Investigating the use of Semantic Web Technologies in OpenKAT

Perry van der Zande
MSc Artificial Intelligence
University of Groningen

Patrick Darwinkel
MSc Information Science
University of Groningen

Rachelle Bouwens
MSc Artificial Intelligence
University of Groningen

Maurits Merks
MSc Artificial Intelligence
University of Groningen

Abstract

This paper explores the application of Semantic Web Technologies (SWT), specifically the Web Ontology Language (OWL), Shapes Constraint Language (SHACL), and Semantic Web Rule Language (SWRL), to model the knowledge representation and reasoning functionality of OpenKAT, a vulnerability analysis tool. Focusing on a subset of OpenKAT, our implementation achieves near feature-parity. We explain the layout of our ontology, how we leverage SHACL shapes for constraint enforcement, and how we use SWRL for logical inference. Challenges such as handling the open world assumption are discussed. Finally, we discuss limitations and provide valuable insights into the potential and constraints of using Semantic Web Technologies for vulnerability analysis tools.

1 Introduction

OpenKAT[1] is a free and open source vulnerability analysis tool. It collects information about (networked) entities[2] through *boefjes*, and then normalises the output into a common data model through *whiskers*. Business rules (*bits*) then draw conclusions from the data and create findings (if any). This results in a constant feedback loop that automatically explores infrastructure and objects and fills the data model with new information. Critically, OpenKAT is recursive in the sense that newly created objects may also be used as input for *boefjes*, *whiskers*, and *bits*. In OpenKAT, this data model is implemented through the *Octopoes* module, as strict-typed Python classes backed by a bitemporal knowledge graph[3]. OpenKAT gains knowledge through three methods:

- Observation by scanning the world;
- Information declared to be true by users;
- Reasoning through defined rules (*bits*).

There is currently no formal modelling regarding the knowledge graph and accompanying rule engine, beyond what is implemented in the Python code. However, as Lisser (2023) notes, Octopoes operates in the domain of first-order logic and open-world reasoning. A formalised ontology can help with identifying inconsistencies in the models and rules. It would be helpful to detect modelling issues that stem from (implicit) inference chaining, where the result of some rule is used as input for another. Occasionally, there have been unexpected bugs in the software which ended up being logical consequences resulting from Octopoes' inference[4]. In addition, semantic web technologies enable versioned, interoperable, and distributed ontologies (e.g. different OpenKAT deployments can have flexible modifications in their data model).

> *While the architecture of KAT is quite clear and documented, the workings of Octopoes are still slightly obscure and not sufficiently formally defined. This lack of documentation and formalization in this core component of KAT is currently resulting in unclear constraints considering the rule-engine, vagueness of to be made trade-offs between expressivity and computational complexity, and more.* - Lisser (2023)

In this paper, we propose to build an ontology combining the Web Ontology Language (OWL), Shapes Constraint Language (SHACL), and Semantic Web Rule Language (SWRL).

1.1 Research questions

Our goal is to create a formal model of a part of OpenKAT using OWL, SHACL, and SWRL. Therefore, our first research question is "Can we reach feature-parity on a subset of OpenKAT with Semantic Web Technologies?"

This project is exploratory: it is at the start unknown what is possible, what the difficulty is, and what fits in the scope of this project. Thus, the goal is to investigate this, so that in the case of future work, it is better known what to keep in mind for similar endeavours.

[1] `https://openkat.nl`
[2] Such as hostnames and IP-addresses.
[3] `https://www.xtdb.com`
[4] For example, see `https://github.com/minvws/nl-kat-coordination/issues/`{1701, 1963, 1087}.

Therefore, our second research question is “What (if any) problems arise when modelling OpenKAT with Semantic Web Technologies?”

2 Related work

Previous attempts at standardising information within the cybersecurity domain include the Unified Cyber Ontology (UCO)[5] by Syed et al. (2016). UCO provides a standardized model for representing, exchanging, and analyzing cyber-related information using RDF/OWL and SHACL. Since the UCO aims to be domain-agnostic and broad, it is very abstract which makes practical applications challenging. However, it does highlight the possibilities for ontologies in the cybersecurity domain. Moreover, Oltramari et al. (2014) mentions that with a rapidly expanding cyber landscape, building a comprehensive model remains a major objective for the cybsersecurity community. However, these attempts at ontologies seem to be based around the purpose of classifying different attacks by incorporating rich catalogs of cyber attacks, exploits and vulnerabilities. This is different than our goal of modelling an existing system with some defaults to pinpoint these defaults.

Rivadeneira and Gómez (2020) provides a systemic literature review of the use of ontologies in cybersecurity. While some papers used a methodology like Methontology, most papers construct ontologies using Protégé. This seems to indicate that Protégé is the industry-standard tool for building (cybersecurity) ontologies.

