Blockstream

Blockstream RESEARCH

Provably Bug-Free BIPs & Implementations

Jonas Nick

nickler.ninja

@n1ckler

2022-10

TABConf 22

Specifications should be **free of bugs, easy to implement, and hard to misinterpret**.

Specifications should be **free of bugs, easy to implement, and hard to misinterpret**.

This makes writing specifications a **slow** and **exhausting** process.

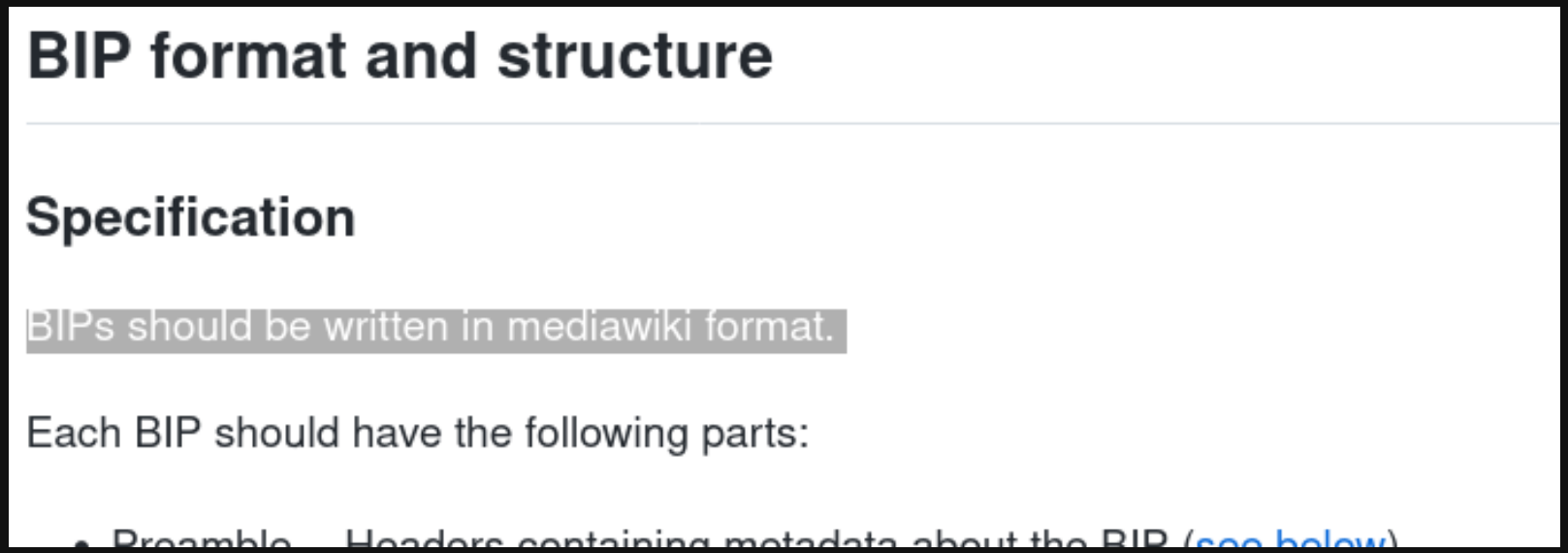
Specifications should be **free of bugs, easy to implement, and hard to misinterpret**.

This makes writing specifications a **slow** and **exhausting** process.

This talk presents a tiny **step** towards improving this.

Bitcoin Improvement Proposal (BIP) 2

Bitcoin Improvement Proposal (BIP) 2



BIP format and structure

Specification

BIPs should be written in mediawiki format.

Each BIP should have the following parts:

- Preamble – Headers containing metadata about the BIP (see below)

BIP-MuSig2 signing specification

BIP-MuSig2 signing specification

Signing

Algorithm *Sign(secnonce, sk, session_ctx)*:

- Inputs:
  - The secret nonce *secnonce* that has never been used as input to *Sign* before: a 64-byte array
  - The secret key *sk*: a 32-byte array
  - The *session_ctx*: a Session Context data structure
- Let *(Q, gacc, _, b, R, e) = GetSessionValues(session_ctx)*; fail if that fails
- Let $k_1' = int(secnonce[0:32])$, $k_2' = int(secnonce[32:64])$
- Fail if $k_i' = 0$ or $k_i' \geq n$ for $i = 1..2$
- Let $k_1 = k_1'$, $k_2 = k_2'$ if *has_even_y(R)*, otherwise let $k_1 = n - k_1'$, $k_2 = n - k_2'$
- Let $d' = int(sk)$
- Fail if $d' = 0$ or $d' \geq n$
- Let $P = d'G$

BIP-MuSig2 signing specification

Signing

Algorithm *Sign(secnonce, sk, session_ctx)*:

- Inputs:
  - The secret nonce *secnonce* that has never been used as input to *Sign* before: a 64-byte array
  - The secret key *sk*: a 32-byte array
  - The *session_ctx*: a Session Context data structure
- Let *(Q, gacc, _, b, R, e) = GetSessionValues(session_ctx)*; fail if that fails
- Let $k_1' = int(secnonce[0:32])$, $k_2' = int(secnonce[32:64])$
- Fail if $k_i' = 0$ or $k_i' \geq n$ for $i = 1..2$
- Let $k_1 = k_1'$, $k_2 = k_2'$ if *has_even_y(R)*, otherwise let $k_1 = n - k_1'$, $k_2 = n - k_2'$
- Let $d' = int(sk)$
- Fail if $d' = 0$ or $d' \geq n$
- Let $P = d'G$

- **readability:** meh

BIP-MuSig2 signing specification

Signing

Algorithm *Sign(secnonce, sk, session_ctx)*:

- Inputs:
  - The secret nonce *secnonce* that has never been used as input to *Sign* before: a 64-byte array
  - The secret key *sk*: a 32-byte array
  - The *session_ctx*: a Session Context data structure
- Let *(Q, gacc, _, b, R, e) = GetSessionValues(session_ctx)*; fail if that fails
- Let $k_1' = int(secnonce[0:32])$, $k_2' = int(secnonce[32:64])$
- Fail if $k_i' = 0$ or $k_i' \geq n$ for $i = 1..2$
- Let $k_1 = k_1'$, $k_2 = k_2'$ if *has_even_y(R)*, otherwise let $k_1 = n - k_1'$, $k_2 = n - k_2'$
- Let $d' = int(sk)$
- Fail if $d' = 0$ or $d' \geq n$
- Let $P = d'G$

- **readability:** meh

BIP-MuSig2 signing specification

Signing

Algorithm *Sign(secnonce, sk, session_ctx)*:

- Inputs:
  - The secret nonce *secnonce* that has never been used as input to *Sign* before: a 64-byte array
  - The secret key *sk*: a 32-byte array
  - The *session_ctx*: a Session Context data structure
- Let *(Q, gacc, _, b, R, e) = GetSessionValues(session_ctx)*; fail if that fails
- Let $k_1' = int(secnonce[0:32])$, $k_2' = int(secnonce[32:64])$
- Fail if $k_i' = 0$ or $k_i' \geq n$ for $i = 1..2$
- Let $k_1 = k_1'$, $k_2 = k_2'$ if *has_even_y(R)*, otherwise let $k_1 = n - k_1'$, $k_2 = n - k_2'$
- Let $d' = int(sk)$
- Fail if $d' = 0$ or $d' \geq n$
- Let $P = d'G$

- **readability:** meh

BIP-MuSig2 signing specification

Signing

Algorithm *Sign(secnonce, sk, session_ctx)*:

- Inputs:
  - The secret nonce *secnonce* that has never been used as input to *Sign* before: a 64-byte array
  - The secret key *sk*: a 32-byte array
  - The *session_ctx*: a Session Context data structure
- Let $(Q, gacc, \_, b, R, e) = GetSessionValues(session\_ctx)$; fail if that fails
- Let $k_1' = int(secnonce[0:32])$, $k_2' = int(secnonce[32:64])$
- Fail if $k_i' = 0$ or $k_i' \geq n$ for $i = 1..2$
- Let $k_1 = k_1'$, $k_2 = k_2'$ if $has\_even\_y(R)$, otherwise let $k_1 = n - k_1'$, $k_2 = n - k_2'$
- Let $d' = int(sk)$
- Fail if $d' = 0$ or $d' \geq n$
- Let $P = d'G$

