpass-proverif> $ wait -f 34  
rosenpass-proverif> $ cpp -P -I/build/source/analysis analysis/01_secrecy.e  
y.i.pv  
rosenpass-proverif> $ cpp -P -I/build/source/analysis analysis/02_availabil  
ity.entry.i.pv  
rosenpass-proverif> $ awk -f marzipan/marzipan.awk target/proverif/01_secre  
y  
rosenpass-proverif> $ awk -f marzipan/marzipan.awk target/proverif/02_avail  
ability  
rosenpass-proverif> 4s ✓ state coherence, initiator: Initiator accepting a  
lled the associated InitHello message  
rosenpass-proverif> 35s ✓ state coherence, responder: Responder accepting a  
lled the associated RespHello message  
rosenpass-proverif> 0s ✓ secrecy: Adv can not learn shared secret key  
rosenpass-proverif> 0s ✓ secrecy: There is no way for an attacker to learn  
rosenpass-proverif> 0s ✓ secrecy: The adversary can learn a trusted kem pk  
rosenpass-proverif> 0s ✓ secrecy: Attacker knowledge of a shared key implie  
rosenpass-proverif> 31s ✓ secrecy: Attacker knowledge of a kem sk implies t
```

- 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



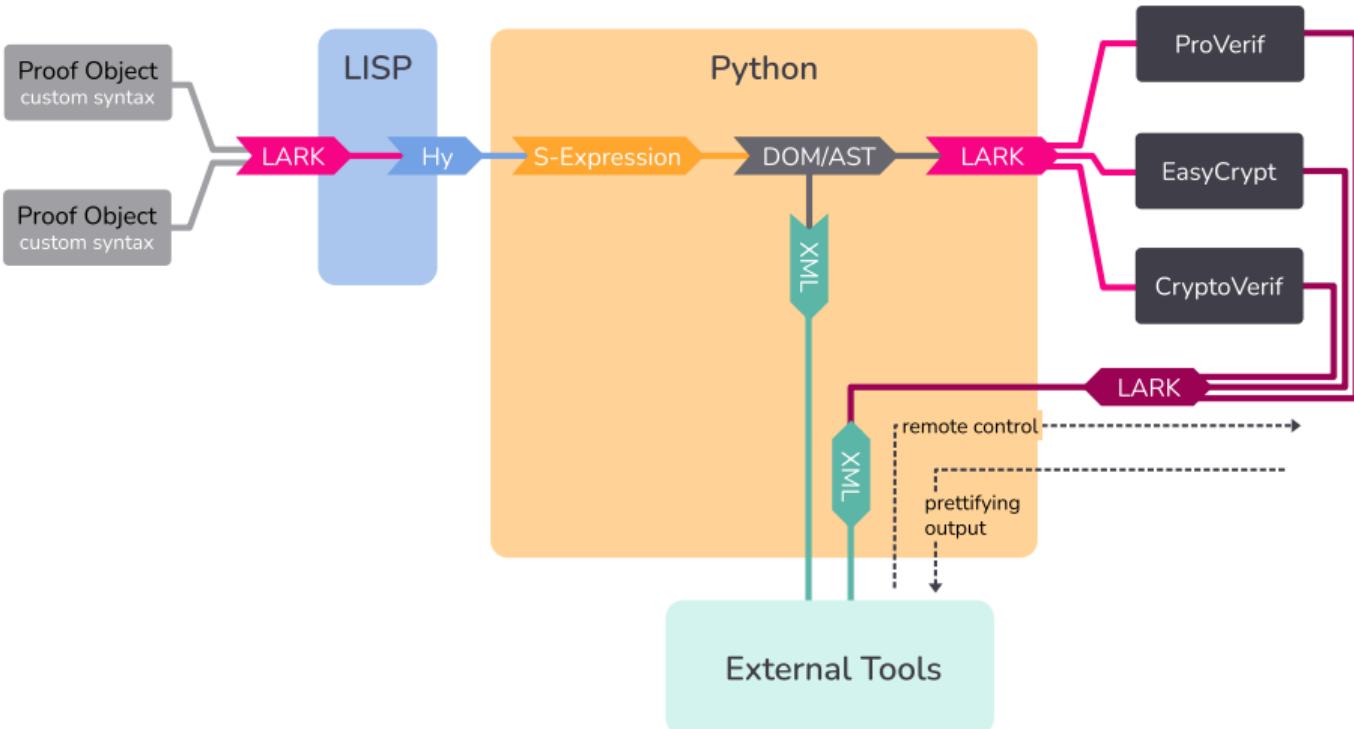
The Day Language Came into My Life

Everything had a name, and each name gave birth to a new thought.

