ORNL/SPR-2017/527

Stucco System

Kelly Huffer
John Goodall
Maria Vincent
Michael Iannacone
Robert Bridges

October 3, 2017

**Approved for public release.
Distribution is unlimited.**

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

DOCUMENT AVAILABILITY

Reports produced after January 1, 1996, are generally available free via US Department of Energy (DOE) SciTech Connect.

Website www.osti.gov

Reports produced before January 1, 1996, may be purchased by members of the public from the following source:

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
Telephone 703-605-6000 (1-800-553-6847)
TDD 703-487-4639
Fax 703-605-6900
E-mail info@ntis.gov
Website http://classic.ntis.gov/

Reports are available to DOE employees, DOE contractors, Energy Technology Data Exchange representatives, and International Nuclear Information System representatives from the following source:

Office of Scientific and Technical Information
PO Box 62
Oak Ridge, TN 37831
Telephone 865-576-8401
Fax 865-576-5728
E-mail reports@osti.gov
Website http://www.osti.gov/contact.html

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ORNL/SPR-2017/527

DHS Science & Technology
Cyber Security Division

Stucco System

Kelly Huffer
John Goodall
Maria Vincent
Michael Iannacone
Robert Bridges

Date Published: June 1, 2018

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-BATTELLE, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

Stucco System

Oak Ridge National Laboratory
Robert Bridges
John Goodall
Kelly Huffer
Mike Iannacone
Maria Vincent

Project Partners
Pacific Northwest National Laboratory
Stanford University
REN-ISAC

1 Problem

Security event data, such as intrusion detection system alerts, provide a starting point for analysis, but are information impoverished. To provide context, analysts must manually gather and synthesize relevant data from myriad sources within their enterprise and external to it. Analysts search system logs, network flows, and firewall data; they search IP blacklists and reputation lists, software vulnerability information, malware and threat data, OS and application vendor blogs, and news sites. All of these sources are manually searched for data relevant to the event being investigated. Relevant results must then be brought together and synthesized to put the event in context and make decisions about its importance and impact.

2 Summary

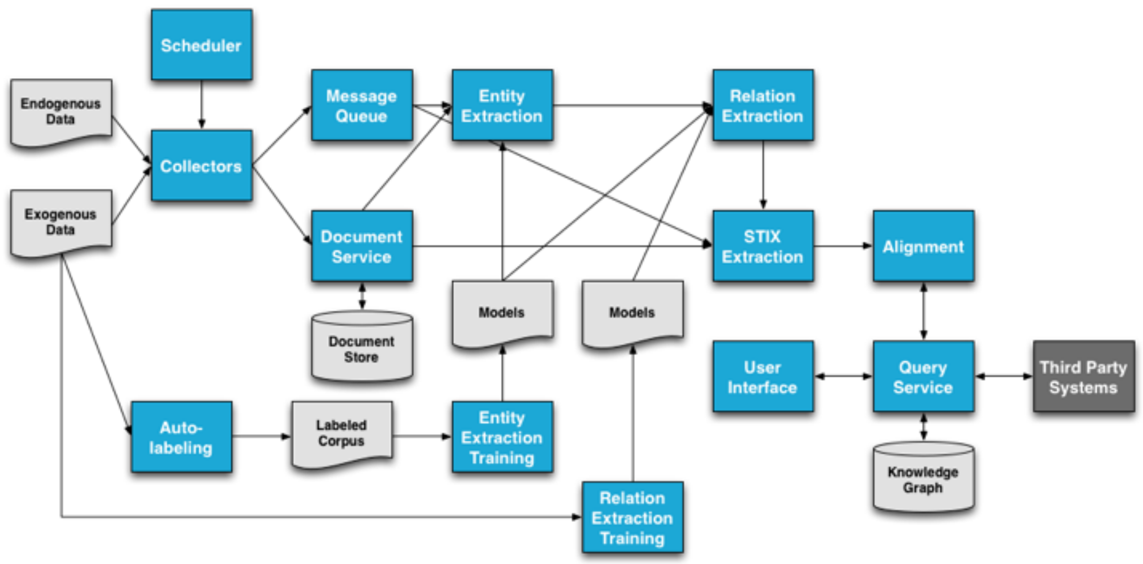
Gathering and fusing relevant context is a manual, tedious process, but the results of this process are required to know how to react to events. Stucco is a cyber intelligence platform to help automate this process and provide relevant information to analysts quickly and easily. Stucco collects data not typically integrated into security systems, extracts domain concepts and relationships, and integrates that information into a cyber security knowledge graph to accelerate decision making.

By organizing data into a knowledge graph, security analysts will be able to rapidly search for domain concepts, speeding up access to the information needed for decision-making. The information returned will only be that which is pertinent to their search. Our approach enables analysts to more quickly identify events that can be discarded as false positives and to perform more thorough analysis with the relevant context to make decisions.

Stucco is open-source software available at https://stucco.github.io/

This report has been prepared by Oak Ridge National Laboratory, Oak Ridge, TN 37831-6283 managed by UT-Battelle, LLC for the US Department of Energy under contract DE-AC05-00OR22725.

3 System Overview



3.1 Data Collection

The collectors pull data or process data streams and push the collected data (documents) into the message queue. Each type of collector is independent of others. The collectors can be implemented in any language. Collectors can either send messages with the document content or without. For messages without content, the collector will add the document to the document store and attach the returned 'id' to the message. Collectors can either be stand-alone and run on any host, or be host-based and designed to collect data specific to that host. Stand-alone collectors may require state (state should be stored with the scheduler, such as the last time a site was downloaded). Host-based collectors may need to store state (e.g. when the last collection was run).

3.1.1 Collector Types

Web collector

Web collectors pull a document via HTTP/HTTPS given a URL. Documents will be decompressed, but no other processing will occur. The documents can be various formats such as HTML, XML, CSV, etc.

Scraping collector

Scrapers pull data embedded within a web page via HTTP/HTTPS given a URL and an HTML pattern. The documents will be in HTML format.

RSS collector

RSS collectors pull an RSS/ATOM feed via HTTP/HTTPS given a URL. The documents will be in XML format.

Twitter collector
Twitter collectors pull Tweet data via HTTP from the Twitter Search REST API given a user (@username), hashtag (#keyword), or search term. The documents will be in JSON format.

Netflow collector
Netflow collectors will collect from Argus (http://www.qosient.com/argus/). The collector will listen for Argus streams using 'ra' tool and convert to XML and send the flow data to the message queue as a string.

Host-based collectors
Host-based collectors collect data from an individual host using agents. Host-based collectors should be able to collect and forward:

- System logs
- Hone (https://github.com/HoneProject/) data
- Installed packages

The documents will be in whatever format the agent uses.

3.1.2 Post-Processing

After collection has taken place the content may require additional handling. For example, the NVD source is tarred and gzipped. We specifically provide a post-processing method that will untar and unzip the file before it is sent on through the pipeline. We've added following post-processing actions on the content:

- unzip: uncompress the content by first determining the compression type based on the file extension .gz, bz2, etc.
- tar-unzip: untar the file content prior to uncompressing the content.
- removeHTML: applies the Boilerpipe (https://github.com/kohlschutter/boilerpipe) process to the content to extract the base text content by ignoring the HTML tags in a webpage. It also uses the Apache TIKA library (https://tika.apache.org/) to extract the documents' metadata. Recommended for use on all unstructured text sources.

3.1.3 Input Transport Protocol

Input transport protocol will depend on the type of collector.

3.1.4 Input Format

Input format will depend on the type of collector.

3.1.5 Output Transport Protocol

Advanced Message Queuing Protocol (AMQP) (http://www.amqp.org/), as implemented in RabbitMQ. See the concepts documentation (http://www.rabbitmq.com/tutorials/amqp-concepts.html) for information about AMQP and RabbitMQ concepts. See the protocol documentation (http://www.rabbitmq.com/amqp-0-9-1-reference.html) for more on AMQP. Examples below are in Go (http://golang.org/) using the AMPQ package (http://godoc.org/github.com/streadway/amqp). Other libraries (http://www.rabbitmq.com/devtools.html) should implement similar interfaces.

The RabbitMQ exchange uses the exchange-type of 'topic' with the exchange-name of 'stucco'. The exchange declaration options should be:

```
"topic",    // type
true,       // durable
false,      // auto-deleted
false,      // internal
false,      // noWait
nil         // arguments
```

The publish options should be:

```
stucco,     // publish to an exchange named stucco
<routingKey>, // routing to 0 or more queues
false,      // mandatory
false       // immediate
```

The '<routingKey>' format should be: 'stucco.in.<data-type>.<source-name>.<data-name (optional)>', where:
data-type (required): the type of data, either 'structured' or 'unstructured'
source-name (required): the source of the collected data, such as cve, nvd, maxmind, cpe, argus, hone.
data-name (optional): the name of the data, such as the hostname of the sensor.

The message options should be:

```
DeliveryMode:          1,    // 1=non-persistent, 2=persistent
Timestamp:             time.Now(),
ContentType:           "text/plain",
ContentEncoding:       "",
Priority:              1,    // 0-9
HasContent:            true, // boolean
Body:                  <payload>
```

'**DeliveryMode**' should be 'persistent'.

'**Timestamp**' should be automatically filled out by your AMPQ client library. If not, the publisher should specify.

'**ContentType**' should be "text/xml" or "text/csv" or "application/json" or "text/plain" (i.e. collectorType from the output format). This is dependent on the data source.

'**ContentEncoding**' may be required if things are, for example, gzipped.

'**Priority**' is optional.

'**HasContent**' is an application-specific part of the message header that defines whether or not there is content as part of the message. It should be defined in the message header field table using a boolean: “HasContent: true” (if there is data content) or “HasContent:

false" (if the document service has the content). The next extraction component will use the document service accordingly. This is the only application-specific data needed.