There has been more work in the field of cybersecurity knowledge graphs, but none are similar to OpenKAT (Chen et al., 2002; Iannacone et al., 2015; Mavroeidis and Bromander, 2017; Liu et al., 2022).

3 Methods

3.1 Ontology

We detail the layout of the ontology, starting with the classes, followed by the object properties and the data properties. For the sake of conciseness, details about the individuals that populate these classes are left out. However, they are visible in the ontology.

3.1.1 Class Hierarchy

The classes in the ontology represent the main structure of the components in the mispo.es site. For instance, it reflects the multiple types of DNSRecords, IPAddress and DNSSPF Mechanisms. A complete overview can be found in Figure 1.

[5] https://unifiedcyberontology.org/

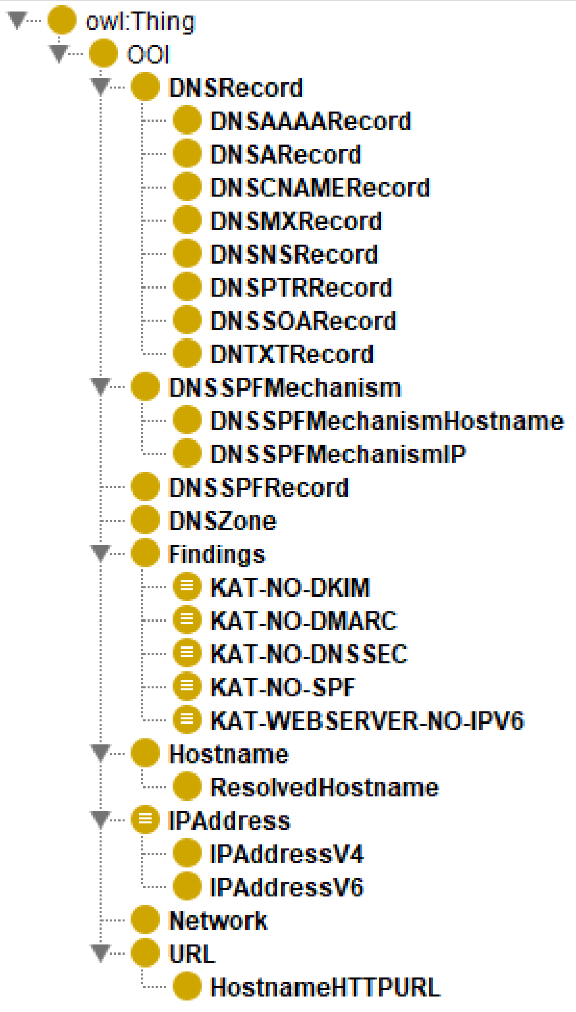
```
owl:Thing
  OOI
    DNSRecord
      DNSAAAARecord
      DNSARecord
      DNSCNAMERecord
      DNSMXRecord
      DNSNSRecord
      DNSPTRRecord
      DNSSOARecord
      DNTXTRecord
    DNSSPFMechanism
      DNSSPFMechanismHostname
      DNSSPFMechanismIP
    DNSSPFRecord
    DNSZone
    Findings
      KAT-NO-DKIM
      KAT-NO-DMARC
      KAT-NO-DNSSEC
      KAT-NO-SPF
      KAT-WEBSERVER-NO-IPV6
    Hostname
      ResolvedHostname
    IPAddress
      IPAddressV4
      IPAddressV6
    Network
    URL
      HostnameHTTPURL
```

Figure 1: Hierarchy of classes in the ontology.

3.1.2 Object Properties

In order to model the different relations between the components of a network, object properties are needed. An overview is presented in Table 1.

3.1.3 Data Properties

Data properties serve to differentiate multiple individuals within the same class. In Table 2, an overview is presented of the data properties, as well as their range and domain. While these properties do not reflect the entirety of a network, they represent the important elements that are needed to make inferences.

3.2 SHACL

When building an ontology as large as this, in terms of classes, data properties and instances, mistakes are bound to happen when creating it. To partially alleviate this issue, a Shapes Constraint Language (SHACL) shape was created for each class. By creating a shape, we are able to enforce certain conditions for each class to ensure an instances satisfies the right properties. For instance, the DNSARecord shape gives a warning if an instance does not have the object property `hasIPAddress`. A complete overview of all the shapes is given in Table 3. Conceptually, SHACL makes a closed-world assumption on a single class.

