TRAIL OF BITS

Implementing X.509 path validation for Python

William Woodruff, Trail of Bits

Introduction

Hello!

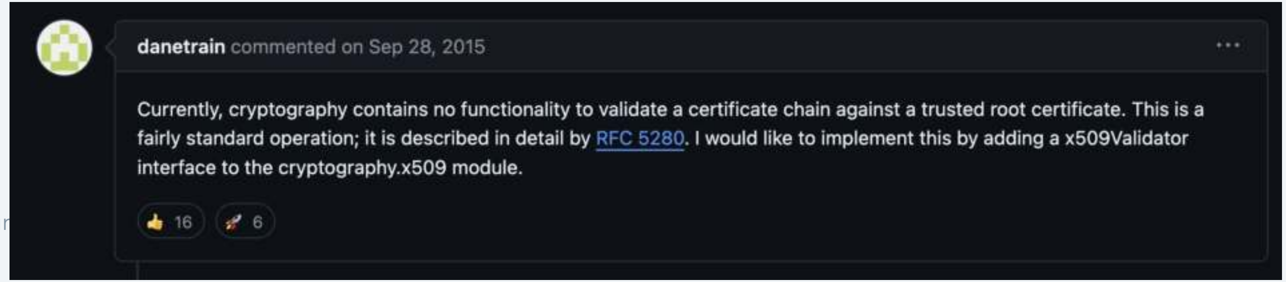
- **William Woodruff (william@trailofbits.com)**
  - open source group engineering director @ trail of bits
  - long-term OSS contributor (Homebrew, LLVM, Python) and maintainer (pip-audit, sigstore-python)
  - @yossarian@infosec.exchange
- **Trail of Bits**
  - ~130 person R&D firm, NYC based
  - specialities: cryptography, compilers, program analysis research, "supply chain", OSS package management, general high assurance software development

Agenda

- **Quick recap/background/mode-set on X.509 and path validation**
- **Path validation for Python**
  - Pre-existing efforts and implementations
  - Designing a new implementation from scratch
  - Our implementation
- **Testing the bejeezus out of it**
  - x509-limbo
  - Testing our implementations and others
  - Bugs everywhere
- **Lessons learned & future work**

Thank-yous

- **This work was funded by the Sovereign Tech Fund**
  - Themselves funded by 🇩🇪 via SPRIN-D
- **PyCA maintainers (Paul Kehrer and Alex Gaynor) defined the bounds of our approach, answered questions, and reviewed (and fixed) our changes throughout ~9 months of development**
  - x509-limbo is their idea, we just built it for them
- **Filippo Valsorda inspired the testvector design (at RWC 2023) and allowed us to home x509-limbo under C2SP**
- **Andrew Pan, Facundo Tuesca, Alex Cameron all designed, discussed, and engineered components of the implementation or tests**

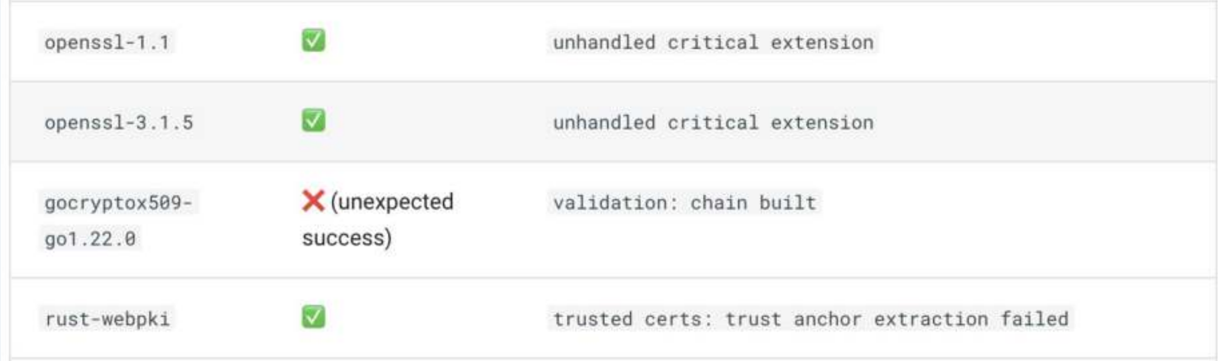


danetrain commented on Sep 28, 2015

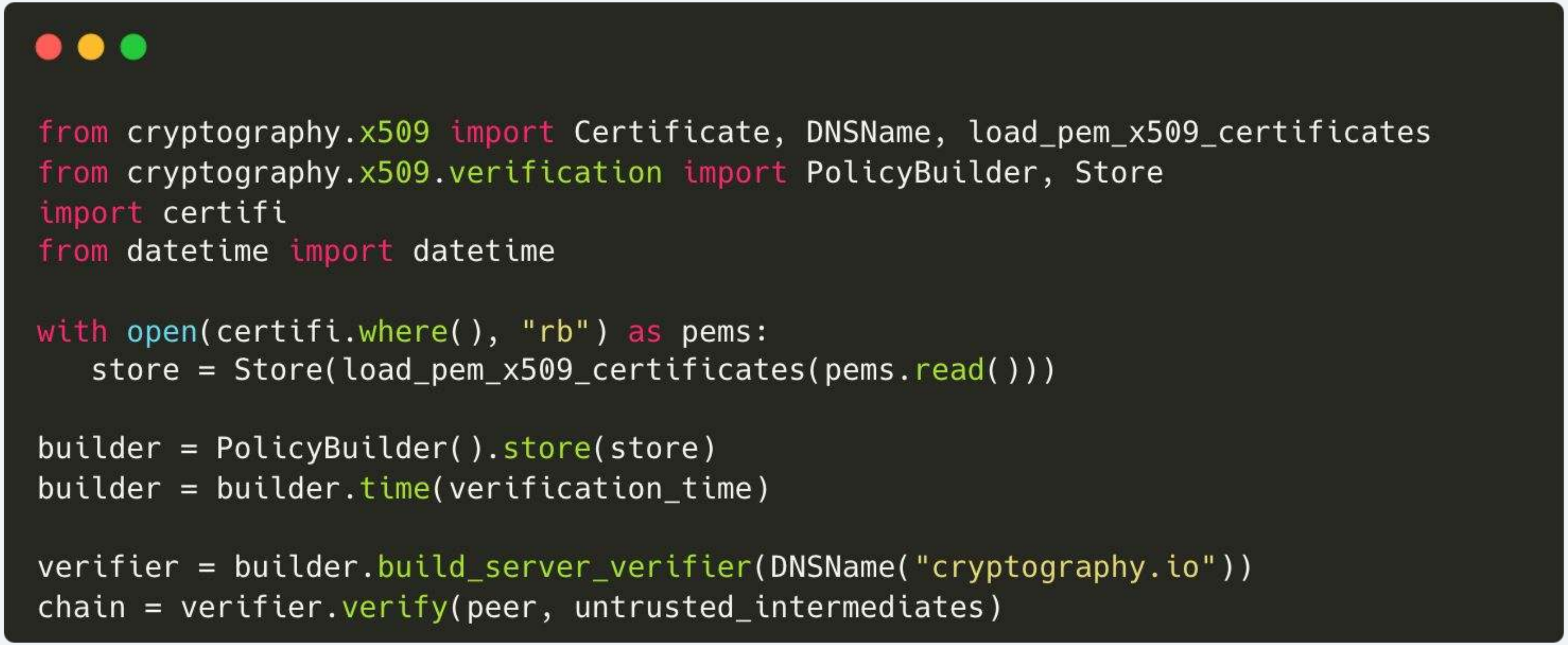
Currently, cryptography contains no functionality to validate a certificate chain against a trusted root certificate. This is a fairly standard operation; it is described in detail by RFC 5280. I would like to implement this by adding a x509Validator interface to the cryptography.x509 module.

Banner items

- **Python Cryptography now has X.509 path validation APIs!**
  - Work right out of the box with `pip install cryptography`!
  - Written in Rust, bound to Python with PyO3!
  - ~2500 new lines of code total
  - No OpenSSL*!
- **X.509 validation is scary, so we tested the crap out of it!**
  - x509-limbo: a new testsuite for path validation implementations!
  - We want **you** (other implementations) to integrate it into your tests!
  - It's already found bugs/discrepancies in other implementations!