'**Body**' is the payload, either the document itself or the ID if 'HasContent' is false.

The corresponding binding keys for the queue (http://www.rabbitmq.com/amqp-0-9-1-quickref.html#class.queue) defined in the extraction pipeline (RT) can use wildcards to determine which extraction component should handle which messages:
* (star) can substitute for exactly one word.
# (hash) can substitute for zero or more words.
For example, 'stucco.in.#' would listen for all input.

3.1.6 Output Format

There are two types of output messages: (1) messages with data and (2) messages without data that reference an ID in the document store.

3.2 Scheduler

The scheduler is a Java application that uses the Quartz Scheduler library (http://www.quartz-scheduler.org) for running tasks. The scheduler instantiates and runs collectors at the scheduled times. The schedule is specified in a configuration file.

The collectors and scheduler are tightly coupled, so it makes sense to discuss major aspects of collection control together. Accordingly, we discuss configuration options and redundancy control here, even though most of their actual implementation is part of the collectors.

3.2.1 Configuration

The schedule is maintained in the main Stucco configuration file, stucco.yml. The scheduler can read directly from file.

3.2.2 Running

The scheduler's main class is gov.pnnl.stucco.utilities.CollectorScheduler. It recognizes the following switches:

-section

This tells the scheduler what section of the configuration to use. It is currently a required switch and should be specified as "–section demo-load".

-file

This tells the scheduler to read the collector configuration from the given YAML file, typically stucco.yml.

-url

This tells the scheduler to read the collector configuration from a URL like for an etcd service, which will typically be http://10.10.10.100:4001/v2/keys/ (the actual IP may vary depending on your setup). Alternatively, inside the VM, you can use localhost instead of the IP

3.2.3 Schedule Format

Each exogenous collector's configuration contains information about how and when to collect a source. Example from a configuration file:

```
 default:
  …
    scheduler:
      collectors:
        -
          source-name: Bugtraq
          type: PSEUDO_RSS
          data-type: unstructured
          source-URI: http://www.securityfocus.com/vulnerabilities
          content-type: text/html
          crawl-delay: 2
          entry-regex: 'href="(/bid/\d+)"'
          tab-regex: 'href="(/bid/\d+/(info|discuss|exploit|solution|references))"'
          next-page-regex: 'href="(/cgi-bin/index\.cgi\?o[^"]+)">Next &gt;<'
          cron: 0 0 23 * * ?
          now-collect: all
```

source-name
The name of the source, used primarily as a key for the document-processing pipeline (i.e. the extraction components).

type
The type key specifies the primary kind of collection for a source. Here's one way to categorize the types.

Generic Collectors - Collectors used to handle the most common cases.

- RSS: An RSS feed
- PSEUDO_RSS: A web page acting like an RSS feed, potentially with multiple pages, multiple entries per page, and multiple subpages (tabs) per entry. This uses regular expressions to scrape the URLs it needs to traverse.
- TABBED_ENTRY: A web page with multiple subpages (tabs). In typical use, this will be a delegate for one of the above collectors, and won't be scheduled directly.
- WEB: A single web page. In typical use, this will be a delegate for one of the above collectors, and won't be scheduled directly.

Site-Specific Collectors - Collectors custom-developed for a specific source.

- NVD: The National Vulnerability Database
- BUGTRAQ: The Bugtraq pseudo-RSS feed. (Deprecated) Use PSEUDO_RSS.
- SOPHOS: The Sophos RSS feed. (Deprecated) Use RSS with a tab-regex.

Disk-Based Collectors - Collectors used for test/debug, to "play back" previously-captured data.

- FILE: A file on disk
- FILEBYLINE: A file, treated as one document per line
- DIRECTORY: A directory on disk with multiple documents

source-uri
The URI for a source.

crawl-delay
The minimum number of seconds to wait between requests to a site.

***-regex**
The collectors use regular expressions (specifically Java regexes) to scrape additional links to traverse. There are currently keys for three kinds of links:

- entry-regex: In a PSEUDO_RSS feed, this regex is used to identify the individual entries.
- tab-regex: In an RSS or PSEUDO_RSS feed, this regex is used to identify the subpages (tabs) of a page.
- next-page-regex: In a PSEUDO_RSS feed, this regex is used to identify the next page of entries.

cron
When to collect is specified in the form of a Quartz Scheduler cron expression.
CAUTION: Quartz's first field is seconds, not minutes like some crons.
There are seven whitespace-delimited fields (six required, one optional):

’s m h D M d [Y]’
These are seconds, minutes, hours, day of month, month, day of week, and year.

- Use * to mean “every”
- Exactly one of the D/d fields must be specified as ? to indicate it isn’t used
- In addition, we support specifying a cron expression of now, to mean “immediately run once”.

now-collect
The now-collect configuration key is intended as an improvement on the now cron option, offering more nuanced control over scheduler start-up behavior. This key can take the following values:

- ‘all’: Collect as much as possible, skipping URLs already collected
- ‘new’: Collect as much as possible, but stop once we find a URL that's already collected
- ‘none’: Collect nothing; just let the regular schedule do it

3.2.4 Reducing Redundant Collection

Most of the scheduler consists of fairly straightforward use of Quartz. The one area that is slightly more complicated is the logic used to try to prevent, or at least reduce, redundant collection and messaging. We’re trying to avoid collecting pages that haven’t changed since the last collection. Sometimes we may not have sufficient information to avoid such redundant collection, but we can still try to detect the redundancy and avoid re-messaging the content to the rest of Stucco. Our strategy is to use built-in HTTP features to prevent redundant collection where possible, and to use internal bookkeeping to detect redundant collection when it does happen. We implement this strategy using the following tactics:

1. We use HTTP HEAD requests to see if GET requests are necessary. In some cases the HEAD request will be enough to tell that there is nothing new to collect.
2. We make both HTTP HEAD and GET requests conditional, using HTTP's If-Modified-Since and If-None-Match request headers. If-Modified-Since checks against a timestamp. If-None-Match checks against a previously returned response header called an ETag (entity tag). An ETag is essentially an ID of some sort, often a checksum.
3. We record a SHA-1 checksum on collected content, so we check it for a match the next time. This is necessary because not all sites run the conditional checks. For a feed, the checksum is performed on the set of feed URLs.

These checks are performed by the collectors and the internal bookkeeping is kept in the CollectorMetadata.db file. Each entry is a whitespace-delimited line containing URL, last collection time, SHA-1 checksum, and UUID.

3.2.5 State

The scheduler runs the schedule as expected, controlling when the collectors execute. Other aspects of collection control are less complete, and need improvements in the following areas:

1. Exception Handling - Minor exceptions during collection are generally ignored. However, no attempt is made to deal with more serious exceptions. In particular, no attempt is made to ensure that the metadata recording, document storage, and message sending are performed in a transactional manner. The scheduler does have a shutdown hook so it can attempt to exit gracefully for planned shutdowns.
2. Collector Metadata Storage - This is currently implemented strictly as proof-of-principle. Metadata is stored to a flat file, requiring constant re-loading and re-writing of the entire file. We know this won't scale, and plan to migrate to an embedded database.
3. Leveraging robots.txt - The code does not currently read a site's robots.txt file. It should do so in order to determine the throttling setting, as well as know if it should avoid collection of some files. Currently, we can honor these in the configuration file by using the crawl-delay setting and by only specifying URLs that are fair game.

3.3 Document Service

This software provides a storage service for text documents and metadata over an HTTP API. The API is exposed on 'host:port/document/' with the following routes:

- Get a document:
  ```
  GET host:port/document/<id>
  ```
  Returns a JSON object of the document and meta-data, which includes the success or failure.
- Post a document:
  ```
  POST host:port/document/  will assign an id
  POST host:port/document/<id>      to specify the id
  ```
  Returns a JSON object that describes the success or failure.

- Delete a document:
  DELETE host:port/document/<id>
  Returns a JSON object that describes the success or failure.

Examples:
Below are examples using ‘curl’ (http://curl.haxx.se).

1) Upload a json file:

```
curl -XPOST localhost:8000/document/12345\?extractor\=test\&title\=test
--data "{key1: 'some data', key2: 'more data'}" -i -H "Content-Type:
application/json"

HTTP/1.1 200 OK
Content-Type: application/json
Date: Fri, 21 Nov 2014 01:53:18 GMT
Content-Length: 61

{"ok":"true","key":"12345","message":"saved document by id"}
```

2) Retrieve a file:

```
curl -XGET localhost:8000/document/12345 –i

HTTP/1.1 200 OK
Content-Type: application/json
Date: Fri, 21 Nov 2014 01:54:41 GMT
Content-Length: 122

{"ok":"true","key":"12345","document":"{key1: 'some data', key2: 'more
data'}","timestamp":1416534798,"extractor":"test"}
```

3) Delete a file:

```
curl -XDELETE localhost:8000/document/12345 -i

HTTP/1.1 200 OK
Content-Type: application/json
Date: Fri, 21 Nov 2014 01:55:38 GMT
Content-Length: 57

{"ok":"true","key":"12345","message":"removed document"}
```

4) Upload an image file:

```
base64 file.png | curl -XPOST localhost:8000/document/ --data @- -i -H
"Content-Type: image/png"
```