3.3 Inference & SWRL

Many bits in OpenKAT parse one or more objects and add new properties or relationships to these objects. To infer object and data properties based on other properties, one can use the Semantic Web Rule Language, or SWRL. Protégé is equipped with a plugin to make such rules. These rules can then be used by the Pellet reasoner to make inferences. Examples of SWRL rules are shown throughout this report.

4 Results

4.1 Reflection on current ontology

In OpenKAT, URL is subclass of OOI, WebURL of OOI and HostnameHTTPURL of WebURL. Our current model has HostnameHTTPURL as subclass for URL. We asserted the URL `https://mispo.es`, and SWRL rules add the scheme, port and path to the existing URL object. Because these data properties' domains are HostnameHTTPURL, the URL object is inferred to be also of class HostnameHTTPURL. This is actually quite convenient: we don't have to create a new object like OpenKAT, but if we were to query for HostnameHTTPURL objects, we would still get something back. So objects having this class is an indication that it is a processed URL.

Similarly, in OpenKAT, ResolvedHostname is not a subclass of Hostname. But as we have modelled it as such, and set the ResolvesToAddress's domain to ResolvedHostname, Protégé infers that a Hostname for which it has been inferred that it resolves to an IP address is a ResolvedHostname. For other rules or for querying purposes, this could be useful.

4.2 Weakened inference due to OWA

OWL makes an open-world assumption. This can be paraphrased by the maxim *the absence of evidence is not evidence of absence*, and means that OWL reasoners only infer things based on positive evidence. In practice, this makes inferring things through classical logical negation non-trivial. For example, consider the class `Hostname` which has two subclasses, `HostnameWithDNSAAAARecord` and `HostnameWithoutDNSAAAARecord`. If one makes the following assertions:

- `Hostname` is the disjoint union of `HostnameWithDNSAAAARecord` and `HostnameWithoutDNSAAAARecord`;
- `HostnameWithDNSAAAARecord` is equivalent to having 1 or more linked `DNSAAAARecords`;
- `HostnameWithoutDNSAAAARecord` is equivalent to having at most 0 linked `DNSAAAARecords`.

Then by these definitions, both subclasses are each others inverse, and membership of them is mutually exclusive. In a closed-world, in which the knowledge base is considered complete and authoritative, an instance of a `Hostname` must also be either one of the subclasses. This makes sense intuitively, as you either have a linked record, or you do not. In logic programming languages such as Prolog, this is implemented through *negation-as-failure* and is known as *weak negation*.

With these definitions, an OWL reasoner correctly infers `Hostnames` with 1 or more linked `DNSAAAARecords` to also be a member of `HostnameWithDNSAAAARecord`. However, `Hostnames` without any linked `DNSAAAARecords` are not inferred to be members of `HostnameWithoutDNSAAAARecord`. At a surface level, this is annoying, as we might think that this piece of obvious knowledge should be applied to the data.

However, this is may actually be desired behaviour in the context of OpenKAT. As OpenKAT has incomplete knowledge about the world (i.e., there may be unseen objects, or knowledge about existing objects that we don't know about, because we have not yet discovered them through boefjes), new information from scans might very well invalidate assumptions made in a closed-world. This can be easily demonstrated with the following simple experiment:

1. Create a fresh installation of OpenKAT[6];
2. Disable all boefjes (but keep bits and whiskers enabled);
3. Add any `Hostname` and let all bits run.

Then, notice that the findings `KAT-MISSING-DKIM`, `KAT-MISSING-SPF`, and `KAT-MISSING-DMARC` are created. At a surface level, this might seem fine. But note that OpenKAT has made a claim about the world based on the absence of evidence, without actually having bothered to scan anything at all. It is entirely possible that these findings are immediately invalidated by a single boefje scan, as it might turn out the object in question does indeed have valid DKIM, SPF, and DMARC. This illustrates a fundamental consideration that has to be made for OpenKAT: is it acceptable to

[6]Any version from before this paper was written.

make claims about security without having gathered any substantial evidence?

The advantage of monotonic (i.e., open-world) reasoning is that additional knowledge will not invalidate previous claims. When the system makes claims, these are always based on the existence of concrete positive evidence. Non-monotonic (i.e., closed-world) reasoning (Strasser and Antonelli, 2019) is more powerful, in the sense that more knowledge can be inferred overall. However, this comes at the risk of new evidence invalidating previous claims. This is related to the concepts of *belief revision* and *defeasible reasoning* in formal logic (Hansson, 2022; Koons, 2022), and can result in awkward situations such as the one described above. Note that negation-as-failure is considered to be a form of non-monotonic, autoepistemic modal logic (Garson, 2023): if something cannot be proven, then it is *believed* to be false.

