Developing a Zeek plugin with Rust

Zeek Workshop
CERN
2026-03-25

Benjamin Bannier

But why would you want to do that???

Today we have

- Zeek script for customizing Zeek runtime behavior and scripting
- JavaScript for handling events
- Spicy for implementing analyzers
- C++ API for adding components and implementing builtin functions (bifs)

Why would anyone want to add yet another language to that pile?

But why would you want to do that???

No way around C++ API if we need

- general purpose tool to extend arbitrary aspects of Zeek
- good control over runtime performance
- implement builtin functions

But why would you want to do that???

No way around C++ API if we need

- general purpose tool to extend arbitrary aspects of Zeek
- good control over runtime performance
- implement builtin functions

Personal observation:

> In the Zeek ecosystem packages providing bifs seem rare.

Possible explanations:

1. Zeek already comes with everything one would ever need
2. Implementing bifs is hard since it requires use of the C++ API

But why would you want to do that???

No way around C++ API if we need

- general purpose tool to extend arbitrary aspects of Zeek
- good control over runtime performance
- implement builtin functions

Personal observation:

> In the Zeek ecosystem packages providing bifs seem rare.

Possible explanations:

1. Zeek already comes with everything one would ever need
2. Implementing bifs is hard since it requires use of the C++ API

Using an API in a different ecosystem can provide a fresh look.

A serializer plugin

```
namespace Zeek_more_serializers {

class Plugin : public zeek::plugin::Plugin {
protected:
  zeek::plugin::Configuration Configure() override;
};

extern Plugin plugin;

} // namespace Zeek_more_serializers
```

A serializer plugin

```
namespace Zeek_more_serializers {

class Plugin : public zeek::plugin::Plugin {
protected:
  zeek::plugin::Configuration Configure() override;
};

extern Plugin plugin;

} // namespace Zeek_more_serializers
```

```
zeek::plugin::Configuration
Zeek_more_serializers::Plugin::Configure() {
  zeek::plugin::Configuration config;
  config.name = "Zeek::more_serializers";
  config.description = "Additional serializers";
  config.version.major = VERSION_MAJOR;
  config.version.minor = VERSION_MINOR;
  config.version.patch = VERSION_PATCH;

  AddComponent(
    new zeek::cluster::EventSerializerComponent(
      "binary",
      cluster::event::Serializer::Instantiate));

  return config;
}
```

A serializer plugin

A cluster event serializer needs to implement

```
class EventSerializer {
  virtual bool SerializeEvent(byte_buffer& buf, const cluster::Event& event) = 0;

  virtual std::optional<cluster::Event> UnserializeEvent(byte_buffer_span buf) = 0;
};
```

A serializer plugin

A cluster event serializer needs to implement

```
class EventSerializer {
  virtual bool SerializeEvent(byte_buffer& buf, const cluster::Event& event) = 0;

  virtual std::optional<cluster::Event> UnserializeEvent(byte_buffer_span buf) = 0;
};
```

```
class Event {
  Event(const EventHandlerPtr&, zeek::Args, zeek::detail::EventMetadataVectorPtr);
};

using Args = std::vector<ValPtr>;
using ValPtr = IntrusivePtr<Val>;
```

A serializer plugin

A cluster event serializer needs to implement

```
class EventSerializer {
  virtual bool SerializeEvent(byte_buffer& buf, const cluster::Event& event) = 0;

  virtual std::optional<cluster::Event> UnserializeEvent(byte_buffer_span buf) = 0;
};
```

```
class Event {
  Event(const EventHandlerPtr&, zeek::Args, zeek::detail::EventMetadataVectorPtr);
};

using Args = std::vector<ValPtr>;
using ValPtr = IntrusivePtr<Val>;
```

For testing we also want builtin functions:

```
function serialize_val%(val: any, format: Format%): string
function deserialize_val%(val: string, ty: any, format: Format%): any
```

Approach

Use `serde` (https://serde.rs/) to implement serializers for Rust types:

- provides generic protocol into which concrete serializers can hook
- implement Rust types which model `Val` and `cluster::Event` which do `#[derive(Serialize, Deserialize)]`

Provide functions for serializing and deserializing, and somehow expose these functions to C++ for use in C++ plugin:

```
fn serialize_event(event: &Event, format: Format, buf: Pin<&mut ByteBuffer>) -> Result<()>;
fn deserialize_event(buf: &[u8], format: Format) -> Result<UniquePtr<Event>>;

fn serialize_val(value: &Val, format: Format, buf: Pin<&mut ByteBuffer>) -> Result<()>;
fn deserialize_val(buf: &[u8], format: Format, ty: &TypePtr) -> Result<UniquePtr<ValPtr>>;
```

Leave the rest of the plugin implementation in C++ for now.

