A Domain-Specific Language for Filtering in Application-Level Gateways

Balldin, Hampus; Reichenbach, Christoph

Published in:
Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences

DOI:
10.1145/3425898.3426955

2020

Document Version:
Publisher's PDF, also known as Version of record

A Domain-Specific Language for Filtering in Application-Level Gateways

Hampus Balldin
Advenica AB
Malmö, Sweden
hampus.balldin@advenica.com

Christoph Reichenbach
Department of Computer Science
Lund University
Lund, Sweden
christoph.reichenbach@cs.lth.se

Abstract

Application-level packet filtering is a technique for network access control in which an "application-level gateway" intercepts network packets at the application level (e.g., HTTP, FTP), scans them for security concerns and optionally logs, rewrites or discards them. Existing application-level filters express their filtering rules in general-purpose languages, which limits the correctness guarantees available for them.

We present the first declarative language for application-level network filtering, developed at Advenica AB. Our DSL uses *security assertions* to express properties that packets must have to be allowed through the network (e.g., "IMAP packet contains no executable attachment" or "SQL reply contains only explicitly permitted columns"), along with remedies that either reject or rewrite undesirable packets.

We have designed the language around the needs of network filter developers, with a focus on correctness: our language can statically verify several properties of filter programs, such as well-formedness of the outcome, confluence, and termination, with the help of an off-the-shelf SMT solver.

Our initial results show that the language can express many typical filtering tasks, closely maps to the application domain, and provides strong correctness guarantees.

CCS Concepts: • **Networks** → *Application layer protocols*; • **Software and its engineering** → **Domain specific languages**; • **Security and privacy** → **Network security**.

Keywords: filtering language, packet filtering, network security, domain-specific languages

ACM Reference Format:
Hampus Balldin and Christoph Reichenbach. 2020. A Domain-Specific Language for Filtering in Application-Level Gateways. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3425898.3426955

GPCE '20, November 16–17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8174-1/20/11.
https://doi.org/10.1145/3425898.3426955

1 Introduction

Any organisation that collects sensitive information (e-mails, photographs, personal data, ...) must consider who should be able to access and modify that information. In computer networks, this means that the organisation must impose access control. The main opportunities for access control are at the level of server applications, and at the level of network transmissions. While access control on the server software level *can* give fine-grained control, the quality of this control is only as good as the quality of the server software (regarding bugs and missing features) and the organisation's effectiveness at enforcing organisation-wide server settings.

Many organisations therefore choose to use network filters instead of or in addition to relying on server-side access control. They can place these filters at any level of the OSI network layer stack, but the further up they place the filter, the more information they have available to make informed decisions. The highest (or "application") layer can provide detailed insight, e.g. whether an FTP [20] request is trying to download or overwrite a sensitive file.

Today, application-level gateways implement filters in general-purpose languages, which makes it difficult to obtain strong safety guarantees. Moreover, such filters spend nontrivial effort on searching through filter packets, which requires substantial boilerplate code for managing iterations.

We have designed a novel declarative domain-specific language for asserting *security conditions* over application-level network packets and for optionally repairing packets that violate these conditions. We have equipped our language with a number of verification techniques that together guarantee that filters will only emit packets that pass all security conditions and never get "stuck". While our verification techniques are not complete (i.e., may discard some valid programs), we argue that they are sound (i.e., they discard all programs that violate well-formedness).

Figure 1 illustrates the architecture of Advenica AB's ZoneGuard[1] system as an example of a typical application-layer network filter: the system takes in network packets intended

[1] https://advenica.com/en/cds/zoneguard, accessed 2020-07-24

Figure 1. Architecture of a typical application-level network filter. The system pipes all incoming packets through a filter, validating that the packet is *well-formed* before and after.

for a specific network service (HTTP, FTP, …) extracts key features from the packet with a parser for the protocol, validates that the packet is *well-formed* according to a subset of the protocol (expressed as a *schema*), and then applies a filter that may log and rewrite or discard the packet. If the filter did not discard the packet, the system re-validates the output packet against a schema (usually same as the input schema), and translates the packet back into a binary network packet that it passes on to the intended target server.

To support multiple protocols, decoding, encoding, and validation operate on a universal binary representation. The two principal challenges then are to implement encoding/decoding logic (once per protocol), and to implement filter logic (once per use case), which is the focus of this paper.

Figure 2 describes a filter for a small security-relevant subset of the Remote Desktop Protocol (RDP), a protocol for remote control of Microsoft Windows desktops. This filter operates on a subset of the RDP protocol that controls the server's mouse movements and mouse button clicks.

Lines 1–4 describe the *schema* in a form similar to an EBNF grammar with start symbol `RDP`. According to this schema, an `RDP` packet is a possibly empty sequence of `Mouse` nodes ('`*`' is the Kleene star), and each `Mouse` node consist of two integers, named `x` and `y`, and an optional integer, called `button`, that represents an optional mouse button press.

Line 6 asserts that any `button` must be an element of the list `allowedButtons` (line 8). Our filter language infers the context of `button` and checks the assertion condition on all `Mouse` nodes in the input packet. Whenever a `Mouse` contains the press of a button that is sanctioned by `allowedButtons` (which only allows button #0), the assertion fails.

Such a failure triggers the recovery strategy in line 7, which removes the (optional) mouse button press. This turns the event into a pure mouse movement event. If an incoming packet contains multiple `Mouse` events, we will transform all of them accordingly.

Our system supports a number of different recovery actions, including direct updates. For instance, we could also remap disallowed button presses to operate on button #0 with an *update action*:

```
Assert button elem allowedButtons where
    @onfail = button <- 0
```

[2] https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-rdpbcgr/, accessed 2020-07-17.

```
1 [Schema @start = RDP]
2 data RDP = [ Mouse * ]
3 data Mouse = [ x : Integer, y : Integer,
4                button : Integer ? ]
5
6 Assert button elem allowedButtons where
7     @onfail = remove(button)
8     @def allowedButtons = [0]
```

Figure 2. Filter for a small subset of the Remote Desktop Protocol[2]. Lines 1–4 describe the input / output schema (a subset of `TS_POINTER_EVENT` messages). Line 6 describes an assertion over input packets with the help of a local definition in line 8, while line 7 describes the recovery action.

However, careless updates can introduce security vulnerabilities. For example, an end user of our filter might decide to only allow mouse button #2 and set `allowedButtons = [2]`. In this case, our update could produce a packet that violates the assertion condition `button elem allowedButtons` — in other words, the update fails to sanitise disallowed packets!

Since our system is intended as a security tool, we statically check for such **ineffective repair**s and report any assertions that we cannot guarantee to be effective.

Ineffective repairs are only one of several pitfalls in filter specification that we have identified. We also statically check for **impossible assertions**, which never succeed, **vacuous assertions**, which never fail, **termination bugs**, **runtime faults**, **schema invalidation**, where filters produce ill-formed output, and **subverted repair**, where one assertion's repair action thwarts the repair action of another.

