Håck ma's 2024, Schloss in Ottenschlag
2024-08-29

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

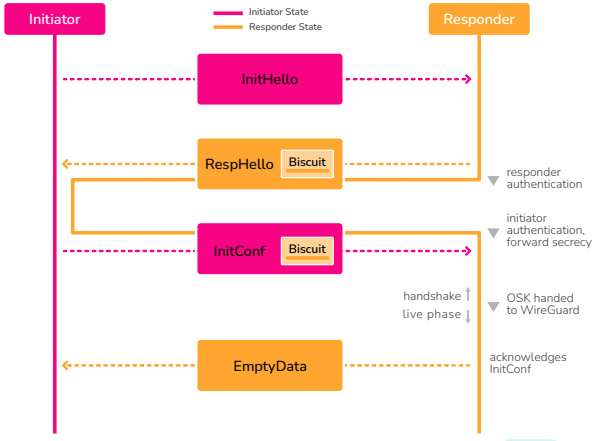
Benjamin Lipp, **Karolin Varner**, and **Lisa Schmidt**
with support from Alice Bowman, and Marei Peischl
https://rosenpass.eu

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 VPN setups against quantum attacks
- A **post-quantum secure VPN**
- A governance **organization** to facilitate development, maintenance, and adoption of said protocol



rosenpass.eu

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.

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

Post-quantum cryptography is ...

... not susceptible to attacks from quantum computers.

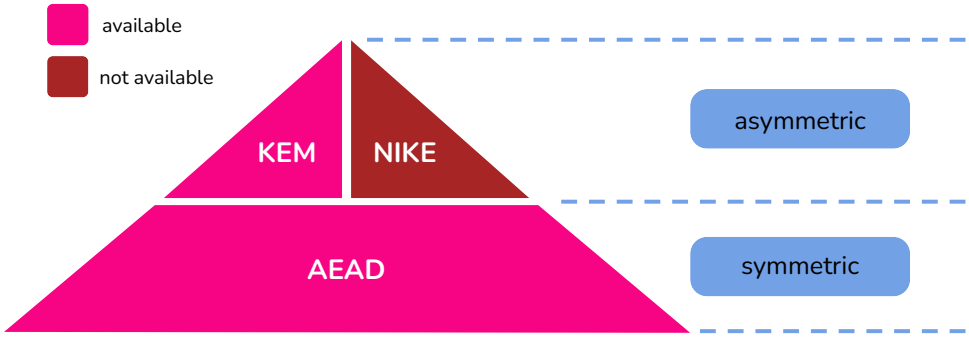
- generally less efficient.
- much bigger ciphertexts.
- less analyzed.

Hybrid cryptography combines ...

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

- 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 a 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. 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 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: For the 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.

KEMs and NIKEs

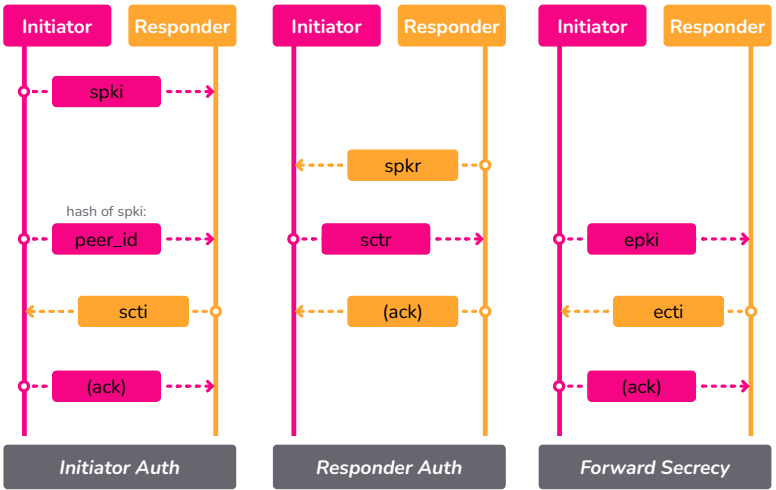
Key Encapsulation Method

```
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);
}
```