Open-world and closed-world reasoning are not fully mutually exclusive, and some implementations of the semantic web support local closed-world reasoning (Grimm and Motik, 2005; Lamy, 2017). This means that the ontology overall is treated as if it is open, but some parts may be explicitly asserted to be complete (such as membership of some class through negation). Regardless, we believe OpenKAT should make a conscious decision which model of inference should be adopted, and that it should be made clear to stakeholders what that means for their security.

As an alternative to findings based on the absence of things, OpenKAT could consider an approach based on the total state of knowledge. For example, instead of throwing `KAT-MISSING-` findings, it could provide users with some sort of pre-defined checklist for certain classes, which could look like this:

Hostname `openhond.nl`

- Discovered valid DKIM
  - `[X]` DKIM was found, but it was not valid (strong negative evidence)
- Discovered valid SPF
  - `[ ]` No SPF found (yet) (weak negative evidence)
- Discovered valid DMARC
  - `[ ]` No DMARC found (yet) (weak negative evidence)
- Discovered valid DNSSEC
  - `[V]` Valid DNSSEC found (strong positive evidence)
- Discovered at least one valid IPv6 record alongside IPv4 record(s)
  - `[-]` Not applicable, as no IPv4 record was found (missing antecedent condition)
  - `[ ]` No IPv6 record(s) found (yet) (weak negative evidence)

Arguably, the above checklist is superior to single findings as it takes into account the strength of evidence. Additionally, it constrains and defines the type of issues that may plague specific classes.

4.3 Inferring existence of new individuals

In OpenKAT, business rules (bits) apply to objects that have been manually asserted or found using boefjes and whiskers. In most cases, the result of the application of bits is the inference of new objects. This poses a problem for modelling bits with semantic web technologies like OWL and Protégé, as these ontology-based technologies are not designed to infer the existence of new individuals. Fortunately, by making some modeling choices, alternative ways of implementing these bits is still possible.

4.3.1 New properties

An insightful example is the `dns-resolving` bit. In OpenKAT, this bit checks if, for a given Hostname, there exists in the knowledge graph a DNS A record or AAAA record which maps the Hostname to an IP address. If so, this bit yields a new object of the type `ResolvedHostname`, which links a hostname to the IP address it resolves to.

This bit cannot be directly implemented in Protégé, as new individuals cannot be inferred. Instead, a SWRL rule can be written as shown in Figure 2. Instead of making a new object, this rules adds an object property, linking to the IP address, to the already existing Hostname. This avoids the need to make a new individual. Arguably, assigning a property to an existing individual is more efficient than creating a new instance of another class. From an ontology perspective, it also makes more intuitive sense to add a property instead of making a new object of a completely different class; a `ResolvedHostname` is nothing more than a Hostname that links to an IP address.

4.3.2 Findings

Next to inferring new links between objects, OpenKAT also produces findings. Findings represent vulnerabilities of scanned system, which is the end result that the user of OpenKAT is interested in. In OpenKAT, bits look for vulnerabilities like misconfigured settings or the absence of some objects, among others (see Section 4.2). If a vulnerability is found, OpenKAT creates

```
Hostname(?h) ^ DNSRecord(?r) ^ IPAddress(?a)
^ hasIPAddress(?r, ?a) ^ hasHostname(?r, ?h)
-> resolvesToAddress(?h, ?a)
```

Figure 2: SWRL rule for resolving DNS.

a new `Finding` object, with an appropriate description of the finding. The system then presents all the findings to the user.

To represent these findings, we made separate classes for each finding (see Figure 1). Instead of creating new objects, we have implemented SWRL rules to make related individuals an instance of the corresponding Findings class. So for example, if a Hostname has no DNSSEC, the Hostname will be inferred to be a member of the class KAT-NO-DNSSEC. This way of modeling findings allows us to retrieve a list of all findings that have been inferred.

4.4 Limited flexibility in rules

Many bits in OpenKAT parse the text content of an object's properties to infer new objects or properties. The parsing of such text can become quite complicated. To infer object and data properties based on other properties, one can use SWRL. Unfortunately, we have found that the flexibility of the SWRL rules that the selected reasoner can work with is quite limited.

A simple illustration of this is the `url-classification` bit. In OpenKAT, it decomposes a URL into a schema, a hostname, and a port. So, for example, it takes the URL `https://mispo.es/home` and assigns to the object the properties that its scheme is `https`, its hostname is `mispo.es`, and its port is `443`.