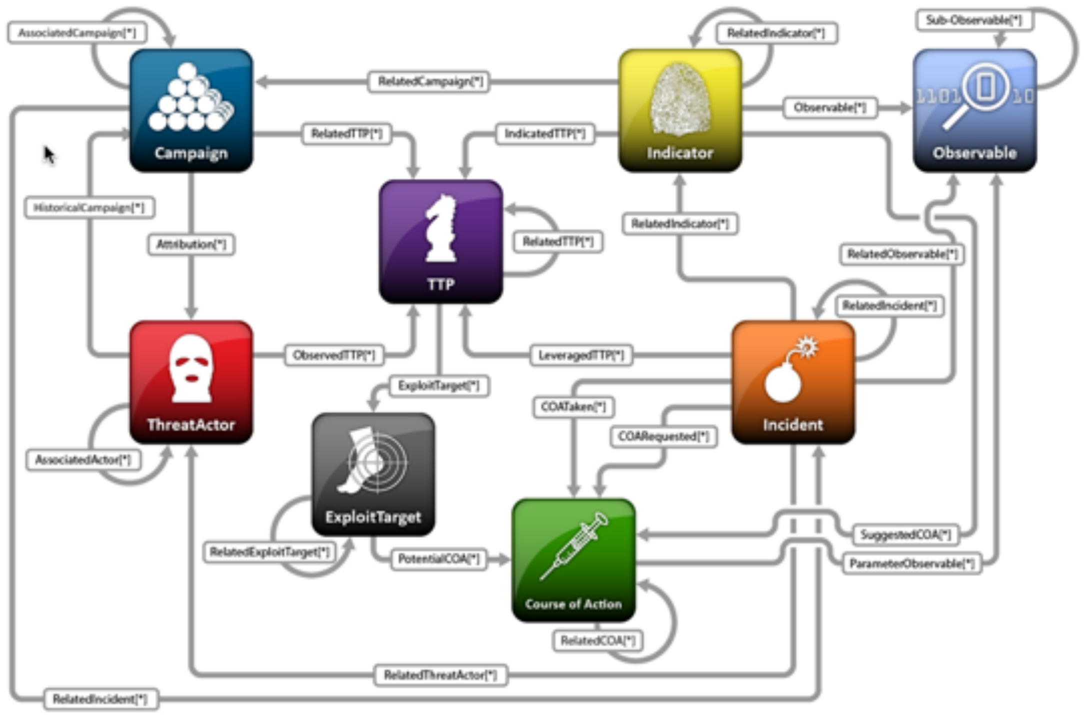
```
HTTP/1.1 100 Continue

HTTP/1.1 200 OK
Content-Type: application/json
Date: Thu, 15 Jan 2015 20:36:16 GMT
Content-Length: 86

{"ok":"true","key":"befc3e40-e3de-4666-b7b5-
155e1b0935d6","message":"saved document"}
```

5) Download an image file; this example uses ‘jq’ (http://stedolan.github.io/jq/) to extract the base64 data from the JSON object:

```
curl -XGET localhost:8000/document/1de60b72-e91b-4a26-9466-86f0d3ccdf7f
--silent | jq --raw-output .document | base64 -D > file.png
```

3.4 Data Model

The Stucco data model aligns with the Structured Threat Information eXpression (STIX) language version 1.2.0. The documentation describes STIX as “a structured language for describing cyber threat information so it can be shared, stored, and analyzed in a consistent manner”. See the STIX version 1.x webpage for more details. (https://stixproject.github.io) Below is the graph representation of STIX data.



3.5 RT

RT is the real-time processing pipeline of Stucco. The data it receives will be transformed into a subgraph, consistent with the STIX data model (previously described). Then, the subgraph will be aligned with the knowledge graph. The RT pipeline consists of two message queue consumers (structured and unstructured), an unstructured information extraction process, a structured information extraction process, an alignment component, and a connection to the graph storage system. The following subsections describe the RT pipeline components.

3.5.1 Message Queue

The message queue accepts input documents, from the collectors (publishers) and holds the documents in separate queues based on the routing key. RT consumes a message from the structured queue for structured information extraction, and consumes from the unstructured queue for unstructured extraction. The message queue uses the technology, RabbitMQ (http://www.rabbitmq.com/), which implements the Advanced Message Queueing Protocol (AMQP) standard version 0.9.1. The queue should hold messages until RT acknowledges its receipt.

3.5.2 Entity Extraction

Once the message is pulled from the unstructured queue, RT gets the document ID from within the message, and queries the document service for the document text and title. The text and title are passed to the entity extraction component. The entity extraction component identifies and labels cyber-domain entities from unstructured text. The document's text is either contained in the message itself, or the entity extraction component requests it from the document service. This library makes use of Stanford's CoreNLP (http://nlp.stanford.edu/software/corenlp.shtml) and Apache's OpenNLP (https://opennlp.apache.org) libraries.

3.5.2.1 Entity Types

- Software
  - Vendor
  - Product
  - Version
- File
  - Name
- Function
  - Name
- Vulnerability
  - Name
  - Description
  - CVE ID
  - MS ID

3.5.2.2 Input

- Trained Apache OpenNLP averaged perceptron model file in binary format that represents a cyber-domain entity model

- Default CoreNLP models for tokenizing, part-of-speech tagging, sentence splitting, and parse-tree building
- Text content of document to be annotated with cyber labels
- Predefined heuristics, including known-entity lists (i.e. gazetteers) and regular expressions
- Mapping of known tokens (i.e. words or punctuation) to a unique label, found during training

3.5.2.3 Process

1. Use the CoreNLP library to tokenize, part-of-speech tag, and build the parse trees of the document's text.
2. Check the tokens (i.e. words and punctuation) against lists of known entities such as Google's Freebase data sets. If the token is found, label it appropriately.
3. Attempt to match a token, or set of tokens against regular expressions. If a match is found, then label the token, or set of tokens.
4. Check token against the token-to-unique-label map and label appropriately, if found.
5. If the token is still unlabeled, generate features/context for the token, and evaluate them against the maximum entropy model (MEM) to determine the label with the highest probability.

3.5.2.4 Features / Context Used

- Token (word or punctuation to be labeled)
- Prefix (first 6 characters of token)
- Suffix (last 6 characters of token)
- Part of speech tag
- Match current token against a set of regular expressions
- Match pervious token against a set of regular expressions

3.5.2.5 Output

An Annotation object that represents the document as a map, where annotator classnames are keys. The document map includes the following values:

- Text: original raw text
- Sentences: list of sentences
    - Sentence: map representing one sentence
        - Token: word within the sentence
        - POSTag: part-of-speech tag
        - CyberEntity: cyber domain label for the token
    - ParseTree: sentence structure as a tree

3.5.3 Relation Extraction

The relation extraction component creates vertices from a document annotated with cyber-entity labels, and creates edges using a set of SVMs and feature models to predict relationships between these cyber entities.

3.5.3.1 Relationship Types

* ExploitTargetRelatedObservable Edge

Exploit Target (e.g. vulnerability) --> Observable (e.g. software)

* Sub-Observable Edge

Observable (e.g. software) --> Observable (e.g. file)

* Software, File, Function, Vulnerability Vertex Properties

Software/file/function/vulnerability properties are part of the same vertex

Example Text: "... **MS15-035**, which addresses a **remote code execution** bug ..."

"MS15-035" is extracted as a vulnerability MS ID property, and "remote code execution" is extracted as a vulnerability description property. This type of relationship indicates that both properties are describing the same vulnerability object.

3.5.3.2 Input

- Output from the entity-extraction component as an Annotation object, which represents the sentences, list of words from the text, along with each word's part of speech tag and cyber-domain label.
- The string name of the document's source
- The string name of the document's title

3.5.3.3 Process

1. Pre-trained Word2Vec model
2. Pre-trained SVM models, one for each relationship and entities' order of appearance
3. Pre-generated feature maps, one for each relationship and entities' order of appearance
4. NVD XML files are used to find examples of the relationships
5. For each Annotated document:
    a. Use NVD files to find known examples of relationships in document
    b. Use Word2Vec model to encode each token of the document
    c. Use feature maps to generate feature vectors for each token of the document
    d. Use pre-trained SVM models with the document's feature vectors to predict relationships between cyber entities

Please refer to relation-bootstrap repo (https://github.com/stucco/relation-bootstrap) for more information on the research related to this process.

3.5.3.4 Output

A JSON-formatted subgraph of the vertices and edges is created, which loosely resembles the STIX data model.

```
{
      "vertices": {
            "1235": {
                  "name": "1235",
                  "vertexType": "software",
                  "product": "Windows XP",
                  "vendor": "Microsoft",
                  "source": "CNN"
            },
            ...
            "1240": {
                  "name": "file.php",
                  "vertexType": "file",
                  "source": "CNN"
            }
      },
      "edges": [
            {
                  "inVertID": "1237",
                  "outVertID": "1238",
                  "relation": "ExploitTargetRelatedObservable"
            },
            {
                  "inVertID": "1240",
                  "outVertID": "1239",
                  "relation": "Sub-Observable"
            }
      ]
}
```

3.5.4 STIX Extraction

There are two ways data can enter the STIX extraction process. The first is from RT pulling a new message from the structured queue. In this case, RT gets the document or ID from within the message, and queries the document service, if necessary. Then, the text is passed to the STIX extraction component for data ingestion. The second method of entry is from the relation extraction component. The JSON-formatted subgraph resulting from relation extraction needs to be transformed into the STIX data model before it can be aligned with the knowledge graph.