Interfacing Rust and C++

Rust can directly interface C, e.g.,

```
#[unsafe(no_mangle)]
pub extern "C" fn add(left: u64, right: u64) -> u64 { left + right }
```

can directly be called from C or C++ as

```
extern uint64_t add(uint64_t left, uint64_t right);
```

Interfacing Rust and C++

Rust can directly interface C, e.g.,

```
#[unsafe(no_mangle)]
pub extern "C" fn add(left: u64, right: u64) -> u64 { left + right }
```

can directly be called from C or C++ as

```
extern uint64_t add(uint64_t left, uint64_t right);
```

C++ has additional features which do not map natively, e.g.,

- RAII with constructors and destructors
- inheritance
- exceptions

Rust has e.g., destructive moves and all types are trivially relocatable which does not align with C++ semantics.

Generics in neither Rust nor C++ translate across the language boundary.

Interfacing Rust and C++: types

Use low-level approach of exposing C++ types and functions by hand with `cxx` (https://cxx.rs).

```
#[cxx::bridge]
mod ffi {
    #[derive(Debug)]
    enum TypeTag {
        TYPE_VOID,
        TYPE_BOOL,
        TYPE_INT,
        ...
    }

    #[namespace = "::zeek"]
    unsafe extern "C++" {
        include!("zeek/Val.h");
        type Val;

        fn AsBool(self: &Val) -> bool;
        fn AsCount(self: &Val) -> u64;
        fn AsInt(self: &Val) -> i64;
        fn AsDouble(self: &Val) -> f64;
        fn AsRecordVal(self: &Val) -> *const RecordVal;
        ...
```

Interfacing Rust and C++: types

Use low-level approach of exposing C++ types and functions by hand with cxx (https://cxx.rs).

```
#[cxx::bridge]
mod ffi {
    #[derive(Debug)]
    enum TypeTag {
        TYPE_VOID,
        TYPE_BOOL,
        TYPE_INT,
        ...
    }

    #[namespace = "::zeek"]
    unsafe extern "C++" {
        include!("zeek/Val.h");
        type Val;

        fn AsBool(self: &Val) -> bool;
        fn AsCount(self: &Val) -> u64;
        fn AsInt(self: &Val) -> i64;
        fn AsDouble(self: &Val) -> f64;
        fn AsRecordVal(self: &Val) -> *const RecordVal;
        ...
```

```
    extern "Rust" {
        fn serialize_val(
                value: &Val,
                format: Format,
                buf: Pin<&mut ByteBuffer>)
            -> Result<()>;

        fn deserialize_val(
                buf: &[u8],
                format: Format,
                ty: &TypePtr)
            -> Result<UniquePtr<ValPtr>>;

        ...
}
```

Interfacing Rust and C++: types

```
// Automatically generated by `cxx`.
auto serialize_val(
    zeek::Val const &value,
    Format format,
    support::ByteBuffer &buf);
  -> void;

auto deserialize_val(
    rust::Slice<std::uint8_t const> buf,
    Format format,
    zeek::TypePtr const &ty);
  -> std::unique_ptr<::zeek::ValPtr>
```

```
    extern "Rust" {
        fn serialize_val(
                value: &Val,
                format: Format,
                buf: Pin<&mut ByteBuffer>)
            -> Result<()>;

        fn deserialize_val(
                buf: &[u8],
                format: Format,
                ty: &TypePtr)
            -> Result<UniquePtr<ValPtr>>;

        ...
}
```

Interfacing Rust and C++: build system

Zeek plugins really want to be built with CMake.

Corrosion (https://corrosion-rs.github.io/corrosion/) allows exposing Rust crates as CMake targets.

```
FetchContent_Declare(
    Corrosion
    GIT_REPOSITORY https://github.com/corrosion-rs/corrosion.git
    GIT_SHALLOW ON
    GIT_TAG v0.5.2
)
FetchContent_MakeAvailable(Corrosion)

corrosion_import_crate(
    MANIFEST_PATH Cargo.toml
    CRATES zeek-more-serializers
    FEATURES corrosion
)
corrosion_add_cxxbridge(
    zeek_more_serializers_cxx
    CRATE zeek_more_serializers
    FILES lib.rs
)
```