- **readability:** meh
- **correctness:** not executable, no tests (surprisingly hard to get rid of errors)

BIP-MuSig2 reference code

BIP-MuSig2 reference code

```
347  def sign(secnonce: bytearray, sk: bytes, session_ctx: SessionContext) -> bytes:
348      (Q, gacc, _, b, R, e) = get_session_values(session_ctx)
349      k_1_ = int_from_bytes(secnonce[0:32])
350      k_2_ = int_from_bytes(secnonce[32:64])
351      # Overwrite the secnonce argument with zeros such that subsequent calls of
352      # sign with the same secnonce raise a ValueError.
353      secnonce[:] = bytearray(b'\x00'*64)
354      if not 0 < k_1_ < n:
355          raise ValueError('first secnonce value is out of range.')
356      if not 0 < k_2_ < n:
357          raise ValueError('second secnonce value is out of range.')
358      k_1 = k_1_ if has_even_y(R) else n - k_1_
359      k_2 = k_2_ if has_even_y(R) else n - k_2_
360      d_ = int_from_bytes(sk)
361      if not 0 < d_ < n:
362          raise ValueError('secret key value is out of range.')
363      R = point_mul(G, d_)
```

BIP-MuSig2 reference code

```python
def sign(secnonce: bytearray, sk: bytes, session_ctx: SessionContext) -> bytes:
    (Q, gacc, _, b, R, e) = get_session_values(session_ctx)
    k_1_ = int_from_bytes(secnonce[0:32])
    k_2_ = int_from_bytes(secnonce[32:64])
    # Overwrite the secnonce argument with zeros such that subsequent calls of
    # sign with the same secnonce raise a ValueError.
    secnonce[:] = bytearray(b'\x00'*64)
    if not 0 < k_1_ < n:
        raise ValueError('first secnonce value is out of range.')
    if not 0 < k_2_ < n:
        raise ValueError('second secnonce value is out of range.')
    k_1 = k_1_ if has_even_y(R) else n - k_1_
    k_2 = k_2_ if has_even_y(R) else n - k_2_
    d_ = int_from_bytes(sk)
    if not 0 < d_ < n:
        raise ValueError('secret key value is out of range.')
    R = point_mul(G, d_)
```

- **readability**: good (for implementers)

BIP-MuSig2 reference code

```python
def sign(secnonce: bytearray, sk: bytes, session_ctx: SessionContext) -> bytes:
    (Q, gacc, _, b, R, e) = get_session_values(session_ctx)
    k_1_ = int_from_bytes(secnonce[0:32])
    k_2_ = int_from_bytes(secnonce[32:64])
    # Overwrite the secnonce argument with zeros such that subsequent calls of
    # sign with the same secnonce raise a ValueError.
    secnonce[:] = bytearray(b'\x00'*64)
    if not 0 < k_1_ < n:
        raise ValueError('first secnonce value is out of range.')
    if not 0 < k_2_ < n:
        raise ValueError('second secnonce value is out of range.')
    k_1 = k_1_ if has_even_y(R) else n - k_1_
    k_2 = k_2_ if has_even_y(R) else n - k_2_
    d_ = int_from_bytes(sk)
    if not 0 < d_ < n:
        raise ValueError('secret key value is out of range.')
    R = point_mul(G, d_)
```

- **readability**: good (for implementers)
- **tested** (random tests, test vectors) & type checked

MuSig2 Paper

$\mathsf{Sign}'(state_1, out, sk_1, m, (pk_2, \ldots, pk_n))$

// Sign' must be called at most once per $state_1$.

$(r_{1,1}, \ldots, r_{1,\nu}) := state_1$

$x_1 := sk_1 \,;\; X_1 := g^{x_1}$

$(R_{1,1}, \ldots, R_{1,\nu}) := (g^{r_{1,1}}, \ldots, g^{r_{1,\nu}})$

$(X_2, \ldots, X_n) := (pk_2, \ldots, pk_n)$

$L := \{X_1, \ldots, X_n\}$

$a_1 := \mathsf{KeyAggCoef}(L, X_1)$

$\widetilde{X} := \mathsf{KeyAgg}(L)$

$(R_1, \ldots, R_\nu) := out$

$b := \mathsf{H}_{\text{non}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$

$R := \prod_{j=1}^{\nu} R_j^{b^{j-1}}$

$c := \mathsf{H}_{\text{sig}}(\widetilde{X}, R, m)$

$s_1 := ca_1x_1 + \sum_{i=1}^{\nu} r_{1,j} b^{j-1} \bmod p$

$state_1' := R \,;\; out_1' := s_1$

return $(state_1', out_1')$

MuSig2 Paper

$\mathsf{Sign}'(state_1, out, sk_1, m, (pk_2, \ldots, pk_n))$

// Sign' must be called at most once per $state_1$.

$(r_{1,1}, \ldots, r_{1,\nu}) := state_1$

$x_1 := sk_1 ;\ X_1 := g^{x_1}$

$(R_{1,1}, \ldots, R_{1,\nu}) := (g^{r_{1,1}}, \ldots, g^{r_{1,\nu}})$

$(X_2, \ldots, X_n) := (pk_2, \ldots, pk_n)$

$L := \{X_1, \ldots, X_n\}$

$a_1 := \mathsf{KeyAggCoef}(L, X_1)$

$\widetilde{X} := \mathsf{KeyAgg}(L)$

$(R_1, \ldots, R_\nu) := out$

$b := \mathsf{H}_{\text{non}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$