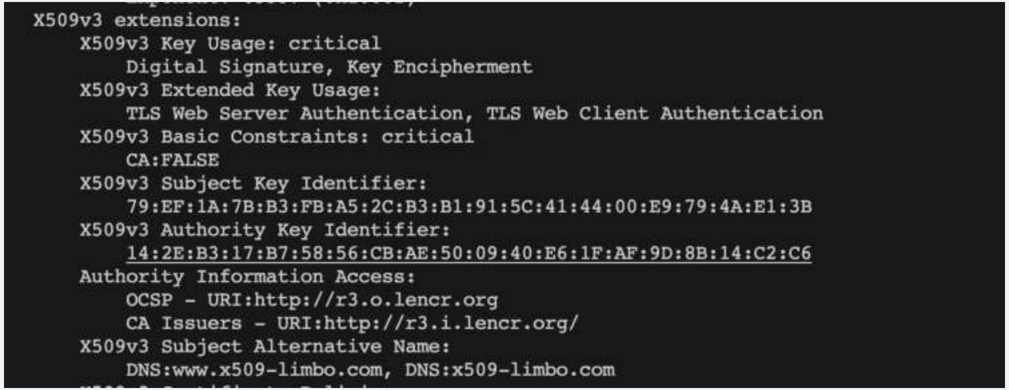
`openssl-1.1`	✅	`unhandled critical extension`
`openssl-3.1.5`	✅	`unhandled critical extension`
`gocryptox509-go1.22.0`	❌ (unexpected success)	`validation: chain built`
`rust-webpki`	✅	`trusted certs: trust anchor extraction failed`



```python
from cryptography.x509 import Certificate, DNSName, load_pem_x509_certificates
from cryptography.x509.verification import PolicyBuilder, Store
import certifi
from datetime import datetime

with open(certifi.where(), "rb") as pems:
   store = Store(load_pem_x509_certificates(pems.read()))

builder = PolicyBuilder().store(store)
builder = builder.time(verification_time)

verifier = builder.build_server_verifier(DNSName("cryptography.io"))
chain = verifier.verify(peer, untrusted_intermediates)
```

X.509 is…

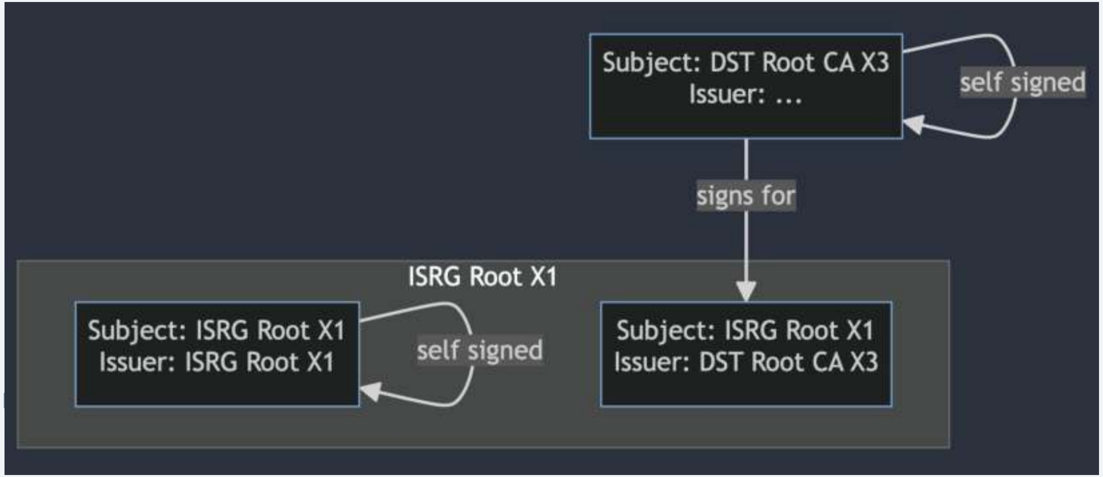
- **…a public key conveyance format**
  - “Public key `0xblahblahblah` is bound to subject `freestuff.example.com`”
  - Metadata includes expiration, machine & human-readable policy, expected usage, etc.
- **(pubkey, metadata) is signed over by a private key**
  - Usually a different private key than the signed-over pubkey’s private half!



```
X509v3 extensions:
    X509v3 Key Usage: critical
        Digital Signature, Key Encipherment
    X509v3 Extended Key Usage:
        TLS Web Server Authentication, TLS Web Client Authentication
    X509v3 Basic Constraints: critical
        CA:FALSE
    X509v3 Subject Key Identifier:
        79:EF:1A:7B:B3:FB:A5:2C:B3:B1:91:5C:41:44:00:E9:79:4A:E1:3B
    X509v3 Authority Key Identifier:
        14:2E:B3:17:B7:58:56:CB:AE:50:09:40:E6:1F:AF:9D:8B:14:C2:C6
    Authority Information Access:
        OCSP - URI:http://r3.o.lencr.org
        CA Issuers - URI:http://r3.i.lencr.org/
    X509v3 Subject Alternative Name:
        DNS:www.x509-limbo.com, DNS:x509-limbo.com
```

X.509 is…

- **… a chaining primitive:**
  - Certificate A can sign for Certificate B by conveying pubkey$_B$
- **…*not* a one-way key ←→ cert mapping**
  - A single CA can have multiple keypairs, each with its own certificate
    - …for rotation, fall over, or “just because” reasons
  - A *single* keypair can have multiple certificates!
    - …for policy reasons (e.g. expiry renewal without key rotation)
    - …for cross-issuance purposes (e.g. a CA certificate signed by a *different* CA)



X.509 is…

- **… a protocol for securely introducing principals to each other:**
  - Chaining allows us to go from a set of locally trusted entities (the OS/language trust store) to a remotely *untrusted* entity that we know (e.g. by DNS name) but don't previously trust
  - And vice versa, for mutual authentication ("client" auth in TLS land)
- **Principals can be introduced through untrusted intermediates!**
  - `example.com` can be securely connected to regardless of how many intermediate certs it sends, *so long as* the final chain construction is shaped like:

```
[example.com, …, untrusted, trusted]
```

x509-limbo.com
ISRG Root X1
DST Root CA X3

X.509 is…

- **… a complete dumpster fire of an ecosystem**
- **75% of the world runs on OpenSSL and forks (Libre, Boring)**
  - Forks are often better but severely constrained by API/ABI compatibility
  - OpenSSL's X.509 implementation is half RFC 3280/5280, half 🤷
  - The other 25% is OS implementations, mbedTLS, WolfSSL, GnuTLS, vendor crap, mystery meat, etc.
  - A bunch of this stuff runs on middleboxes and appliances that are **deployed and promptly forgotten about**
- **…a playground for exploitable logic and memory safety bugs**
  - Geometric combination of ASN.1/DER parsing bugs and X.509 protocol/policy errors
  - Severity ranges from remote triggerable DoS (moderately bad) to false positive verification (very bad) to remote disclosure of process memory (very, very bad)
  - Some of these bugs are baked into the specs themselves!
    - CVE-2023-0464, CVE-2023-23524

X.509 validation in Python

X.509 in Python: status quo

- **Standard library provides `ssl`, which is a relatively thin wrapper over OpenSSL**
  - Boring/LibreSSL can be used, but not officially supported upstream
  - Tied closely to the `socket` API; hard to reuse outside of TLS
  - Very lightly maintained (nobody wants to deal with OpenSSL)
- **Third-party libraries are (typically) wrappers over OpenSSL or platform APIs and are easy to misuse**
  - pyOpenSSL (OpenSSL via CFFI), M2Crypto (OpenSSL via SWIG)
  - Exception: `certvalidator` is pure Python, pulls from OS APIs via FFI
    - With a generic API!