Interfacing Rust and C++: build system

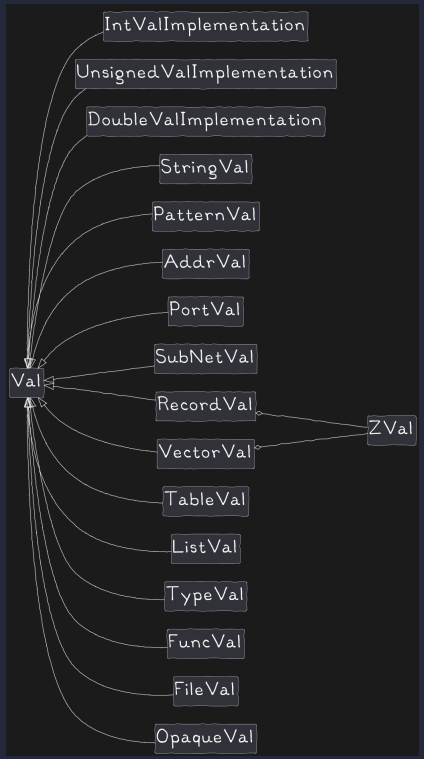
Zeek plugins really want to be built with CMake.

Corrosion (https://corrosion-rs.github.io/corrosion/) allows exposing Rust crates as CMake targets.

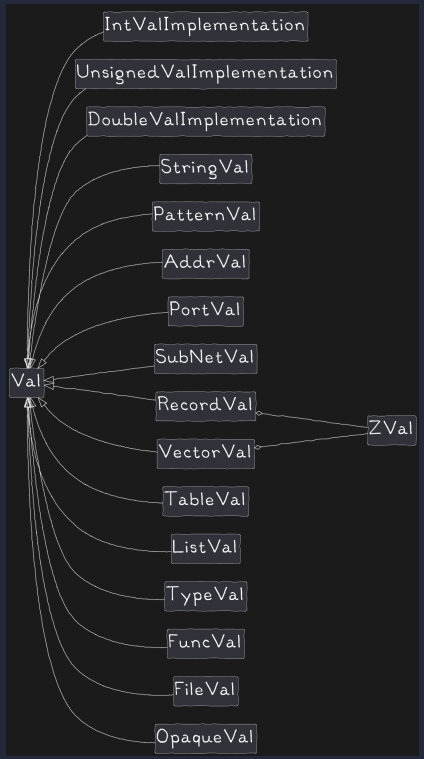
```
FetchContent_Declare(
    Corrosion
    GIT_REPOSITORY https://github.com/corrosion-rs/corrosion.git
    GIT_SHALLOW ON
    GIT_TAG v0.5.2
)
FetchContent_MakeAvailable(Corrosion)

corrosion_import_crate(
    MANIFEST_PATH Cargo.toml
    CRATES zeek-more-serializers
    FEATURES corrosion
)
corrosion_add_cxxbridge(
    zeek_more_serializers_cxx
    CRATE zeek_more_serializers
    FILES lib.rs
)
```

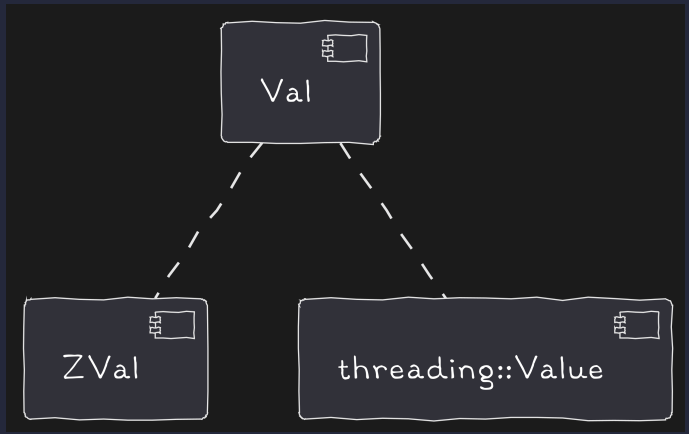
```
zeek_add_plugin(Zeek more_serializers
  SOURCES plugin/src/Plugin.cc
  BIFS plugin/src/more_serializers.bif
  DIST_FILES README.md LICENSE
  DEPENDENCIES zeek_more_serializers_cxx
  INCLUDE_DIRS plugin/src)
```

Mapping `zeek::Val`



Mapping `zeek::Val`





- `Val` uses virtual inheritance
- `ZVal` and `threading::Value` are union types
- `threading::Value` supports less tags than `Val` or `ZVal`