$R := \prod_{j=1}^{\nu} R_j^{b^{j-1}}$

$c := \mathsf{H}_{\text{sig}}(\widetilde{X}, R, m)$

$s_1 := ca_1x_1 + \sum_{i=1}^{\nu} r_{1,j} b^{j-1} \bmod p$

$state'_1 := R ;\ out'_1 := s_1$

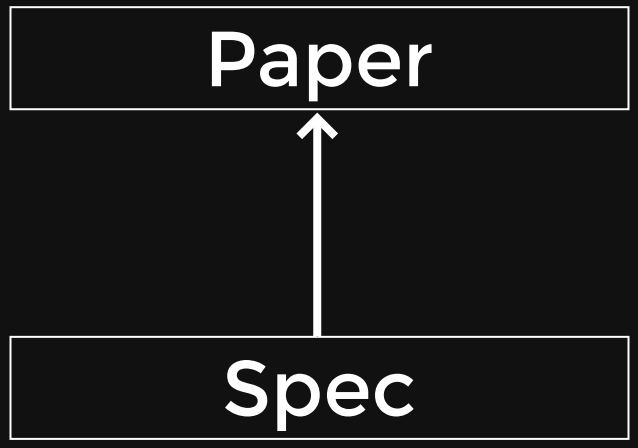
return $(state'_1, out'_1)$

- not a specification

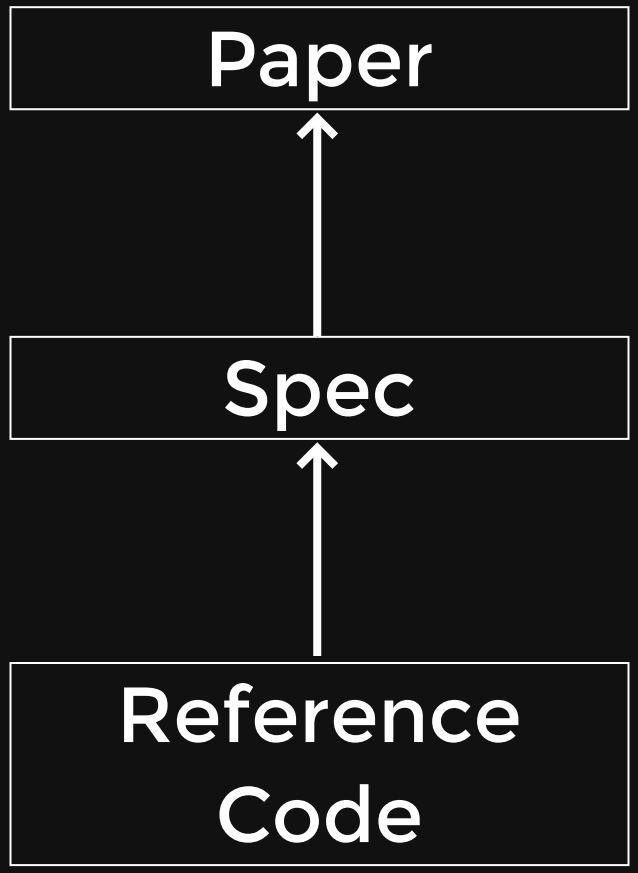
BIP Schnorr/MuSig/etc

Paper

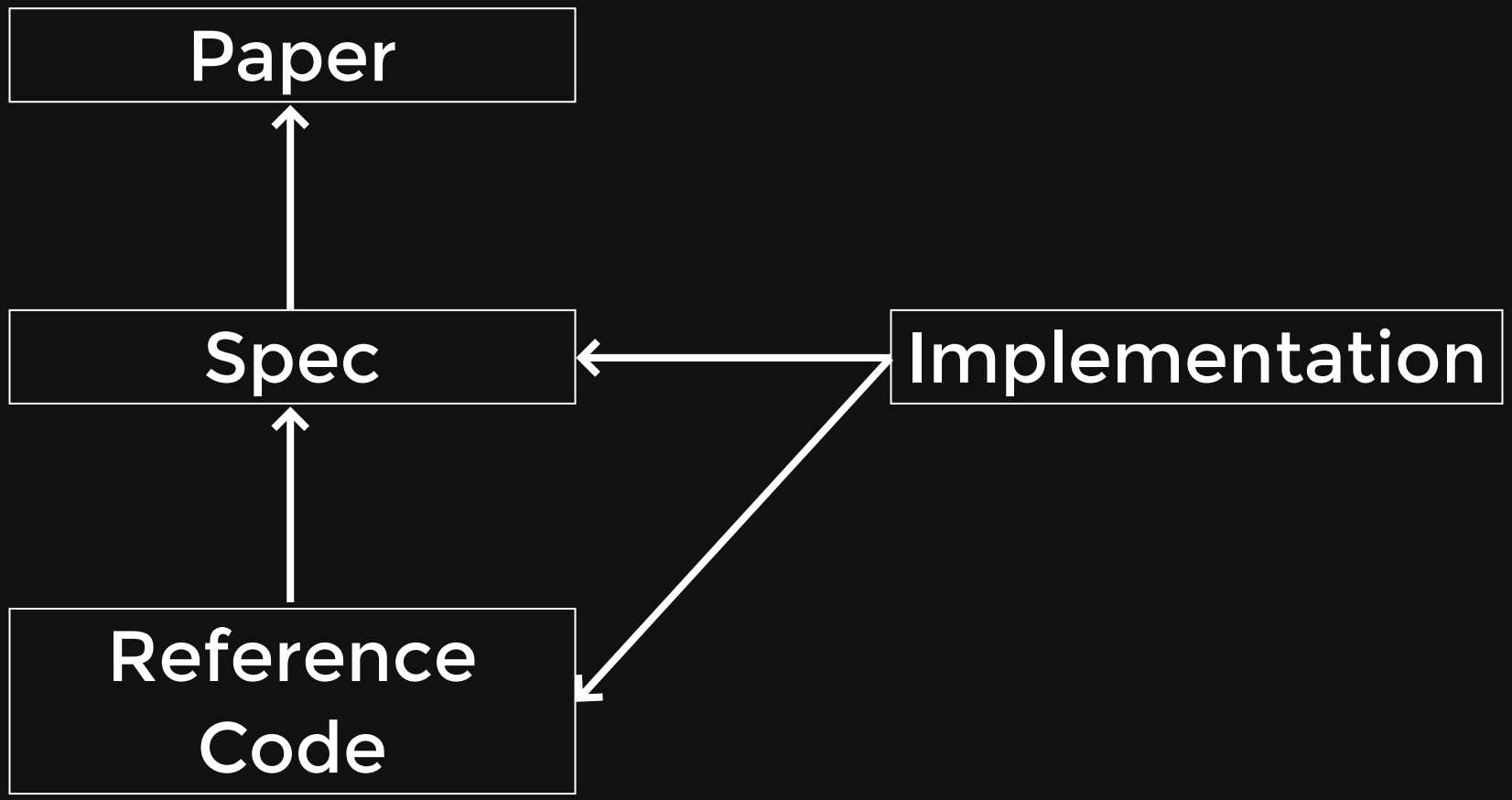
BIP Schnorr/MuSig/etc



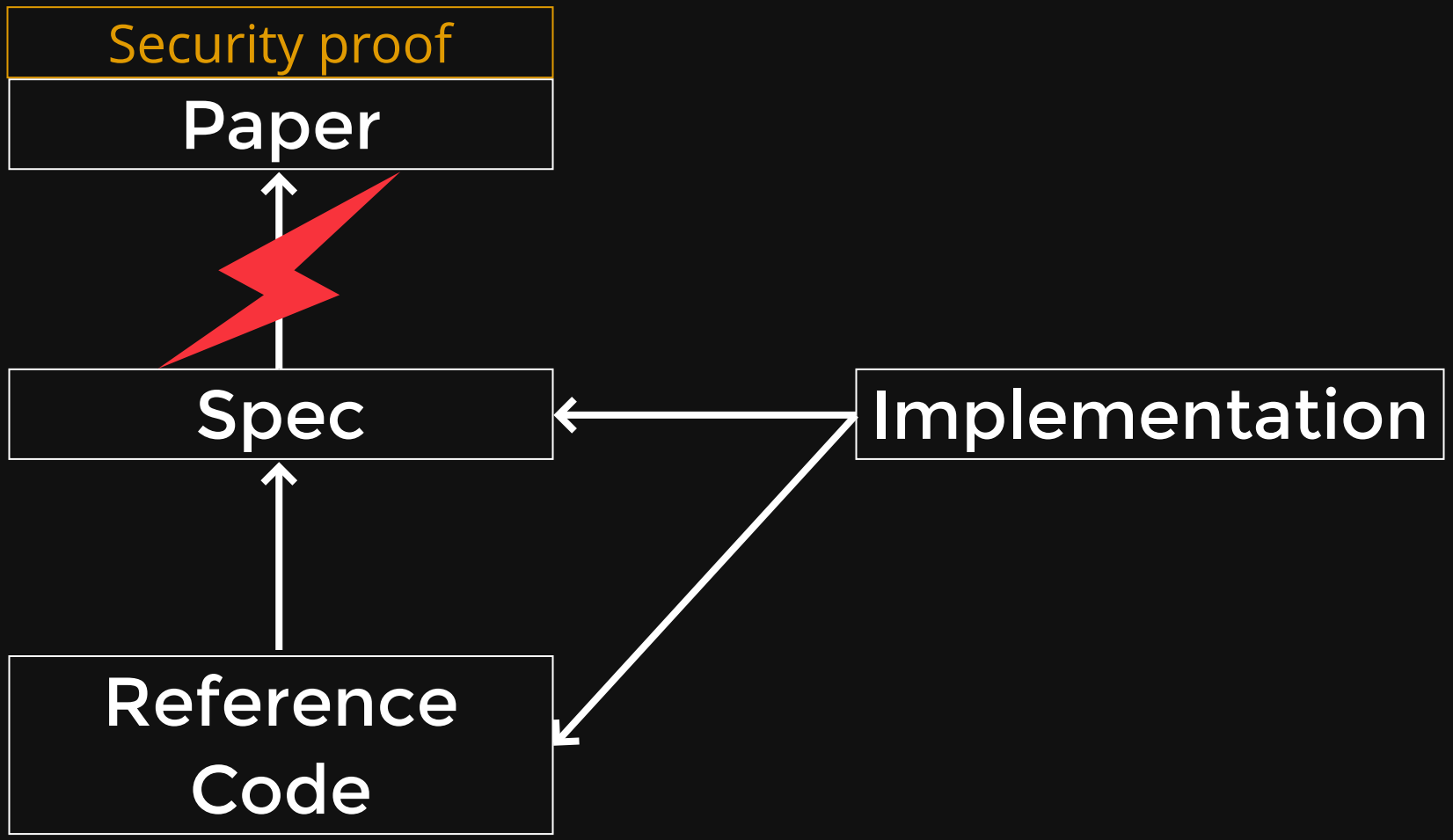
BIP Schnorr/MuSig/etc



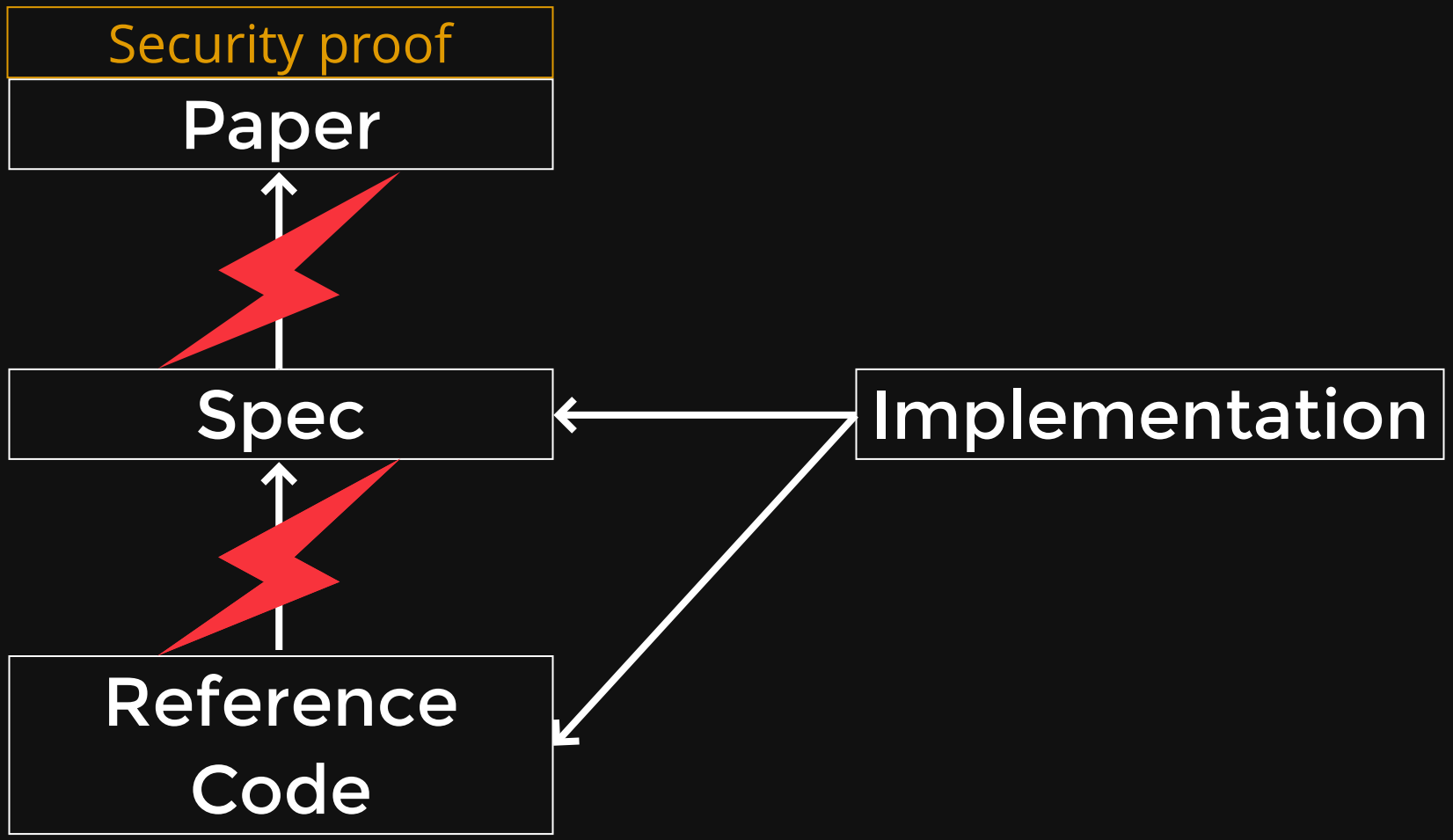
BIP Schnorr/MuSig/etc



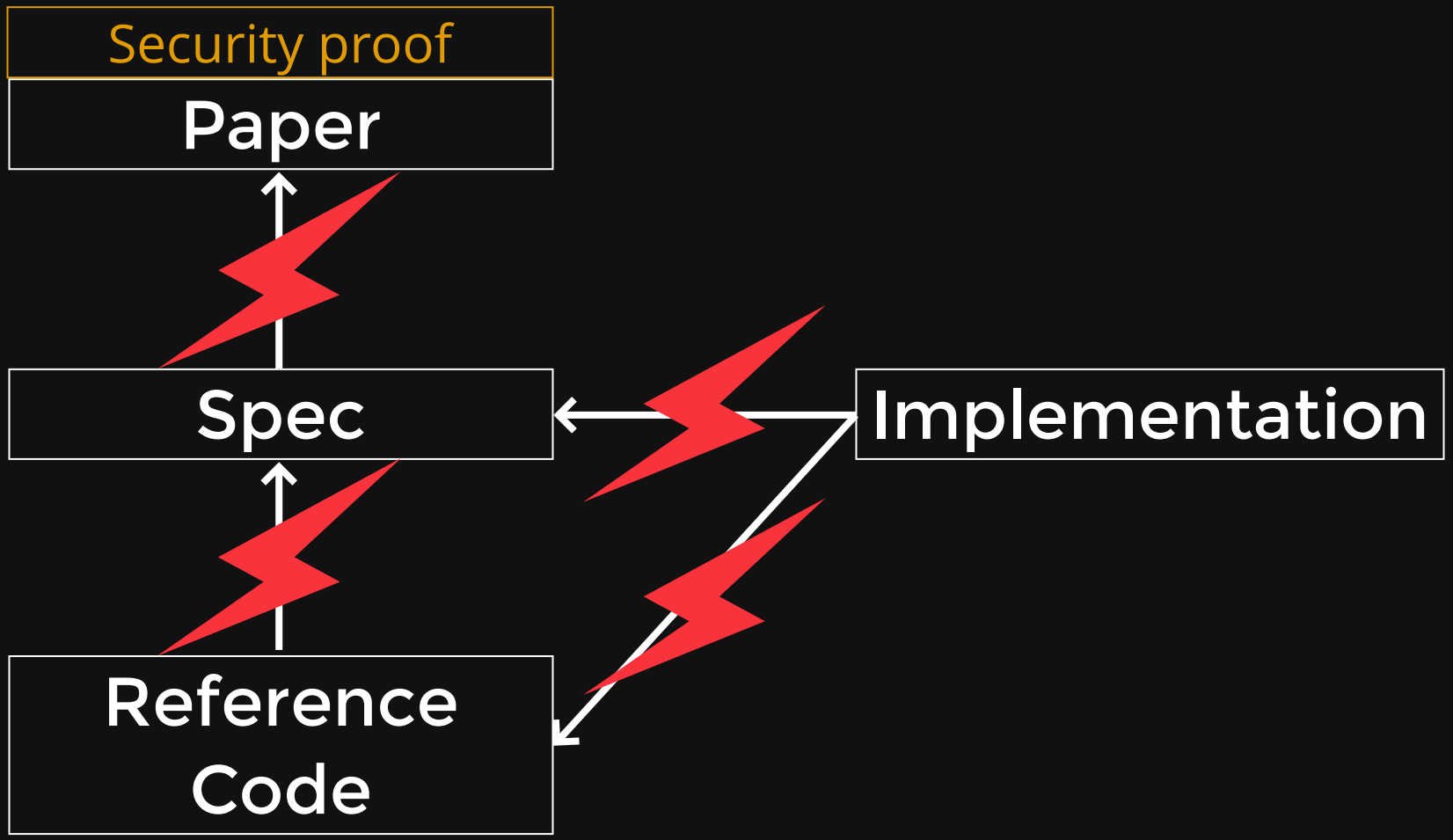
BIP Schnorr/MuSig/etc



BIP Schnorr/MuSig/etc

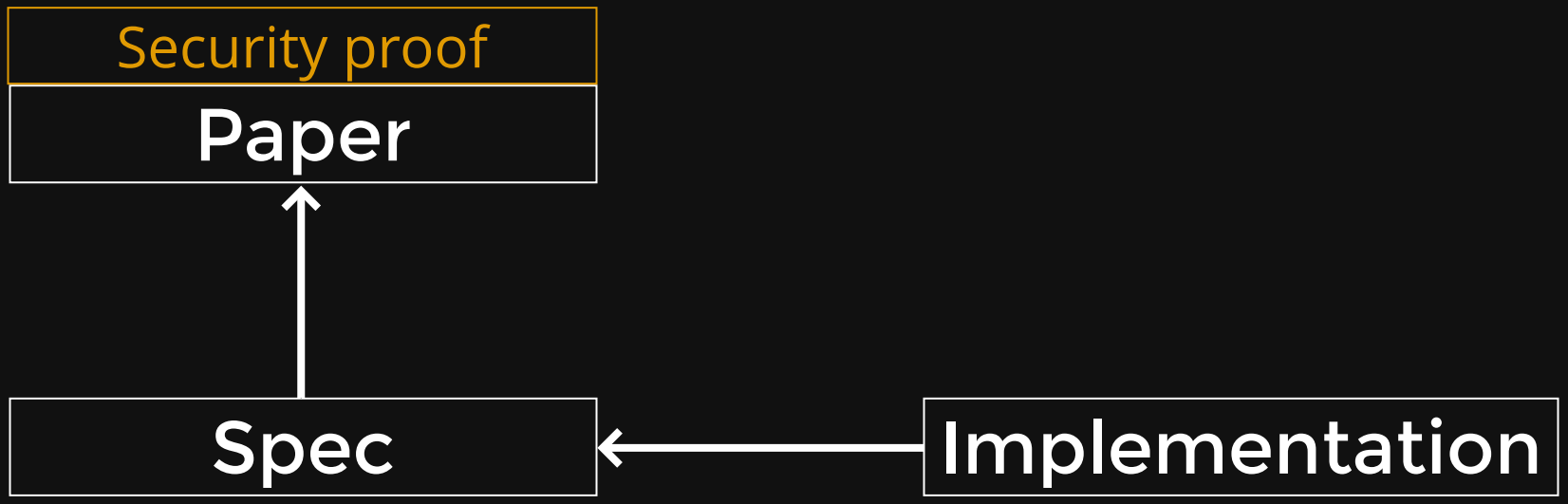


BIP Schnorr/MuSig/etc



BIP Half Agg

BIP Half Agg



BIP Half Agg Spec

BIP Half Agg Spec

Non-interactive aggregation ("compression") of BIP 340 signatures

BIP Half Agg Spec

Non-interactive aggregation ("compression") of BIP 340 signatures

```
32  pub type AggregateResult = Result<AggSig, Error>;
33  pub fn aggregate(pms: &Seq<(PublicKey, Message, Signature)>) -> AggregateResult {
```

BIP Half Agg Spec

Non-interactive aggregation ("compression") of BIP 340 signatures

```
32  pub type AggregateResult = Result<AggSig, Error>;
33  pub fn aggregate(pms: &Seq<(PublicKey, Message, Signature)>) -> AggregateResult {
```

```
89  pub fn verify_aggregate(aggsig: &AggSig, pm_aggd: &Seq<(PublicKey, Message)>) -> VerifyResult {
90      if pm_aggd.len() > 0xffff {
91          VerifyResult::Err(Error::AggSigTooBig)?;
92      }
93      if aggsig.len() != 32 * (pm_aggd.len() + 1) {
94          VerifyResult::Err(Error::InvalidSignature)?;
95      }
96      let u = pm_aggd.len();
97      let mut terms = Seq::<(Scalar, AffinePoint)>::new(2 * u);
98      let mut pmr = Seq::<(PublicKey, Message, Bytes32)>::new(u);
99      for i in 0..u {
```

BIP Half Agg Spec