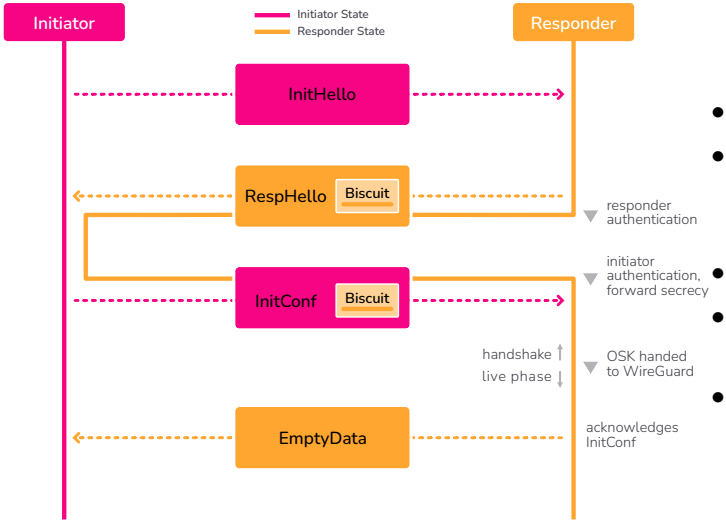
Non-Interactive Key Exchange

```
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



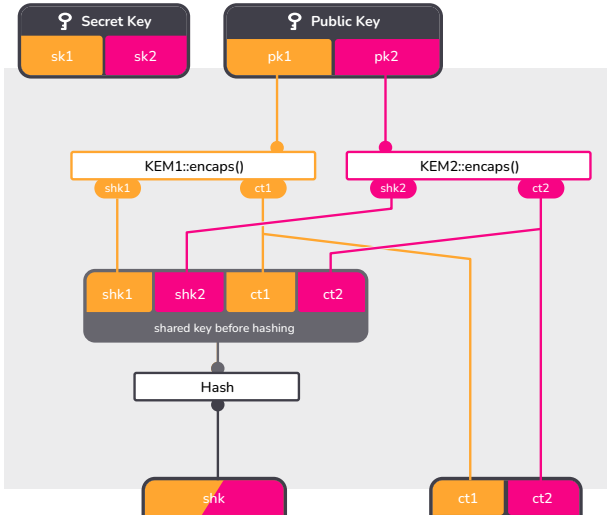
- Authenticated key exchange
- Three KEM operations interleaved to achieve mutual authentication and forward secrecy
- No use of signatures
- First package (InitHello) is unauthenticated
- Stateless responder to avoid disruption attacks

Hybridization

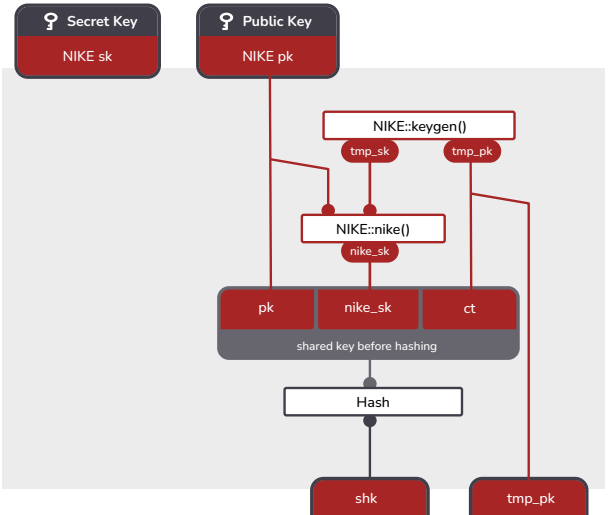
In the following slides you will learn ...

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

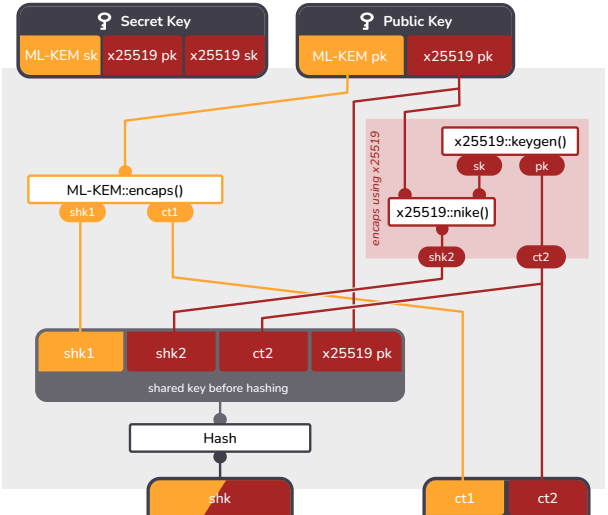
Combining two KEMs with the GHP Combiner



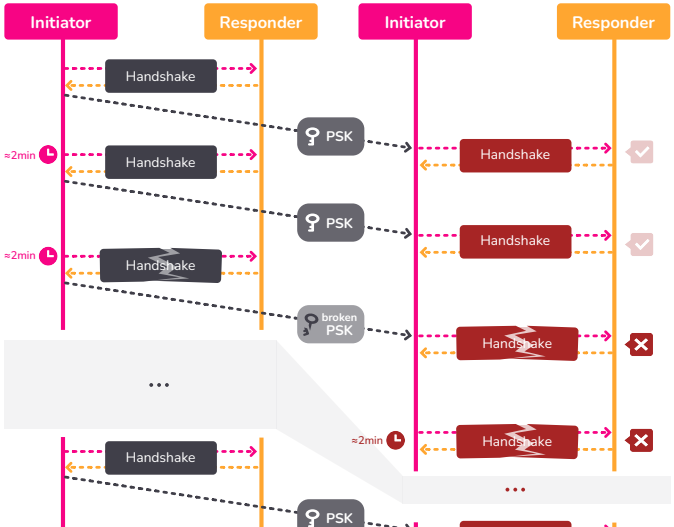
Turning a NIKE into a KEM



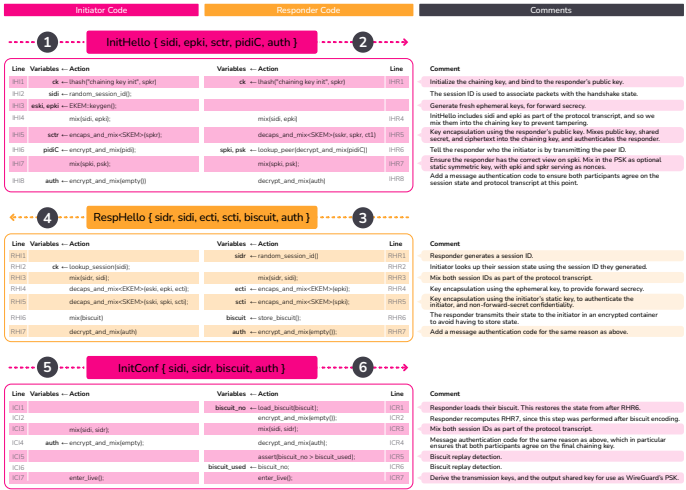
X-Wing


Secret Key
ML-KEM sk
x25519 pk
x25519 sk
Public Key
ML-KEM pk
x25519 pk
encaps using x25519
x25519::keygen()
sk
pk
ML-KEM::encaps()
shk1
ct1
x25519::nike()
shk2
ct2
shk1
shk2
ct2
x25519 pk
shared key before hashing
Hash
shk
ct1
ct2

Rosenpass & WireGuard Hybridization



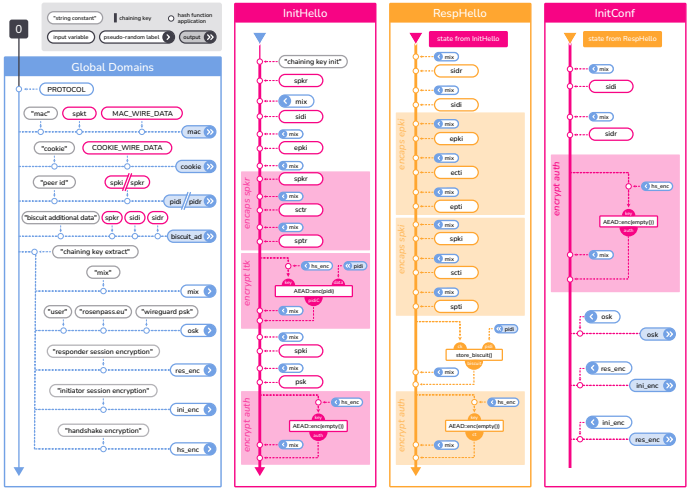
Full Protocol Reference in the Whitepaper



Initiator Code			Responder Code			Comments
			1 InitHello { sidi, epki, sctr, pidiC, auth } 2			
Line	Variables ← Action		Variables ← Action	Line		Comment
IH1	ck ← lhash("chaining key init", spkr)		ck ← lhash("chaining key init", spkr)	IHR1		Initialize the chaining key, and bind to the responder's public key.
IH2	sidi ← random_session_id();					The session ID is used to associate packets with the handshake state.