Our contributions are as follows:

- The first (to the best of our knowledge) DSL for protocol-independent application-level network filtering
- A technique for statically verifying the absence of the seven pitfalls listed above
- A validation of our language via re-implementations of four of the 15 currently existing protocol filters from Advenica AB's ZoneGuard system in our prototype
- A systematic case study with a network filter engineer from Advenica AB that explored re-implementing an existing filter in our language

2 Extended Example

We introduce our language with a synthetic example that showcases several core features. Figure 3 shows the schema of a hypothetical protocol for reporting time series of sensor measurements. Line 2 defines a *node type* `SensorData` for the root node of incoming and outgoing packets. A `SensorData` node contains a `String` with the *name tag* `sensor`, and another node with time series data: either as a `TSeries` (another node type), or as a zero or more `BoundedTSeries` nodes.

```
1 [Schema @start = SensorData]
2 data SensorData = [ sensor : String,
3                     ( TSeries | BoundedTSeries * ) ]
4 data TSeries = [ val : Integer + ]
5 data BoundedTSeries = [ max : Integer, TSeries ]
```

Figure 3. Schema for a protocol for transmitting sensor data. Each packet contains either a single time series or a sequence of bounded time series, where each bounded time series stores a regular time series and the series' maximal element.

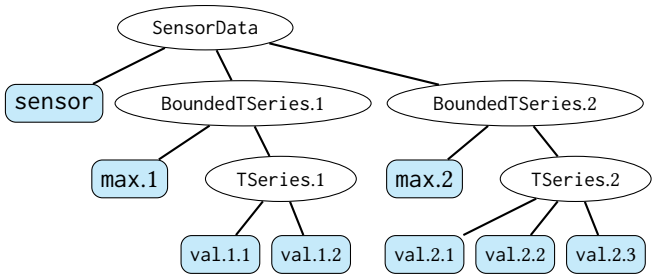

Figure 4. Example packet for the schema in Figure 3

Line 4 then defines `TSeries` as a nonempty list of Integer nodes, each named `val`. Here, the `+` stands for the Kleene plus. Finally, line 5 defines `BoundedTSeries` as a pair of an integer named `max` and an unnamed `TSeries`. Figure 4 gives an example of a well-formed example packet in this protocol.

We now write a filter to ensure properties over the packets: (**R-1**) they come from a named node, (**R-2**) bounded time series report a correct maximum, and (**R-3**) no time series contains negative numbers. Along the way, we explore how our system guarantees what we call *filter correctness*:

Whenever a filter processes a well-formed packet that violates an assertion, the filter either discards the packet or emits a different, well-formed packet that satisfies all assertions.

Our system also guarantees that well-formed packets that pass all assertions pass the filter unmodified and that the filter discards all ill-formed packets.

2.1 Rule R-1: Sensors Are Named

We begin with the following assertion:

```
Assert sensor /= ""
```

This assertion will discard any packets in which the name of the sensor is empty. Since we left out the `@onfail` repair action, our system will discard any packet that fails the assertion condition and record a default log message.

2.2 Rule R-2: Values Do Not Exceed the Maximum

This rule requires comparing all `val` nodes in `BoundedTSeries` to the same series' `max` value. We write it as:

```
Assert BoundedTSeries..val <= BoundedTSeries.max where
  @onfail = BoundedTSeries..val <- BoundedTSeries.max
```

Here, `BoundedTSeries..val` refers to a `val` node that is a *descendant* of a `BoundedTSeries` in the tree structure of the input packet, while `BoundedTSeries.max` refers to a `max` node that is an immediate *child* of a `BoundedTSeries` node.

This condition illustrates a semantic subtlety. As we see in Figure 4, a `SensorData` packet can contain more than one `BoundedTSeries` node. When our language iterates over all five possible bindings for `val`, we want to compare `val` nodes under `TSeries.1` only to `max.1` and not to `max.2`. This is implicit in our semantics: as we bind `BoundedTSeries..val`, we also implicitly bind `BoundedTSeries` accordingly, and re-use that binding to resolve `BoundedTSeries.max`. Section 3.1 elaborates on this mechanism.

2.3 Rule R-3: No Negative Numbers in Time Series

Finally, we ensure that the time series contains no negative numbers. Below is a first attempt at such a rule, though this rule suffers from an off-by-one error:

```
Assert val > 0 where @onfail = val <- 0    # broken
```

Written in an imperative language, this rule would make sense: Whenever `val` is greater than zero, set it to zero. However, our language is declarative, and its semantics guarantee that any packet emitted by a filter satisfies all assertions. We will thus reject the above rule, with an error to show that the rule does not resolve the failed assertion.

Instead, we can write the following rule:

```
Assert val >= 0 where @onfail = val <- 0
```

While this rule captures our intent, our system will still reject the rule, due to the way that it interacts with rule **R-2**: first, if the filter changes a `val` both by rule **R-3** and by rule **R-2**, we introduce an ambiguity into the filter's semantics. Our system will report this issue as *ambiguous assertions*: "the order of evaluating the @onfail action of [the two assertions] may lead to different filter results"

Second, setting `val` to zero may *invalidate the assertion of rule **R-2***. The reason for that is that in rule **R-2** we require `val` to be less than `max`, but we have never ruled out that `max` can be negative. Our tool reports this error as follows:[3]

```
[UnstableAssertions]
After evaluating the @onfail action of {assertion2},
  a previously satisfied precondition of {assertion1}
  may evaluate to False, in {model}
where
 {assertion2} = Assert@Line=28
 {assertion1} = Assert@Line=32
 {model} = SensorData..BoundedTSeries..max%2 -> -1
           SensorData..BoundedTSeries..val%2 -> 0
           SensorData..val%1 -> 0 :: Integer
```

[3]Edited for space, removing two type annotations (`:: Integer`) and three trailing variable assignments that were less useful than the first three.

The concrete counterexample from this error message alerts us that we must add a rule **R-4** to the program:

```
Assert max >= 0
```

After we ensure that no `max` value can be negative, our system can verify that our specification guarantees that any filtered packet will satisfy all assertions and be well-formed.

3 Language Overview

Figure 5 summarises the core syntax of our language. Each filter program consists of a schema and a set of assertions.

The schema consists of type definitions that together impose a structure over the tree-structured input and output packets. Our types include primitive types (`Integer`, `String`, `Bool`), product types (`T1, T2`), sum types (`T1 | T2`) for variants, and *multiplicities*, which state how many elements of the type a given node may contain: One element (`T`), any number of elements (`T*`), one or more (`T+`), zero or one (`T?`), or any number i where $a \le i \le b$ (`T{`a`,`b`}`). By default, multiplicities apply equally to input and output packets, but users can designate output-only types by marking the multiplicity with square brackets (`T[?]`). Since these output-only types are invalid in input packets, we only allow them within sum types. Types can also name their primitive component types; e.g. `[ x : Integer, y : Integer ]` gives two different *name tags* to the integers in this product type. Finally, types can reference each other, but as in ZoneGuard, we currently disallow types to recursively depend on themselves (i.e., the types are *stratifiable*) since we have not found this feature to be needed and it may indicate a bug in the specification.

Assertions consist of a *condition* expression and a number of optional *decorators*, which are mainly `@onfail` recovery actions and, occasionally, precedence levels (Section 3.3).

Recovery actions can `discard` the packet (the default action), `remove` a node, `log` a message, *update* a primitive value in a node, notation `v <- expr`, `replace` a node with an output-only variant (Section 3.2), or any combination of these.

We use expressions in two places: to compute truth values for assertion conditions, and to compute node updates and node replacements. Expressions can access packet nodes via *path variables* that may bind to multiple values (Section 3.1). Whenever we can evaluate an expression with multiple bindings, we explore all possible bindings. For example, consider the following assertion:

```
data T = [ a : Integer *, b : Integer * ]
Assert a > b
```

If we process a packet with two `a` elements and three `b` elements, we will check all six combinations of these elements.