X.509 in Python: design constraints

- **Baseline: TLS certificate validation, consistent with CABF**
  - Meaning: DNS/IP names + constraints, RFC 5280 + 6125, some additional stuff on top
- **Explicitly excluded:**
  - X.509v1, RFC 3280 and earlier profiles
  - CRLs, OCSP, AIA chasing, Certificate Transparency
  - Policy Constraints/Mapping
- **Needs to fully exist within PyCA Cryptography**
  - Only Python and (mostly) dep-free Rust
    - Absolutely no unsafe Rust
  - `pip install cryptography` must continue to work on all supported platforms
  - 100% test coverage + correctness over known problematic chains
  - Maintainers are speed freaks

X.509 in Python: design principles

On top of everything, follow Sleevi's laws:

- **There is no one true chain**
- **Treat path building as an abstract DFS problem**
  - With changing constraints (e.g. accumulated Name Constraints)
- **All rejections must be encoded into the search itself**
  - Rejecting a chain after it's built means potentially leaving other chains undiscovered
- **"Know your limits"**
  - Don't allow arbitrarily deep chains, excessive name/policy comparisons, or anything else that would not appear in a real PKI

X.509 in Python: implementation details

90% of the implementation is Rust, 10% is Python

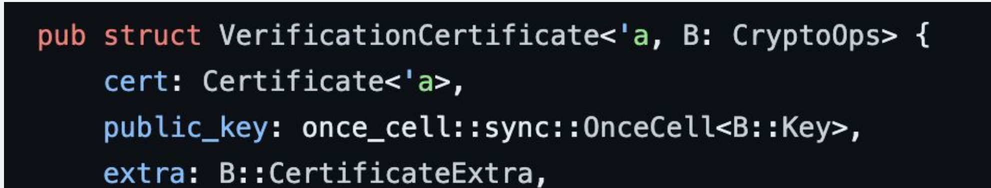
Rust:

- **Trust store, policy, key/SAN/NC parsing, core path building**
- **PyO3 types that get exposed to Python (but are themselves Rust)**

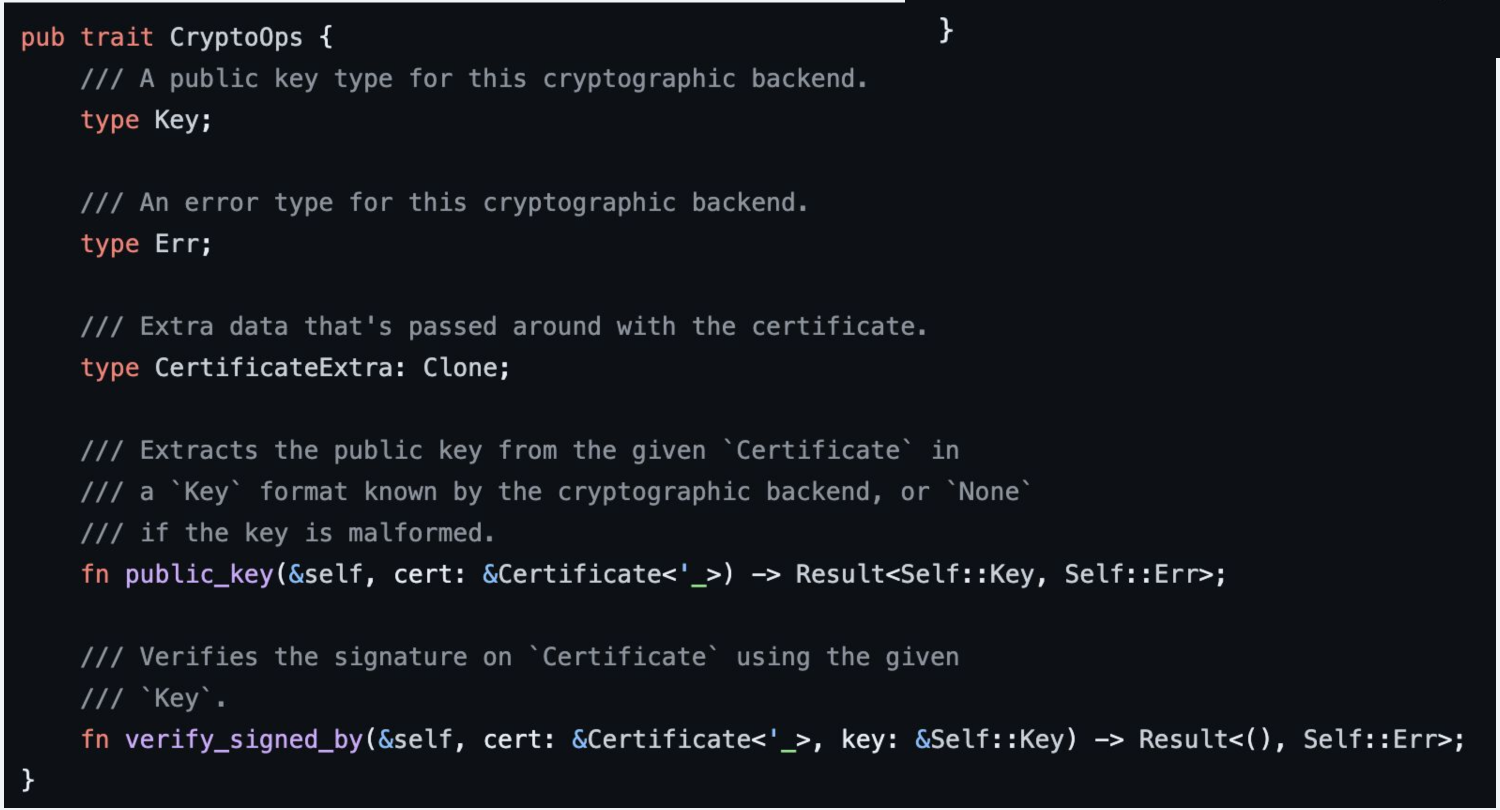
Python:

- `PolicyBuilder`**, misc. small types and type aliases**
- **Pre-existing** `cryptography.x509` **APIs for certs, extensions, names, etc.**

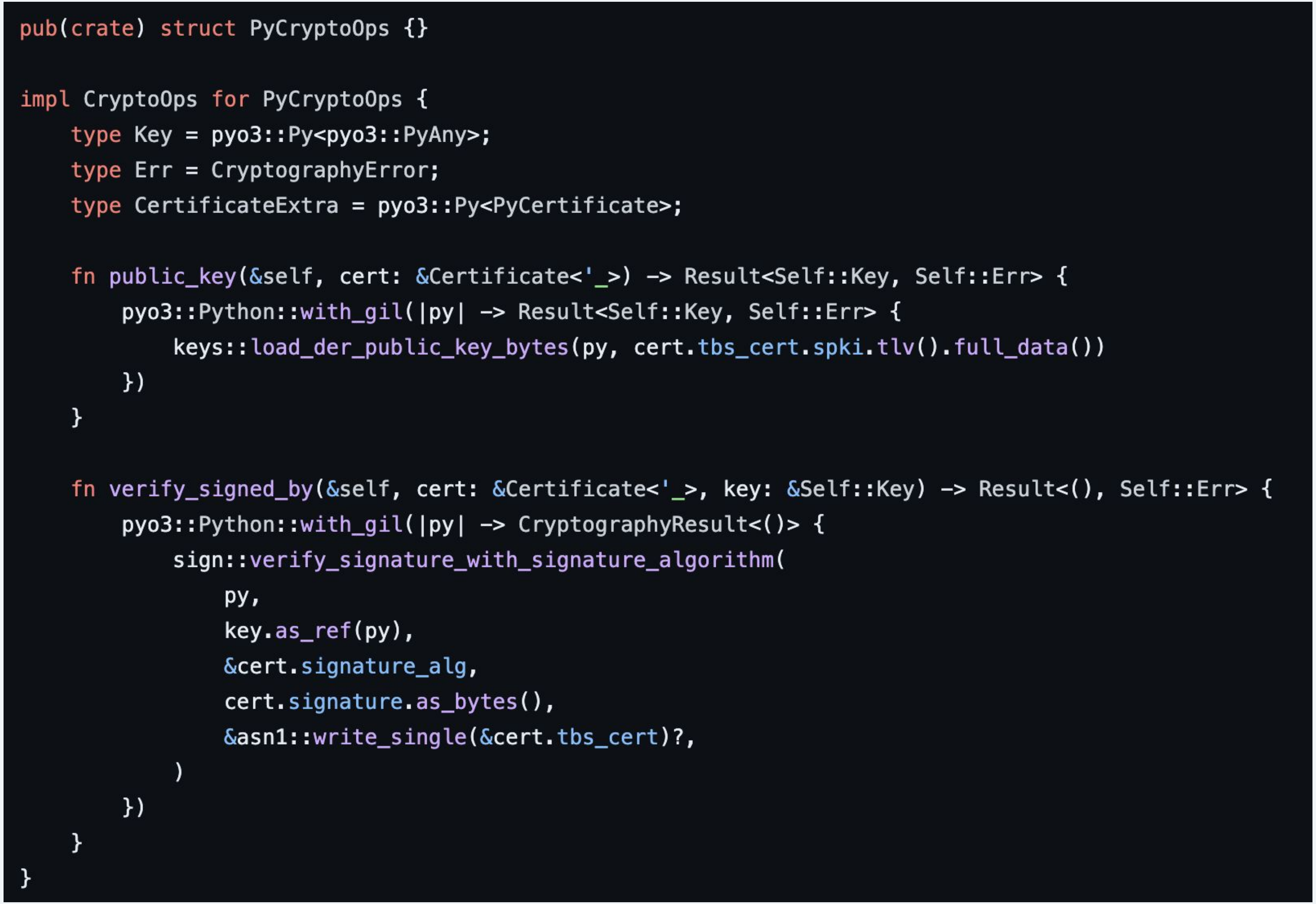
Where is the actual cryptography????