Mapping `zeek::Val`: Rust implementation

```rust
#[derive(Debug, Clone, PartialOrd, Ord, PartialEq, Eq, Hash,
         Serialize, Deserialize)]
pub enum Val<'a> {
    None,
    Bool(bool),
    Count(u64),
    Int(i64),
    Double(OrderedFloat<f64>),
    Enum(TypeId<'a>, u64),
    String(Cow<'a, [u8]>),
    Pattern { exact: Cow<'a, str>, anywhere: Cow<'a, str> },
    Port { num: u32, proto: TransportProto },
    Addr(Addr),
    Subnet(Subnet),
    Interval(Duration),
    Time(OffsetDateTime),
    Vec(Vec<Val<'a>>),
    List(Box<[Val<'a>]>),
    Set(BTreeSet<Box<[Val<'a>]>>),
    Table(BTreeMap<Box<[Val<'a>]>, Val<'a>>),
    Record(TypeId<'a>, BTreeMap<Cow<'a, str>, Val<'a>>),
}
```

Mapping `zeek::Val`: Rust implementation

```rust
#[derive(Debug, Clone, PartialOrd, Ord, PartialEq, Eq, Hash,
         Serialize, Deserialize)]
pub enum Val<'a> {
    None,
    Bool(bool),
    Count(u64),
    Int(i64),
    Double(OrderedFloat<f64>),
    Enum(TypeId<'a>, u64),
    String(Cow<'a, [u8]>),
    Pattern { exact: Cow<'a, str>, anywhere: Cow<'a, str> },
    Port { num: u32, proto: TransportProto },
    Addr(Addr),
    Subnet(Subnet),
    Interval(Duration),
    Time(OffsetDateTime),
    Vec(Vec<Val<'a>>),
    List(Box<[Val<'a>]>),
    Set(BTreeSet<Box<[Val<'a>]>>),
    Table(BTreeMap<Box<[Val<'a>]>, Val<'a>>),
    Record(TypeId<'a>, BTreeMap<Cow<'a, str>, Val<'a>>),
}
```

```rust
/// Trait so code can work with both
/// `ffi::Val` and `ffi::ZVal`.
pub trait ValInterface {
    fn as_bool(&self) -> bool;
    fn as_int(&self) -> i64;
    fn as_count(&self) -> u64;
    fn as_double(&self) -> f64;
    fn as_interval(&self) -> f64;
    fn as_time(&self) -> f64;
    ...
    fn as_record_val(&self)
         -> Option<&ffi::RecordVal>;
    ...
}
```

- implement `ValInterface` for both `ffi::Val` and `ffi::ZVal`
- blanket implement of `TryInto<Val>` for types implementing `ValInterface`
- implemented conversion from `Val` to `ValPtr` for the other direction

Mapping `zeek::Val`: Rust implementation

```rust
#[derive(Debug, Clone, PartialOrd, Ord, PartialEq, Eq, Hash,
         Serialize, Deserialize)]
pub enum Val<'a> {
    None,
    Bool(bool),
    Count(u64),
    Int(i64),
    Double(OrderedFloat<f64>),
    Enum(TypeId<'a>, u64),
    String(Cow<'a, [u8]>),
    Pattern { exact: Cow<'a, str>, anywhere: Cow<'a, str> },
    Port { num: u32, proto: TransportProto },
    Addr(Addr),
    Subnet(Subnet),
    Interval(Duration),
    Time(OffsetDateTime),
    Vec(Vec<Val<'a>>),
    List(Box<[Val<'a>]>),
    Set(BTreeSet<Box<[Val<'a>]>>),
    Table(BTreeMap<Box<[Val<'a>]>, Val<'a>>),
    Record(TypeId<'a>, BTreeMap<Cow<'a, str>, Val<'a>>),
}
```

```rust
impl Val<'_> {
    /// Convert this `Val` into an owned version.
    pub fn into_owned(self) -> Val<'static> {
        match self {
            Val::String(x) => Val::String(
                Cow::from(x.into_owned())
            ),
            ...
        }
    }
}
```

The current implementation converts containers `vector`, `set`, `list`, `table`, `record` eagerly which could be made lazy by storing e.g.,

```rust
enum Record<'a> {
    Eager(&'a ffi::RecordVal),
    Lazy(
        TypeId<'a>,
        BTreeMap<Cow<'a, str>, Val<'a>>
    ),
}
```

Example: Conversion from Val to zeek::ValPtr

```
impl Val<'_> {
    pub fn to_valptr(self, ty_orig: Option<&ffi::TypePtr>) -> Result<UniquePtr<ffi::ValPtr>> {
        // Absent type indicates `TYPE_ANY`.
        let ty = ty_orig.and_then(|t| {
            t.val().filter(|ty| !matches!(ty.Tag(), zeek::TypeTag::TYPE_ANY))
        });

        Ok(match self {
            Val::None => ffi::make_null(),
            Val::Bool(x) => {
                check_dest_type!(TypeTag::TYPE_BOOL);
                ffi::make_bool(x)
            }
            Val::Count(x) => {
                check_dest_type!(TypeTag::TYPE_COUNT);
                ffi::make_count(x)
            }
            Val::Int(x) => {
                check_dest_type!(TypeTag::TYPE_INT);
                ffi::make_int(x)
            }
            ...
```

Making Zeek API types easier to consume: set/table iteration

Zeek's `zeek::TableVal` has a historically grown API and is not easy to iterate.