Expressions can alternatively materialise all bindings of a given variable as a list (in a deterministic order), notation `valuesOf(`*var*`)`. We provide aggregator functions, existential and universal checks, and list comprehensions to operate on lists. The example below illustrates using `valuesOf` with one of the built-in aggregator functions, `sum`, to assert that all `a` elements and all `b` elements sum up to the same value:

```
Assert sum(valuesOf(a)) = sum(valuesOf(b))
```

Our language also supports local variables (e.g., for list comprehensions or `exists`/`forall`), but the majority of our variables are path variables.

3.1 Referencing Tree Nodes with Path Variables

Typical application-level filter assertions check or compare nodes as individuals, rather than considering their siblings or node list offset. We thus designed our language to make it easy to avoid explicit iteration, via *path variables* (though users can use `valuesOf` for explicit iteration when needed).

The simplest path variables ($\langle Var \rangle$ in Figure 5) consist of only a name n and reference all nodes that have the type named n *or* occur under the name tag n. If we want only variables n that are immediate children of path variable ℓ, we can write $\ell.n$, while $\ell..n$ describes variables n that are *descendants* of ℓ (cf. *child* and *descendant axes* in XPath [8]).

For variables with a common prefix, as in

```
Assert BoundedTSeries..val <= BoundedTSeries.max
```

our semantics ensure that both variables refer to the *same* `BoundedTSeries`: when we bind the first `BoundedTSeries..val`, we simultaneously bind the corresponding `BoundedTSeries`.

Informally, our semantics ensure that for any two path variables in the same expression, their *longest common prefix* must bind to the same node. We capture this formally below.

Assume that $s_{\mathcal{N}}$ is the packet's root (`@start`) node, $p \longrightarrow c$ the child relation (p is the parent of c), and $T(n,t)$ a predicate that holds whenever n is a node and t a type such that $n : t$, *or* n has the name tag t. We define the relation

$$C, E \vdash v \triangleleft E'$$

to describe all environments E' that contain bindings for variable v, when encountering v in environment E with the *context* C. In the core language, $C = s_{\mathcal{N}}$ (but see Section 3.4).

Figure 6 captures our environment semantics formally. For each expression e we extract all path variables $v_1, \ldots, v_k$. At runtime, we set $E_0 = \emptyset$ and iterate $C, E_{i-1} \vdash v \triangleleft E_i$, to find all minimal environments E_k and test e once for each E_k. Our system reports a static error if the schema guarantees that there can be no such E_k.

Note that our semantics are order-independent: no matter in which order we encounter the variables of an expression, we will produce the exact same set of environments.

3.2 Replacement Semantics

Most of our recovery actions have straightforward semantics, but `replace` introduces two subtleties that reflect its main use case of *error reporting*.

Node replacement, unlike updates, can change the type of a node, so we restrict replacement to variant nodes, as in:

```
⟨Program⟩   →  ⟨Schema⟩ ⟨Assertion⟩*
⟨Schema⟩    →  [ Schema @start = id ] ⟨TyDef⟩*
⟨TyDef⟩     →  data id = [ ⟨TyBody⟩ ]
⟨TyBody⟩    →  id : ⟨Ty⟩
            |  ⟨Ty⟩
            |  ⟨TyBody⟩ (, ⟨TyBody⟩)*
            |  ⟨TyBody⟩ (| ⟨TyBody⟩)*
            |  ⟨TyBody⟩⟨Multi⟩
⟨Ty⟩        →  id | ⟨PrimTy⟩
⟨PrimTy⟩    →  Integer | String | Bool
⟨Multi⟩     →  ⟨Quant⟩ | [ ⟨Quant⟩ ]
⟨Quant⟩     →  * | + | ? | { int , int }

⟨Var⟩       →  id
            |  ⟨Var⟩ . id
            |  ⟨Var⟩ .. id

⟨Assertion⟩ →  Assert ⟨Expr⟩ [ where ⟨Decorator⟩* ]
⟨Decorator⟩ →  @prec = int
            |  @onfail = ⟨Action⟩
⟨Action⟩    →  discard
            |  remove ( ⟨Var⟩ )
            |  log ( ⟨Var⟩ )
            |  ⟨Var⟩ <- ⟨Expr⟩
            |  replace ( ⟨Var⟩ , id , ⟨Expr⟩)
⟨Expr⟩      →  ⟨Var⟩
            |  valuesOf( ⟨Var⟩ )
            |  id (⟨Expr⟩(, ⟨Expr⟩) + )
            |  id (⟨Expr⟩(, ⟨Expr⟩) + )
            |  exists (id(, id) + ) <- ⟨Expr⟩ : ⟨Expr⟩
            |  forall (id(, id) + ) <- ⟨Expr⟩ : ⟨Expr⟩
            |  [ ⟨Expr⟩ | ⟨ListComp⟩(, ⟨ListComp⟩) * ]
            |  [ ] | int | string | True | False
⟨ListComp⟩  →  id <- ⟨Expr⟩ | ⟨Expr⟩
```

Figure 5. Simplified syntax of our core filter language in EBNF.

$$\frac{v \in \operatorname{dom} E}{C, E \vdash v \triangleleft E} \ (env)$$

$$\frac{id \notin \operatorname{dom} E}{C, E \vdash id \triangleleft \{E[id \mapsto n] | C \longrightarrow^* n \wedge T(n, id)\}} \ (context)$$

$$\frac{v\,.\,id \notin \operatorname{dom} E \quad C, E \vdash v \triangleleft E'}{C, E \vdash v\,.\,id \triangleleft \{E'[v\,.\,id \mapsto n] | E'(v) \longrightarrow n \wedge T(n, id)\}} \ (cld)$$

$$\frac{v\,..\,id \notin \operatorname{dom} E \quad C, E \vdash v \triangleleft E'}{C, E \vdash v\,..\,id \triangleleft \{E'[v\,..\,id \mapsto n] | E'(v) \longrightarrow^+ n \wedge T(n, id)\}} \ (dsc)$$

Figure 6. Semantics of environment production for a single name, given an existing environment E and a context C.

```
data Packet = [ D* ]
data D = [ x : Integer | err : String [{1,1}] ]

Assert D.x > 0 where
  @onfail = replace(D.x, err, "Bad: " + toString(D.x))
```

Here we replace integer nodes that fail an assertion with error messages, in output-only string nodes of multiplicity 1. In practice, the same node may have multiple errors:

```
Assert D.x elem passlist where
  @onfail = replace(D.x, err, "Not in passlist")
```

For `D.x` that fail both assertions, our system collects *both* replacements, but only if we give `err` suitable multiplicity:

```
data D = [ x : Integer | err : String [{1,2}] ]
```

3.3 Stratification

Our declarative language design treats all assertions as independent of their order of appearance. However, sometimes we need explicit order dependence, as we have (so far) observed in one case, where assertion (a) removed list elements and assertion (b) removed empty lists. Here, we must apply assertion (a) exhaustively before testing assertion (b).

For these cases. we allow rules to supply the decorator `@prec` = n, where the number n is the desired precedence level (higher means later). The default stratum is 1, or -1 for rules that `discard` the packet. We use stratum separation extensively while checking filter correctness (Section 4.2).