IH3	eski, epki ← EKEM::keygen();					Generate fresh ephemeral keys, for forward secrecy.
IH4	mix(sidi, epki);		mix(sidi, epki)	IHR4		InitHello includes sidi and epki as part of the protocol transcript, and so we mix them into the chaining key to prevent tampering.
IH5	sctr ← encaps_and_mix<SKEM>(spkr);		decaps_and_mix<SKEM>(sskr, spkr, ct1)	IHR5		Key encapsulation using the responder's public key. Mixes public key, shared secret, and ciphertext into the chaining key, and authenticates the responder.
IH6	pidiC ← encrypt_and_mix(pidi);		spki, psk ← lookup_peer(decrypt_and_mix(pidiC))	IHR6		Tell the responder who the initiator is by transmitting the peer ID.
IH7	mix(spki, psk);		mix(spki, psk);	IHR7		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.
IH8	auth ← encrypt_and_mix(empty())		decrypt_and_mix(auth)	IHR8		Add a message authentication code to ensure both participants agree on the session state and protocol transcript at this point.
			4 RespHello { sidr, sidi, ecti, scti, biscuit, auth } 3			
Line	Variables ← Action		Variables ← Action	Line		Comment
RH1			sidr ← random_session_id()	RHR1		Responder generates a session ID.
RH2	ck ← lookup_session(sidi);			RHR2		Initiator looks up their session state using the session ID they generated.
RH3	mix(sidi, sidi);		mix(sidr, sidi);	RHR3		Mix both session IDs as part of the protocol transcript.
RH4	decaps_and_mix<EKEM>(eski, epki, ecti);		ecti ← encaps_and_mix<EKEM>(epki);	RHR4		Key encapsulation using the ephemeral key, to provide forward secrecy.