Rust code uses iterator pervasively to work with containers, e.g.,

```
let values = vec![1, 2, 3, 4];
let odd_doubled: Vec<_> = values
    .into_iter()
    .filter(|x| x % 2 != 0)
    .map(|x| x * 2)
    .collect();
assert_eq!(odd_doubled, vec![2, 6]);
```

Making Zeek API types easier to consume: set/table iteration

Zeek's zeek::TableVal has a historically grown API and is not easy to iterate.

Rust code uses iterator pervasively to work with containers, e.g.,

```
let values = vec![1, 2, 3, 4];
let odd_doubled: Vec<_> = values
    .into_iter()
    .filter(|x| x % 2 != 0)
    .map(|x| x * 2)
    .collect();
assert_eq!(odd_doubled, vec![2, 6]);
```

- add FFI type TableEntry which holds reference to a set/table entry
- add FFI type TableIterator which adapts iteration
- to make this zero-copy tie lifetime of TableEntry to underlying container

Making Zeek API types easier to consume: set/table iteration

Zeek's `zeek::TableVal` has a historically grown API and is not easy to iterate.

Rust code uses iterator pervasively to work with containers, e.g.,

```
let values = vec![1, 2, 3, 4];
let odd_doubled: Vec<_> = values
    .into_iter()
    .filter(|x| x % 2 != 0)
    .map(|x| x * 2)
    .collect();
assert_eq!(odd_doubled, vec![2, 6]);
```

- add FFI type `TableEntry` which holds reference to a `set`/`table` entry
- add FFI type `TableIterator` which adapts iteration
- to make this zero-copy tie lifetime of `TableEntry` to underlying container

```
let val/*: &TableVal*/ = val.as_table_val().ok_or
(Error::ValueUnset)?;

let it/*: UniquePtr<TableIterator<'_>>*/ = val.iter();
let it/*: &TableIterator<'_>*/ = it.as_ref().ok_or
(Error::ValueUnset)?;

let xs/*: Result<Vec<_>>*/ = std::iter::from_fn(move || {
    let val/*: UniquePtr<TableEntry<'_>>*/ = it.next();
    let cur/*: &TableEntry<'_>*/ = val.as_ref()?;

    let key: Result<Box<[Val]>> = cur.key().try_into();
    let key = key.map(|x| x.into_iter().map(Val::into_owned).collect
());

    let val = cur.value_ref().map_or(Ok(Val::None),
TryInto::try_into);

    Some((key, val))
}).map(|(k, v)| Ok((k?, v?)))
.collect();
```

Serialization formats

Human readable

```
count 9:         {"Count":9}
string "abc":    {"String":[97,98,99]}
port 1/icmp:     {"Port":{"num":1,"proto":"Icmp"}}
port 2/unknown:  {"Port":{"num":2,"proto":"Unknown"}}
```

- implemented with `serde_json`
- useful for debugging
- not particularly performant or space efficient

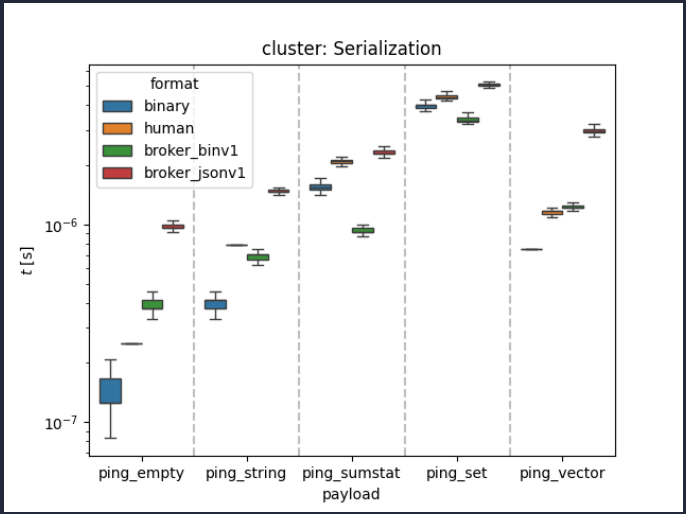
Binary format

```
count 9:         \x02\x09
string "abc":    \x06\x03abc
port 1/icmp:     \x07\x01\x03
port 2/unknown:  \x07\x02\x00
```