Helen Keller (1880-1968) in The Day Language Came into My Life



Rosenpass going Rube-Goldberg



Idea: Build a framework around existing tools

Keep expressivity and preciseness

Generate and parse their languages

Open them up to ecosystems in Python, Lisp, XML

Epilogue



Epilogue

Rosenpass

- Post-quantum secure AKE
- Same security as WireGuard
- Improved state disruption resistance
- Transfers key to WireGuard for hybrid security

About Protocols

- 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

Talk To Us

- 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

Appendix — Here Be Dragons



Bibliography

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



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>

Random slides — The dragon just
ate you!



Rosenpass and WireGuard: Advanced Security

Limited Stealth:

- Protocol should not respond without pre-auth.
- Proof of IP ownership (cookie mechanism) prevents full stealth
- Adv. needs to know responder public key

CPU DOS mitigation:

- Attacker should not easily trigger public key operations
- Preventing CPU exhaustion using network amplification
- Proof of IP ownership

Limited Identity Hiding:

- Adversary cannot recognize peers unless their public key is known
- This is incomplete!

Triumphs ~ Secrecy & Non-Interruptability

Modeling of Rosenpass

Using ProVerif



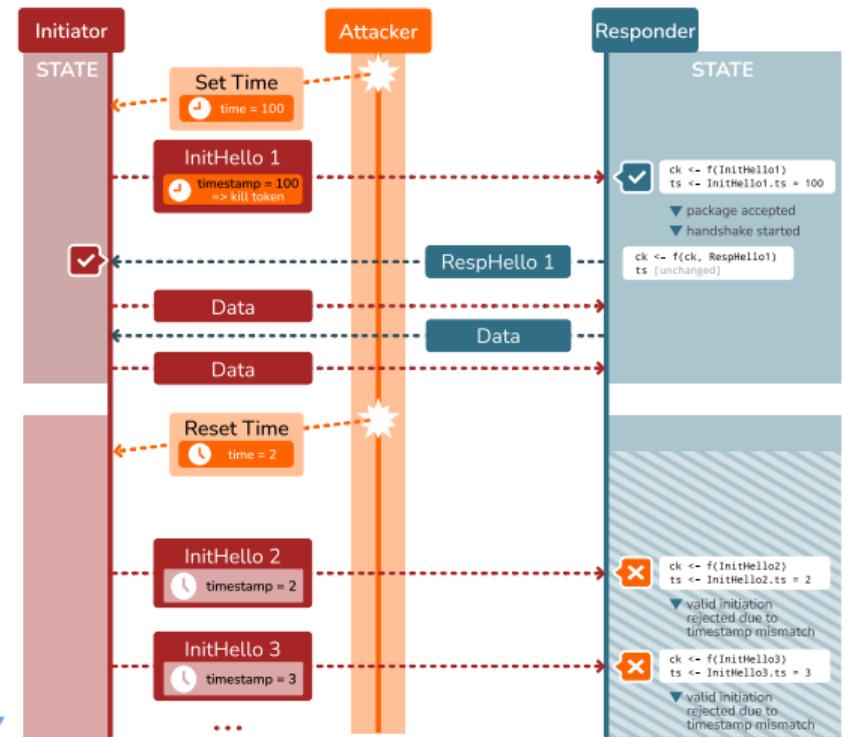
Non-Interruptability: More Formally

For every pair of traces t_{\min}, t_{\max} where trace t_{\max} can be formed by insertion of messages/oracle calls into t_{\min} , the result of t_{\min} and t_{\max} should remain the same.