Non-interactive aggregation ("compression") of BIP 340 signatures

```
32  pub type AggregateResult = Result<AggSig, Error>;
33  pub fn aggregate(pms: &Seq<(PublicKey, Message, Signature)>) -> AggregateResult {
```

```
89  pub fn verify_aggregate(aggsig: &AggSig, pm_aggd: &Seq<(PublicKey, Message)>) -> VerifyResult {
90      if pm_aggd.len() > 0xffff {
91          VerifyResult::Err(Error::AggSigTooBig)?;
92      }
93      if aggsig.len() != 32 * (pm_aggd.len() + 1) {
94          VerifyResult::Err(Error::InvalidSignature)?;
95      }
96      let u = pm_aggd.len();
97      let mut terms = Seq::<(Scalar, AffinePoint)>::new(2 * u);
98      let mut pmr = Seq::<(PublicKey, Message, Bytes32)>::new(u);
99      for i in 0..u {
```

halfagg.rs

BIP Half Agg Spec

Non-interactive aggregation ("compression") of BIP 340 signatures

```rust
pub type AggregateResult = Result<AggSig, Error>;
pub fn aggregate(pms: &Seq<(PublicKey, Message, Signature)>) -> AggregateResult {
```

```rust
pub fn verify_aggregate(aggsig: &AggSig, pm_aggd: &Seq<(PublicKey, Message)>) -> VerifyResult {
    if pm_aggd.len() > 0xffff {
        VerifyResult::Err(Error::AggSigTooBig)?;
    }
    if aggsig.len() != 32 * (pm_aggd.len() + 1) {
        VerifyResult::Err(Error::InvalidSignature)?;
    }
    let u = pm_aggd.len();
    let mut terms = Seq::<(Scalar, AffinePoint)>::new(2 * u);
    let mut pmr = Seq::<(PublicKey, Message, Bytes32)>::new(u);
    for i in 0..u {
```

halfagg.rs

```
$ cargo test
```

rust > python (proper type system, better error handling, cargo package manager, etc.)

rust > python (proper type system, better error handling, cargo package manager, etc.)

but perhaps **less readable** for some

rust > python (proper type system, better error handling, cargo package manager, etc.)

but perhaps **less readable** for some

However, the halfagg spec is **not** actually written in rust but in **hacspec**