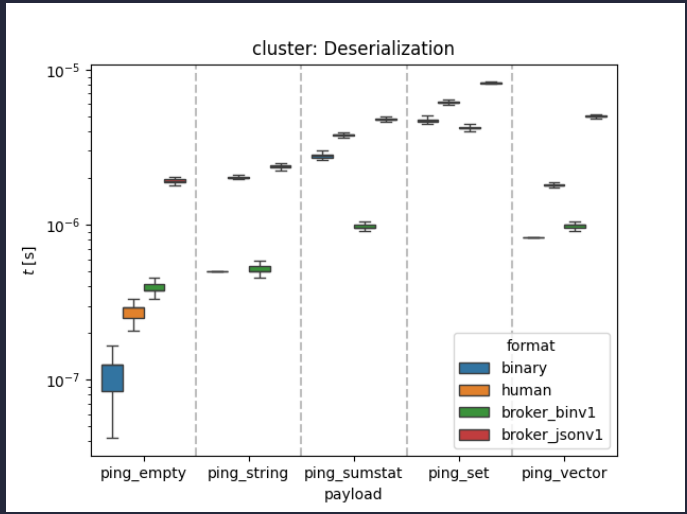
- implemented with `postcard`
- space efficient format with variable-width data fields

```rust
pub enum Val<'a> {
    None,
    Bool(bool),
    Count(u64),
    Int(i64),
    Double(OrderedFloat<f64>),
    Enum(TypeId<'a>, u64),
    String(Cow<'a, [u8]>),
    Port { num: u32, proto: TransportProto },
    Addr(Addr),
    Subnet(Subnet),
    Interval(Duration),
    Time(OffsetDateTime),
    Vec(Vec<Val<'a>>),
    List(Box<[Val<'a>]>),
    Set(BTreeSet<Box<[Val<'a>]>>),
```

Benchmark: cluster event serializer

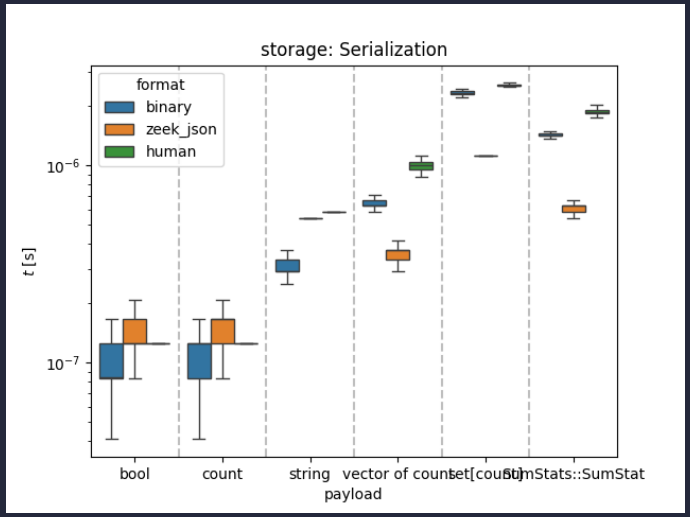


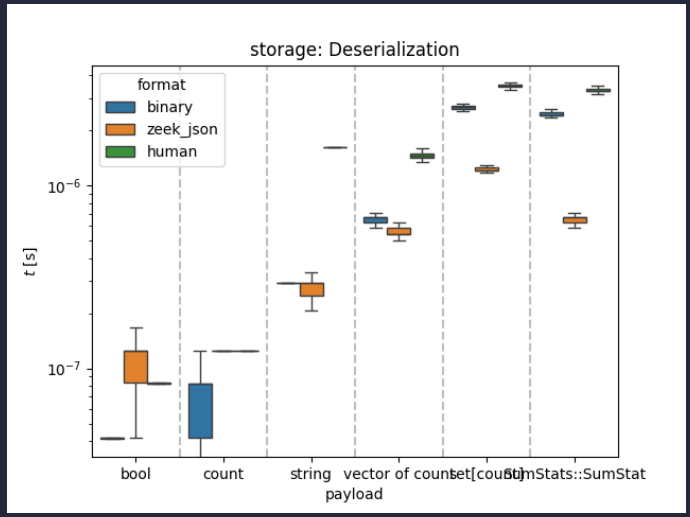
1. convert `zeek::Val` into representation
2. serialize representation to bytes



1. deserialize bytes into representation
2. convert representation into `zeek::Val`

Benchmark: storage serializer





- default serializer `zeek_json` can directly serialize/deserialize `zeek::Val` to JSON string
- `binary` and `human` need to go through intermediate representation

Testing: btest

The canonical way to test Zeek plugins is with BTest:

- run `zeek` with the plugin
- snapshot side effect, typically some log output
- `assert` results

This is a poor fit for unit testing Rust code since most functionality never exposed.