To model this with a SWRL rule we can use the built-in string operator `swrlb:stringConcat(?y,?x1,?x2,. . . ,?xn)`, which is satisfied when the first parameter equals the concatenation of all other parameters. So, `swrlb:stringConcat(?uri, "https", "://", "mispo.es")` is satisfied if the URI's scheme is `https` and its hostname is `mispo.es`. This operator works as expected. To make the rule more generalised, we have to avoid hardcoding hostnames and schemes. However, if instead we bind the scheme and hostname to variables (`swrlb:stringConcat(?uri, ?scheme, "://", ?hostname)`), the atom will not be satisfied, even though there is exactly one combination of variable bindings that would satisfy it. Similar problems occur when trying to use regular expressions to bind a specific part of a string to a variable. Although we have not found any conclusive evidence, it seems very likely that this is a limitation of the reasoner that we have used (Pellet). Unfortunately, as no other reasoner handles SWRL as well as Pellet, there is no way to make these general rules work at the time of writing. Other rules that use variable binding slightly differently do work. For example, Figure 3 can infer the scheme and path of a URI. Separate rules are required for when the scheme is not explicitly present in the URI.

Given that many bits perform text parsing (often more complicated than this example), the limitations of the reasoner makes it very convoluted to implement rules that might easily be implemented in Python.

4.5 Primary keys and distributed ontologies

In the semantic web, the name of an individual is not of significant concern. Although it is used to construct an individual's IRI, it has no inherent semantic or logical meaning. An IRI is used for linking entities across ontologies. In OpenKAT, the name of an individual is the natural key (functional determinant) of its properties. The reason for this is deduplication: if the natural keys of two entries are the same, then it must be the case that they are the same individual. This reduces storage cost, prevents redundancy, and allows for easy querying by unique primary key. This implies the fundamental assumption that all individuals are uniquely named, which is known as the *unique name assumption*.

However, this can be problematic when dealing with versioned or distributed ontologies. For example, if there would be an update of the `DNSTXTRecord` model such that it modifies one of the natural keys, the primary key of related individuals would also change, even though one would probably still consider them to be fundamentally the same individual. The only way to avoid this, is by never changing the definitions of natural keys. This makes implementing future revisions and corrections tricky, as well as using an external ontology that extends or modifies the one in OpenKAT. After all, there is in the current implementation no reliable way to do consistency checking. Additionally, considering that Octopoes must have a backwards-compatible schema in order to support bi-temporality, this is something that should be addressed.

OWL does not have the unique name assumption. The reason for this is that different IRI's (for example, from different ontologies) may still

```
URI(?u, ?uri) ^ swrlb:substringBefore(?scheme, ?uri, "://")
^ swrlb:substringAfter(?hostpath, ?uri, "://")
^ swrlb:substringBefore(?host, ?hostpath, "/")
^ swrlb:substringAfter(?path, ?hostpath, "/")
-> hasScheme(?u, ?scheme) ^ hasPath(?u, ?path)
```

Figure 3: One of the SWRL rules for url classification, which fires if the URI has the scheme explicitly stated.

refer to the same real-world individual. OWL requires explicitly asserting this through `owl:sameAs` and `owl:differentFrom`. At the cost of potential redundancy, OWL enables mixing various (backwards-compatible) ontologies. Critically, mixed ontologies are then automatically checked for consistency by the reasoner, provided there are sufficient defined constraints in class definitions (OWL) as well as class shapes (SHACL).

To illustrate, a third-party ontology[7] that implements a hypothetical `hond:IPAddressV8` could make it a subclass of `kat:IPAddress` and declare it disjoint with `kat:IPAddressV4` and `kat:IPAddressV6`. A reasoner will then automatically apply all rules and axioms related to a `kat:IPAddress` to the new subclass, and will throw an error if anything in the combined ontology conflicts. If another class, `hond:IPAddressV4`, is declared to be equivalent to `kat:IPAddressV4`, all existing `kat:IPAddressV4` individuals will automatically benefit from any revisions or updates in `hond:IPAddressV4`. Indeed, the reasoner will also immediately detect whether `hond:IPAddressV4` contradicts anything in the original ontology. As we explained earlier, in a monotonic reasoning system adding additional knowledge should not be a problem.

4.6 Modelling scan levels

In OpenKAT, scan levels apply to objects, range between 0 and 4, and are either declared by a user, or inherited from a related object. They are used by a boefje to determine whether it is allowed to run on a given object. Non-invasive scans require only a low scan level, whereas invasive scans require a high scan level.

4.6.1 Scan level inheritance algorithm

Although it is not explicitly documented, it appears that the inherited scan level of an object is calculated by:

- For all incoming relations of the object:
  - Retrieve the scan level of the neighbour;
  - Retrieve the maximum allowed inheritance of that type of relationship;
  - Take the minimum of those.
- Return the highest of all.