3.4 Language Extensions

Our language provides numerous convenience features beyond its core, such as a typical set of infix and prefix operators for arithmetic, comparison, and string operations. While it is easy to add new operations to the language runtime, implementers must take special care to encode the semantic properties of our operations so that our static checks can reason about them (Section 4.2).

We summarise the most salient remaining features below.

Regular Expressions allow e.g. checking server names or file names. We not only include regular expressions in our system but also provide full support for reasoning about them in our correctness checker (Section 4.2).

User-Defined Functions and Constants such as the `@def allowedButtons` in line 8 of Figure 2 give names to common values and computation to increase readability and reuse, either globally or within one assertion. User-defined functions may call each other, but we reject (mutually) recursive function definitions to guarantee termination.

Table 1. Pitfalls and how we prevent them

Pitfall	Approach
Impossible Assertion	Satisfiability (Section 4.2.1)
Vacuous Assertion	Tautology (Section 4.2.1)
Schema Invalidation	Validity (Section 4.2.3)
Termination Bug	Totality (Section 4.2.4)
Runtime Fault	Totality (Section 4.2.4)
Ineffective Repair	Effectiveness (Section 4.2.5) and Limited Confluence (Section 4.2.2)
Subverted Repair	Global Stability (Section 4.2.6)

Configuration Files externalise parts of the filter settings via user-defined functions and constants. End-users can thus configure filters without editing their logic. Semantically, we treat configuration files as if they were inlined.

Advanced Type Support comes mainly in the form of *constrained types*, which are `Integer`, `String`, `Bool` types with value restrictions that we express in a boolean expression.

For example, we could have incorporated rule **R-4** from Section 2.3 into the definition of the field `max` by writing

```
Constrained(max : Integer, max >= 0)
```

When a constraint fails, we `discard` the packet, making the constraint checks part of stratum −1 (cf. Section 3.3).

Path Variable Context Operators such as `@context` allow us to add context for path variables (Section 3.1). This allows us to e.g. shorten rule **R-2** from Section 2.2:

```
Assert val <= max where
  @onfail = val <- max
  @context = BoundedTSeries
```

3.5 Additional Language Design Considerations

Our language design was driven partly by the existing ZoneGuard system (motivating our `@onfail` actions and our non-recursive schemas), but especially by considering features for reducing the risk of program bugs (Section 4).

Many other design decisions were driven by prototyping. For example, our initial design used rules of the form "if condition then action", until we noticed that we were mostly writing negative conditions ("if not desired-property"), which led us to our current syntax ("assert desired-property").

4 Verification and Correctness

Since network filters are security devices, their correctness is crucial. We have designed our language to facilitate a number of correctness checks that together avoid the known pitfalls that we list in the introduction. Table 1 summarises the relationship between pitfalls and correctness checks.

Together, our checks give a strong *correctness* guarantee:

- Any well-formed packet that enters a filter will either be discarded or the filter will emit another well-formed packet (Section 4.2.3)
- All filters terminate (Section 4.2.4)
- Any packet emitted by the filter satisfies all of the assertions in the filter specification (Section 4.2.6)

4.1 Type Correctness

Our system uses monomorphic type inference to ensure type correctness in expressions. The type system provides three built-in types (`Integer`, `String`, `Bool`), parametric types for lists and tuples, and user-defined schema types (`data`).

The type system provides limited overloading for built-in operators, namely addition (overloaded for strings and integers) and the `toString()` function (supported on all types). It also supports sum types `e.g., [ Integer | String ]`.

We check types throughout, including in update and `replace` actions, which must preserve the well-typedness of the packet. The type checker delegates multiplicities to a separate multiplicity checker (Section 4.2.3).

4.2 Satisfiability Checks

Our correctness checks can depend on the satisfiability of arbitrary arithmetic expressions over unknown inputs. Since this means that these properties are undecidable [14], we apply a semi-decision procedure in the form of the off-the-shelf SMT solver Z3 [9]: whenever Z3 cannot confirm a safety property, we treat this result as a static error.

Much of our encoding to Z3 is straightforward. We map most operators to corresponding Z3 functions (integer addition in both formalisms corresponds to "unbounded" integers rather than machine integers), string operations and regular expressions to the corresponding Z3 operations [27], and other built-in functions such as the string uppercase conversion function `toUpper` to uninterpreted function symbols for which we provide additional axioms (e.g., that `toUpper` returns no lower-case characters).

We use *assertion stratification* to promote assertion conditions to lemmas, and exploit schema information to provide additional constraints to Z3 beyond type information, primarily through *multiplicity analysis* and *alias analysis*.

Assertion stratification describes the order in which we check assertions. Due to *stability* (Section 4.2.6), each stratum guarantees that all its assertions hold before we evaluate the next stratum. Assertion conditions from earlier strata thus become axioms when checking later strata. For example, in Section 2.3 we proved rule **R-3** (stratum 1) correct with the help of rule **R-4**. Rule **R-4** discards packet that fail its condition, so it is in stratum −1 (Section 3.3) and we can use its condition as an axiom for checking properties of **R-3**.

Multiplicity analysis determines the multiplicity of an expression by examining how many distinct bindings the schema permits. For example, in Figure 3, `sensor` has multiplicity $\{1, 1\}$, i.e., must occur precisely once, while `max` has multiplicity $\{0, \infty\}$, i.e., might occur any number of times in a packet, since its parent node, `BoundedTSeries`. also has multiplicity $\{0, \infty\}$. We also use this knowledge to bound the size of lists for SMT checks involving `valuesOf`.

Alias analysis checks if two path variables must reference the same node, i.e., if, for any binding of their longest common prefix (Section 3.1), the schema constrains the variables to bind to the same node. If two variables are aliases, we unify their names in our SMT checks.

4.2.1 Assertion Satisfiability and Tautology.

For an example of alias analysis, consider the following filter:

```
data A = [ B* ]
data B = [ C ]
data C = [ d : Integer ]

Assert B..d > B.C.d
```

Due to the common prefix rule, `B..d` and `B.C.d` will always refer to the same `d`, making this assertion unsatisfiable (since `data B = [C]` and not `data B = [C*]`).

Our system can detect such unsatisfiable assertions by iterating over all assertions and asking Z3 whether their conditions are satisfiable. Here, alias analysis tells us to unify `B..d` and `B.C.d`, so we ask Z3 (essentially) whether $x > x$. We then report Z3's failure result (`UNSAT`) result as an error.

Analogously, we check for and report assertion conditions that are tautologies, i.e., always true.

The above checks can find suspicious rules but are not essential to correctness, so we allow users to override them

4.2.2 Limited Confluence.

Limited confluence guarantees that multiple `@onfail` actions in one stratum — due to one rule or multiple rules — yield a predictable result.

For example, consider an assertion with two `@onfail` actions `v <- 1` and `v <- 0`. These actions are contradictory.

To ensure confluence, we search each stratum for pairs of recovery actions (within the same assertion or in separate assertions) that can modify the same variable, such as

```
Assert P1 where @onfail = x <- A1
Assert P2 where @onfail = x <- A2
```

and ask Z3 for each pair to confirm that

$$\neg P_1 \wedge \neg P_2 \implies (A_1 = A_2)$$

If Z3 times out or finds a counterexample, we report an error.

We prioritise the recovery actions (from highest to lowest) as follows: `discard`, `remove`, `replace`, and update. Higher-precedence actions override lower-precedence actions; e.g., `replace` actions overwrite update actions. We ignore `log` actions, treat `discard` and `remove` as idempotent, and use checks like the above to ensure confluence of updates. For all actions other than `replace`, our semantics guarantee confluence.