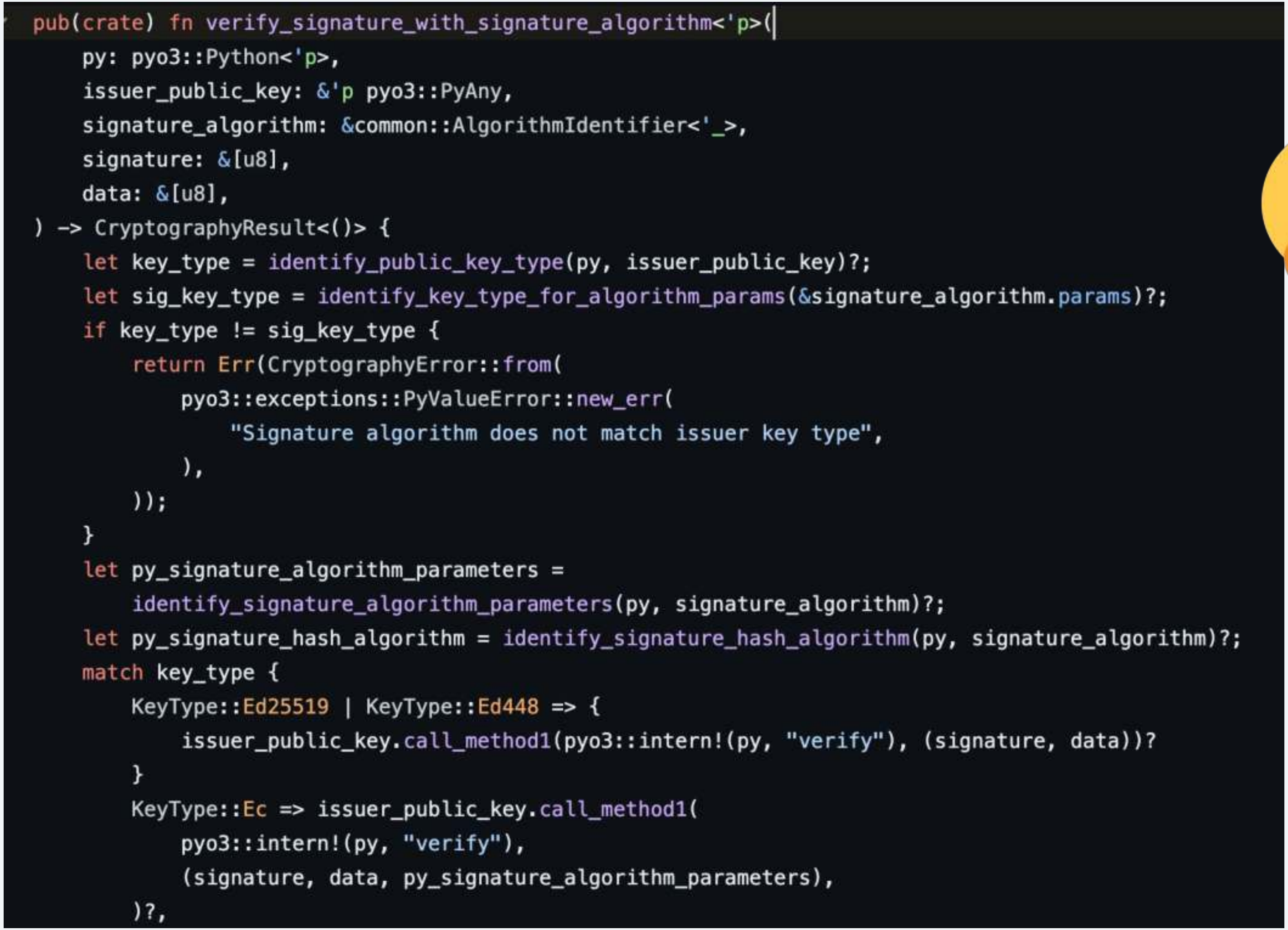
```rust
pub struct VerificationCertificate<'a, B: CryptoOps> {
    cert: Certificate<'a>,
    public_key: once_cell::sync::OnceCell<B::Key>,
    extra: B::CertificateExtra,
}
```



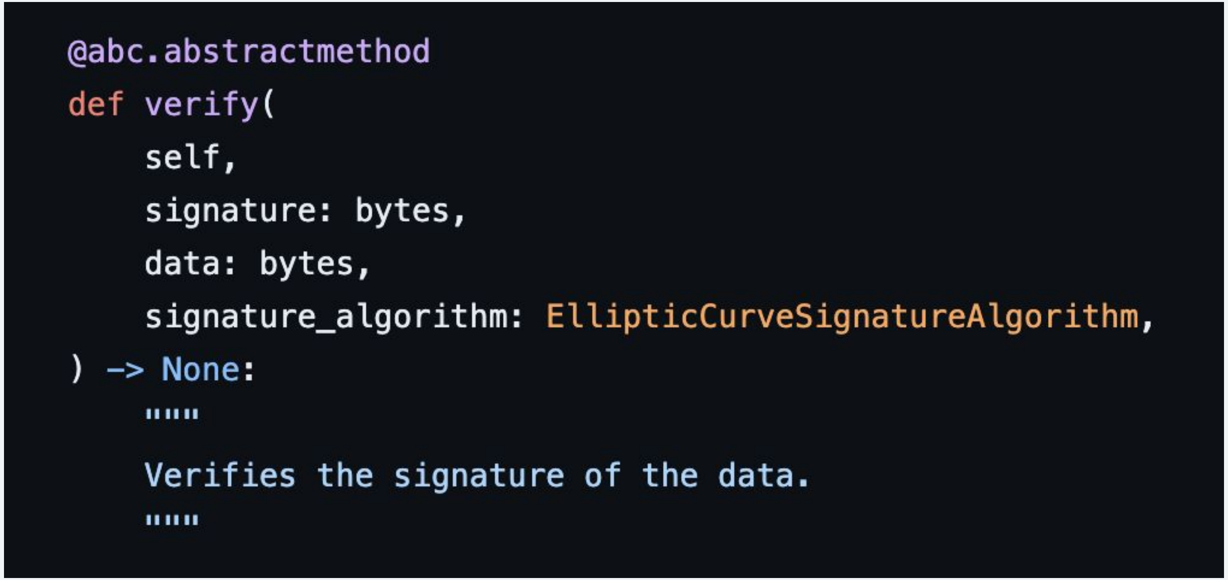
```rust
pub trait CryptoOps {
    /// A public key type for this cryptographic backend.
    type Key;

    /// An error type for this cryptographic backend.
    type Err;

    /// Extra data that's passed around with the certificate.
    type CertificateExtra: Clone;

    /// Extracts the public key from the given `Certificate` in
    /// a `Key` format known by the cryptographic backend, or `None`
    /// if the key is malformed.
    fn public_key(&self, cert: &Certificate<'_>) -> Result<Self::Key, Self::Err>;

    /// Verifies the signature on `Certificate` using the given
    /// `Key`.
    fn verify_signed_by(&self, cert: &Certificate<'_>, key: &Self::Key) -> Result<(), Self::Err>;
}
```