- Let Result be the set of possible protocol results
- Let Trace be the set of possible protocol traces
- Let $\text{res}(t) : \text{Trace} \rightarrow \text{Result}$ determine the protocol result given $t : \text{Trace}$
- Let $t_1 \supseteq t_2 : \text{Trace} \rightarrow \text{Trace} \rightarrow \text{Prop}$ denote that t_2 can be formed by insertion of elements into t_1
- $\forall(t_{\min}, t_{\max}) : \text{Trace} \times \text{Trace}; t_{\min} \supseteq t_{\max} \rightarrow \text{res}(t_{\min}) = \text{res}(t_{\max})$



ChronoTrigger Attack: Immediate Execution



A. Preparation phase:

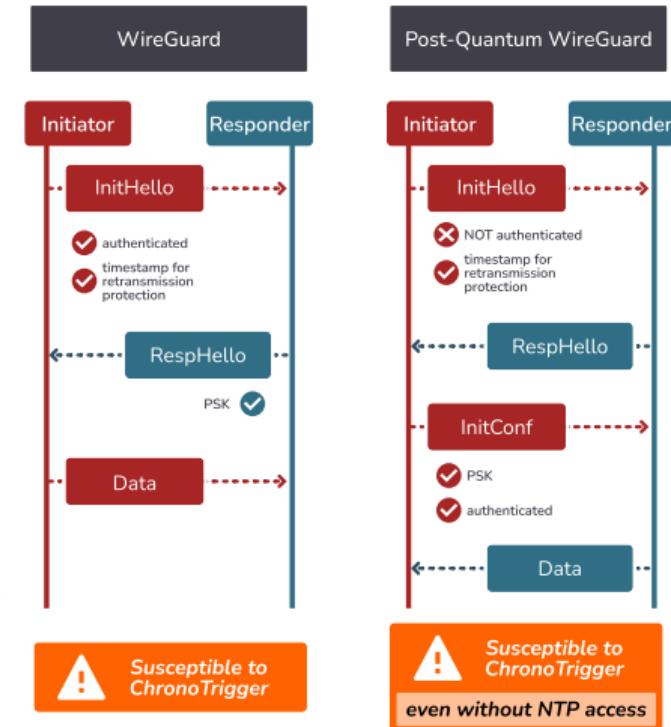
1. **Attacker** sets *initiator system time* to a future value
2. **Attacker** waits while both peers are performing a valid handshake

B. Direct execution phase:

1. **Attacker** lets system time on initiator reset
=> Initiation now fails due to counter mismatch



ChronoTrigger: Changes in Post-Quantum WG

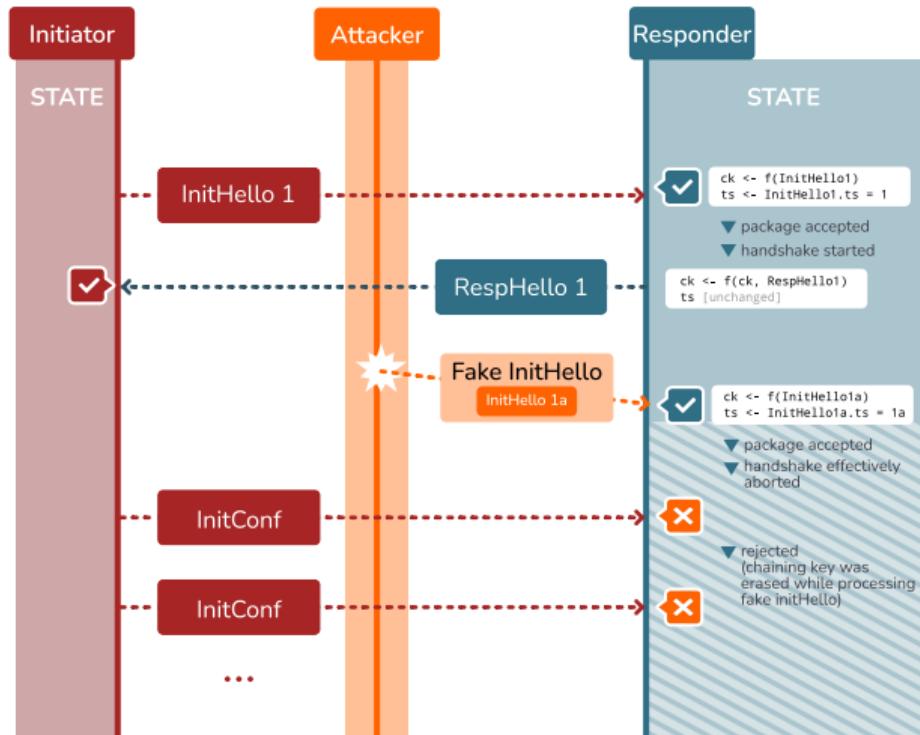


- *InitHello* is unauthenticated
- Retransmission counter is kept
- PQWG assumes a pre-shared key to authenticate *InitHello* instead (the authors recommend deriving the PSK from both public keys)
- PSK evaluated twice, during *InitHello* and *InitConf* processing



ChronoTigger against Post-Quantum WireGuard

time ↓



No PSK/Public keys as PSK

- Attacker needs access to public keys
- The attack is trivial (attacker just forges *InitHello*)

With PSK

- Replay attack with NTP access from classic WireGuard still applies



Trials ~ Attacks found

CookieCutter



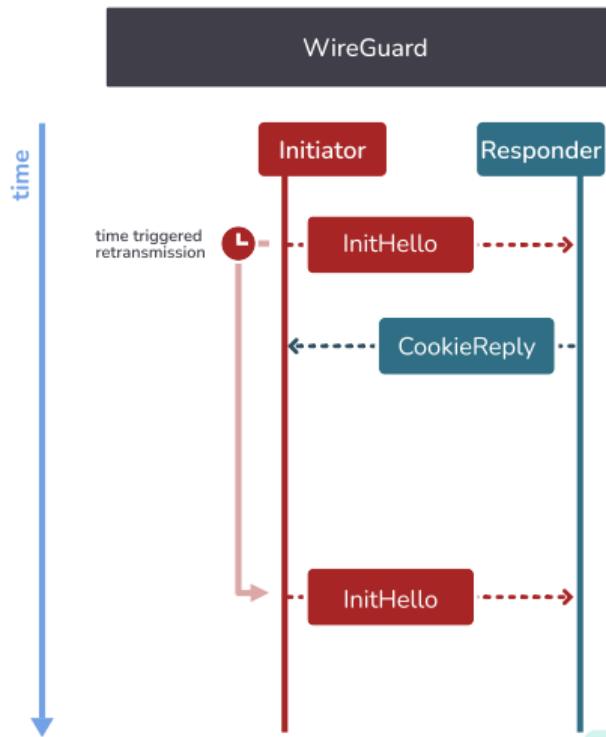


CookieCutter Attack

I am under load. Prove that you are not using IP address impersonation before I process your handshake!