RH5	decaps_and_mix<SKEM>(ski, spki, sct);		scti ← encaps_and_mix<SKEM>(spki);	RHR5		Key encapsulation using the initiator's static key, to authenticate the initiator, and non-forward-secret confidentiality.
RH6	mix(biscuit)		biscuit ← store_biscuit();	RHR6		The responder transmits their state to the initiator in an encrypted container to avoid having to store state.
RH7	decrypt_and_mix(auth)		auth ← encrypt_and_mix(empty());	RHR7		Add a message authentication code for the same reason as above.
			5 InitConf { sidi, sidr, biscuit, auth } 6			
Line	Variables ← Action		Variables ← Action	Line		Comment
IC1			biscuit_no ← load_biscuit(biscuit);	ICR1		Responder loads their biscuit. This restores the state from after RHR6.
IC2			encrypt_and_mix(empty());	ICR2		Responder recomputes RHR7, since this step was performed after biscuit encoding.
IC3	mix(sidi, sidr);		mix(sidi, sidr);	ICR3		Mix both session IDs as part of the protocol transcript.
IC4	auth ← encrypt_and_mix(empty);		decrypt_and_mix(auth);	ICR4		Message authentication code for the same reason as above, which in particular ensures that both participants agree on the final chaining key.
IC5			assert(biscuit_no > biscuit_used);	ICR5		Biscuit replay detection.
IC6			biscuit_used ← biscuit_no;	ICR6		Biscuit replay detection.
IC7	enter_live();		enter_live();	ICR7		Derive the transmission keys, and the output shared key for use as WireGuard's PSK.

rosenpass.eu/docs

Rosenpass Key Derivation Chain



Trials ~ Attacks found

ChronoTrigger

In the following slides, you will learn ...

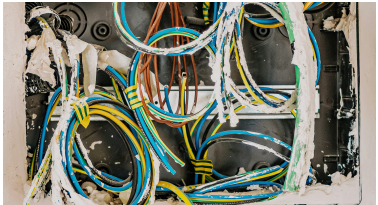
... that denial of service can happen on the level of cryptographic protocols!