https://github.com/hacspec/hacspec

Hacspec

succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust

https://github.com/hacspec/hacspec

Hacspec

succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust

verifiable: has formal semantics ("precise definition what each operation does")

https://github.com/hacspec/hacspec

Hacspec

succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust

verifiable: has formal semantics ("precise definition what each operation does")

- Pseudocode is whatever your interpretation is

https://github.com/hacspec/hacspec

Hacspec

succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust

verifiable: has formal semantics ("precise definition what each operation does")

- Pseudocode is whatever your interpretation is
- Python is whatever the python interpreter does

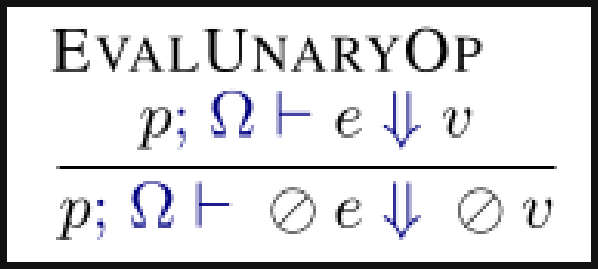
https://github.com/hacspec/hacspec

Hacspec

succinct, executable, verifiable specifications for high-assurance cryptography embedded in Rust

verifiable: has formal semantics ("precise definition what each operation does")

- Pseudocode is whatever your interpretation is
- Python is whatever the python interpreter does
- hacspec:



"Structured Operational Semantics"

Why Formal Semantics?

Why Formal Semantics?

Specification is **unambiguous**, which allows proper **reasoning** about it.

Why Formal Semantics?

Specification is **unambiguous**, which allows proper **reasoning** about it.

Can use computer tools, i.e., **proof assistants**, to prove that

Why Formal Semantics?

Specification is **unambiguous**, which allows proper **reasoning** about it.

Can use computer tools, i.e., **proof assistants**, to prove that

- The spec is free of array out-of-bound access or integer overflows

Why Formal Semantics?

Specification is **unambiguous**, which allows proper **reasoning** about it.