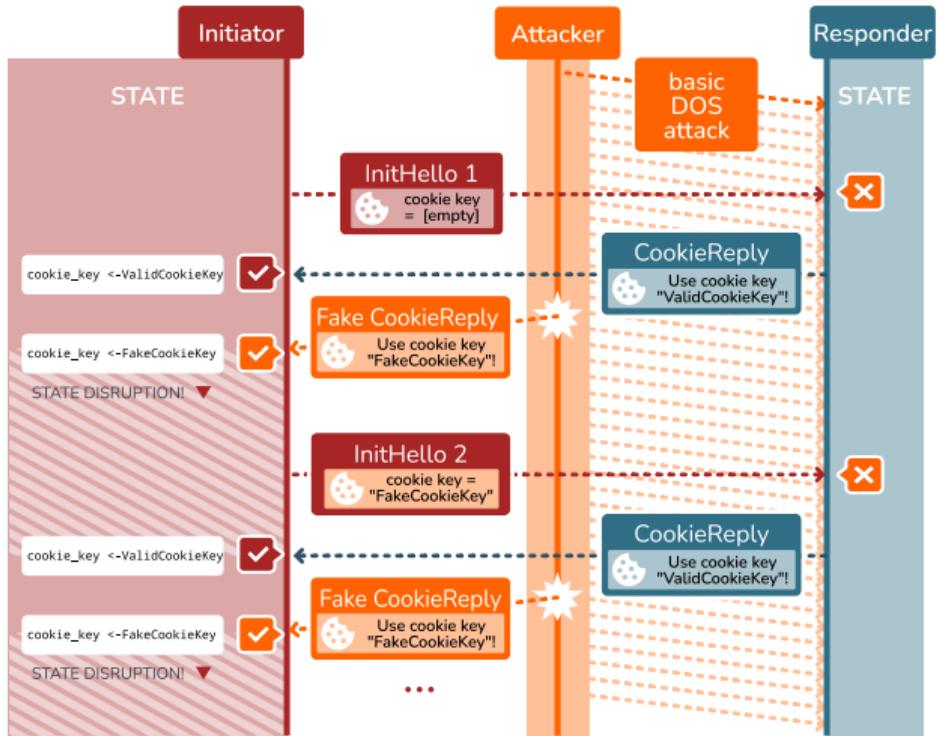
This message contains a *cookie* key. Use it to prove that you can receive messages sent to your address when retransmitting your *InitHello* packet.