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

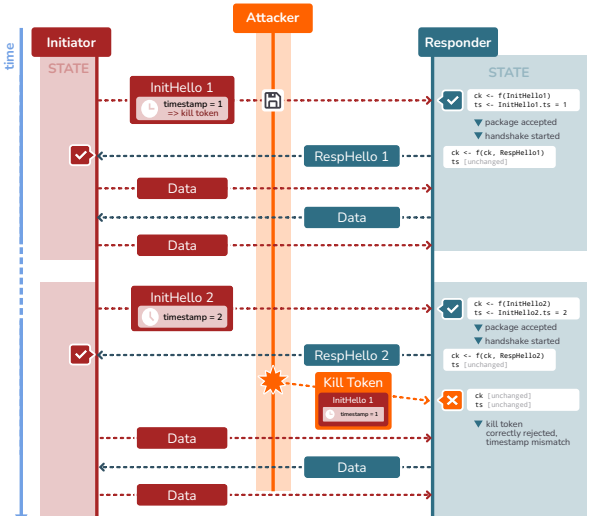
... how to accept replay attacks and face them without fear!

What are State Disruption Attacks?

Protocol-level DoS



Retransmission Protection in WireGuard



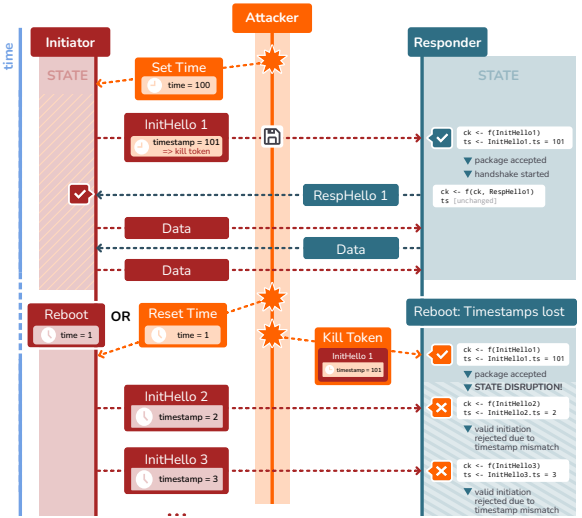
- Replay attacks thwarted by counter
- Counter is based on real-time clock
- Responder is semi-stateful (one retransmission at program start may be accepted, but this does not affect protocol security)

⇒ 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

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

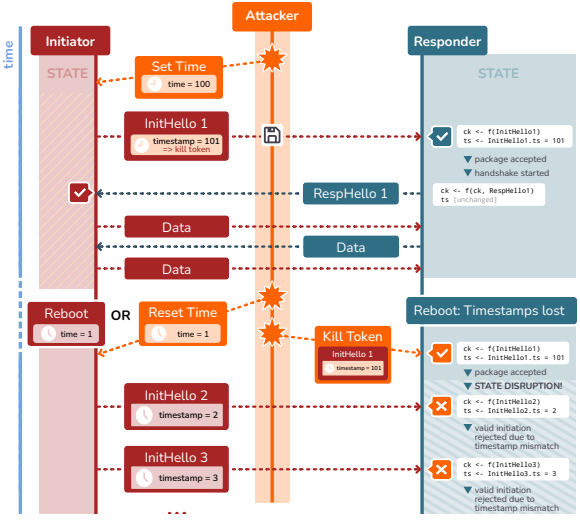
... both peers are being reset ...

B. Delayed execution phase:

1. **Attacker** sends *KillToken* to responder, setting their timestamp to a future value

⇒ Initiation now fails again due to time stamp mismatch

ChronoTrigger Attack

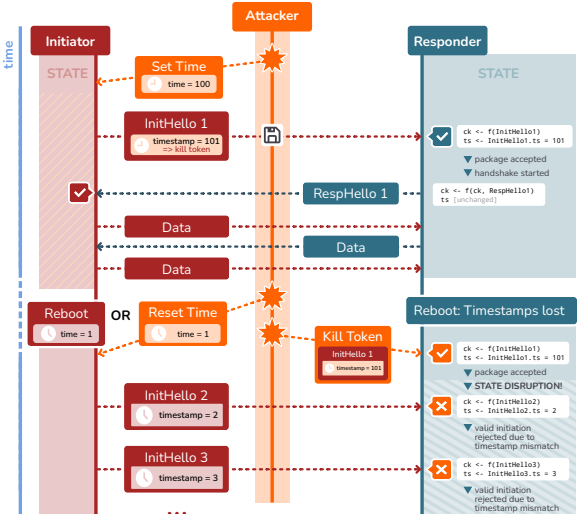


Gaining access to system time:

- 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

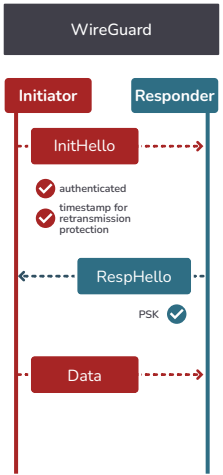
Preparation phase, attacker needs:

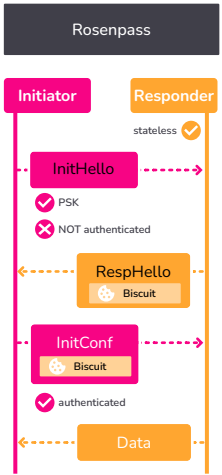
- Eavesdropping of initiator packets
- Access to system time

Delayed execution, attacker needs:

- No access beyond message transmission to responder

ChronoTrigger: Changes in Rosenpass

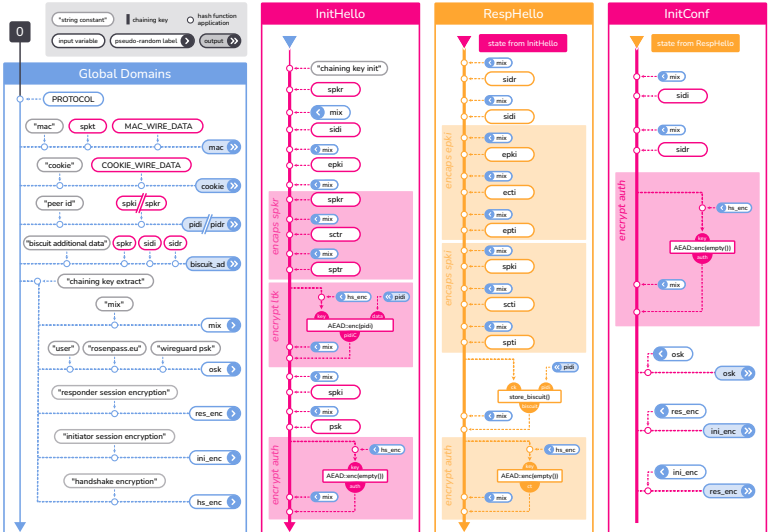




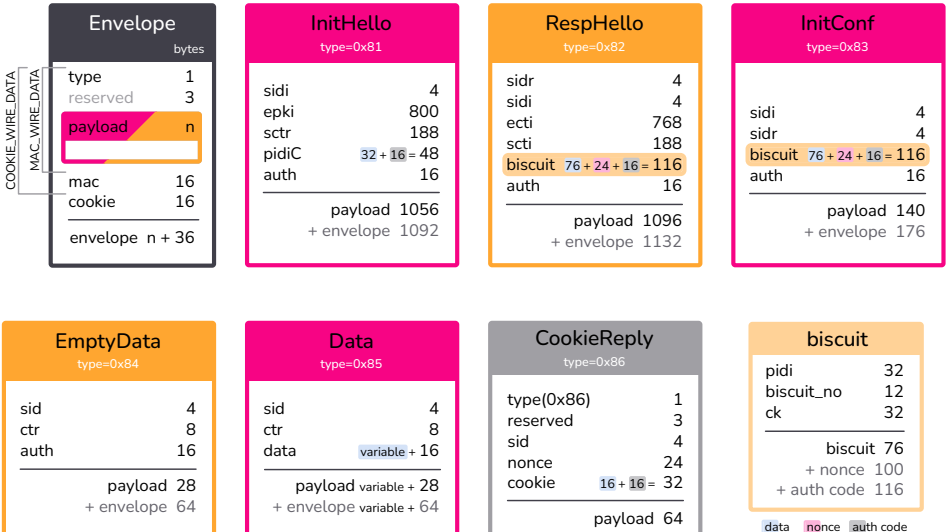
- InitHello is unauthenticated because responder still needs to encapsulate secret with initiator key