When we detect two conflicting `replace` actions (Section 3.2), we allow them if the replacements are on the same node and name tag. Our semantics here are to incorporate all replacements into that list *in an undefined order*, meaning that we do not guarantee confluence in this case.

4.2.3 Validity.

We ensure statically that the output packet is well-formed according to the schema. This allows us to eliminate the *Validator-Out* phase from Figure 1 in Figure 7.

Validity checking has three parts: *type checking* (Section 4.1), *multiplicity checking*, and *type constraint checking*.

Multiplicity checking relies heavily on our multiplicity analysis (Section 4.2) and checks that no transformation (`remove` / `replace` action) can violate multiplicity constraints.

For `remove`, we ensure that the action cannot reduce the number of nodes in a node list below its minimum multiplicity by requiring the minimum multiplicity to be zero.

For `replace` actions, we compute the minimum and maximum number of possible replacements (Section 3.2) and compare against the declared multiplicity.

Finally, type constraint checking ensures that `replace` and update actions on a node of a `Constrained` type (Section 3.4) substitute only values that pass the requirements imposed by the type constraint, which we verify with a query to Z3.

4.2.4 Totality.

We ensure that filters are *total*, i.e., evaluation cannot crash or hang for any well-formed input packet.

This guarantee hinges on the following properties:

- *Language lacks recursion*: Our language does not allow (indirect) recursion or equivalent language features.
- *Finite input*: All input packets are of finite size.
- *All transformations are finite*: We only transform packets during `@onfail` actions. Due to *limited confluence*, each stratum updates an individual node at most once, with the exception of `replace` actions. These may multiply the number of nodes by $O(\#assertions \times \#bindings)$. Nodes created via `replace` can only be bound to path variables in *later* strata.
- *All operations are effectively total and return finite results*: All built-in operations return primitive values (`Integer`, `String`, `Bool`) and are total, with two exceptions: division and modulo. We check all right-hand sides of these operations with Z3 to ensure that they are always nonzero, i.e., our language will reject the expression `a / b` unless Z3 can prove that `b` is nonzero.

Given *finite input*, we see for each stratum that each variable in the stratum can bind to only a finite number of nodes. With a finite number of possible bindings, we evaluate a finite number of expressions in assertion conditions. Since *all operations are effectively total and return finite results*, and since our *language lacks recursion*, all of these evaluations terminate. Since *all transformations are finite*, each stratum will produce a finite number of nodes, from which we conclude termination for filters of finite size.

4.2.5 Effectiveness.

Effectiveness ensures that whenever we cannot satisfy the condition of an assertion, the `@onfail` action of that assertion will either discard the packet or repair it so that the same condition now holds.

To check for effectiveness for actions other than `discard`, we again defer to Z3. For an assertion with condition $P(x)$ and `@onfail` action x <- $A(x)$, we emit the proof obligation

$$\neg P(x) \implies P(A(x))$$

In practice, x may denote multiple variables. This obligation ensures that the repair is immediate, so we will e.g. reject an attempt to repair $x > 0$ via x <- $x + 1$. We handle `replace` by disallowing `replace` actions from producing nodes that are matched in the same stratum. For `remove`, effectiveness is straightforward: all path variables in an action must also be bound in the assertion condition, so `remove` trivially eliminates condition failures.

4.2.6 Local and Global Stability.

Global stability ensures that whenever the filter does not discard the packet, the output packet satisfies the conditions imposed by all assertions. We derive it from three properties: *effectiveness* (Section 4.2.5), *local stability*, and *stability preservation*.

Local stability extends the *effectiveness* guarantee to the entire stratum: each stratum either discards the packet or ensures that it satisfies all assertion conditions in the stratum. We ensure local stability by finding all rules R_2 that may update a node or a descendant of a node that is referenced in a rule R_1. Here, R_1 and R_2 may be the same rule, with different path variable bindings. For each such case, we extract the conditions P_1 and P_2 of R_1 and R_2, respectively, along with the `@onfail` action A_2 of R_2, and query Z3 to confirm the *stability condition* that applying A_2 will not invalidate P_1:

$$P_1(x') \wedge \neg P_2(x) \implies P_1(A_2(x))$$

where x are the affected variables. Specifically, we handle three cases: (i) Limited confluence (Section 4.2.2), which ensures this property for all updates to the *same* node except via `replace`, (ii) `replace`, which we prohibit from introducing new nodes that can be bound in the same stratum, (iii) `remove` / `replace` / update actions $A_2(x)$ on node x that can affect $P_1(x')$ if x' contains x *or a descendant* of x, where we assert the above stability condition. Our system extends this last case to path variables in `valuesOf` expressions, whose stability can depend on both ancestor and descendant nodes.

Stability preservation guarantees that if a packet satisfies all conditions of all assertions in a stratum, then no `@onfail` action in a later stratum may invalidate these conditions. Stability preservation is analogous to local stability, except that (i) we check preservation for a stratum by checking preservation of assertion conditions from all *previous* strata, and (ii) we emit assertion conditions from previous strata as premises, allowing proofs in later strata to exploit properties guaranteed in earlier strata. Global stability follows by induction from *local stability* and *stability preservation*.

4.3 Example Generation

While we can *verify* many useful properties of our filters, we cannot automatically *validate* that they capture the user's intent, i.e., that a filter removes and retains exactly what the user wants the filter to remove and retain.

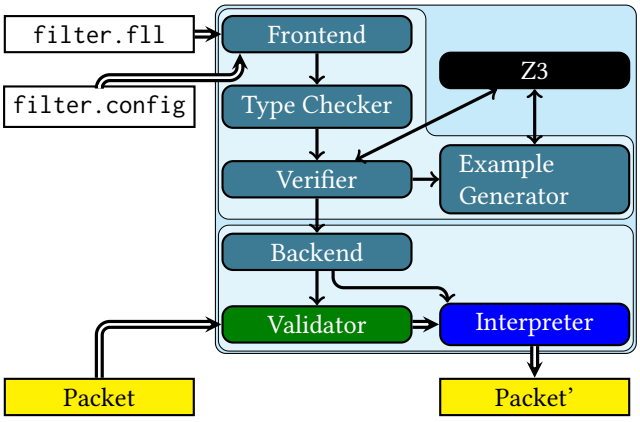

Figure 7. Overview over our system. After type checking and verifying filters, we can run the example generator or run the validator and interpreter (for packet filtering).

We help users validate their filters by providing an *example generator* that synthesises random packets that satisfy the schema. This process is straightforward except for constrained types, for which we defer to Z3 to search for variable bindings that satisfy the constraints. Since we cannot control the randomness in Z3's search for variable bindings, we instead ask Z3 to generate multiple bindings and sample from them. Since Z3 is incomplete, we we cannot guarantee that we can produce an example, but we have not observed this to be a limitation in practice.

5 Implementation

Figure 7 summarises our system. We implemented our prototype in Haskell [16], using Alex[4]/Happy[5] for the frontend.

We translate the input program into an AST, desugar it (e.g., inlining user-defined functions and translating constraint types into assertions), and decorate it with types and expanded variable names (Section 3.1). We then run our SMT-based correctness checks (Section 4.2) on Z3, via the SBV library[6]. Due to limitations in SBV, we preprocess queries that involve explicit existential or universal quantification to translate them into Prenex Normal Form (specifically queries that use our explicit `forall` and `exists` subexpressions).