[7] OpenHOND - Handige Objectgeoriënteerde Netwerk Dumper

We can illustrate this through an example:

- An individual `Hostname` has a declared scan level of 4;
- The `Hostname` has an outgoing relation to an individual `DNSARecord`;
- Any relation from a `Hostname` towards a `DNSARecord` maximally inherits 2.

In this scenario, the `DNSARecord` individual receives a scan level of 2.

4.6.2 Modelling declared scan levels in SWRL

If a manual declaration is given, it always overrides any inherited scan level. The naive implementation through data properties can be seen in Figure 4. However, this does not work when the OWA is taken into account. A potential workaround is setting `hasDeclaredScanLevel` explicitly to -1 for every individual by default, and enforcing that every individual has one through SHACL constraints. The new rules can be seen in Figure 5. Note that in this scenario, `hasScanLevel` is only added to individuals which have explicitly inherited or declared one. Other individuals are *not* inferred to have a scan level of 0, as their scan level is considered unknown.

4.6.3 Modelling neighbour relationships

If all relevant object properties are made a subproperty of `hasOutgoingLinkTo(OOI, OOI)`, then modelling incoming and outgoing relations becomes trivial. This can be seen in Figure 6. If `isIncomingNeighbourOf(OOI, OOI)` is declared to be the inverse of `isOutgoingNeighbourOf(OOI, OOI)`, then all OOI's correctly receive either one depending on whether they are the *subject* or *object* of the predicate. If they are both made subproperties of `isNeighbourOf(OOI, OOI)`, then that is also automatically added to individuals. This superproperty can be useful if directionality does not matter, for example if it just needs to be established that two individuals are neighbours.

```
hasDeclaredScanLevel(?ooi, ?declared)
-> hasScanLevel(?ooi, ?declared)

hasInheritedScanLevel(?ooi, ?inherited)
^ not(hasDeclaredScanLevel(?ooi))
-> hasScanLevel(?ooi, ?inherited)
```

Figure 4: Declaration and inheritance resolution using logical negation.

```
hasDeclaredScanLevel(?ooi, ?declared)
^ swrlb:lessThan(?declared, 0)
^ hasInheritedScanLevel(?ooi, ?inherited)
-> hasScanLevel(?ooi, ?inherited)

hasDeclaredScanLevel(?ooi, ?declared)
^ swrlb:greaterThanOrEqual(?declared, 0)
-> hasScanLevel(?ooi, ?declared)
```

Figure 5: Declaration and inheritance resolution taking into account the OWA.

4.6.4 Modelling `min(a, b)` in SWRL

SWRL does not have a built-in operator to calculate the minimum of two arguments[8]. However, since we have access to basic arithmetic operators, we can add our own conditional based on the formula:

$$\min(a, b) = \frac{a + b - |a - b|}{2}$$

In figure 7 it can be seen how this translates to SWRL. Unfortunately, we have not found an easy way[9] to write a shorthand for this, such as `kat:min(?result, ?arg1, ?arg2)`. This means that the above rule has to be added in full to all rules which need to calculate a minimum.

4.6.5 Modelling inheritance

Using both the `min(a, b)` rule and the inferred neighbour relationships, we can model inheritance as seen in Figure 8. In this figure, we maximally assign an inherited scan level of 2 to a `DNSARecord` from a neighbouring `Hostname` (e.g. through `hasHostname`). Note that with this implementation, an individual may be inferred to have multiple inherited scan levels if it has multiple relations. This is not a problem, however, as multiple scan levels will be separately used as input for any rules which need them.

5 Conclusion

As discussed in the previous section, we have shown that it is possible to model the selected subset of OpenKAT with semantic web technologies. This has some advantages, most notably that it results in highly interpretable code with first-class support for consistency checking and validation. However, it turns out that doing so is non-trivial. Specifically, very specialized knowledge is required to implement the business rules used in OpenKAT. This goes far beyond Python programming, for which less specialized knowledge is required and more documentation exists to implement the same functionality. Furthermore, the limited flexibility of the Pellet reasoner may make certain rules very convoluted to implement. Finally, using semantic web technologies requires a very deliberate approach on how to tackle the *open-world assumption* in order to make findings. Arguably, this is a positive property of these technologies: even though the Python-based OpenKAT system does not require a full solution to the OWA problem in order to give useful results, having to tackle the problem directly and explicitly can improve the reliability and soundness of the produced findings. We recommend the OpenKAT project to make a conscious policy decision on this conceptual issue, as this also affects the current implementation of OpenKAT.