We can access Zeek symbols in Rust crate, but not link them. This breaks building of simple test targets.

Testing: btest

The canonical way to test Zeek plugins is with BTest:

- run `zeek` with the plugin
- snapshot side effect, typically some log output
- `assert` results

This is a poor fit for unit testing Rust code since most functionality never exposed.

We can access Zeek symbols in Rust crate, but not link them. This breaks building of simple test targets.

▒▒▒ Workaround: `cargo t --workspace`

1. Wrap "unit test" code in a separate plugin `test-plugin` crate with `crate-type = ["dylib"]`.
2. `test-plugin` runs tests in `Plugin::InitPreExecution` via Rust wrapper for `Plugin`.
3. Introduce a `runner` crate which builds a Zeek plugin from `test-plugin` with `xtask` to create plugin artifacts.
4. `runner` has a single `#[test]` function which starts Zeek and loads `test-plugin`.

Testing: property testing

We can access coverage information with e.g., `cargo llvm-cov`.

Combine this with a property testing framework (here: `proptest`).

> Proptest is a property testing framework (i.e., the QuickCheck family) inspired by the Hypothesis framework for Python. It allows to test that certain properties of your code hold for arbitrary inputs, and if a failure is found, automatically finds the minimal test case to reproduce the problem. Unlike QuickCheck, generation and shrinking is defined on a per-value basis instead of per-type, which makes it more flexible and simplifies composition.

Testing: property testing

We can access coverage information with e.g., `cargo llvm-cov`.

Combine this with a property testing framework (here: `proptest`).

> Proptest is a property testing framework (i.e., the QuickCheck family) inspired by the Hypothesis framework for Python. It allows to test that certain properties of your code hold for arbitrary inputs, and if a failure is found, automatically finds the minimal test case to reproduce the problem. Unlike QuickCheck, generation and shrinking is defined on a per-value basis instead of per-type, which makes it more flexible and simplifies composition.

```
let strategy = any::<Type>()
    .prop_flat_map(|ty| arbitrary_val(ty.clone())
         .prop_map(move |val| (ty.clone(), val)));

runner.run(&strategy, |(ty, x0)| {
    let ty: cxx::UniquePtr<_> = ty
        .try_into()
        .expect("type should be compatible with Zeek");

    // Check value roundtrip.
    let x = x0.clone().to_valptr(Some(&*ty))?;
    let x = x.val().ok_or(Error::ValueUnset)?;
    let x1: Val = x.try_into()?;

    assert_eq!(x0, x1);
    Ok(())
})?;
```

Testing: property testing

We can access coverage information with e.g., `cargo llvm-cov`.

Combine this with a property testing framework (here: `proptest`).

> Proptest is a property testing framework (i.e., the QuickCheck family) inspired by the Hypothesis framework for Python. It allows to test that certain properties of your code hold for arbitrary inputs, and if a failure is found, automatically finds the minimal test case to reproduce the problem. Unlike QuickCheck, generation and shrinking is defined on a per-value basis instead of per-type, which makes it more flexible and simplifies composition.

```
let strategy = any::<Type>()
    .prop_flat_map(|ty| arbitrary_val(ty.clone())
         .prop_map(move |val| (ty.clone(), val)));

runner.run(&strategy, |(ty, x0)| {
    let ty: cxx::UniquePtr<_> = ty
        .try_into()
        .expect("type should be compatible with Zeek");

    // Check value roundtrip.
    let x = x0.clone().to_valptr(Some(&*ty))?;
    let x = x.val().ok_or(Error::ValueUnset)?;
    let x1: Val = x.try_into()?;

    assert_eq!(x0, x1);
    Ok(())
})?;
```

- `any` is tricky on the Zeek side, exclude some cases
- not all containers support holes or `any` values
- precision loss for `timstamp` and `interval` near edges