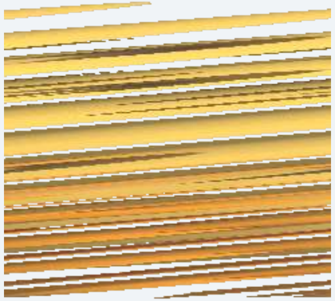
```rust
pub(crate) struct PyCryptoOps {}

impl CryptoOps for PyCryptoOps {
    type Key = pyo3::Py<pyo3::PyAny>;
    type Err = CryptographyError;
    type CertificateExtra = pyo3::Py<PyCertificate>;

    fn public_key(&self, cert: &Certificate<'_>) -> Result<Self::Key, Self::Err> {
        pyo3::Python::with_gil(|py| -> Result<Self::Key, Self::Err> {
            keys::load_der_public_key_bytes(py, cert.tbs_cert.spki.tlv().full_data())
        })
    }

    fn verify_signed_by(&self, cert: &Certificate<'_>, key: &Self::Key) -> Result<(), Self::Err> {
        pyo3::Python::with_gil(|py| -> CryptographyResult<()> {
            sign::verify_signature_with_signature_algorithm(
                py,
                key.as_ref(py),
                &cert.signature_alg,
                cert.signature.as_bytes(),
                &asn1::write_single(&cert.tbs_cert)?,
            )
        })
    }
}
```

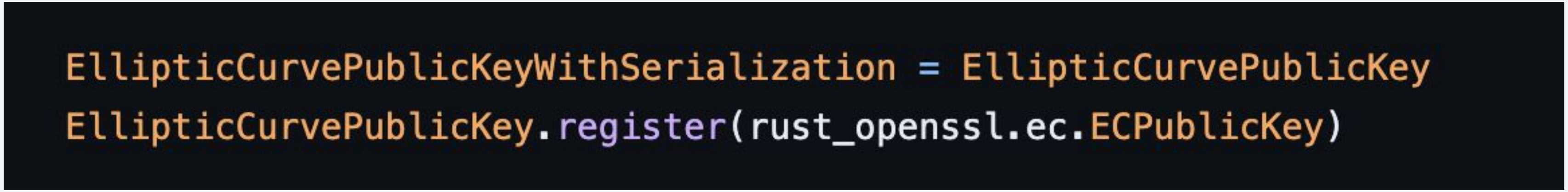


```rust
pub(crate) fn verify_signature_with_signature_algorithm<'p>(
    py: pyo3::Python<'p>,
    issuer_public_key: &'p pyo3::PyAny,
    signature_algorithm: &common::AlgorithmIdentifier<'_>,
    signature: &[u8],
    data: &[u8],
) -> CryptographyResult<()> {
    let key_type = identify_public_key_type(py, issuer_public_key)?;
    let sig_key_type = identify_key_type_for_algorithm_params(&signature_algorithm.params)?;
    if key_type != sig_key_type {
        return Err(CryptographyError::from(
            pyo3::exceptions::PyValueError::new_err(
                "Signature algorithm does not match issuer key type",
            ),
        ));
    }
    let py_signature_algorithm_parameters =
        identify_signature_algorithm_parameters(py, signature_algorithm)?;
    let py_signature_hash_algorithm = identify_signature_hash_algorithm(py, signature_algorithm)?;
    match key_type {
        KeyType::Ed25519 | KeyType::Ed448 => {
            issuer_public_key.call_method1(pyo3::intern!(py, "verify"), (signature, data))?
        }
        KeyType::Ec => issuer_public_key.call_method1(
            pyo3::intern!(py, "verify"),
            (signature, data, py_signature_algorithm_parameters),
        )?,
```

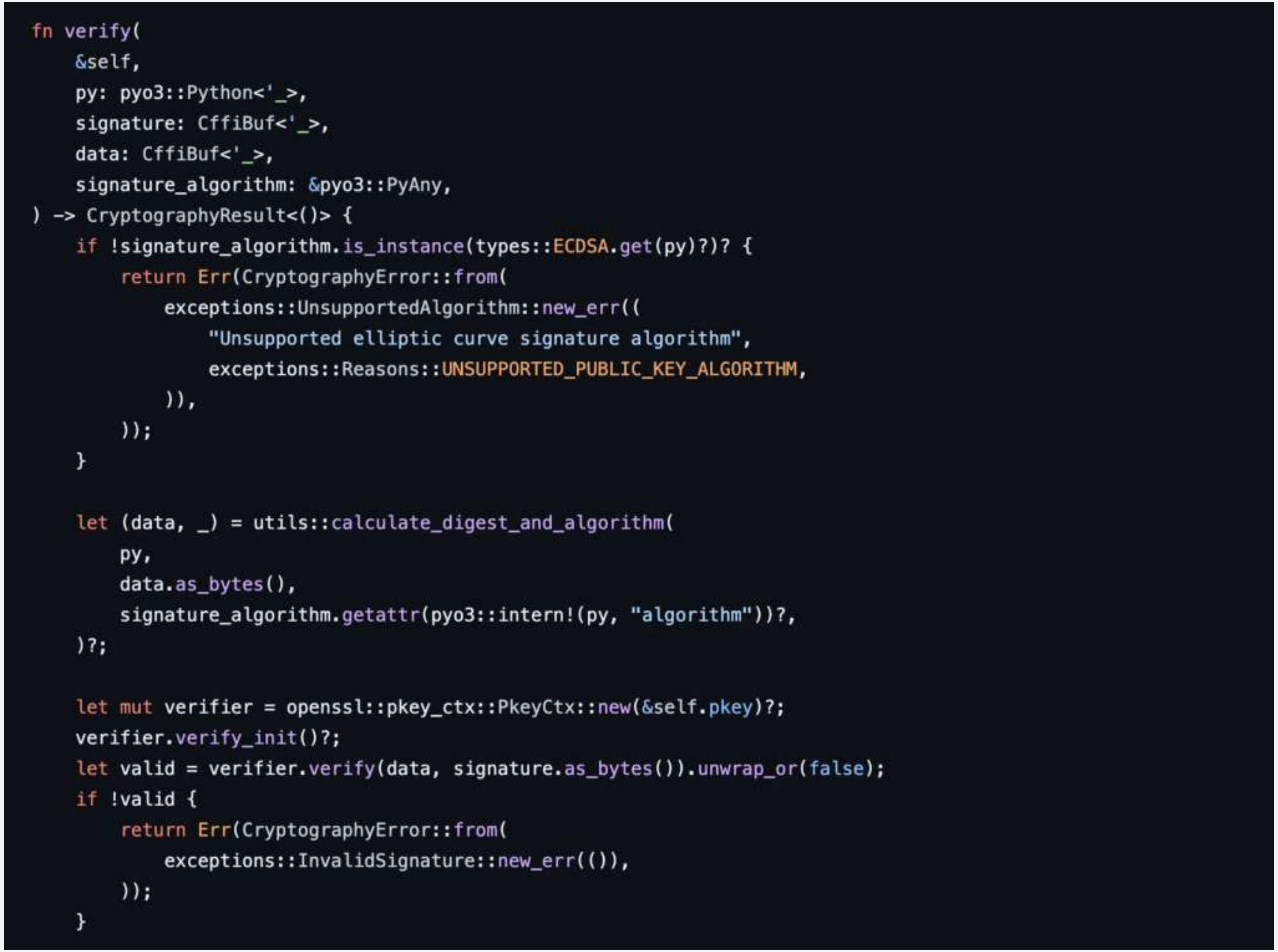


```python
@abc.abstractmethod
def verify(
    self,
    signature: bytes,
    data: bytes,
    signature_algorithm: EllipticCurveSignatureAlgorithm,
) -> None:
    """
    Verifies the signature of the data.
    """
```