- Since InitHello is unauthenticated, retransmission protection is impossible
- Responder state is moved into a cookie called *Biscuit*; this renders the responder stateless
- Retransmission of InitHello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

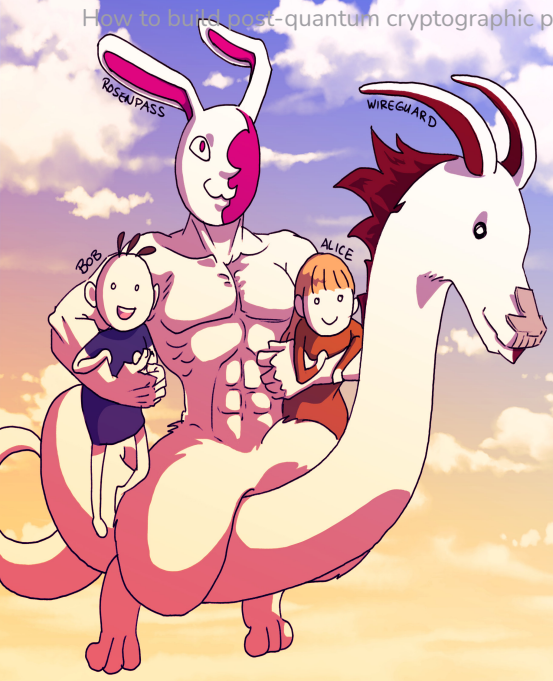
⇒ Stateless responder prevents ChronoTrigg

Rosenpass Key Derivation Chain: Spot the Biscuit



Rosenpass Protocol Messages: Spot the Biscuit





Tribulations ~ Tooling

Oh These Proof Tools

Vive la Révolution! Against the

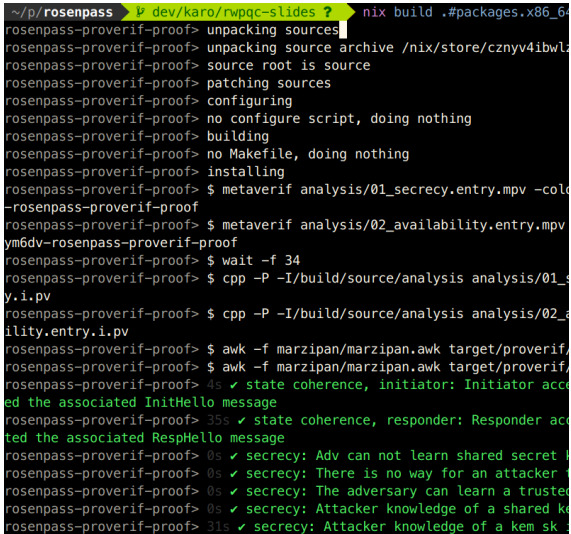
Bourgeoisie of Proof Assistants!

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

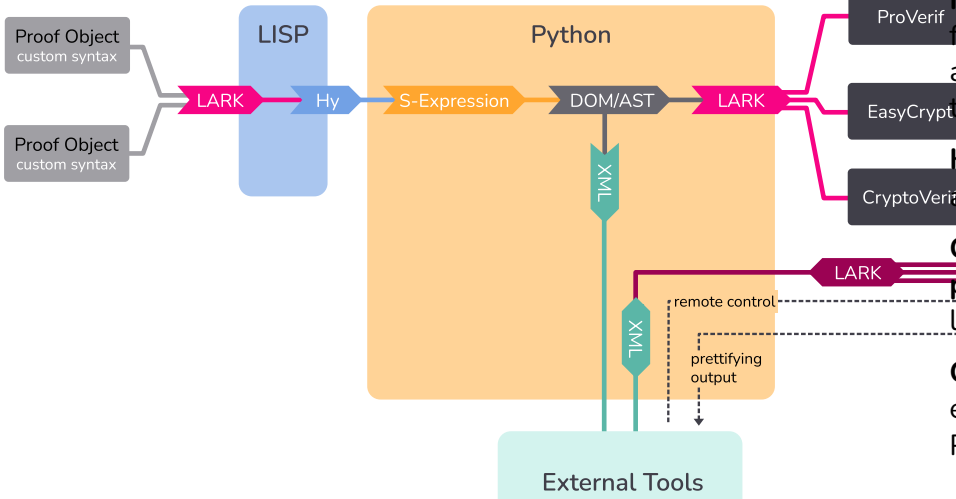


- Symbolic modeling using ProVerif
- Proofs treated as part of the codebase
- Uses a model internally that is based on a fairly comprehensive Maximum Exposure Attacks (MEX) variant
- Covers non-interruptability (resistance to disruption attacks)
- Mechanized proof in the computational model is an open 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-iOLHAIaxpDA
- 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

Limited Identity Hiding:

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

CPU DoS mitigation:

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

Triumphs ~ Secrecy & Non-Interruptability

Modeling of Rosenpass

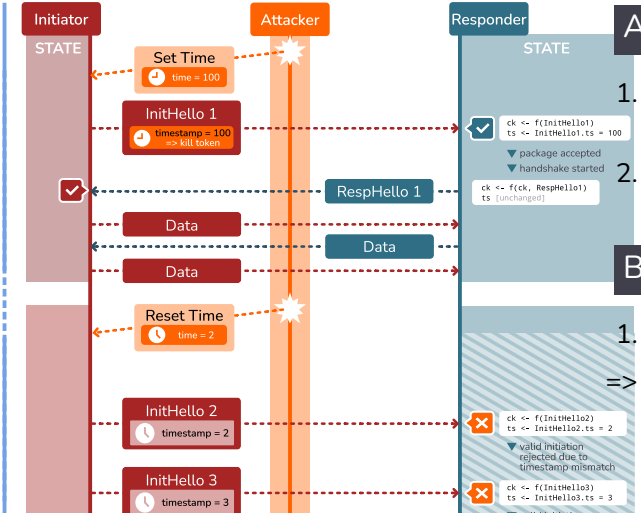
Using ProVerif

Non-Interruptability: More Formally

For every pair of traces `tmin`, `tmax` where trace `tmax` can be formed by insertion of messages/oracle calls into `tmin`, the result of `tmin` and `tmax` should remain the same.