The structured data ingested into the Stucco system can be of various formats. Information extraction requires transforming the raw data into a subgraph based on the STIX data model. The subgraph can then be aligned with the knowledge graph. The following data types are currently implemented:

- Argus network flows
- Bugtraq exploit targets and remediation
- 1d4 malware
- CAIDA autonomous systems mapping
- CleanMX virus
- Common Platform Enumeration (CPE) software

- Common Vulnerabilities and Exposures (CVE) database
- DNS records
- F-Secure threat descriptions
- Maxmind GeoIP
- HTTP header requests
- Hone process and port listing
- Login events from auth.log
- Malware domain list indicators
- Metasploit exploit data
- National Vulnerability Database (NVD)
- Debian package list
- Service list
- Sophos virus alerts and indicators
- Zeus Tracker malware

3.5.5 Alignment

Alignment is the process of merging a new subgraph, generated by the extraction components, with the full knowledge graph. The alignment code receives all content from a single source as a single subgraph. For example, when Stucco loads content from a CVE source file, all the content is transformed into a JSON structure (similar to GraphSON), and passed to alignment. The alignment code “assumes" the string it receives is a JSON subgraph with a set of vertices and an array of edges. Alignment handles the merging of new content as individual vertices and edges without taking into account any topology/connectivity. There are two broad categories of alignment:

1. Merging new nodes that have unique names / IDs (e.g. CVE #):
    - If a matching name / ID is not found in the knowledge graph, add the node.
    - If a matching name / ID is found in the knowledge graph, merge properties and merge edges.
2. Merging nodes without names / IDs (e.g. malware). Some of these nodes may not have a name, others may have a name but it is not available:
    - Identify equivalent nodes and score the confidence that the two nodes refer to the same domain concept.
    - If a suitable match is found, merge properties and merge edges.
    - If a suitable match is not found, add the new node, and merge edges, if needed.

Of the two broad categories for alignment our first implementation is only of the first category. Here are the steps:

1. Using only the vertices first:
    - Each vertex’s unique name / ID is searched for within the knowledge graph (i.e., unique name / alias in Postgres).
    - If no vertex is found, then this vertex is created within the knowledge graph.
    - If a vertex is found then the properties are “merged” with the vertex in the knowledge graph. Properties that were not present are added and existing properties are appended to, overridden, or retained if they are newer than the

vertex being merged. There are two types of merge methods for Postgres: appendList and keepUpdates.

2. Once all the vertices have been added, then the edges can be added.
    - Note, vertices must be added first or the new edges won't find the vertices within the knowledge graph.
    - Using the edge definition (i.e., which vertices ID's define an edge) we look for incoming and outgoing vertices as defined in the knowledge graph.
    - If an edge's definition can't find all the vertices, an error is logged and the process moves to the next edge.
    - When the respective vertices are found the process then creates a property map for the edge and adds the edge properties to that map, finally committing that edge to the knowledge graph. If an edge already exists we are not performing alignment with it, which will create duplicate edges.

To perform alignment with Postgres we load alignment rules written in PL/pgSQL into database during initialization, and all alignment process is occurring inside of a database.

3.5.5.1 Alignment Research Avenues

There are several venues to deal with the alignment problem in other domains. In the database domain, this is called the merge/purge problem of combining different databases. The theory is similar however the underlying structures are different because we are using a graph database whereas your standard relational database is row-column oriented. Part of this task will be exploring what functional pieces can be leverage from the database community and what pieces can be leveraged from the graph community. The following list attempts to highlight recommendations and considerations for future improvements to the alignment process.

1. The alignment rule set will need to be based on the STIX data model.
2. Rule construction may want to leverage a domain-specific language (DSL) to make construction and verification of the rules easier to manage.
3. As rules are constructed are these rules maintained in a database or loaded via file
4. Manual Correction Tool
    - Ability to revert/override modifications to the knowledge graph if there are incorrect insertions
    - Ability to add content without having to go through the pipeline
    - See the provenance on a node/edge and know what entries made that contribution
5. Consider provide a holding queue for entries that have enough conflicting evidence that manual intervention is needed.
6. Log provenance information for changes/updates on edges and nodes.
7. When updates occur on either a node or edge the result is:
    - Overwrite content
    - Append content (simple merge)
    - Merge Content (identify what portions should be combined)

8. Need to determine for different nodes/edges what comparison measure should be used. What kinds of comparison measures are needed? How much of deviation results in creation of a new node/edge, updating existing, or holding?
    - For canonical names or IDs the comparison function should be an equality measure
    - For dates, we need to consider timestamps that vary with only year down to the second (i.e. general to precise). How will we deal with this broad range (unless we provide range values)?
    - For unstructured text, there are several approaches but this will depend upon the property in question.
9. Meta-Rules will need to be used to make sure that updates will be smart. For example, new sources of information may provide old content and shouldn't overwrite current content. Checking timestamps to know what content is most recent.
10. Approximate subgraph matching with graph edit distance. This will help identify which subgraphs are most likely a match. However, it won't be conclusive as additional functions need to be applied at the individual levels to determine the update/insertion action.

3.5.5.1.1 Merge Properties

When merging two nodes or edges where the new and existing values of a property differ, the updated value will be determined by some function that is specified for that property. The updated value may be (a) one of the two conflicting values, (b) a new value derived from both input values, or (c) an array-like object with both values. These functions may make use of any node properties, such as the new or existing node's confidence score, source(s), or published date(s). General process when merging nodes (properties that had "null" for either the existing or new value can be handled in the same way):

A) resolve value: for each conflicting property, identify the updated value to insert into the knowledge graph. e.g.:

```
existing_node["conflicting_property"] =
resolve_property_with_strategy(conflicting_property, existing_node, new_node)
```

B) update graph: update 'existing_node.conflicting_property' in the knowledge graph, and 'new_node' will not be added to graph.

Edges to/from 'new_node' in the subgraph will be created in the knowledge graph to 'existing_node'. This assumes all nodes from the subgraph are added to the knowledge graph before edges.

Example resolution functions:

```
  //publishedDate is an integer unix timestamp
  resolve_property_with_newest(property_name, existing_node, new_node)
  {
      if (existing_node["publishedDate"] < new_node["publishedDate"])
        return new_node["property_name"]
      else
        return existing_node["property_name"]
   }

   //confidence ("score") is a float between 0 and 1
   resolve_property_by_confidence(property_name, existing_node, new_node)
   {
      if (existing_node["score"] < new_node["score"])
        return new_node["property_name"]
      else
        return existing_node["property_name"]
    }
```

Other examples could include a weighted average by confidence scores, or functions that may be unique to a specific property, e.g. an account's ‘lastLogin’ property might always take the newest value, or a vulnerability's ‘patchAvailable’ property might never change to ‘false’ once a ‘true’ value has been seen. Merging node confidence score properties will always use the same function across all node types. Other properties may share the same functions.

3.5.5.1.2 Merge Edges

Nodes can be added or merged into the knowledge graph. The edges associated with those nodes need to be added or merged as well. If both nodes were merged or added to the knowledge graph, the strategy for merging is based on whether or not there is an existing edge. If no existing edge exists, add the edge. If an existing edge exists, merge the properties of both edges, as described above in the merge properties subsection.

3.5.5.1.3 Identify Equivalent Nodes

This process starts with a new node, with no matching ID found in the database. The database is searched for existing nodes which may be equivalent, and if a match is found, the node properties and edges are merged as above. If an equivalent node is not found, one is created, and its edges are merged or added as needed. Some node types, such as IP addresses, should always have matching IDs, and should not search for approximate matches. However other node types, such as malware, will very rarely have matching IDs even when there is a matching node present. The nodes are equivalent if they represent the same real-world entity, even if they do not have the same ID.

When searching for an equivalent node, the first step is to build a restricted set of potential matches. The purpose of this step is to reduce the number of expensive in-depth comparisons that are needed, by replacing most of them with a much quicker comparison that eliminates most nodes. To start, only nodes of the same node type should be considered (e.g. malware can only possibly match malware, etc.) Next, a "canopy" is found, which contains all potentially-matching nodes. The specifics of this depend on the comparison techniques for the field and node pairs chosen below, but as an example, assume that nodes are matched based on distance, and that the node distance depends on the weighted sum of property distances. If one pair of properties have a large enough distance, that alone could make a match impossible, then comparing the remaining fields is not needed.

For each potentially equivalent node that remains, calculate the distance for each of their properties. There are many approaches to finding these distances, and the distance metric used may vary based on the data types and the field's meaning. Choosing a suitable distance metric depends on the data type and the meaning of the field, but it also depends on the types of errors anticipated. Most of the literature focuses on human error, such as typos, misspellings, and inconsistent representation (eg. "Avenue" vs. "Ave.") In our case, we anticipate most of the errors will originate in the text extraction process, and handling these types of errors has not been studied extensively.

1. Token distance - Token distance compares two multi-word strings, breaks them into individual words, and compares the counts of words in each string. (This is sometimes described as a "bag of words.") This can be expanded to consider word frequency and misspellings in the final distance. This is best suited to reasonably long sections of text, such as a description field.
2. Character distance - Character distance, in the simplest case, is the "edit distance" or "Levenshtein distance" between two strings - the total number of insert, delete, or replace operations needed to transform one string into another. This can be expensive, but some optimizations are possible. There are numerous variations on this basic approach, such as giving different weights to the different operations, or reducing the cost of adjacent insertions, or varying the cost based on position within the string. This can also include varying the cost based on the specific substitution performed, to account for misspellings and phonetic similarity. One interesting approach is to break the strings down into "q-grams" (overlapping substrings of some fixed length) and then finding the token distance using one of the techniques from item 1.
3. Numeric Distance - The techniques for finding distance between numeric fields are generally much simpler than the above categories. In most cases, this is simply the difference between the values. However often in the literature, numeric fields are simply treated as strings, and one of the above methods are used.
4. Domain-specific distance - This involves finding a distance based on some domain specific rules. For example, if a field contained a log level (Emergency, Alert, Critical, Error, Warning, Notice, Info, Debug) then "Debug" may have a distance of 1 from "Info", and a distance of 4 from "Error".

After all property distances have been found, they should be combined to find nodes which are equivalent overall. Again, there are many techniques available to achieve this. Most or all of these techniques can be extended to add a "reject region" for nodes that are too uncertain to be automatically assigned as equivalent, but are instead added to a queue for further (generally manual) review.

1. Probabilistic approaches - There are many approaches that find the probability of a node matching based on the probability of the pairs of fields matching. This requires either learning or estimating these probabilities for each field. Some approaches add an adjustable cost factor, which is useful in cases where false positives and false negatives have different impacts on the use of the data.