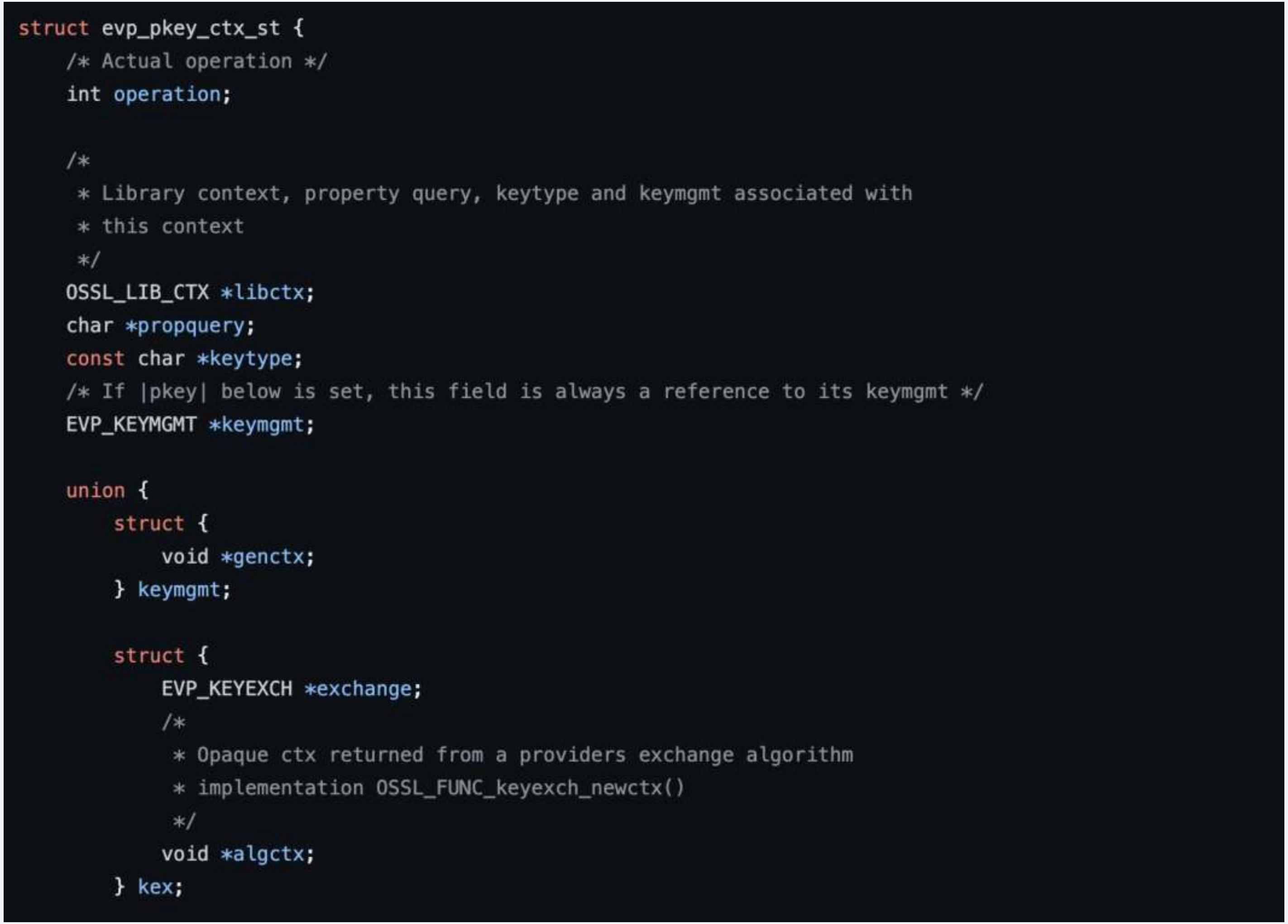




```python
EllipticCurvePublicKeyWithSerialization = EllipticCurvePublicKey
EllipticCurvePublicKey.register(rust_openssl.ec.ECPublicKey)
```



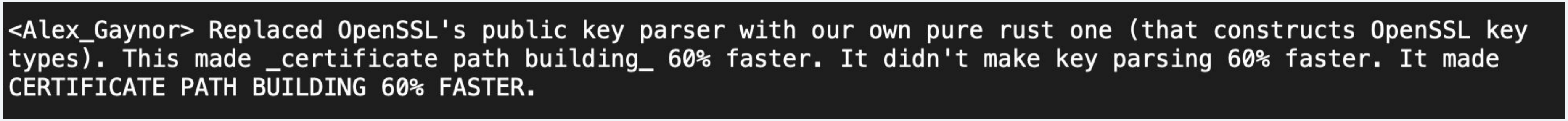
```rust
fn verify(
    &self,
    py: pyo3::Python<'_>,
    signature: CffiBuf<'_>,
    data: CffiBuf<'_>,
    signature_algorithm: &pyo3::PyAny,
) -> CryptographyResult<()> {
    if !signature_algorithm.is_instance(types::ECDSA.get(py)?)? {
        return Err(CryptographyError::from(
            exceptions::UnsupportedAlgorithm::new_err((
                "Unsupported elliptic curve signature algorithm",
                exceptions::Reasons::UNSUPPORTED_PUBLIC_KEY_ALGORITHM,
            )),
        ));
    }

    let (data, _) = utils::calculate_digest_and_algorithm(
        py,
        data.as_bytes(),
        signature_algorithm.getattr(pyo3::intern!(py, "algorithm"))?,
    )?;

    let mut verifier = openssl::pkey_ctx::PkeyCtx::new(&self.pkey)?;
    verifier.verify_init()?;
    let valid = verifier.verify(data, signature.as_bytes()).unwrap_or(false);
    if !valid {
        return Err(CryptographyError::from(
            exceptions::InvalidSignature::new_err(()),
        ));
    }
```



```c
struct evp_pkey_ctx_st {
    /* Actual operation */
    int operation;

    /*
     * Library context, property query, keytype and keymgmt associated with
     * this context
     */
    OSSL_LIB_CTX *libctx;
    char *propquery;
    const char *keytype;
    /* If |pkey| below is set, this field is always a reference to its keymgmt */
    EVP_KEYMGMT *keymgmt;

    union {
        struct {
            void *genctx;
        } keymgmt;

        struct {
            EVP_KEYEXCH *exchange;
            /*
             * Opaque ctx returned from a providers exchange algorithm
             * implementation OSSL_FUNC_keyexch_newctx()
             */
            void *algctx;
        } kex;
```

OpenSSL (or ilk) still provides the crypto itself

- **Carefully nestled behind abstractions, but still there in the shadows…**
  - Accessed through rust-openssl for invariant preservation
  - Abstractions mean that it *could* be removed, some day
- **But nothing else!**
  - X.509 parsing = pure Rust!
  - Chain building + profile/policy conformance = pure Rust!!
  - SPKI parsing/key extraction = pure Rust!!!

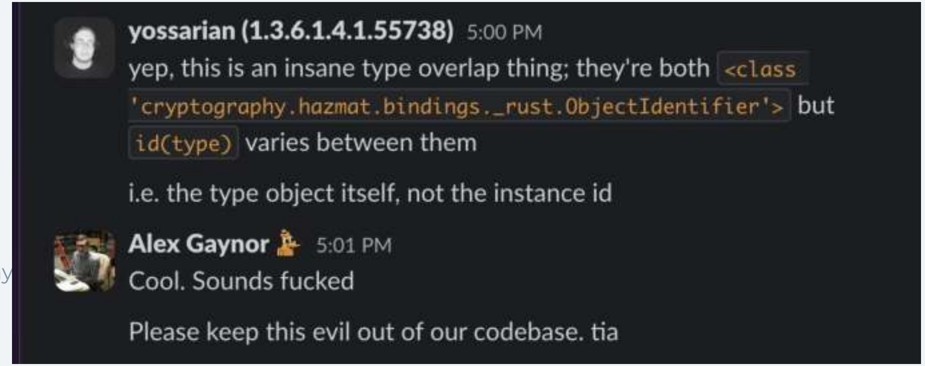


```
<Alex_Gaynor> Replaced OpenSSL's public key parser with our own pure rust one (that constructs OpenSSL key
types). This made _certificate path building_ 60% faster. It didn't make key parsing 60% faster. It made
CERTIFICATE PATH BUILDING 60% FASTER.
```