- Let `Result` be the set of possible protocol results
- Let `Trace` be the set of possible protocol traces
- Let $\texttt{res}(\texttt{t}) : \texttt{Trace} \to \texttt{Result}$ determine the protocol result given $\texttt{t} : \texttt{Trace}$
- Let $\texttt{t1} \supseteq \texttt{t2} : \texttt{Trace} \to \texttt{Trace} \to \texttt{Prop}$ denote that $\texttt{t2}$ can be formed by insertion of elements into $\texttt{t1}$
- $\forall(\texttt{tmin}, \texttt{tmax}) : \texttt{Trace} \times \texttt{Trace}; \texttt{tmin} \supseteq \texttt{tmax} \to \texttt{res}(\texttt{tmin}) = \texttt{res}(\texttt{tmax})$

ChronoTrigger Attack: Immediate Execution



A. Preparation phase:

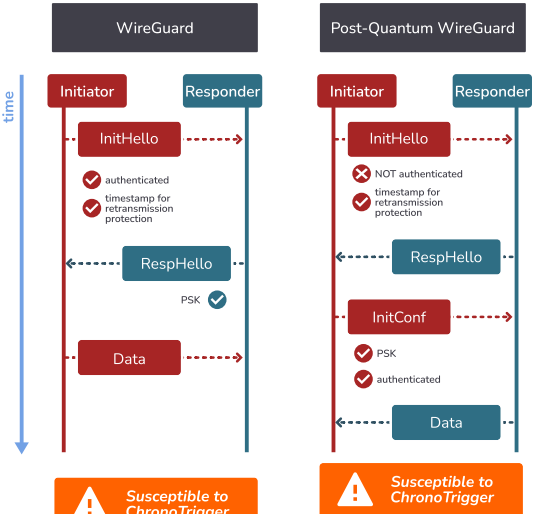
1. **Attacker** sets *initiator system time* to a future value
2. **Attacker** waits while both peers are performing a valid handshake

B. Direct execution phase:

1. **Attacker** lets system time on initiator reset

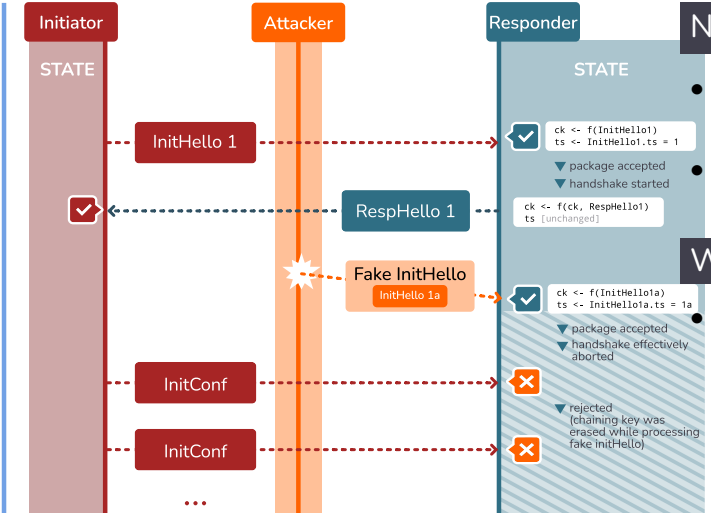
=> Initiation now fails due to counter mismatch

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



No PSK/Public keys as PSK

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

With PSK

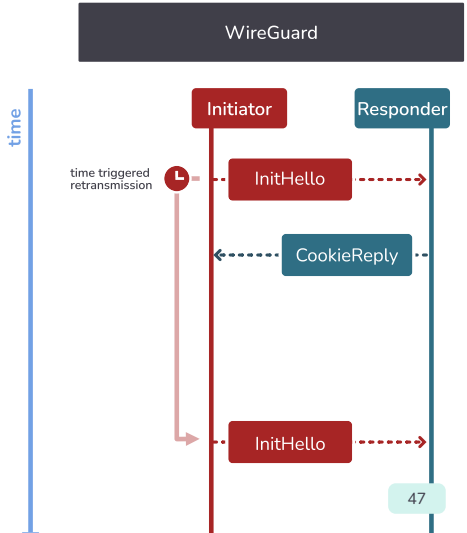
- Replay attack with NTP access from classic WireGuard still applies

Trials ~ Attacks found

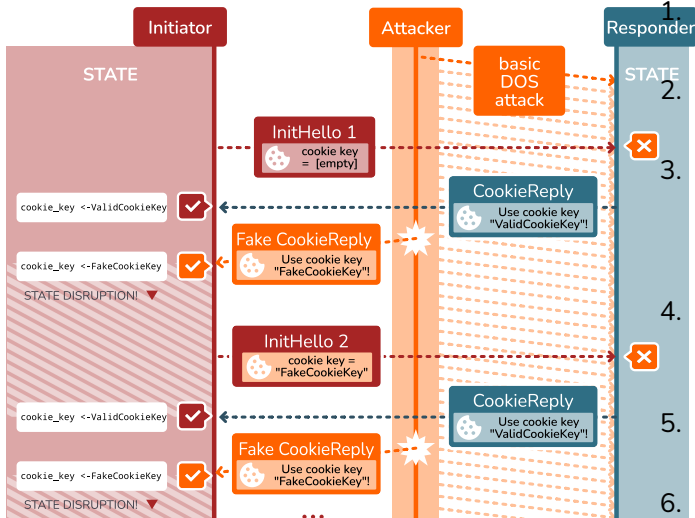
CookieCutter

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

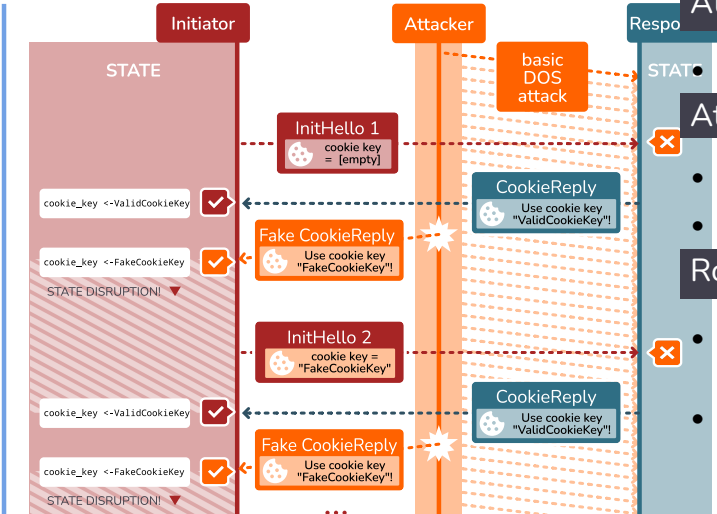


CookieCutter Attack



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