Our implementation interprets filter programs and communicates with packet (un)parsers through a general-purpose tree representation. We expect that implementing a future backend for compilation, targeting Rust [18] or C, will be substantially simpler for our system than for the existing ZoneGuard system, which has access to unconstrained Python.

[4] https://www.haskell.org/alex/, accessed 2020-07-24
[5] https://www.haskell.org/happy, accessed 2020-07-24
[6] https://hackage.haskell.org/package/sbv, accessed 2020-07-19

6 Evaluation

We have evaluated our language by comparing it to the current state-of-the-practice technique used at Advenica AB, which implements filters in the embedded system ZoneGuard. The system's Python-based filters provide none of the static guarantees that our system offers (Section 4) and rely on testing and logging to detect bugs. We compare the two languages for compactness (Section 6.1) and report on the language's suitability for filtering by conducting a case study with a filter engineer familiar with ZoneGuard (Section 6.2).

6.1 Filters

To explore the suitability of our DSL for different filtering tasks, we translated four of the 15 existing ZoneGuard filters (three production filters, for Syslog [11], FTP, and RDP, one full-featured prototype for SQL) and an RDP filter variant (Section 1) into our language. Each filter addressed threats related to information leakage, the SQL filter also addressed tampering threats. Table 2 summarises our results.

Here, the *Schema size* describes the information responsible for the *Validator* processes in Figure 1 (and written in a separate specification language, for ZoneGuard), while the *Filter size* corresponds to the *Filter* process in Figure 1.

We found that our filter specifications are roughly 2–3× smaller than the existing ZoneGuard specifications. The difference is less extreme for schema specifications. Our Syslog schema even exceeds the original schema in size: ZoneGuard schemas have syntactic support for subrange types (integers with upper/lower bounds), which we encoded in our more general (and more verbose) constrained types. ZoneGuard does not support general constrained types.

We also found that for all of the existing filters, we could summarise the existing Python code using only three `Assert` declarations, with the exception of the SQL filter, which required 14 `Assert`ions. This filter processes SQL statements and offers a high degree of configurability. Only one filter (`Syslog`) used explicit stratification, for a single rule.

Overall, our language mapped closely to the needs of the existing filters and allowed us to re-implement them much more concisely. We did not observe any practical problems due to the restrictions that we impose on expressivity.

When compiling, we spend most of our time in Z3 (Table 2). On an Intel i7-7700 CPU with 16 GiB RAM on Ubuntu 18.04 with Z3 4.8.5 (64 bit) we measured a maximum of 1.01s, checking the FTP filter. No individual check took more than 75 ms. On an Intel i5-8250U laptop (8 GiB RAM) we observed similar results (3% to 20% slower). We conclude that the turnaround time for our checks is suitable for practical use.

6.2 Case Study

To explore the utility of our language, we conducted a case study with a software engineer who was closely familiar with the Python-based ZoneGuard filters. The objective of our study was to explore the suitability of our language for the filtering task, and its strengths and weaknesses as a notational mechanism, with focus on the following questions:

- **RQ 1**: How does the new filter language's *writability* compare to the existing system?
- **RQ 2**: How does the new filter language help write *less error-prone code* than the existing system?
- **RQ 3**: Is the filter language suitable as a *replacement* for the ZoneGuard Python system?

To develop and run our case study, we followed best practices as described by Runeson and Höst [21].

6.2.1 Test Subject. The test subject for our case study was a software engineer at Advenica AB who was part of the design and development team behind ZoneGuard, and has substantial experience implementing filters and schemas for it. The subject was also the first author's industrial M.Sc. co-supervisor. This M.Sc work developed an earlier version of our system. The subject's involvement in this thesis work and the development of the filter language was confined to requesting a small number of features (most notably regular expression matching) and to providing feedback on the thesis document, approximately one year prior to the study.

6.2.2 Study. We first confirmed the modalities of the study with the test subject. We then began the study with the first author (the implementer of our system) giving a one-hour tutorial of the language and its semantics.

We then asked the test subject to implement a filter for the Syslog protocol [11] in our new filter language. The test subject had previously implemented the same protocol for ZoneGuard. While implementing this filter, the test subject had access to the first author and could ask questions if needed. The first author kept a log of all interactions. The first author had previously implemented the Syslog protocol in our language himself, without making any changes to the language. We selected Syslog because we expected it to be of average complexity, compared to other candidate protocols.

Afterwards, the second author asked the test subject to fill out the "Cognitive Dimensions of Notation questionnaire" [6] which asks a number of questions related to the different "Cognitive Dimensions of Notation" [5]. We replaced section headings for questions on the different cognitive dimensions by letters and asked the subject to not use internet search while filling out the questionnaire, to reduce bias.

The second author reviewed the replies, and conducted a follow-on online interview about unclear points in the questionnaire and a number of questions aimed at **RQ 3**, on the same day. The second author then summarised the key points from the replies and sent them to the test subject for review before sharing them with the first author.

6.2.3 Observations. The test subject completed the task in 52 minutes, of which they spent 15 minutes (29%) bug-fixing. The subject spent the last of these minutes generating

Table 2. Summary of our filters: lines of code and number of tokens in the filter specifications of our tool vs. the corresponding ZoneGuard filters, plus Z3 verification effort for our tool. We list the lines of code for the schemas separately from the filters. We report LoC via sloccount for ZoneGuard (excluding imports), and by counting non-blank non-comment lines for our tool. For token counts, we used the Python3 tokeniser, discarding comment tokens. We used the same tokeniser for our language, but also discarded indent, dedent, and newline tokens. For Z3 verification, we list the number of invocations of Z3 for our optional rules (Section 4.2.1) and mandatory rules (all other rules) separately. All times in milliseconds, averaged over 100 runs.

	Schema Size				Filter Size				Z3 Queries				Compile
	ZoneGuard		Our Tool		ZoneGuard		Our Tool		Mandatory		Optional		Time
Filters	LoC	Tok	LoC	Tok	LoC	Tok	LoC	Tok	#	Time	#	Time	(other)
RDP	13	83	7	53	38	208	9	77	0	0	0	0	44
RDP (Section 1)	9	59	3	34	31	249	12	115	10	127	6	68	46
Syslog	17	172	11	198	45	384	12	117	0	0	22	287	47
Syslog (case study)	17	172	13	210	45	384	11	108	0	0	22	260	47
FTP	18	189	17	153	81	593	32	312	0	0	34	1012	62
SQL (prototype)	45	446	19	227	258	1854	80	825	20	621	28	444	180

four synthetic examples with our example generator (Section 4.3), of which one was a duplicate.

The subject worked on the specification offline (without running our tool) for 37 minutes, submitting a specification that was complete except for the following bugs that we extracted from the compiler logs:

- *Missing type annotations*: The code used eight constrained type declarations but initially omitted the specification of the underlying primitive type. The test subject added all annotations in less than one minute.
- The test subject initially used `@onfail` handlers without following them by '=' signs but fixed this issue after feedback from the language frontend.
- The test subject implemented a type that they had referenced but not included in the filter specification after feedback from the language frontend.
- The test subject encountered and fixed a bug in string concatenation during logging. The fix required him to explicitly convert an integer to a string.
- The test subject encountered two more complex bugs that we analysed in more detail (Sections 6.2.4 and 6.2.5).