Testing

Testing: requirements

- **We can do a lot with unit tests in Rust and Python, but not enough**
  - Remember: 100% coverage requirement!
  - Cryptography ops/backend abstraction means that anything after sig verification needs to be in Python
  - PyO3 and `cargo test` don't always play nicely
- **Need a way to reach edge cases and annoying/pathological states within the validator**
  - Possibly gated on extensions, criticality, current NC set, etc.
  - In effect, we need a framework for rapidly churning out graphs/chains that contain each behavior under evaluation



yossarian (1.3.6.1.4.1.55738) 5:00 PM

yep, this is an insane type overlap thing; they're both `<class 'cryptography.hazmat.bindings._rust.ObjectIdentifier'>` but `id(type)` varies between them

i.e. the type object itself, not the instance id

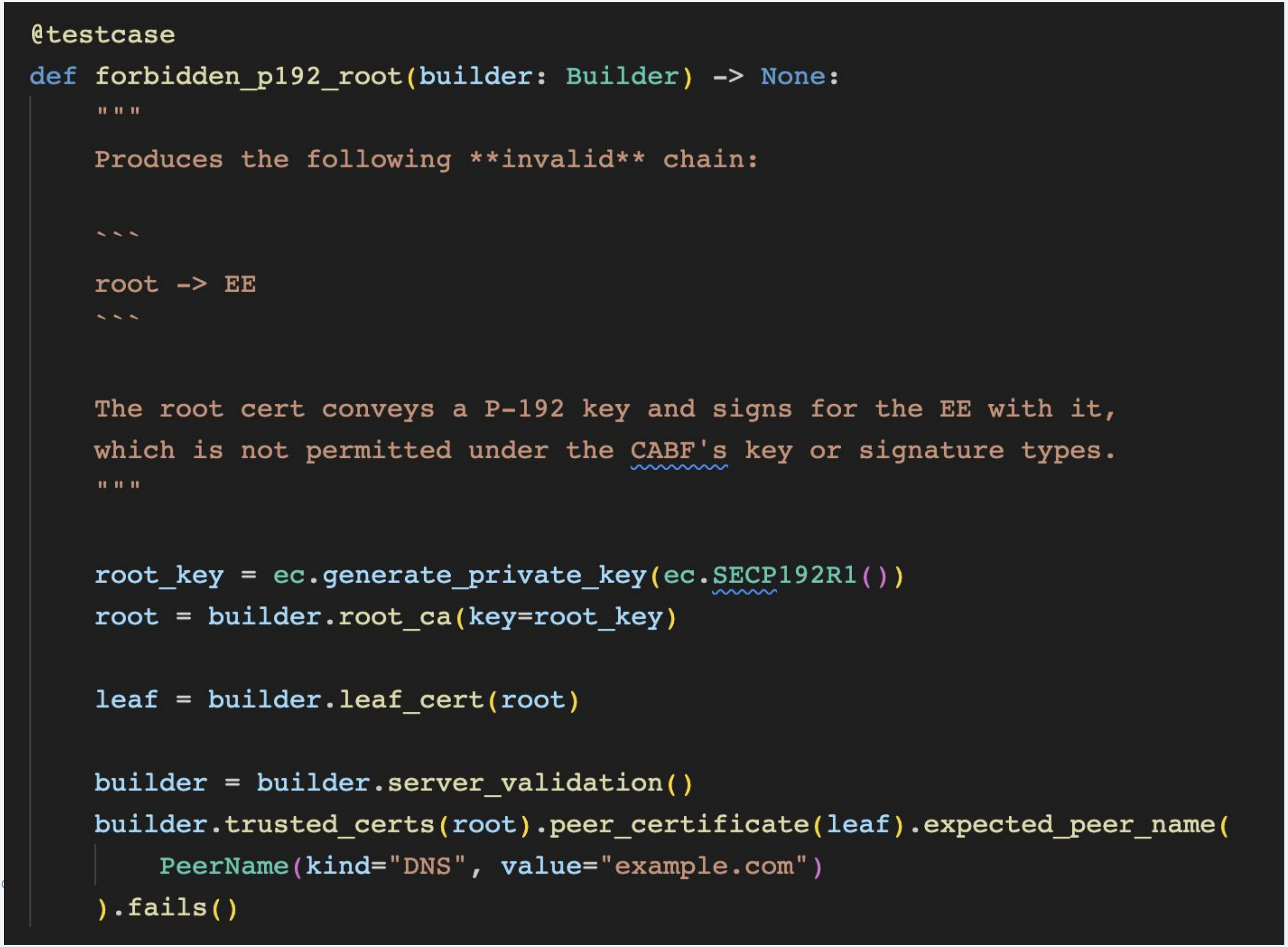
Alex Gaynor 5:01 PM

Cool. Sounds fucked

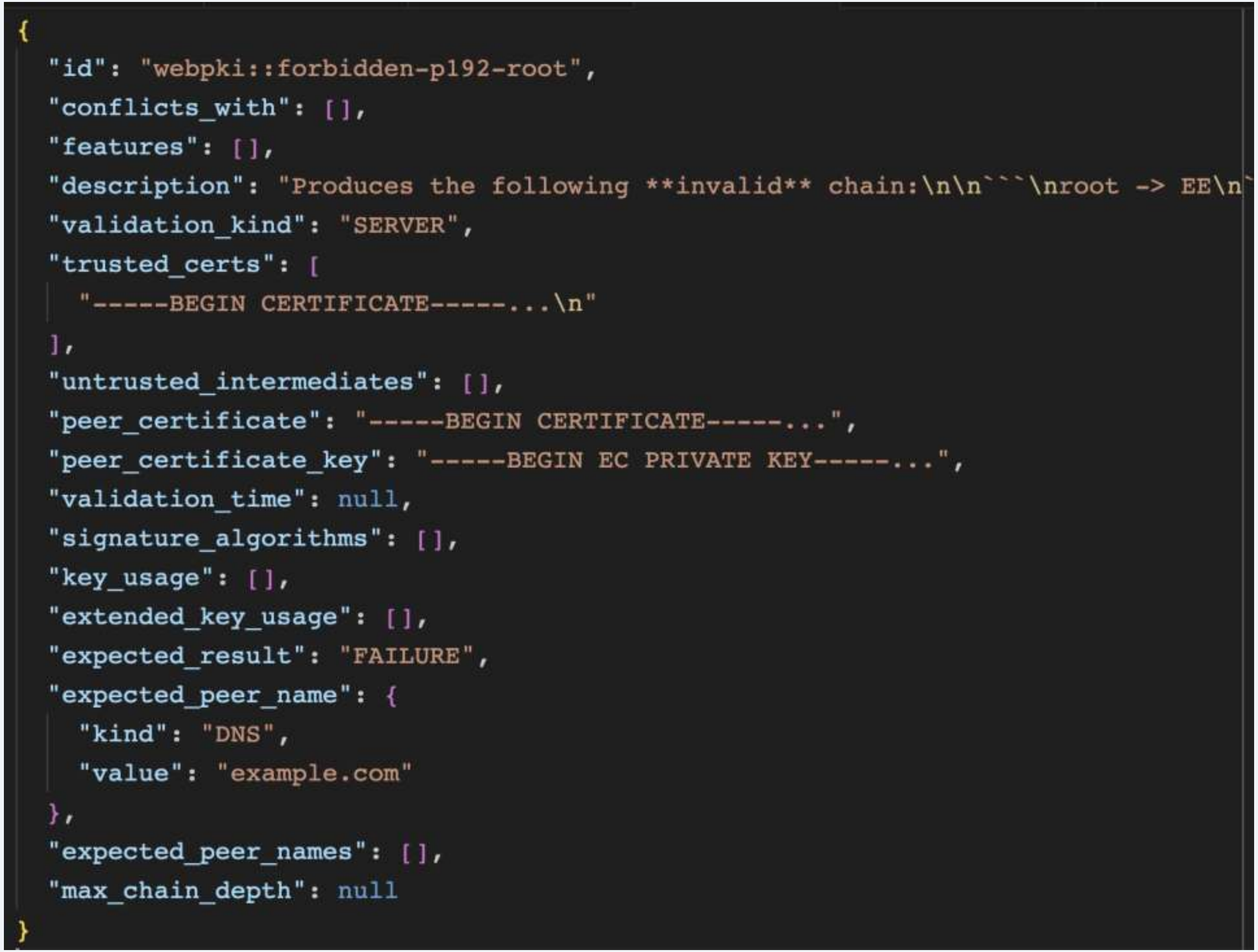
Please keep this evil out of our codebase. tia

Testing: x509-limbo

- **X.509 path validation testcases are inherently reusable, lots of prior art:**
  - BetterTLS (Netflix)
  - x509test (Google)
  - PITTv1/2/3, NIST PKITS, etc.
  - Each has a bespoke testcase/vector format, we want to unify them and add our own tests!
- **x509-limbo is our testsuite + testvector format**
  - Each testcase is (roots, intermediates, leaves, expected-result, extra-constraints)
  - Compiled to a giant blob of JSON
  - Contains both 3p tests (BetterTLS) and our own artisanal tests
  - Grouped into namespaces, e.g. `webpki::nc::*` for NC handling under the CABF profile