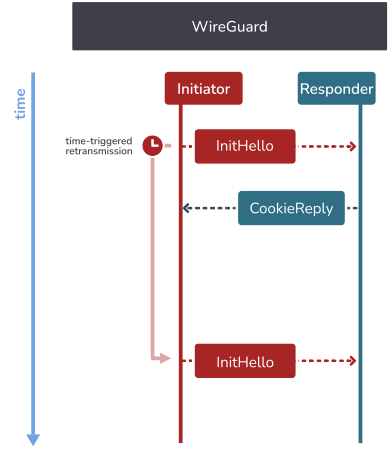
Attacker needs:

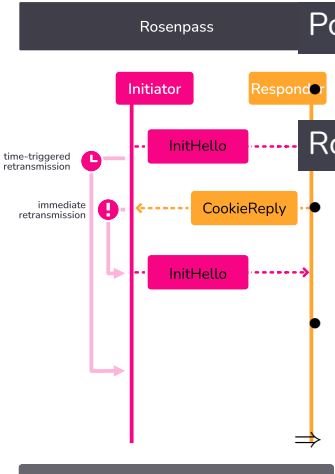
- Knowledge of public keys
- Good timing

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 amplication 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.

CPU DOS Protection

asymmetric functions are not called unless pre-auth has succeeded

KNOCK PATTERNS

proof of IP ownership

interactive proof of work

KNOCK PATTERNS

initiator public key

responder public keys

weaker + more performant form of authentication (e.g. EC-based NIKE)

WireGuard

Stealth

no interactive pre-auth allowed

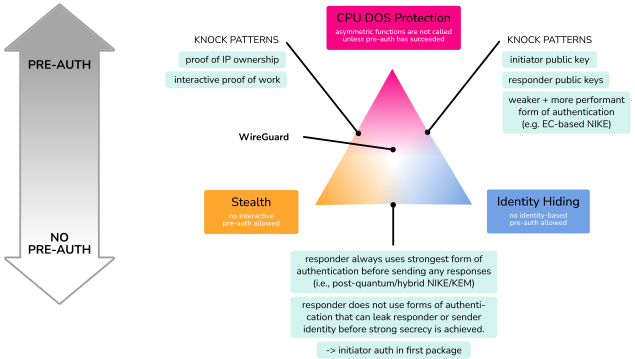
Identity Hiding

no identity-based pre-auth allowed

responder always uses strongest form of authentication before sending any responses (i.e., post-quantum/hybrid NIKE/KEM)

responder does not use forms of authenti-cation that can leak responder or sender identity before strong secrecy is achieved.

WireGuard and Rosenpass Trade-Offs



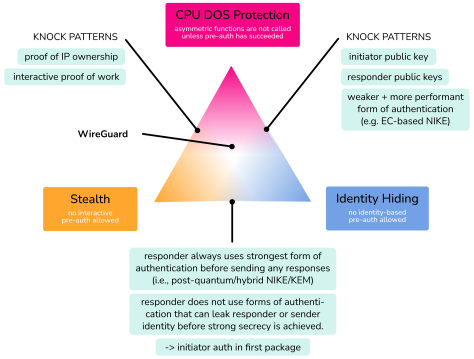
CPU DoS Mitigation:

- 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:

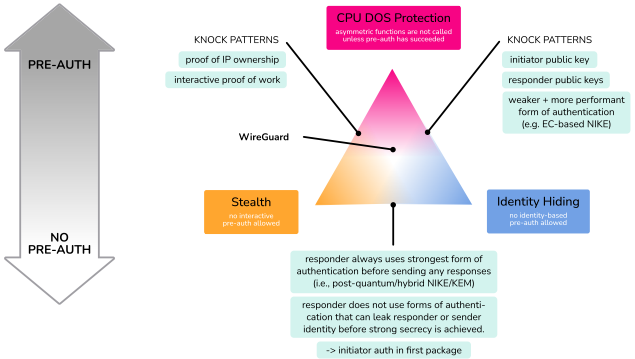
Stealth:

- Broken on DoS attacks assuming recipient is known

⇒ This seems acceptable

Identity hiding:

WireGuard and Rosenpass Trade-Offs



CPU DoS Mitigation:

Stealth:

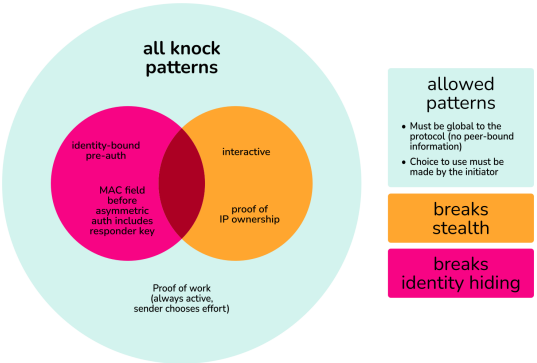
Identity hiding:

- Broken on knowledge of public keys

⇒ This seems inacceptable!

⇒ 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 patter

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)