A **WireGuard CookieReply**, ca. 2014





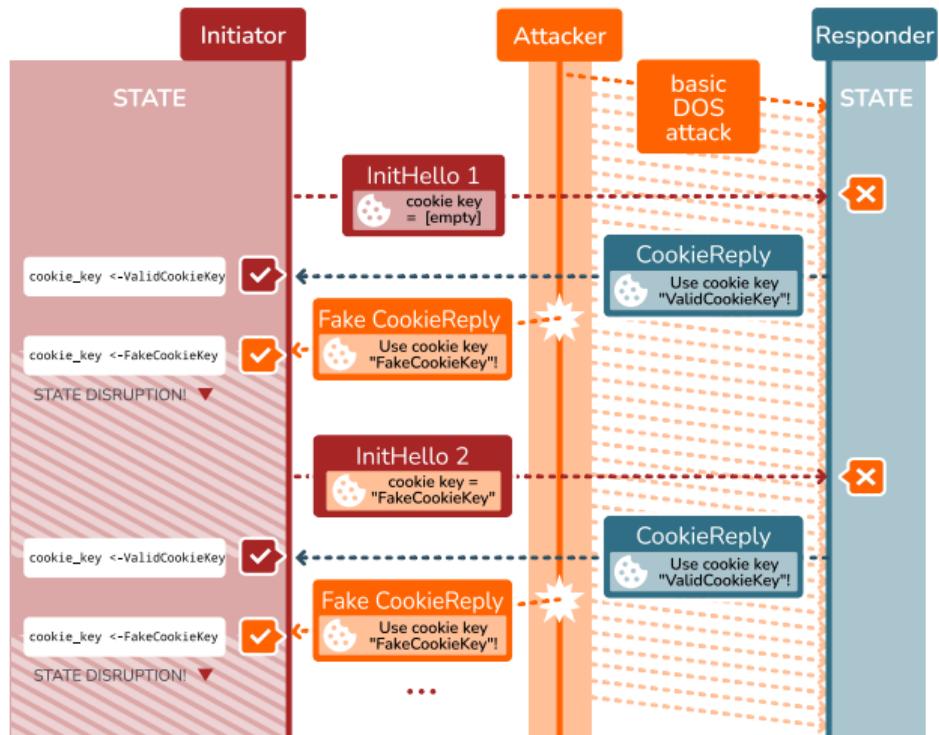
CookieCutter Attack



1. **Attacker** begins continuous DOS attack against responder
2. **Initiator** begins handshake, sends *InitHello*
3. **Responder** replies with **CookieReply**
CookieReply: I am under load.
 Prove you are not using an IP spoofing attack with this cookie key.
4. **Initiator** Initiator stores cookie key and waits for their retransmission timer
5. **Attacker** forges a cookie reply with a fake cookie key
6. **Initiator** Initiator overwrites the valid



CookieCutter Attack



Attacker gains:

- Cheap protocol-level DOS

Attacker needs:

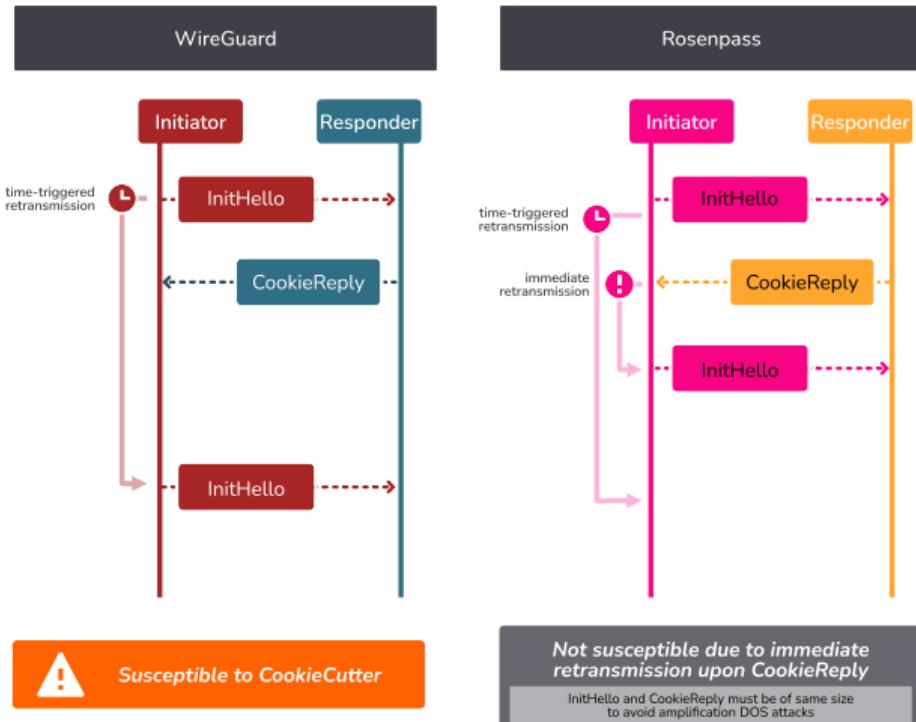
- Knowledge of public keys
- Good timing

Role switching:

- WireGuard sometimes uses role switching
- To account for that, the attack can be performed against both peers



CookieCutter: Post-Quantum WG & Rosenpass



Post-Quantum WireGuard

- No change.

Rosenpass

- Immediate retransmission of *InitHello* upon receiving *CookieReply*
 - CookieReply* and *InitHello* must be of same size to prevent DOS amplification attacks
- ⇒ Rosenpass is protected from CookieCutter attacks

Trials ~ Advanced Security Properties

Knock Patterns





Rosenpass and WireGuard: Advanced Security

CPU DOS mitigation:

- No change on the protocol level.