2. Supervised and semi-supervised approaches - if labeled training data is available, a variety of supervised and semi-supervised machine learning techniques are available, using the list of distances and/or the node properties as the input vector. Examples include using Support Vector Machines (SVM), clustering approaches, and graph partition approaches. Note that some of these are intended to find groups of matching entries, instead of matching pairs as in our case.
3. Unsupervised approaches - These generally rely on clustering to find groups of similar nodes. In some cases, there is an additional step to review and label these clusters. In some cases, after labeling these clusters, this data is then used to "bootstrap" a different approach.
4. Active-learning approaches - These are similar to the approaches above, but they make use of the fact that most cases are either obvious matches or obvious non-matches. They find the relatively few ambiguous cases, prompt for human labeling, and then adjust their parameters as needed. These approaches seem promising, but somewhat less studied than the previous two categories.
5. Distance-based approaches - These approaches also make use of the fact that most non-matching nodes are very distant ("sparse neighborhood"), and matching nodes tend to be few and close ("compact set"). In the simplest case, this involves finding a distance from a weighted sum of the field distances, and then comparing that node distance with some threshold. However, the problem becomes finding suitable weights for each field, and finding an appropriate threshold for a match, which tends to lead back to the above approaches.
6. Rule-based approaches - These approaches are based on constructing domain-specific rules that must be satisfied for a match. These rules are often expressed in some domain-specific language. These approaches tend to be highly accurate, but they require a large amount of manual effort from a domain expert to create and troubleshoot these rules. One interesting approach uses labeled training data to create lists of potential rules, which are then reviewed and adjusted by a domain expert.

All of these approaches are adopted from record matching in conventional databases, which is a well-studied problem. Unfortunately, there is still no overall best approach for that problem, instead, it is highly dependent on the domain, on the data, and on what (if any) training data or domain expertise is available. Another consideration is that these approaches vary greatly in speed, so a suitable choice will depend on the fraction of nodes that must be matched with this process, the number of potential matches in the "canopy" for each node, and the overall rate of incoming data vs. available resources.

3.6 Graph Database

The Stucco system has had many types of graph databases, including Neo4j, Titan, OrientDB, and PostgreSQL graph databases. Initially we researched and evaluated different graph storage technologies, settling on a technology stack called TinkerPop, which is easy to work with in the short term, but will be scalable in the long term. TinkerPop provides a common API, called Blueprints, for many graph databases, including Neo4j, a lightweight graph database that is easy to install and run, and Titan, a scalable graph database that can use distributed storage to scale out. The input to Neo4j and Titan was GraphSON, a superset of JSON. GraphSON can be used with many graph databases, including those that use the Blueprints API, offering us flexibility to change out the graph database without changing how Stucco ingested data. Unfortunately, the freely available version of Neo4j limits the number of vertices that could be stored. The amount of data we planned to ingest into Stucco quickly surpassed this limit.

Titan provided for greater scaling by using horizontally scalable storage backends (e.g. HBase, and Cassandra), which could handle the amount of data in Stucco. Titan allowed us to put multiple instances of Cassandra on different machines to handle the workload. Titan can also use an Elasticsearch instance to do the data indexing. However, we discovered reliability and performance issues with Cassandra and Titan. At that time, no new development on the Titan open-source codebase had been done for over four months, leading us to believe that we would not see improvements with this technology.

So, we moved to OrientDB because it also used the TinkerPop stack. However, an obscure document revealed that indexes within OrientDB were not being utilized by its implementation of the TinkerPop query API. Luckily, OrientDB did have a Java and SQL query API that did exploit the indexes. Unlike Titan, OrientDB had no automated method to handle scaling. The current OrientDB strategy for scaling does not reconfigure itself as more machines are added. Users need to know ahead of time how much data, how many machines, and of what concept type the data will be (vulnerability, software, DNS, etc.). However, automated sharding of OrientDB was proposed for the next major version release.

Due to the lack of automated sharding, we explored database technologies that were stable, easily portable, and included thorough documentation. We decided to use PostgreSQL, a cross-platform, SQL compliant open-source database that has been around for almost twenty years. PostgreSQL includes native full-text search capabilities. The PostgreSQL technology has a bulk-loading option, which improved performance of data ingestion into Stucco. The flexibility of PostgreSQL allowed us to further improve data-ingestion performance by implementing alignment logic functions within the PostgreSQL database. However, this custom logic added a complexity to the database that made it incompatible with sharding technologies, such as Citus. We still have not implemented a sharing technique for the PostgreSQL database.

3.7 Graph Database API

The graph database API is an interface with specific implementations for each supported database. The current PostgreSQL database has an implementation that reads from, writes to, and searches the knowledge graph through SQL statements. The interface allows data storage technologies to be switched and added to suit users' needs.

3.8 Query Service

The query service provides a RESTful web service to communicate with the graph database API so that the user interface and any third-party applications can interface with the knowledge graph. The query service will provide functions that facilitate common operations (eg. get a node by ID).

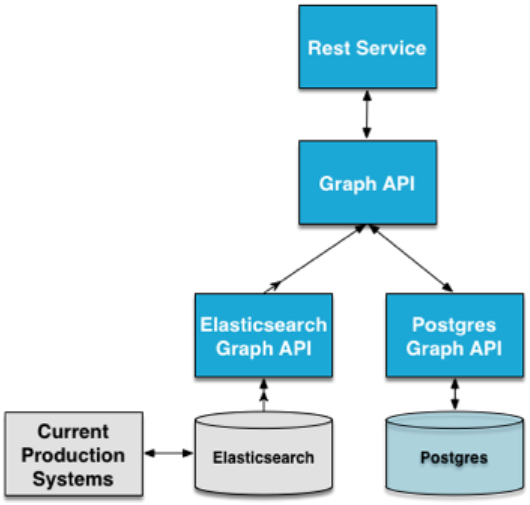
3.8.1 Routes

- `host:port/api/search`
  Returns a list of all nodes that match the search query.

- `host:port/api/vertex/vertexType=<vertType>&name=<vertName>&id=<vertID>`
  Returns the node with the specified <vertName> or <vertID>.
- `host:port/api/inEdges/vertexType=<vertType>&name=<vertName>&id=<vertID>`
  Returns the in-bound edges to the specified node.
- `host:port/api/outEdges/vertexType=<vertType>&name=<vertName>&id=<vertID>`
  Returns the out-bound edges to the specified node.
- `host:port/api/count/vertices`
  Returns a count of all nodes in the knowledge graph.
- `host:port/api/count/edges`
  Returns a count of all edges in the knowledge graph.

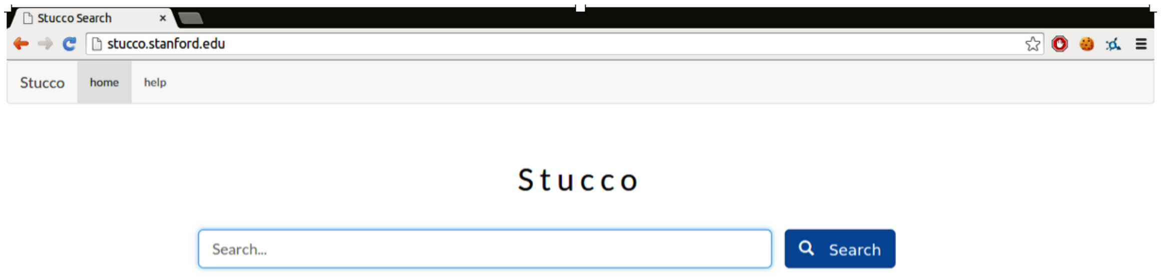
3.9 External Data Fusion

Since many enterprises already have a data store for their endogenous data (e.g. Elasticsearch, Splunk) we decided to modify Stucco to utilize the data where it currently resides, instead of ingesting it into the Stucco knowledge graph. One particular deployment site ingests flow data into Elasticsearch, so we implemented a version of the graph database API for Elasticsearch. Then, we modified the query service to request data from both the Stucco PostgreSQL and Elasticsearch databases, then merge the results together. This essentially moved a lot of the alignment work from ingest time to query time.

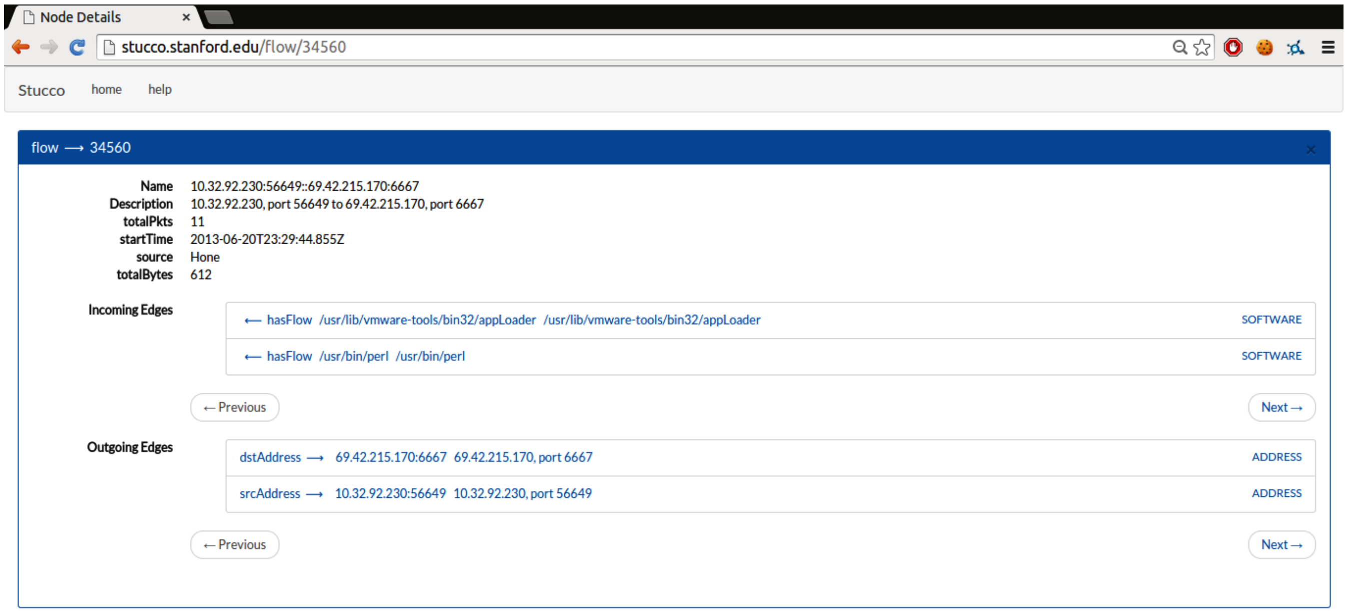


3.10 User Interface

The user interface utilizes a RESTful HTTP service to query the necessary graph APIs and return the results. The user interface is built using the React framework and the state management library, Redux. This screenshot represents the main page of the user interface, where users can search for key terms and see examples on the "Help" tab.