6 Future work

A useful addition to the ontology would be to investigate the use of atomic functions for string parsing. For instance, by adding the ability to retrieve a scheme, host and port from an URL, the ontology gains enhanced capabilities for reasoning about URLs. This facilitates a more straightforward and streamlined approach to URL analysis, simplifying the implementa-

[8]http://www.daml.org/2004/04/swrl/builtins.html

[9]The Pellet reasoner supports custom built-ins, but this requires extending the Java source code.

```
hasOutgoingLinkTo(?a, ?b) -> isOutgoingNeighbourOf(?a, ?b)
```

Figure 6: Every subproperty implies an outgoing neighbouring relationship.

```
swrlb:subtract(?subtraction, LEVEL, MAX_LEVEL)
^ swrlb:abs(?abs, ?subtraction)
^ swrlb:add(?addition, LEVEL, MAX_LEVEL)
^ swrlb:subtract(?divided, ?addition, ?abs)
^ swrlb:divide(?final, ?divided, 2)
```

Figure 7: `min(LEVEL, MAX_LEVEL)` in SWRL, where `LEVEL` is the scan level of the neighbour from which is inherited, and `MAX_LEVEL` is the maximum inheritance level of a relationship.

tion process. It is likely that this will require extending an OWL reasoner.

Another direction worth exploring would be the use of Prolog or Datalog for knowledge representation and reasoning. These languages are well-suited for expressing and querying ontological relationships due to their declarative nature. Datalog especially seems promising, as it is specifically designed for database querying and reasoning tasks. In addition, it is supported by the bi-temporal database used in OpenKAT. Moreover, both of these languages have built-in reasoning mechanisms, which could be a solution to the shortcomings encountered with the technologies that we used.

Finally, it seems that there are some options to solve the aforementioned problems while still using SWT. For instance, RDFox[10] allows for the implementation of negation as failure in SWRL. Alternatively, the owlready2[11] project allows mixing Python and SWT, and supports both local and full closed-world reasoning.

References

Chang-Sheng Chen, Shian-Shyong Tseng, Chien-Liang Liu, and Chia-Hao Ou. 2002. Building a DNS Ontology using METHONTOLOGY and Protege-2000.

James Garson. 2023. Modal Logic. In *The Stanford Encyclopedia of Philosophy*, Spring 2023 edition. Metaphysics Research Lab, Stanford University.

Stephan Grimm and Boris Motik. 2005. Closed world reasoning in the semantic web through epistemic operators. In *OWLED*.

Sven Ove Hansson. 2022. Logic of Belief Revision. In *The Stanford Encyclopedia of Philosophy*, Spring 2022 edition. Metaphysics Research Lab, Stanford University.

Michael Iannacone, Shawn Bohn, Grant Nakamura, John Gerth, Kelly Huffer, Robert Bridges, Erik Ferragut, and John Goodall. 2015. Developing an Ontology for Cyber Security Knowledge Graphs. In *Proceedings of the 10th Annual Cyber and Information Security Research Conference*, CISR '15, New York, NY, USA. Association for Computing Machinery.

Robert Koons. 2022. Defeasible Reasoning. In *The Stanford Encyclopedia of Philosophy*, Summer 2022 edition. Metaphysics Research Lab, Stanford University.

Jean-Baptiste Lamy. 2017. Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies. *Artificial intelligence in medicine*, 80:11–28.

Jesse Lisser. 2023. Octopoes Architectural Report. *Internal OpenKAT development team memo*.

Kai Liu, Fei Wang, Zhaoyun Ding, Sheng Liang, Zhengfei Yu, and Yun Zhou. 2022. Recent progress of using knowledge graph for cybersecurity. *Electronics*, 11(15):2287.

Vasileios Mavroeidis and Siri Bromander. 2017. Cyber Threat Intelligence Model: An Evaluation of Taxonomies, Sharing Standards, and Ontologies within Cyber Threat Intelligence. In *2017 European Intelligence and Security Informatics Conference (EISIC)*, pages 91–98. IEEE.

Alessandro Oltramari, Lorrie Faith Cranor, Robert J Walls, and Patrick D McDaniel. 2014. Building an Ontology of Cyber Security. In *STIDS*, pages 54–61. Citeseer.

William Fernando Borja Rivadeneira and Omar S Gómez. 2020. Cybersecurity Ontologies: A Systematic Literature Review. *ReCIBE. Revista electrónica de Computación, Informática, Biomédica y Electrónica*, 9(2):1–18.

Christian Strasser and G. Aldo Antonelli. 2019. Nonmonotonic Logic. In *The Stanford Encyclopedia of Philosophy*, Summer 2019 edition. Metaphysics Research Lab, Stanford University.

Zareen Syed, Ankur Padia, Tim Finin, Lisa Mathews, and Anupam Joshi. 2016. UCO: A unified cybersecurity ontology. *UMBC Student Collection*.