- Slightly worsened in practice because PQ operations are more expensive than elliptic curves

Limited Stealth:

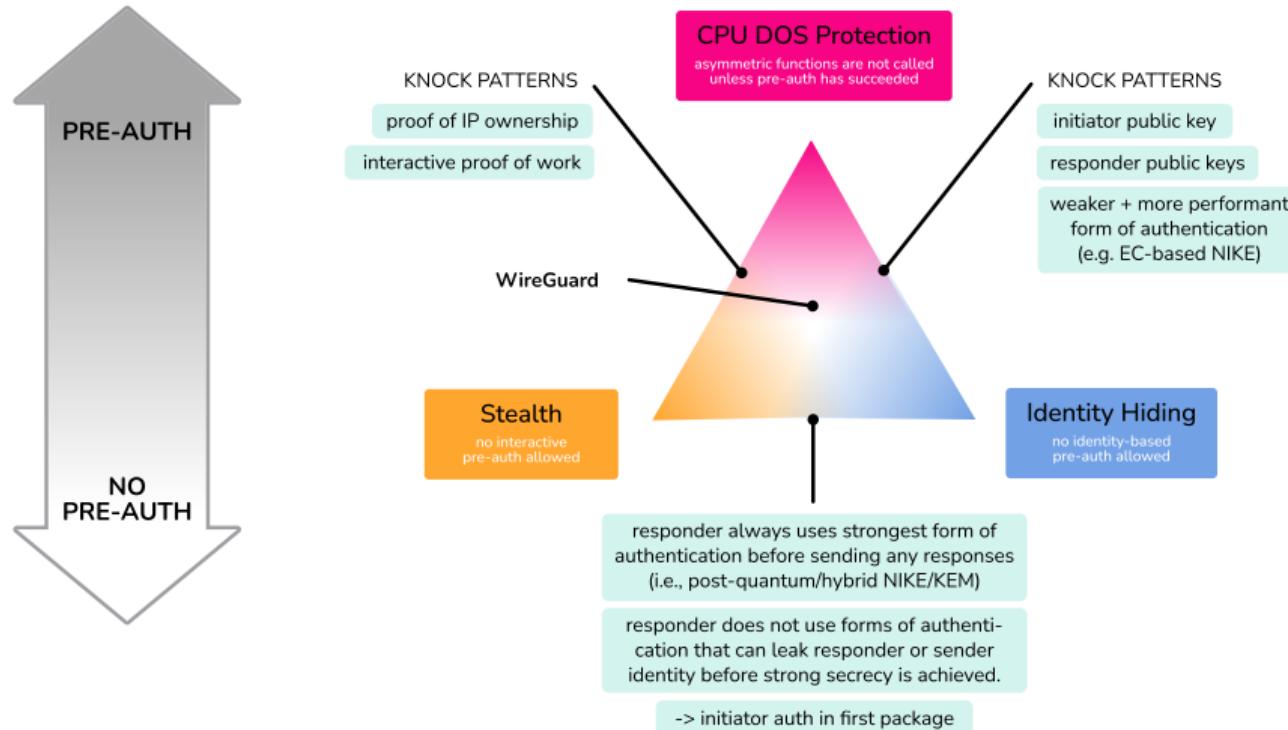
- No change in Rosenpass, but we should have **full stealth!**
 - ⇒ Remove cookie mechanism?
- This would affect the CPU DOS mitigation too much.

Limited Identity Hiding:

- No change in Rosenpass, but we should have **full identity hiding!**
 - ⇒ Do not use pre-authentication with public key?
- This would affect the CPU DOS mitigation, possibly too much.

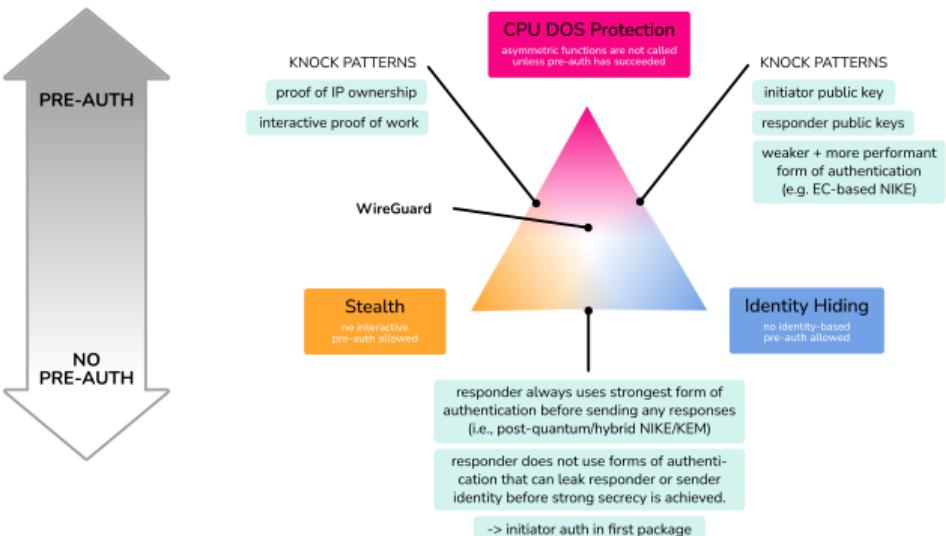


Choose Two: Stealth, Identity Hiding, CPU DOS Mit.