Below is an example of a view within the user interface. The flow data object shown here provides the user with properties of the flow, as well as information on its relationships with other data objects. This flow has two processes associated with it, namely VMware and perl.



Name	10.32.92.230:56649::69.42.215.170:6667
Description	10.32.92.230, port 56649 to 69.42.215.170, port 6667
totalPkts	11
startTime	2013-06-20T23:29:44.855Z
source	Hone
totalBytes	612

Incoming Edges

⟵ hasFlow /usr/lib/vmware-tools/bin32/appLoader /usr/lib/vmware-tools/bin32/appLoader SOFTWARE

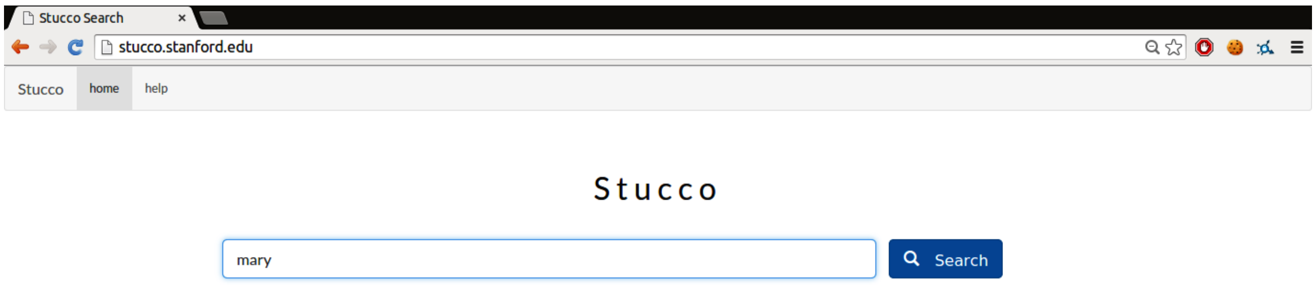
⟵ hasFlow /usr/bin/perl /usr/bin/perl SOFTWARE

Outgoing Edges

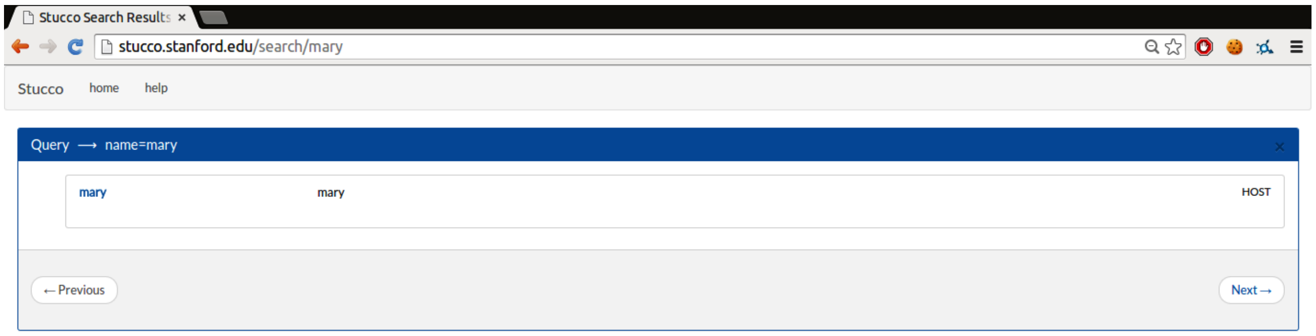
dstAddress ⟶ 69.42.215.170:6667 69.42.215.170, port 6667 ADDRESS

srcAddress ⟶ 10.32.92.230:56649 10.32.92.230, port 56649 ADDRESS

The following set of screenshots illustrates how to use Stucco to learn more about a targeted local host called "mary".



The results of a query for “mary”:

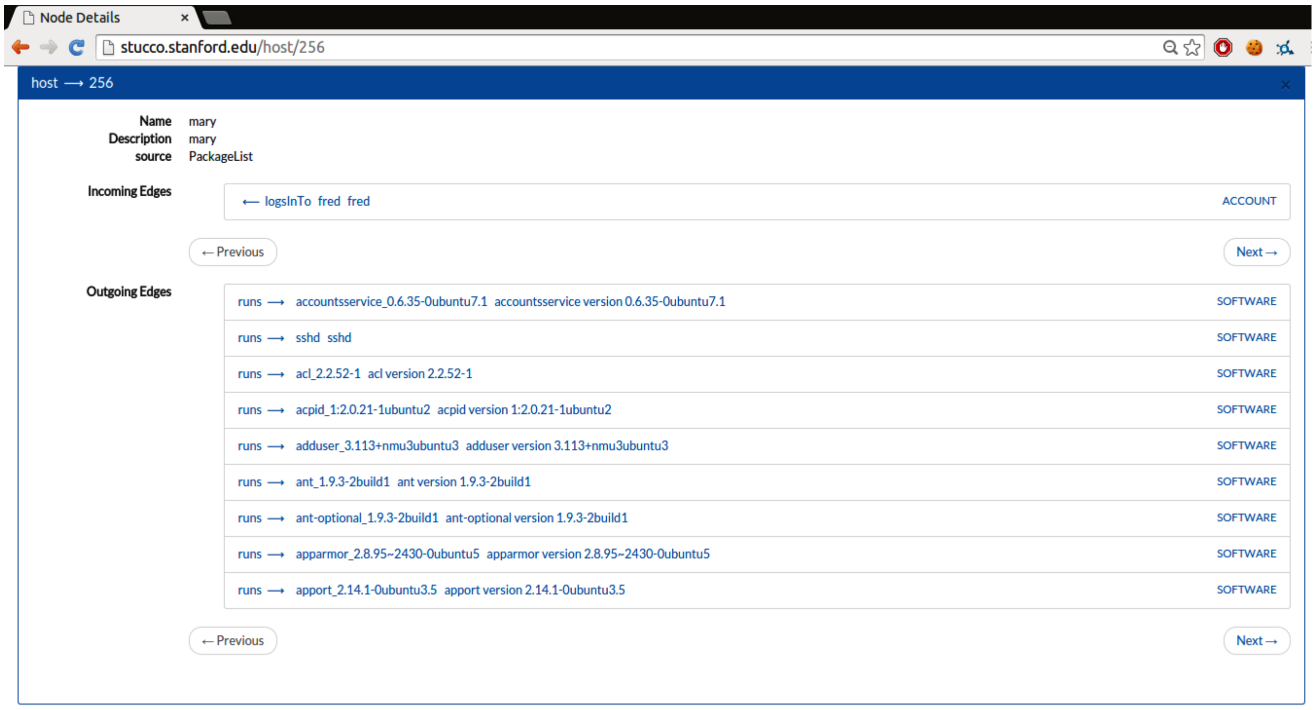


Query ⟶ name=mary

mary	mary	HOST

← Previous Next ⟶

Then, we click on the host data type named “mary” and find properties of the host as well as related information.