[10]https://docs.oxfordsemantic.tech
[11]https://owlready2.readthedocs.io

```
Hostname(?a) ^ DNSARecord(?b)
^ hasScanLevel(?a, ?level) ^ isOutgoingNeighbourOf(?a, ?b)
^ swrlb:subtract(?subtraction, ?level, 2)
^ swrlb:abs(?abs, ?subtraction)
^ swrlb:add(?addition, ?level, 2)
^ swrlb:subtract(?divided, ?addition, ?abs)
^ swrlb:divide(?final, ?divided, 2)
-> hasInheritedScanLevel(?b, ?final)
```

Figure 8: Modelling scan level inheritance in our ontology.

A Acknowledgements

We thank Jesse Lisser for his interesting insights about the challenges he encountered while working on Octopoes, as well as his willingness to clarify the current system to us. This research has been directly inspired by his prior work on the intricacies of Octopoes, without which this would not have been possible.

B Tables

Table 1: Summary of Object Properties.

Object Property	Domain (union)	Range
fallsUnderDNSZone	Hostname	DNSZone
hasAddress	Hostname	IPAddress
hasDNSAAAARecord	Hostname	DNSAAAARecord
hasDNSRecord	Hostname	DNSRecord
hasDNSTXTRecord	DNSSPFRecord	DNTXTRecord
hasHostname	DNSRecord, DNSSPFMechanismHostname, DNSZone	Hostname
hasIPAddress	DNSAAAARecord, DNSARecord, DNSPTRRecord, DNSSPFMechanismIP	IPAddress
hasMailHostname	DNSMXRecord	Hostname
hasNameServerHostname	DNSNSRecord	Hostname
hasParent	DNSZone	DNSZone
hasRDNS	Hostname	IPAddress
hasSOAHostname	DNSSOARecord	Hostname
hasSPFRecord	DNSSPFMechanism	DNSSPFRecord
isInNetwork	Hostname, IPAddress	Network
noDNSAAAARecordFound	Hostname	Hostname
resolvesToAddress	ResolvedHostname	IPAddress

Table 2: Summary of Data Properties.

Data Property	Domain (union)	Range
AddressValue	IPAddress	xsd:string
Preference	DNSMXRecord	xsd:int
URI	URL	xsd:string
hasAll	DNSSPFRecord	xsd:string
hasDKIM	Hostname	xsd:boolean
hasExpire	DNSSOARecord	xsd:int
hasMechanism	DNSSPFMechanism	xsd:string
hasMinimum	DNSSOARecord	xsd:int
hasName	Hostname, Network	xsd:string
hasPath	HostnameHTTPURL	xsd:string
hasPort	HostnameHTTPURL	xsd:integer
hasRefresh	DNSSOARecord	xsd:int
hasRetry	DNSSOARecord	xsd:int
hasScheme	HostnameHTTPURL	xsd:string
hasSerial	DNSSOARecord	xsd:int
hasValue	DNSSPFRecord, DNSRecord	xsd:string
ttl	DNSSPFRecord, DNSRecord	xsd:int
hasDeclaredScanLevel	OOI	xsd:decimal
hasInheritedScanLevel	OOI	xsd:decimal
hasScanLevel	OOI	xsd:decimal

Table 3: Summary of SHACL Properties.

Target Class	**Properties**
DNSSOARecord	hasSOAHostname, Class: SOAHostname hasHostname, Class: Hostname
DNSAAAARecord	hasHostname, Class: Hostname hasIPAddress, Class: IPAddressV6
DNSARecord	hasHostname, Class: Hostname hasIPAddress, Class: IPAddressV4
DNSMXRecord	hasMailHostname, Class: Hostname hasHostname, Class: Hostname Preference, Datatype: xsd:int
DNSNSRecord	hasNameServerHostname, Class: Hostname hasHostname, Class: Hostname
DNSPTRRecord	hasIPAddress, Class: IPAddress hasHostname, Class: Hostname
DNTXTRecord	hasIPAddress, Class: IPAddress hasHostname, Class: Hostname
DNSSPFMechanismHostname	hasSPFRecord, Class: DNSSPFRecord hasHostname, Class: Hostname hasMechanism, Datatype: xsd:string
DNSSPFMechanismIP	hasMechanism, Datatype: xsd:string hasSPFRecord, Class: DNSSPFRecord hasIPAddress, Class: IPAddress
DNSZone	hasParent, Class: DNSZone
Hostname	isInNetwork, Class: Network
IPAddressV4	isInNetwork, Class: Network AddressValue, Datatype: xsd:string
IPAddressV6	isInNetwork, Class: Network AddressValue, Datatype: xsd:string
ResolvedHostname	resolvesToAddress, Class: IPAddress