Can use computer tools, i.e., **proof assistants**, to prove that

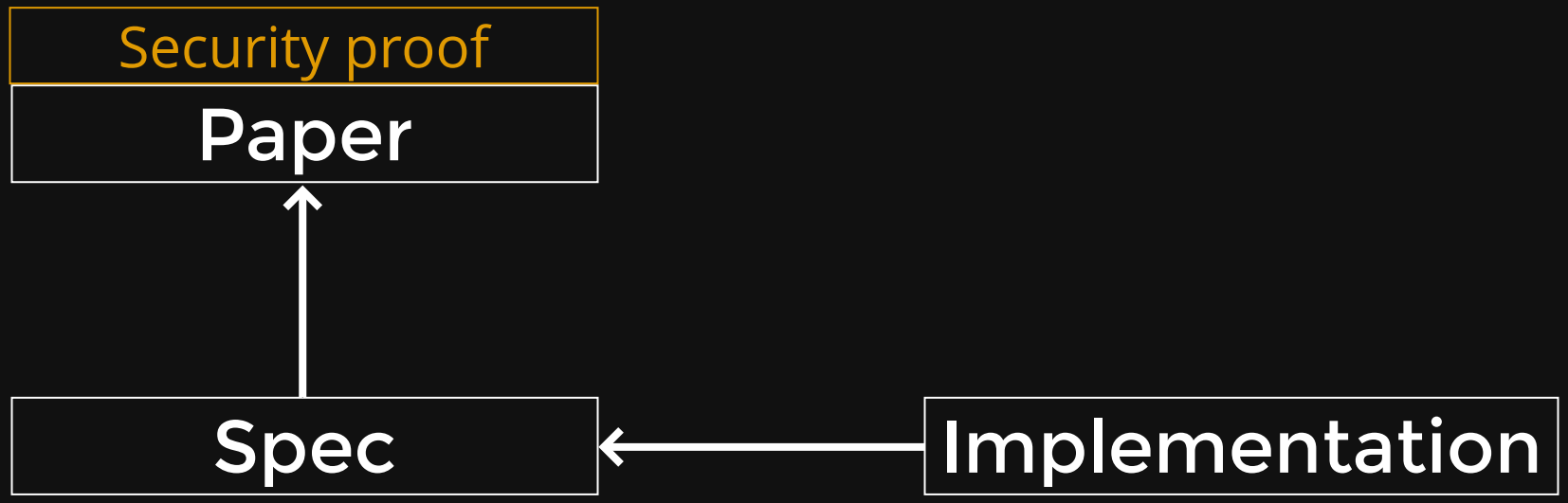
- The spec is free of array out-of-bound access or integer overflows
- The spec is complete: applying the aggregation algorithm to valid BIP 340 sigs **always** yields a valid half-aggregate sig

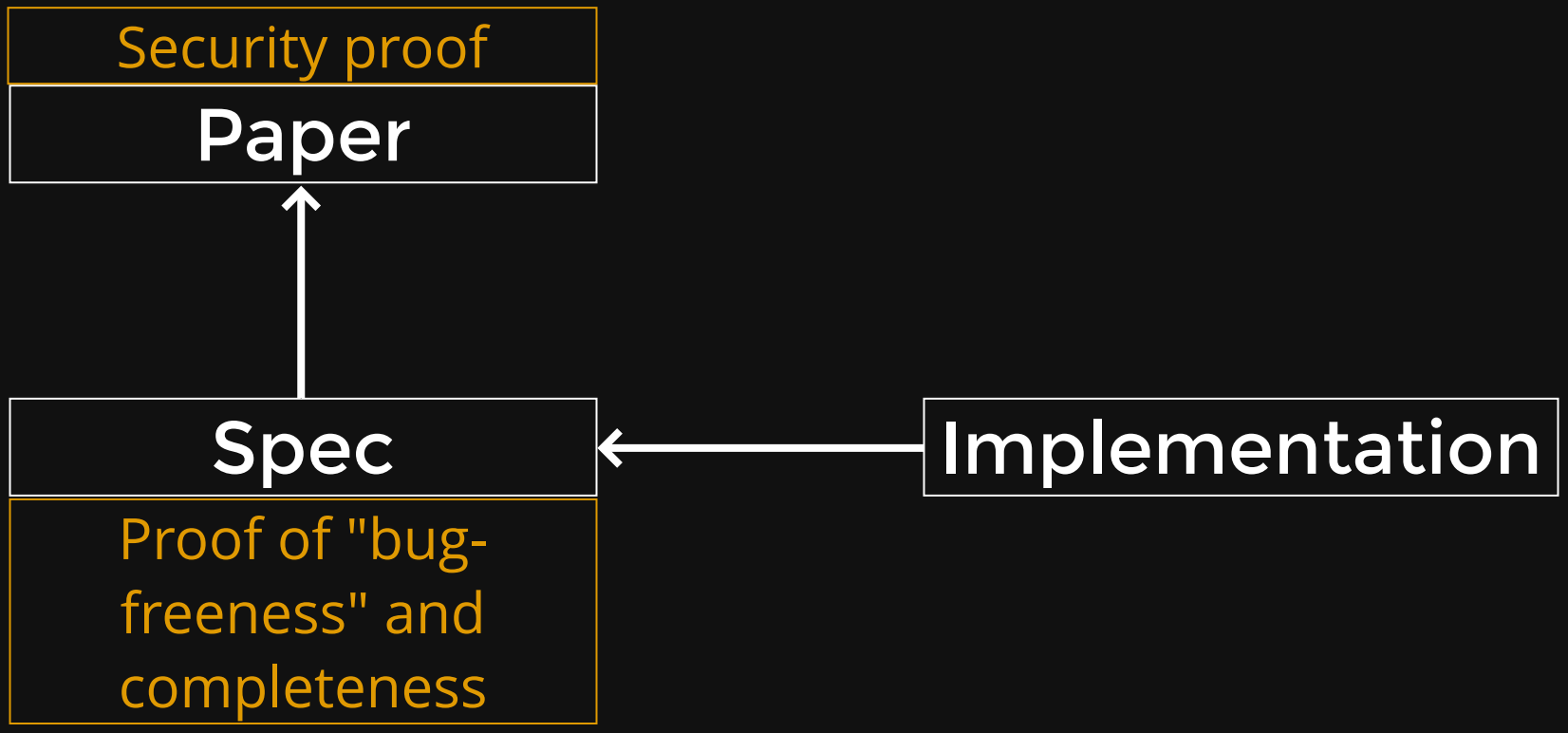
Why Formal Semantics?

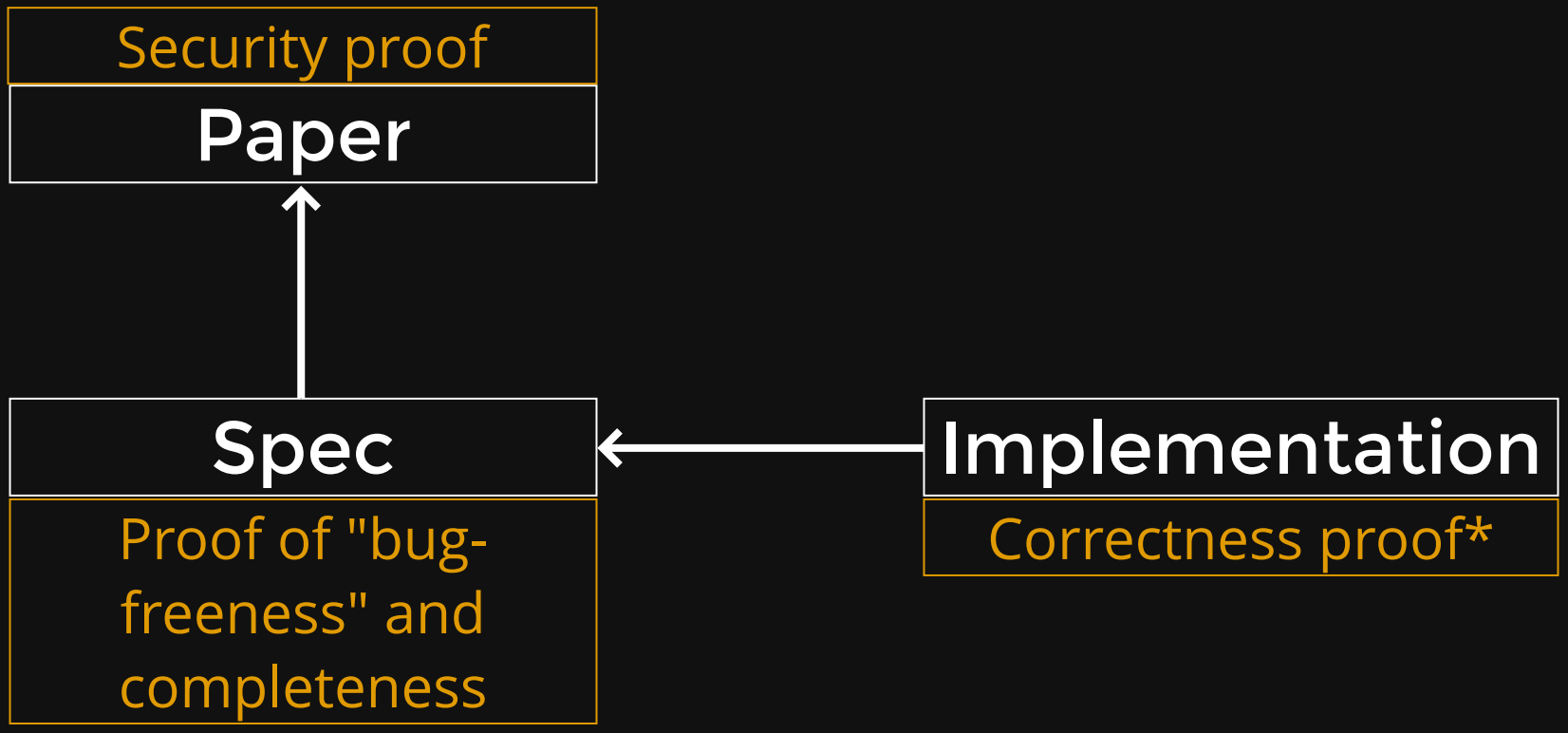
Specification is **unambiguous**, which allows proper **reasoning** about it.