Except for the two more complex bugs, the test subject solved all bugs in less than two minutes.

6.2.4 Access Path Bug. The test subject encountered an error message stating that the variable `Syslogs..Hostname` was undefined (without giving a line number). This access path does not literally occur in the program; our compiler generated it implicitly from the variable '`Hostname`' in the test subject's code, since the root node has node type `Syslogs`.

The test subject spent approximately four minutes editing an occurrence of the name `Syslog..Hostname` (note the singular `Syslog`, a different node type) in a nearby line before asking the first author for help. The first author clarified that the compiler prepends the root node type internally (a leaked implementation detail during context handling).

The subject spent 4–5 minutes to resolve this bug, and less than half a minute to resolve a second, analogous bug.

6.2.5 Unstable Assertions Bug. The system reported the final bug in the user study as `UnstableAssertions`, i.e., lacking *stability* (Section 4.2.6)[7]:

```
After evaluating the @onfail action of {assertion2},
    a previously satisfied precondition of
    {assertion1} may evaluate to False, in {model}
where
  {assertion2} = Assert@Line=18
  {assertion1} = Assert@Line=28
  {model} = Syslogs..Syslog..Hostname%2 -> "\NUL"
```

Here, `{assertion2}` referenced an assertion that would remove an entry from the system log under certain conditions, while `{assertion1}` ensured that the filter would not propagate empty system logs. The four variable bindings reported in `{model}` were unrelated to the conflict.

The test subject reported initially experiencing confusion for why the two assertions would be incompatible, until it became clear to him that the two conditions could not be true at the same time. They remembered a related example from the tutorial that introduced ordering via `@prec` and addressed the bug using the same technique.

The test subject resolved this bug in less than 3.5 minutes and reported it as easier to resolve than the Access Path bug.

6.2.6 Interactions. In total, the test subject asked the first author for help four times during the study:

1. To clarify the availability of empty strings (omitted from the grammar by accident). The test subject's code ultimately did not utilise empty strings.
2. At what time schema constraints apply, which the first author clarified (both for input and for output).

[7] `{model}` shortened from 4 to 1 variable assignments for space.

3. Regarding whether schema nodes had to follow the exact naming scheme as outlined in the task description (not necessary for the study).
4. Regarding the Access Path bug (Section 6.2.4).

6.2.7 Implementation Outcome. Upon manual inspection, we found the implementation to meet the specification in all points but one: the specification had asked for a certain type of strings to have a length of "more than one", the implementation allowed strings to have a length of "at least one". This was in fact the intended behaviour as exhibited by the existing Syslog filter and a bug in our specification.

6.2.8 Insights from the Questionnaire and Interviews. This was the test subject's first interaction with our system. They estimated that they spent 50% of his time reading documentation, 30% translating the specification into the notation of our DSL, and 20% of his time bug-fixing (our measurements put this number at 29%).

The subject reported answers on most of the questions related to the Cognitive Dimensions of Notation [6].

Visibility and Juxtaposibility: The subject found the notation well-suited to finding and to comparing information, with two exceptions: they found that working with the implicit and explicit context of assertions could be "non-trivial to figure out for reviewers".

Viscosity: The subject reported that changes were easy to make, but noted (also regarding **Consistency**) that changing from type constraints to stand-alone assertions required substantial typing, and that such changes were necessary to produce custom-tailored log messages.

Hard Mental Operations , **Error Proneness**, and support for **Role Expressiveness**: The subject listed as challenges the syntactic dissimilarity between our language and mainstream programming languages, as well as the interactions with explicit and implicit scoping, noting that these interactions might lead to misunderstandings when reading code. Moreover, they noted that multiplicities for variable bindings could be challenging for "newcomers".They otherwise found the syntax to have an "overall good structure".

Closeness of Mapping: The subject reported that the declarative nature of the DSL made it a close match to the reasoning that filter developers use, but observed that the syntax might offer readability challenge to "newcomers".

Hidden Dependencies: The subject expressed concern about the implicit reference to the root node (the *context* in Section 3.1), which they argued might lead to name capture during refactoring (due to `@context` decorators).

Progressive Evaluation and **Premature Commitment:** The subject noted that the system can only be tested once the schema structure is specified, but found that it was possible to incrementally add assertions afterwards while generating synthetic permissible packets to aid in identifying the need for additional assertions. They expressed the opinion that early commitment to a schema structure was inherent in the intended use of the system.

Provisionality: They argued that the language was not very supportive of experimentation, due to its strict syntax, apart from omitting assertions.

Overall, they found the system to have a "well-built foundation" with "some rough edges". They expected the system to provide superior correctness guarantees and potentially higher performance than the existing ZoneGuard system if expanded with a C or Rust backend, but noted that the code base was likely bigger than that of the existing system and thus likely to require more maintenance and quality assurance effort. They proposed using the new system concurrently with the existing Python system, with the latter handling legacy specifications and use cases with unforeseen feature requirements, but noted potential bias due his authorship of the previous system.

Finally, they proposed a new feature for logging unmodified packets.

6.2.9 Case Study Conclusions. The case study helped highlight strengths and weaknesses of our language, as well as avenues for future work. Some of the latter are technical (improving error reporting, adding logging support, custom error messages for constrained types), though we also see a need for further studies on the readability of path variables, especially with user-defined contexts (`@context`).

RQ 1: Is code written in the new filter language easier to write? Our declarative approach differs substantially from the dominant imperative programming paradigm. The test subject mentioned this difference as a potential concern, especially to "newcomers", but expressed satisfaction with the language's syntax and *closeness of mapping* to the domain concepts. Since the user was able to construct a correct filter in our language in less than an hour with minimal support, we find no major concerns about the language's writability but cannot conclude the writability of our system is higher than that of Python in the ZoneGuard system.

RQ 2: Does the new filter language help make code less error-prone? By design, our language disallows errors that are possible to make in languages like Python. The subject's interactions with our system confirm that our mechanisms are effective. Moreover, the size of the subject's filter implementation in our language was substantially smaller than the original implementation in Python (between 40% and 75% smaller, depending on how we count).

We conclude that our static checks offer effective safety guarantees beyond that of the existing system.

RQ 3: Is the new filter language adequately powerful to replace ZoneGuard Python? The study subject recommended using the two approaches in complementary roles, with the existing framework supporting legacy use cases and use cases with unexpected feature requirements (since

ZoneGuard allows unconstrained modifications to packets, we cannot, by design, have equal expressivity). While our language supports a variety of use cases (Table 2) and is extensible (Section 6.3), we are not currently able to assuage concerns about expressivity. This suggests a need for additional studies on a broader class of filters and addressing known limitations, especially logging without filtering.

Threats to Validity. Regarding *external validity*: Our case study explored our language with one test subject and one protocol, exposing the test subject to the language for two hours (plus time for questionnaire and interview). To obtain more robust answers to our research questions, we are considering using this study as a pilot for a larger user study.

Regarding *internal validity*: The former 'industrial thesis co-supervisor' relationship between study subject and first author raises the risk of more favourable comments towards our system, while the study subject's authorship of the existing (essentially competing) system raises the risk of the converse. We have tried to reduce this risk by having a separate author collect and evaluate the results and giving the study subject the opportunity to veto any observations reported by the second author before they reached the first author and to abort the study at any time.