host ⟶ 256

Name	mary
Description	mary
source	PackageList

Incoming Edges

⟵ logsInTo fred fred	ACCOUNT

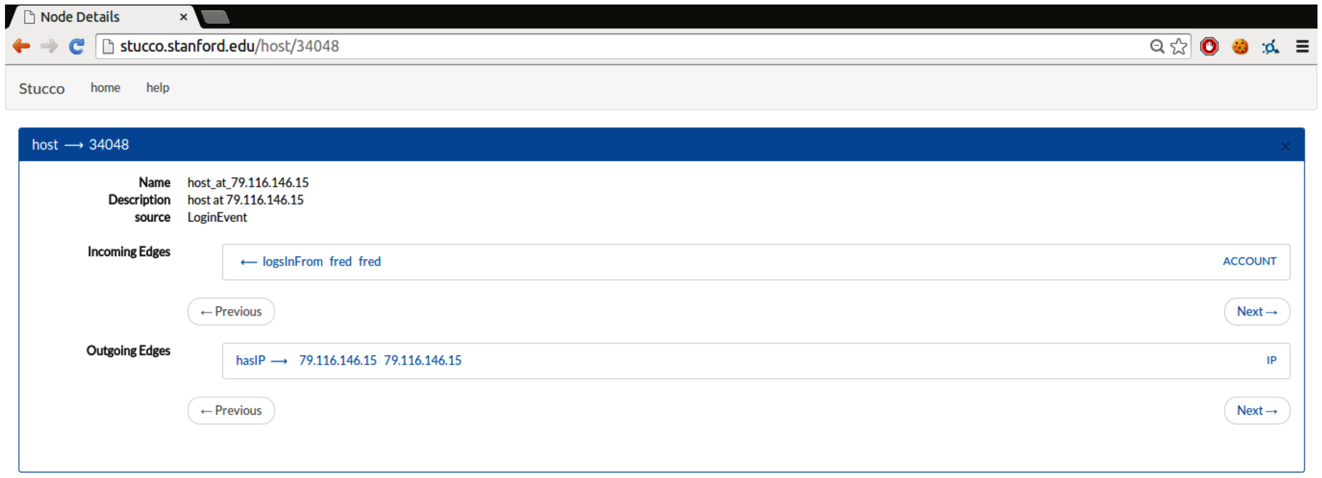
← Previous Next ⟶

Outgoing Edges

runs ⟶ accountsservice_0.6.35-0ubuntu7.1 accountsservice version 0.6.35-0ubuntu7.1	SOFTWARE
runs ⟶ sshd sshd	SOFTWARE
runs ⟶ acl_2.2.52-1 acl version 2.2.52-1	SOFTWARE
runs ⟶ acpid_1:2.0.21-1ubuntu2 acpid version 1:2.0.21-1ubuntu2	SOFTWARE
runs ⟶ adduser_3.113+nmu3ubuntu3 adduser version 3.113+nmu3ubuntu3	SOFTWARE
runs ⟶ ant_1.9.3-2build1 ant version 1.9.3-2build1	SOFTWARE
runs ⟶ ant-optional_1.9.3-2build1 ant-optional version 1.9.3-2build1	SOFTWARE
runs ⟶ apparmor_2.8.95~2430-0ubuntu5 apparmor version 2.8.95~2430-0ubuntu5	SOFTWARE
runs ⟶ apport_2.14.1-0ubuntu3.5 apport version 2.14.1-0ubuntu3.5	SOFTWARE

← Previous Next ⟶

We see that an account called "fred" logged onto the host and we want to see more information about "fred", so we click on it.



host ⟶ 34048

Name	host_at_79.116.146.15
Description	host at 79.116.146.15
source	LoginEvent

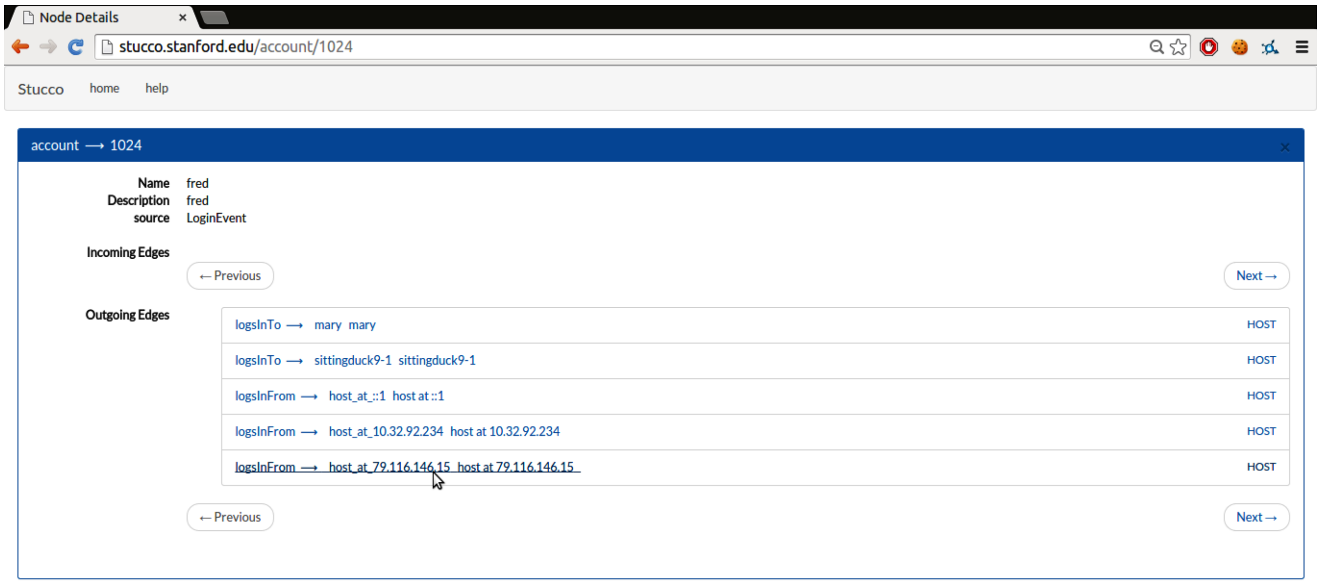
Incoming Edges

⟵ logsInFrom fred fred	ACCOUNT

Outgoing Edges

hasIP ⟶ 79.116.146.15 79.116.146.15	IP

It turns out that "fred" logged in from a host machine with an IP 79.116.146.15, by clicking that host object we might be able to find more information.



account ⟶ 1024

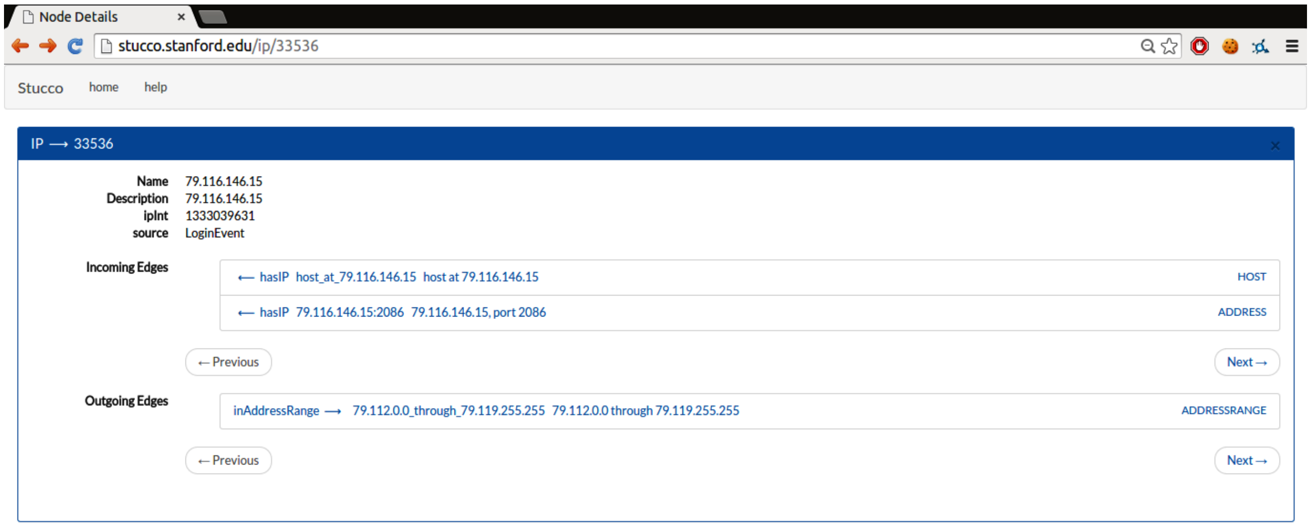
Name	fred
Description	fred
source	LoginEvent

Incoming Edges

Outgoing Edges

logsInTo ⟶ mary mary	HOST
logsInTo ⟶ sittingduck9-1 sittingduck9-1	HOST
logsInFrom ⟶ host_at_::1 host at ::1	HOST
logsInFrom ⟶ host_at_10.32.92.234 host at 10.32.92.234	HOST
logsInFrom ⟶ host_at_79.116.146.15 host at 79.116.146.15	HOST

There is not much information about the host itself, but let's explore the IP object.


Node Details

stucco.stanford.edu/ip/33536

Stucco home help

IP ⟶ 33536

Name	79.116.146.15
Description	79.116.146.15
ipInt	1333039631
source	LoginEvent

Incoming Edges

⟵ hasIP host_at_79.116.146.15 host at 79.116.146.15 HOST

⟵ hasIP 79.116.146.15:2086 79.116.146.15, port 2086 ADDRESS

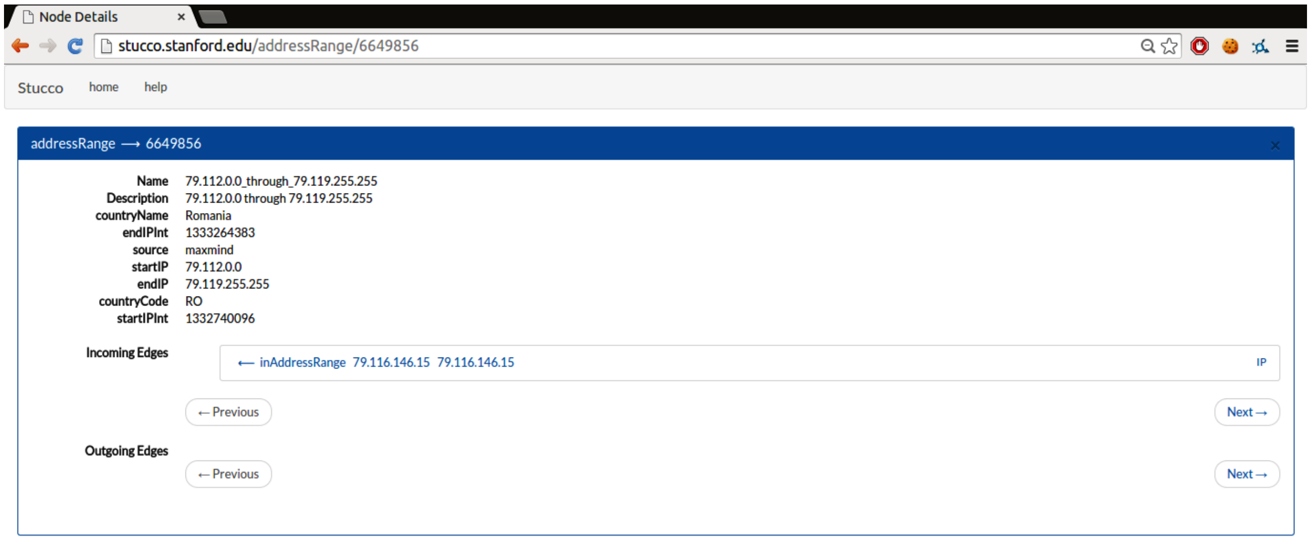
← Previous Next →

Outgoing Edges

inAddressRange ⟶ 79.112.0.0_through_79.119.255.255 79.112.0.0 through 79.119.255.255 ADDRESSRANGE

← Previous Next →

From here we can see this IP falls within a range of addresses. So, we click the address range object.