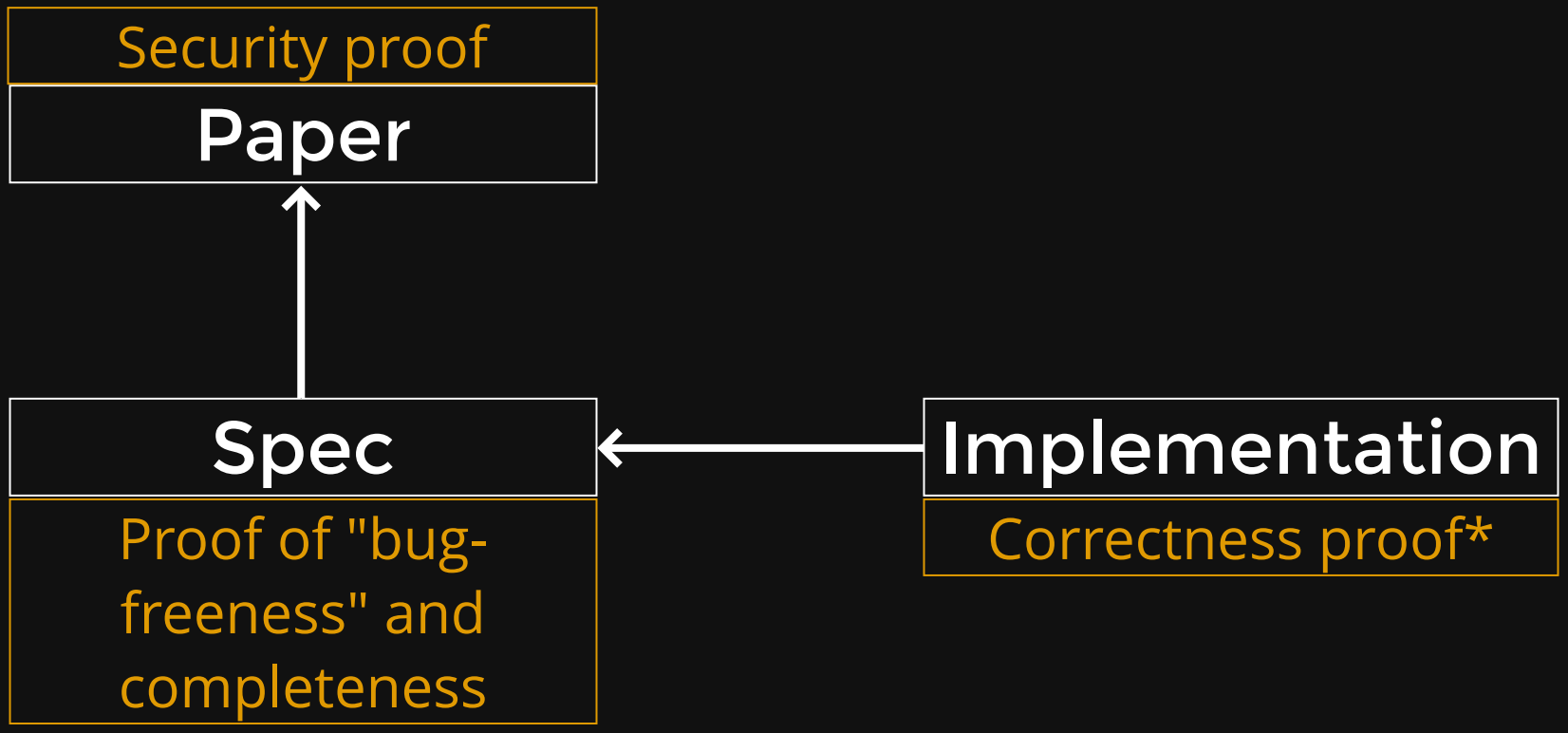
Can use computer tools, i.e., **proof assistants**, to prove that

- The spec is free of array out-of-bound access or integer overflows
- The spec is complete: applying the aggregation algorithm to valid BIP 340 sigs **always** yields a valid half-aggregate sig
- This implementation is correct: its behavior exactly matches that of the spec ("formal verification")

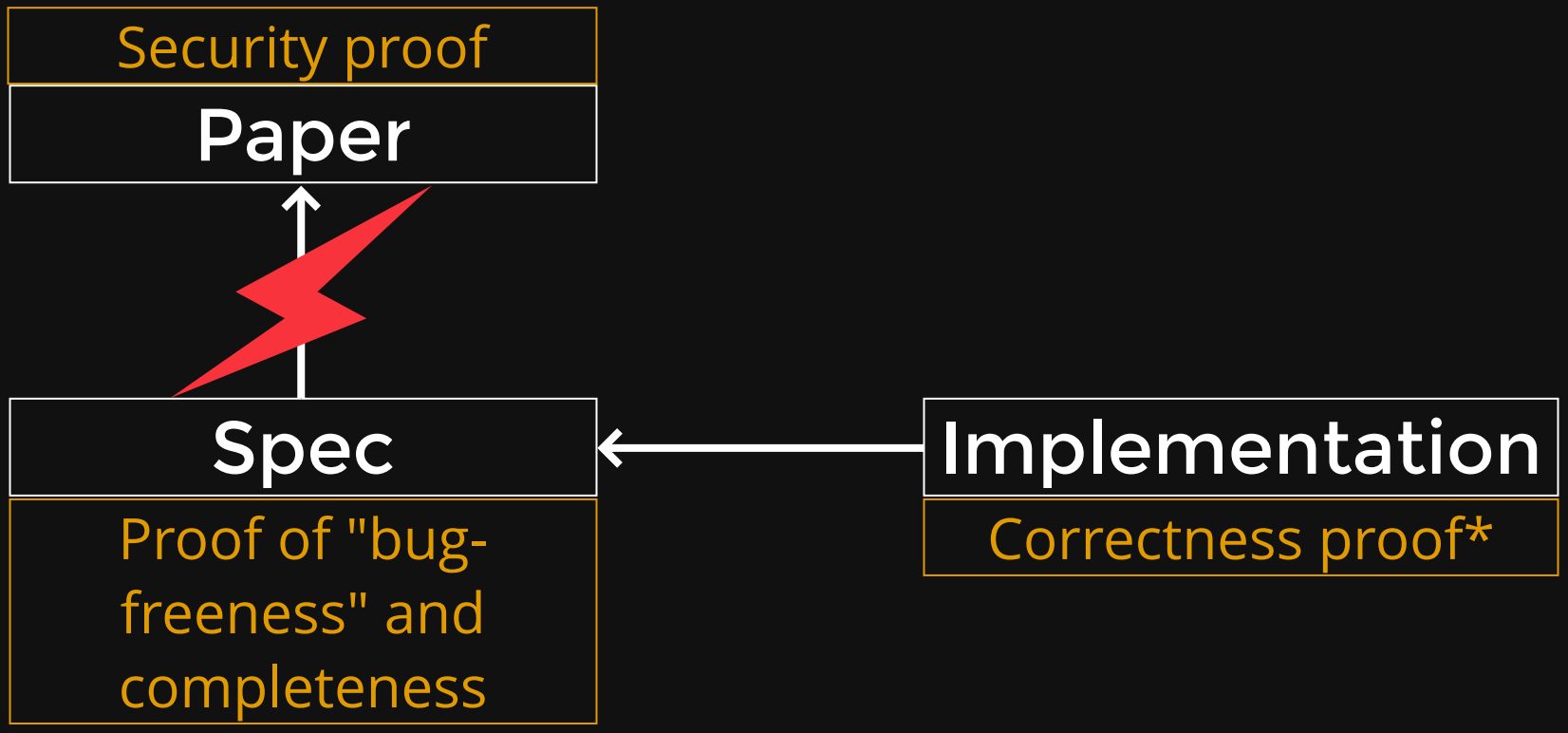

Security proof
Paper
Spec
Implementation


Security proof
Paper
Spec
Implementation
Proof of "bug-freeness" and completeness