Testing: property testing

```
$ PROPTEST_VERBOSE=2 cargo t --workspace
...
proptest: Next test input: (Double, Double(3.078281734093319e113))
proptest: Test case passed
proptest: Next test input: (Vec(Enum(TypeId("Notice::Type"))), Vec([None, None, Enum(TypeId("Notice::Type"), 2),
Enum(TypeId(
proptest: Test case passed
proptest: Next test input: (Table(TableType([Time, Addr], Some(Interval))), Table({[Time(1975-05-26 16:58:58.0 +00:00:00),
Ad
proptest: Test case passed
proptest: Next test input: (Table(TableType([Bool], Some(Enum(TypeId("Notice::Type"))))), Table({[Bool(false)]:
Enum(TypeId("
proptest: Test case passed
proptest: Next test input: (Table(TableType([String], Some(Pattern))), Table({[String([5, 180, 212])]: Pattern { exact:
"OxWH
proptest: Test case passed
proptest: Next test input: (Table(TableType([String, Interval, Enum(TypeId("Notice::Type"))], Some(Double))),
Table({[String(
proptest: Test case passed
proptest: Next test input: (Set(SetType([Subnet, Bool])), Set({[Subnet(Subnet(V4(Ipv4Network { addr: 79.185.128.0, prefix:
18
proptest: Test case passed
proptest: Next test input: (Set(SetType([Int])), Set({[Int(7389619598776464190)], [Int(8323406496964592315)]}))
proptest: Test case passed
    successes: 10000
    local rejects: 948
        266 times at addrs in set have special semantics
        262 times at double in sets are hard
        278 times at patterns in sets have special semantics
        142 times at table of any is unsupported
    global rejects: 0
```

Demo

```
git clone https://github.com/bbannier/zeek-more-serializers
cd zeek-more-serializers

cmake -GNinja -Bbuild
ninja -C build

btest -vjc testing/btest.cfg
```

snippet +exec is disabled, run with -x to enable

The Good

- CMake integration just worked
- non-intrusive serialization format with serde (modulo `zeek::OpaqueValue`)
- much easier to create `zeek::Val` values via `Val` enum, e.g., for `set` or `table` values
- explicit lifetimes simplified zero-copy abstractions
- reusable crate for working with `zeek::Val`

The Bad

Bindings

- needed to implement non-trivial amount of helper functions in C++ (~290 loc)

The Bad

Bindings

- needed to implement non-trivial amount of helper functions in C++ (~290 loc)

Plugin

- no libzeek: impossible to "link" against Zeek, symbols get resolved when plugin is loaded.

The Bad

Bindings

- needed to implement non-trivial amount of helper functions in C++ (~290 loc)

Plugin

- no `libzeek`: impossible to "link" against Zeek, symbols get resolved when plugin is loaded.

`zeek::Val`

- incomplete support for converting between Zeek's value types `Val`, `ZVal`, `threading::Value`
- functions with nullable return value might abort instead of returning a `nullptr`
- Zeek API often does not distinguish clearly between bytes `&[u8]` and UTF-8 strings `&str`
- `zeek::Val` variants which do not clearly map script-level values, e.g., `TYPE_FUNC`, `TYPE_FILE`, `TYPE_TYPE` or `TYPE_ERROR`
- API inconsistencies, e.g., `fn Size(self: &VectorVal) -> u32` vs. `fn Length(self: &ListVal) -> i32`
- ports with values outside of `u16` range for ICMP support

The Bad

Bindings

- needed to implement non-trivial amount of helper functions in C++ (~290 loc)

Plugin

- no `libzeek`: impossible to "link" against Zeek, symbols get resolved when plugin is loaded.

`zeek::Val`

- incomplete support for converting between Zeek's value types `Val`, `ZVal`, `threading::Value`
- functions with nullable return value might abort instead of returning a `nullptr`
- Zeek API often does not distinguish clearly between bytes `&[u8]` and UTF-8 strings `&str`
- `zeek::Val` variants which do not clearly map script-level values, e.g., `TYPE_FUNC`, `TYPE_FILE`, `TYPE_TYPE` or `TYPE_ERROR`
- API inconsistencies, e.g., `fn Size(self: &VectorVal) -> u32` vs. `fn Length(self: &ListVal) -> i32`
- ports with values outside of `u16` range for ICMP support

C++isms

- types without stable ABI in public interfaces, e.g., `int ListVal::Length() const`
- preference for access via `IntrusivePtr` values instead of references so wrapper cannot be 100% zero-copy

Possible next steps

- clean up `Val` interface to make states not possible in Zeek C++ API unrepresentable
- cover unhandled types `TYPE_FUNC`, `TYPE_FILE`, `TYPE_TYPE`, `TYPE_ERROR`, `TYPE_OPAQUE`
- alternative, view-like value type to avoid eager conversion
- trait to expose Rust types as `OpaqueVal`
- helper to directly expose crate as Zeek plugin
- proc macro to directly expose Rust functions as Zeek bifs

Plugin

https://github.com/bbannier/zeek-more-serializers

Thanks!