6.3 Language Limitations

By design, our language is not Turing-complete, nor does it allow recursive tree construction. We have inherited the requirement that schemas may not recurse from the original Python-based ZoneGuard system and note that this constraint simplifies the semantics of our "longest common prefix" semantics for variables. We expect that we could easily extend our system to recursive schemas, if necessary.

Our use of an incomplete external SMT solver makes our correctness checking incomplete: our system may reject some correct filters. In this case, we may sometimes be forced to add new axioms to the SMT solver that increase the size of our trusted base. We have only observed this challenge once, during the implementation of our first filter (SQL), but believe it to be an acceptable trade-off for the additional correctness guarantees that we offer.

Finally, we do not currently support adding new subtrees to the output beyond replacing single nodes. We anticipate no substantial challenges in adding this feature, which will simplify support for future use cases (e.g., HTTP redirection).

7 Related Work

DSLs have been used before for security in networking, e.g. for configuring network protocol analysers,[8] or intrusion detection systems [24]. Hamdi et al. describe PPL, a DSL for securing distributed systems [15]. They use *rules* with *triggers* and *actions*, analogous our system. They can configure multiple systems at once, but operate below the application level and cannot inspect or modify packet contents. Similarly, Youssef et al. [4, 26] check firewall configurations against a *security policy* but again only at the network protocol level.

Backes et al. describe the Zelkova system [2], which can check user-defined Amazon Web Services security policies against both best practices and custom organisational rules. Zelkova's ability to check that a given policy is at least as strict as another policy, again using an SMT solver, is an appealing feature that we expect to be easy to adopt.

At network layers below the application level, domain-specific languages for network filtering are widely available [19]. Recent languages include NetKAT [1, 23], which provides a broad mathematical foundation for reasoning about global and local network packet processing, and P4 [7], which can target a wide variety of switches.

Outside of the networking domain, Schematron [10] and CLiX[9] are languages for checking properties similar to our assertion conditions. Schematron Quickfix[10] adds facilities analogous to our `@onfail` rules. These XML-based languages do not provide correctness guarantees.

Our language is also related to other declarative languages in the tradition of Prolog [25] and Datalog [3], though we permit updates (similarly to some Datalog dialects [13]) and offer features finely tuned to our application domain.

Our current example generator is related to the general domain of *fuzzing*. We expect that we can improve its utility through dynamic-symbolic execution, or concolic testing [12, 22] in the style of Li et al. [17]'s work on systematically exploring all possible paths through a dataflow program.

8 Conclusions

We have presented a novel application-level network packet filtering language developed at Advenica AB. Our declarative language provides strong correctness guarantees at a modest cost in analysis time, a close mapping to the problem domain, and sufficient expressive power to handle many known use cases. We have explored our language with four filter implementations that suggest that our language is substantially more concise than general-purpose languages, and with a systematic case study that found evidence that the language is suitable for its intended task and that its correctness guarantees are effective.

Acknowledgments

We thank Advenica AB and especially the anonymous test subject for support and feedback, and Per Runeson for advice on experimental setup. This work was partially supported by Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation.

[8] https://wiki.wireshark.org/CaptureFilters, accessed 2020-07-18

[9] http://www.clixml.org/clix/1.0/clix.xml, accessed 2020-07-22

[10] http://www.schematron-quickfix.com/, accessed 2020-07-22

References

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. *SIGPLAN Not.* 49, 1 (Jan. 2014), 113–126. https://doi.org/10.1145/2578855.2535862

[2] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. Luckow, N. Rungta, O. Tkachuk, and C. Varming. 2018. Semantic-based Automated Reasoning for AWS Access Policies using SMT. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. 1–9.

[3] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. 1985. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*. 1–15.

[4] N. Ben Youssef, A. Bouhoula, and F. Jacquemard. 2009. Automatic Verification of Conformance of Firewall Configurations to Security Policies. In *2009 IEEE Symposium on Computers and Communications*. 526–531.

[5] Alan F. Blackwell, Carol Britton, Anna Louise Cox, Thomas R. G. Green, Corin A. Gurr, Gada F. Kadoda, Maria Kutar, Martin Loomes, Chrystopher L. Nehaniv, Marian Petre, Chris Roast, Chris Roe, Allan Wong, and Richard M. Young. 2001. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Proceedings of the 4th International Conference on Cognitive Technology: Instruments of Mind (CT '01)*. Springer-Verlag, Berlin, Heidelberg, 325–341.

[6] Alan F. Blackwell and Thomas R. G. Green. 2000. A Cognitive Dimensions questionnaire optimised for users. In *Proceedings of the 12th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2000, Cosenza, Italy, April 10-13, 2000*. Psychology of Programming Interest Group, 10. http://ppig.org/library/paper/cognitive-dimensions-questionnaire-optimised-users

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[8] James Clark and Steve DeRose. 1999. *XML Path Language (XPath) version 1.0*. Recommendation. World Wide Web Consortium. See http://www.w3.org/TR/xpath.html.

[9] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo Theories: Introduction and Applications. *Commun. ACM* 54, 9 (Sept. 2011), 69–77. https://doi.org/10.1145/1995376.1995394

[10] Philip Fennell. 2014. Schematron-more useful than you'd thought. *XML LONDON* (2014).

[11] R. Gerhards. 2009. The Syslog Protocol. RFC 5424 (Proposed Standard). http://www.ietf.org/rfc/rfc5424.txt

[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[13] Todd J Green. 2015. Logiql: A declarative language for enterprise applications. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 59–64.

[14] Kurt Gödel. 1931. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik* 38, 1 (1931), 173–198.

[15] Hedi Hamdi, Mohamed Mosbah, and Adel Bouhoula. 2007. A Domain Specific Language for Securing Distributed Systems. In *ICSNC '07: Proceedings of the Second International Conference on Systems and Networks Communications*. IEEE Computer Society, Washington, DC, USA.

[16] Simon Peyton Jones. 2003. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

[17] Kaituo Li, Christoph Reichenbach, Yannis Smaragdakis, Yanlei Diao, and Christoph Csaller. 2013. SEDGE: Symbolic example data generation for dataflow programs. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 235–245. https://doi.org/10.1109/ASE.2013.6693083

[18] Nicholas D Matsakis and Felix S Klock. 2014. The rust language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104.

[19] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture.. In *USENIX winter*, Vol. 46.

[20] J. Pastel and J. Reynolds. 1985. File Transfer Protocol. RFC 959. https://doi.org/10.17487/RFC0959

[21] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (2009), 131–164. https://doi.org/10.1007/s10664-008-9102-8

[22] Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 419–423.

[23] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. 2015. A Fast Compiler for NetKAT. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 328–341. https://doi.org/10.1145/2784731.2784761

[24] Diomidis Spinellis and Dimitris Gritzalis. 2000. A Domain-specific Language of Intrusion Detection. In *Proceedings of the 1st ACM Workshop on Intrusion Detection Systems*. ACM. http://www.spinellis.gr/pubs/conf/2000-CCS-DSLID/html/paper.html

[25] David HD Warren, Luis M Pereira, and Fernando Pereira. 1977. Prolog-the language and its implementation compared with Lisp. *ACM SIGPLAN Notices* 12, 8 (1977), 109–115.

[26] N. B. S. B. Youssef and A. Bouhoula. 2010. Automatic Conformance Verification of Distributed Firewalls to Security Requirements. In *2010 IEEE Second International Conference on Social Computing*. 834–841.

[27] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 114–124.