Security proof
Paper
Spec
Proof of "bug-freeness" and completeness
Implementation
Correctness proof*

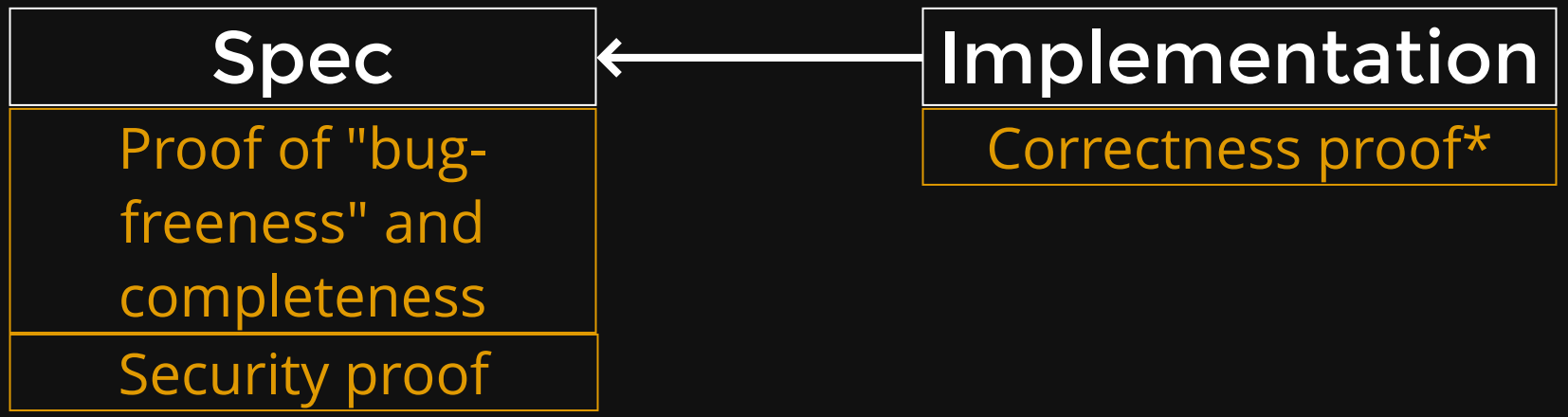


*typically requires implementations to be written in specialized language



*typically requires implementations to be written in specialized language

Far far future



Security proof

[bitcoin-dev] MuSig2 BIP

Jonas Nick jonasdnick at gmail.com
Tue Oct 11 15:34:23 UTC 2022

- Previous message: [bitcoin-dev] MuSig2 BIP
- Next message: [bitcoin-dev] Third version of Silent Payment implemen
- **Messages sorted by:** [date] [thread] [subject] [author]

It is still true that cryptography is hard, unfortunately. Yannick Seurin, Tim Ruffing, Elliott Jin, and I discovered an attack against the latest version of BIP MuSig2 in the case that a signer's individual key A = a*G is tweaked before giving it as input to key aggregation.

In more detail, a signer may be vulnerable if _all_ of the following conditions are true:

Security proof

```
cargo hacspec -o coq.v -e v hacspec-halfagg
```

```
cargo hacspec -o coq.v -e v hacspec-halfagg
```

```
183 Definition verify_aggregate
184   (aggsig_34 : agg_sig_t)
185   (pm_aggd_35 : seq (public_key_t × message_t))
186   : verify_result_t :=
187   ifbnd (seq_len (pm_aggd_35)) >.? (usize 65535) : bool
188   thenbnd (bind (@Err unit error_t (AggSigTooBig)) (fun _ => Ok (tt)))
189   else (tt) >> (fun 'tt =>
190   ifbnd (seq_len (aggsig_34)) !=.? ((usize 32) * ((seq_len (pm_aggd_35)) + (
191         usize 1))) : bool
192   thenbnd (bind (@Err unit error_t (InvalidSignature)) (fun _ => Ok (tt)))
193   else (tt) >> (fun 'tt =>
```

```
cargo hacspec -o coq.v -e v hacspec-halfagg
```

```
183 Definition verify_aggregate
184   (aggsig_34 : agg_sig_t)
185   (pm_aggd_35 : seq (public_key_t × message_t))
186   : verify_result_t :=
187   ifbnd (seq_len (pm_aggd_35)) >.? (usize 65535) : bool
188   thenbnd (bind (@Err unit error_t (AggSigTooBig)) (fun _ => Ok (tt)))
189   else (tt) >> (fun 'tt =>
190   ifbnd (seq_len (aggsig_34)) !=.? ((usize 32) * ((seq_len (pm_aggd_35)) + (
191         usize 1))) : bool
192   thenbnd (bind (@Err unit error_t (InvalidSignature)) (fun _ => Ok (tt)))
193   else (tt) >> (fun 'tt =>
```

```
257 Lemma No_Integer_overflow ...
258 Proof.
259 ...
260 Qed.
```

```
cargo hacspec -o coq.v -e v hacspec-halfagg
```

```
183 Definition verify_aggregate
184   (aggsig_34 : agg_sig_t)
185   (pm_aggd_35 : seq (public_key_t × message_t))
186   : verify_result_t :=
187   ifbnd (seq_len (pm_aggd_35)) >.? (usize 65535) : bool
188   thenbnd (bind (@Err unit error_t (AggSigTooBig)) (fun _ => Ok (tt)))
189   else (tt) >> (fun 'tt =>
190   ifbnd (seq_len (aggsig_34)) !=.? ((usize 32) * ((seq_len (pm_aggd_35)) + (
191         usize 1))) : bool
192   thenbnd (bind (@Err unit error_t (InvalidSignature)) (fun _ => Ok (tt)))
193   else (tt) >> (fun 'tt =>
```

```
257 Lemma No_Integer_overflow ...
258 Proof.
259 ...
260 Qed.
```

requires very specialized skills & a lot of work

Conclusion

Conclusion

- Consider writing (crypto) specifications in hacspec instead of pseudocode

Conclusion

- Consider writing (crypto) specifications in hacspec instead of pseudocode
- Strip python, add supplemental pseudocode (as in BIP half-agg)
  - do we need the pseudocode?

Conclusion

- Consider writing (crypto) specifications in hacspec instead of pseudocode
- Strip python, add supplemental pseudocode (as in BIP half-agg)
  - do we need the pseudocode?
- The above would already be an improvement for crypto BIPs

Conclusion

- Consider writing (crypto) specifications in hacspec instead of pseudocode
- Strip python, add supplemental pseudocode (as in BIP half-agg)
  - do we need the pseudocode?
- The above would already be an improvement for crypto BIPs
- Formal methods is an established research field with little practical relevance outside of some specialized areas, perhaps Bitcoin (crypto) specs are such an area

Conclusion

- Consider writing (crypto) specifications in hacspec instead of pseudocode
- Strip python, add supplemental pseudocode (as in BIP half-agg)
  - do we need the pseudocode?
- The above would already be an improvement for crypto BIPs
- Formal methods is an established research field with little practical relevance outside of some specialized areas, perhaps Bitcoin (crypto) specs are such an area
- Are there applications of this outside of cryptographic BIPs?

Conclusion

- Consider writing (crypto) specifications in hacspec instead of pseudocode
- Strip python, add supplemental pseudocode (as in BIP half-agg)
  - do we need the pseudocode?
- The above would already be an improvement for crypto BIPs
- Formal methods is an established research field with little practical relevance outside of some specialized areas, perhaps Bitcoin (crypto) specs are such an area
- Are there applications of this outside of cryptographic BIPs?
- Perhaps proof engineering gets easier in the future (GPT-3/Codex?). Until then hopefully we get people who do formal analysis on our specs.