WireGuard and Rosenpass Trade-Offs



CPU DOS Mitigation:

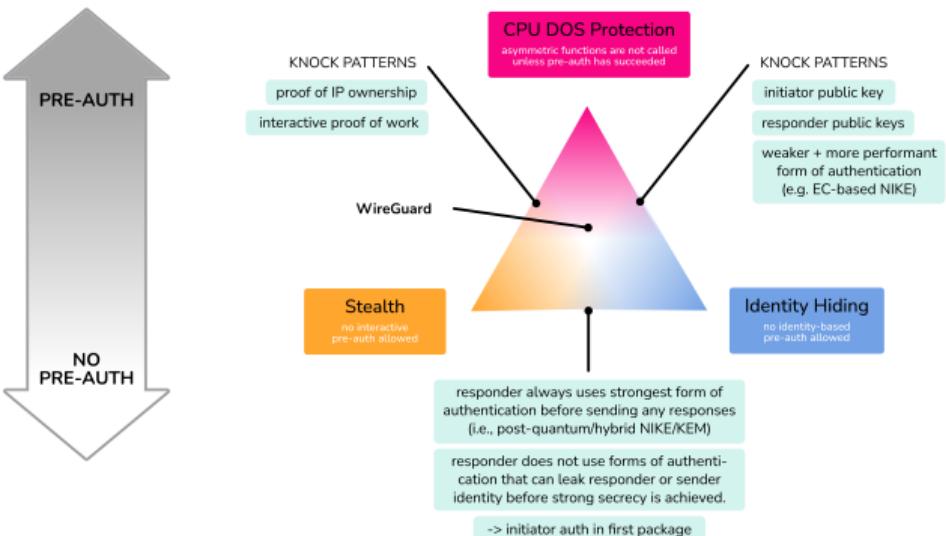
- There is no clear optimum here.
- CPU DOS mitigation is never calling asymmetric crypto unless we know it succeeds (circular reasoning)

Stealth:

Identity hiding:



WireGuard and Rosenpass Trade-Offs



CPU DOS Mitigation:

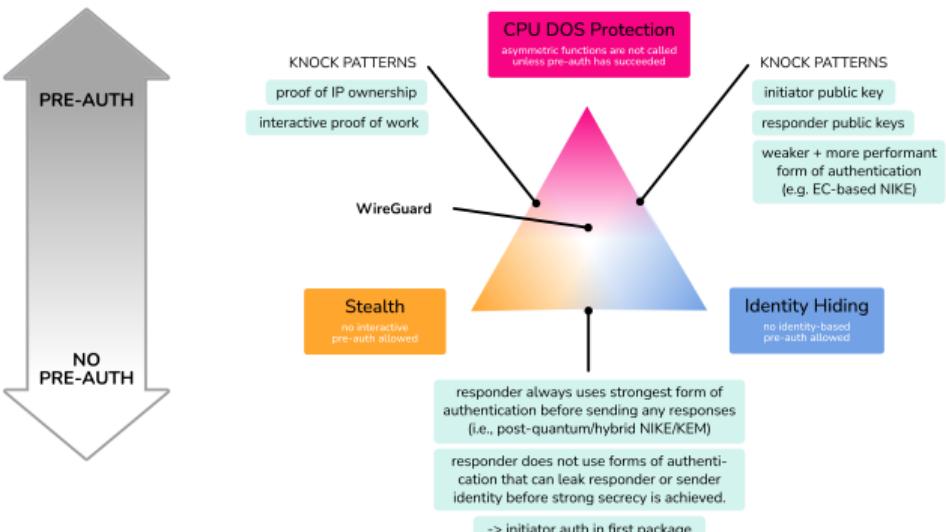
Stealth:

- Broken on DOS attacks assuming recipient is known
- ⇒ This seems acceptable

Identity hiding:



WireGuard and Rosenpass Trade-Offs



CPU DOS Mitigation:

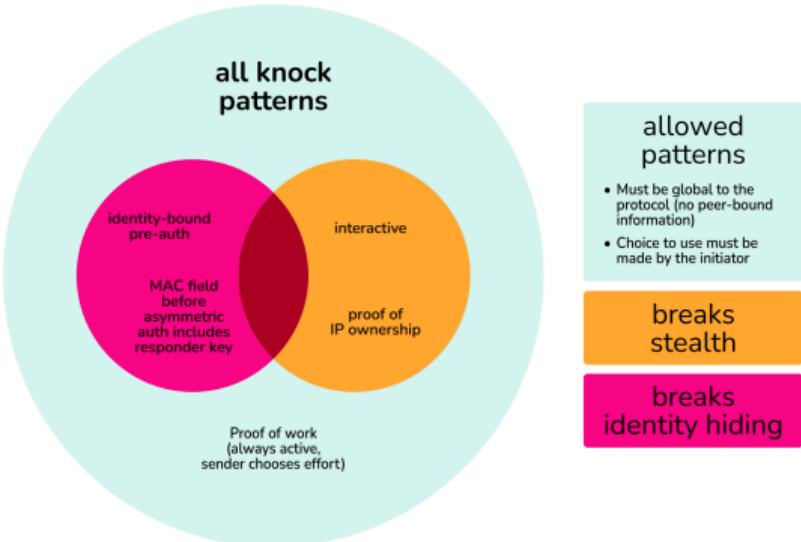
Stealth:

Identity hiding:

- Broken on knowledge of public keys
- ⇒ This seems unacceptable!
- ⇒ Investigate proper identity hiding without overly impacting stealth and CPU DOS mitig.



Knock Patterns



- We choose to think of WireGuard's and Rosenpass' pre-auth as "Knock Patterns"
 - These knock patterns have severe trade-offs.
 - Interactive knock pattern (cookie mechanism) breaks stealth
 - Identity-based knock patterns (e.g., knowledge of public key) breaks identity hiding
- ⇒ Avoid identity-bound knock patterns
- ⇒ Minimize interactive knock patterns
- ⇒ Explore other (allowed) knock patterns



Tools to the Table!

Bellare and Rogaway [BR06], Halevi [Hal05]:

Call for “automated tools, that can help write and verify game-based proofs”



Tools to the Table!

Bellare and Rogaway [BR06], Halevi [Hal05]:

Call for “automated tools, that can help write and verify game-based proofs”

Do the tools *actually help?* And if yes, whom?

ProVerif, Tamarin, CryptoVerif, EasyCrypt:

We like these tools, they are good!



Tools to the Table!

Bellare and Rogaway [BR06], Halevi [Hal05]:

Call for “automated tools, that can help write and verify game-based proofs”

Do the tools *actually help?* And if yes, whom?

ProVerif, Tamarin, CryptoVerif, EasyCrypt:

We like these tools, they are good!

They mostly help the *formal verification experts* to:

- do analyses themselves, write papers
- develop proof methodologies, foundation work for formal methods

Epilogue

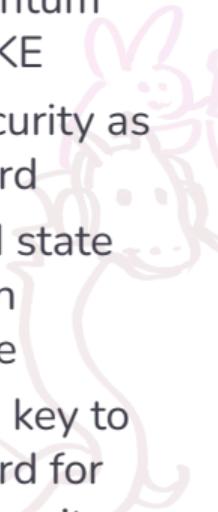




Conclusion

Rosenpass

- Post-quantum secure AKE
- Same security as WireGuard
- Improved state disruption resistance
- Transfers key to WireGuard for hybrid security



Protocol Findings

- **CookieCutter:** DOS exploiting WireGuard cookie mechanism
- **ChronoTrigger:** DOS exploiting insecure system time to attack WireGuard
- There is a **trade-off** between identity hiding, stealth, and CPU-exhaustion DOS protection

Talk To Us

- About why we should use Tamarin (or SAPIC+?) over ProVerif
- State disruption attacks
- Stealth and Identity hiding
- Adding syntax rewriting to the tool belt of mechanized verification in cryptography



Rosenpass going Rube-Goldberg: The Details

- Embed cryptographic proof syntax in Lisp S-Expressions
- Translate Lisp code to Python using the Hy language (Lisp that compiles to Python)
- Translate S-Expression code to AST or DOM
- Translate AST or DOM to ProVerif/Tamarin/CryptoVerif/EasyCrypt code using the LARK code parser/generator
- Remote control ProVerif/Tamarin/CryptoVerif/EasyCrypt by
 - Parsing their command line output using LARK
 - (Possibly using the language server interface for more interactive features)
- Provide custom syntax using
 - Lisp Macros
 - Extending LARK-based syntax parsers (to add custom syntactic elements)