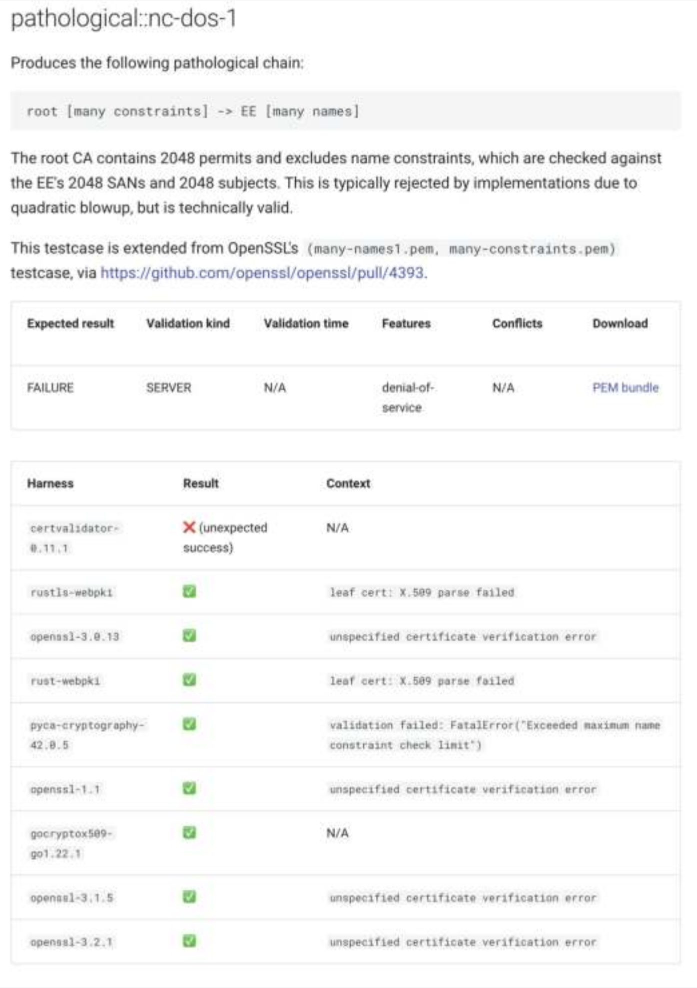
```python
@testcase
def forbidden_p192_root(builder: Builder) -> None:
    """
    Produces the following **invalid** chain:

    ```
    root -> EE
    ```

    The root cert conveys a P-192 key and signs for the EE with it,
    which is not permitted under the CABF's key or signature types.
    """

    root_key = ec.generate_private_key(ec.SECP192R1())
    root = builder.root_ca(key=root_key)

    leaf = builder.leaf_cert(root)

    builder = builder.server_validation()
    builder.trusted_certs(root).peer_certificate(leaf).expected_peer_name(
        PeerName(kind="DNS", value="example.com")
).fails()
```



~~~~
{
  "id": "webpki::forbidden-p192-root",
  "conflicts_with": [],
  "features": [],
  "description": "Produces the following **invalid** chain:\n\n```\nroot -> EE\n`
  "validation_kind": "SERVER",
  "trusted_certs": [
    "-----BEGIN CERTIFICATE-----...\n"
  ],
  "untrusted_intermediates": [],
  "peer_certificate": "-----BEGIN CERTIFICATE-----...",
  "peer_certificate_key": "-----BEGIN EC PRIVATE KEY-----...",
  "validation_time": null,
  "signature_algorithms": [],
  "key_usage": [],
  "extended_key_usage": [],
  "expected_result": "FAILURE",
  "expected_peer_name": {
    "kind": "DNS",
    "value": "example.com"
  },
  "expected_peer_names": [],
  "max_chain_depth": null
}
~~~~

Testing: x509-limbo

- **We want other implementations (Go, OpenSSL, etc.) to integrate x509-limbo into their testsuites!**
- **Created our own basic harnesses for each as reference**
- **Found/surfaced a bunch of bugs in the process**
    - False verifications, incorrect failures (rejecting chains that should be accepted)
    - Lots of Name Constraint bugs
    - Memory corruption: CVE-2024-28835: OOB caused by long cert chain in GnuTLS



pathological::nc-dos-1

Produces the following pathological chain:

```
root [many constraints] -> EE [many names]
```

The root CA contains 2048 permits and excludes name constraints, which are checked against the EE's 2048 SANs and 2048 subjects. This is typically rejected by implementations due to quadratic blowup, but is technically valid.

This testcase is extended from OpenSSL's `(many-names1.pem, many-constraints.pem)` testcase, via https://github.com/openssl/openssl/pull/4393.

Expected result	Validation kind	Validation time	Features	Conflicts	Download
FAILURE	SERVER	N/A	denial-of-service	N/A	PEM bundle

Harness	Result	Context
`certvalidator-0.11.1`	✗ (unexpected success)	N/A
`rustls-webpki`	✅	`leaf cert: X.509 parse failed`
`openssl-3.0.13`	✅	`unspecified certificate verification error`
`rust-webpki`	✅	`leaf cert: X.509 parse failed`
`pyca-cryptography-42.0.5`	✅	`validation failed: FatalError("Exceeded maximum name constraint check limit")`
`openssl-1.1`	✅	`unspecified certificate verification error`
`gocryptox509-go1.22.1`	✅	N/A
`openssl-3.1.5`	✅	`unspecified certificate verification error`
`openssl-3.2.1`	✅	`unspecified certificate verification error`

Lessons learned & future work

Lessons learned

RFCs and CABF say one thing, implementations do another

- **CABF says to prefer SAN always; many implementations check Subject or both**
- **Many implementations don't bother with SN checks**
  - Guessable/fixed serials were what made Flame (2008) possible!
- **Most implementations have at least one NC bug, causing false negatives**
  - Typically in cross-issuance/self-issuance graphs
- **Most implementations handle self-issued cert pathlens incorrectly**
- **Many implementations ignore/don't enforce presence of SKI/AKI**

Lessons learned

X.509 has plenty of quirks to lawyer over

- **Negative 20-octet serial numbers that are really 21 octets when DER encoded, valid or not?**
- `0.999… != 1` **according to RFC 5280**
- **“Root” certs are a lie, only “trust anchors” exist**
  - Unless you’re in CABF, in which case trust anchors are a lie and only roots exist
- **Unclear whether conveying a TA as a certificate means respecting its exts/constraints**
  - PyCA respects constraints on roots, other impls (notably rustls) do not
  - …but some browsers do? Some of the time?
- **Unclear whether the leaf’s pubkey is subject to CABF constraints**

Lessons learned

Despite complexity and ambiguity, things are Pretty Good™:

- **The Web PKI is in *much* better shape in 2024 than 2014, thanks to CABF, shorter validities, Certificate Transparency, and *higher expectations***
  - No more secretly issued intermediate CAs, no more toothless audit failures
  - No more BER or malformed DER roots, no more V1 roots
  - Only a small handful of invalid serials left
- **Hardest part was determining and testing PyCA's "break budget" vs. other implementations, not the actual code itself**
  - Core path building is <400 lines of well-commented Rust
- **Some bugs still snuck through!**
  - Root key strength check bug, unknown NC handling bug, datetime object TZ bug, …
  - Each got a new x509-limbo case, revealing bugs in other impls too!

Future work

- **Only TLS server cert validation is implemented**
  - TLS client cert validation ~~soon™~~: https://github.com/pyca/cryptography/pull/10345
    - Merged, but not in a release yet
- **Other profiles? More configuration knobs?**
  - Intel SGX, Authenticode, etc.
  - Users want even more control over which key/signature algorithms are allowed
    - Sometimes users want bad things
  - Guiding principle: future knobs/config points must not make the current APIs harder to use/easier to misuse
- **Spread the gospel of x509-limbo**
  - Get more implementations to adopt it upstream
  - Cannibalize more related test suites
  - Support profiles other than RFC 5280 and CABF

Thank you!

Slides will be available here:

https://yossarian.net/publications#oscw-2024

Resources:

- **ToB blog: We build X.509 chains so you don't have to**
- **Ryan Sleevi: Path Building vs Path Verifying: The Chain of Pain**
- **Andrew Ayer: Fixing the Breakage from the AddTrust External CA Root Expiration**
- **Robert Alexander: Name "Constrain't" on Chrome**
- **Many specs:**
  - RFC 5280 (X.509 PKIX)
  - RFC 6125 (domain-based identities, wildcards in PKIX)
  - CABF BRs