Node Details

stucco.stanford.edu/addressRange/6649856

Stucco home help

addressRange ⟶ 6649856

Name	79.112.0.0_through_79.119.255.255
Description	79.112.0.0 through 79.119.255.255
countryName	Romania
endIPInt	1333264383
source	maxmind
startIP	79.112.0.0
endIP	79.119.255.255
countryCode	RO
startIPInt	1332740096

Incoming Edges

⟵ inAddressRange 79.116.146.15 79.116.146.15 IP

← Previous Next →

Outgoing Edges

← Previous Next →

The address range object includes properties like geolocation of the IPs in that range. Based on our steps through the Stucco knowledge graph, we now know that the host "mary" was accessed by a user on a machine located in Romania. Since "fred" has never been to Romania, it appears that his credentials have been compromised.

4 Deployment

We worked with PNNL to deploy a pilot Stucco system to CPPNet for the Cyber Intelligence Center (CIC) evaluation. The data involved with CPPNet is about 34k records per second, or 3 billion records per day.

5 Impact

The initial impact of Stucco can be seen in the hours, possiblly days, worth of manual searching that a cyber analyst would need to perform to determine the impact of an incident and how to remedy the situation. One member of the project team was faced with an incident alert prior to the development of Stucco. The team member had to manually run commands on the machine in question, and search through thousands of documents on Google in order to discover contextual information about the incident. This undertaking took days to discover what the incident was, how it happened, and how to remedy the situation. If our collegue had Stucco, this search would have involved a few clicks through the user interface to discover the same information. (The use case shown in the User Interface subsection represents the majority of the search that would be performed by our team member.) Stucco can save cyber analysts days worth of work, and the organization money for this invesitgation.

The Stucco project has provided an opportunity for many college students to get involved in cyber security research. Internships are a great way to train and educate the next generation of cyber researchers and developers. Since the beginning of Stucco, we have had 12 student interns contribute to the project.

We have received many emails from organizations interested in the Stucco technology. In particular, a team of University of California Riverside and University of Pittsburg researchers were interested in the NLP research for their Hacker-Chatter project. An organization called Leidos contacted us saying they are using Stucco and found it very helpful in their research to predict cyber attacks.

The Stucco system is broken up into multiple code repositories, one for each component. The codebases are open-source on GitHub, so anyone interested in using or modifying the system component can fork their own copy of the code. The following is a list of components and the number of forks:

- Data Collectors and Scheduler - 1
- Document Service - 3
- Entity Extractor - 6
- Relation Extractor - 3
- STIX Extractors - 1
- Main Extraction Pipeline - 1
- Graph Alignment - 2
- Graph Database Connection API - 2
- Query Service - 1
- User Interface - 1
- Relation Bootstrap Research - 2
- Development Environment Setup - 2
- Demo - 1
- Auto-labeled Corpus (used by Entity Extractor)- 1
- Data Source Listing – 8

Another metric to show public interest is number of downloads from the Vagrant Cloud site (https://app.vagrantup.com/stucco), which is where we host pre-built Stucco instances as virtual machines. We have three versions of Stucco, each with multiple downloads:

- Development environment with an empty knowledge graph – 58
- Demo instance with data from testbed – 438
- Production instance that actively collects new data – 95

6 Outreach

- Conference Paper Submitted: R.A. Bridges, K.M.T. Huffer, C.L.Jones, M.D. Iannacone, J.R. Goodall, "Cybersecurity Automated Information Extraction Techniques: Drawbacks of Current Methods, and Enhanced Extractors", submitted to IEEE ICMLA 2017 on December 18-21, 2017.
- Paper: C.R. Harshaw, R.A. Bridges, M.D. Iannacone, J.R. Goodall, "GraphPrints: Towards a Graph Analytic Method for Network Anomaly Detection" to CISRC 2016, Jan 28, 2016.
- Poster: C.R. Harshaw, R.A. Bridges, M.D. Iannacone, J.R. Goodall, "Graph-Prints: A Contextual, Model-Free, Multi-Scale Network Analysis Framework for Characterizing Network Flow Data" FloCon, 2016.
- Paper: C.L. Jones, R.A. Bridges, K.M.T. Huffer, J.R. Goodall "Towards a relation extraction framework for cyber-security objects," Cyber and Information Security Research Conference, 2015. Runner-up Best Short Paper Award.
- Corresponding conference presentation of above publication
- Paper: M.D. Iannacone, S. Bohn, G. Nakamura, J. Gerth, K.M.T. Huffer, R.A. Bridges, E.M. Ferragut, J.R. Goodall "Developing an Ontology for Cyber Security Knowledge Graphs," Cyber and Information Security Research Conference, 2015.
- Presentation: J. Gerth and J. R. Goodall. "Stucco - Situation and Threat Understanding by Correlating Contextual Observations". FloCon, 2014.
- Presentation: R.A.Bridges "New Techniques for Entity Extraction of Cyber Security Concepts". CISML Seminar, March 28, 2014.
- Presentation: R.A.Bridges "Stucco - Situation and Threat Understanding by Correlating Contextual Observations". ORNL short presentation to Mitre visitors, May 06, 2014.
- Paper: R.A. Bridges, C. Jones, M. Iannacone, K.M. Testa, J.R. Goodall, "Automatic Labeling for Entity Extraction in Cyber Security", ASE Open Scientific Digital Library, May 28, 2014 Stanford, CA.
- Correpsonding conference presentation of this publication: R.A.Bridges "New Techniques for Entity Extraction of Cyber Security Concepts". ASE Conference, May 28, 2014 Stanford, CA.

- Poster: N. McNeil, R. A. Bridges, and J. R. Goodall. "Bootstrapping for Text Extraction in Cyber Security". Joint Math Meeting, 2014.
- Poster: C. L. Jones, R. A. Bridges, M. D. Iannacone, and J. R. Goodall. "Text Analysis for Timely Discovery of Cyber Security Concepts". Joint Math Meeting, 2014.
- Paper: R. A. Bridges, N. McNeil, M. D. Iannacone, B. Czejdo, N. Perez, and J. R. Goodall. "PACE: Pattern Accurate Computationally Efficient Bootstrapping for Timely Discovery of Cyber Security Concepts". International Conference on Machine Learning and Applications (ICMLA) Special Session on Machine Learning Challenges in Cyber Security Applications, 2013.
- Corresponding conference presentation of above publication
- Poster: A. Athalye, J. Goodall, M. Iannacone. “Morph: A Framework and DSL for Transforming Structured Data”. ORNL Summer Poster Session 2013.
- Poster: C. Jones, R. Bridges, M. Iannacone, J. Goodall. “Text Analysis for Timely Discovery of Cyber Security Concepts”. ORNL Summer Poster Session 2013.
- Poster: N. McNeil, R. Bridges, J. Goodall. “Bootstrapping for Text Extraction in Cyber Security”. ORNL Summer Poster Session 2013.

7 Lessons Learned

The most important lesson we learned had to do with complexity. There were too many different components associated with the Stucco system, and each component had at least one code repository. On top of that, many components depended on multiple libraries and utilities, which only added to the complexity of building the Stucco system. The intricacies of the system components also made it difficult to thoroughly test each piece separately and as a whole system.

Another issue we encountered was the performance, stability, and usability of a data storage technology. More details about this topic are discussed in the “Graph Database” subsection. We still have not solved this problem because our use case required a storage technology to handle many reads and writes as single transactions, and not as batch jobs. We also needed the storage system to be easily distributed for scalability, preferably done ad hoc as the amount of data increases.

The data model, STIX, was very difficult to work with. The focus of the STIX model did not align nicely to the data types we were ingesting. More specifically, Stucco focused on adding context to an incident or event, so we gathered data on IPs, blacklists, software, software vulnerabilities, etc. STIX seemed to center around the actor, or “bad guy”, and how the incident or campaign occurred, so our data mainly covered Observables, and loosely covered Exploit Targets, TTPs, Indicators, and Course Of Action. We tried to use existing STIX libraries to parse and ingest structured data; however, the libraries were incredibly slow and did not implement the XML format properly.

Because of the issues with the data storage system and the data format, alignment became a slow and complex process.

The NLP research into extraction of cyber information from unstructured text was promising, but a lack of labeled data made training algorithms difficult. By using semi-structured data such as NVD, we were able to train models using an unstructured description field because the structured fields provided labels for the entities. We trained models on the labeled description fields, but the writing style of NVD was very different from cyber security blogs, which were the targeted documents. It turned out that using a two-step process to label entities was the best approach. In this process, the first step was to use a gazetteer to do a lookup of known entities, and the second step used a machine learning (ML) model, if the entity was not found in the gazetteer. Our efforts moved the NLP in cyber research forward, but it was not at the point where it was useful in a real-world system just yet. We needed more training data to cover more of the STIX concepts. We believed the best approach to using NLP in a real-world system involved an offline training, or curation step so that performance at data ingest would not slow to a crawl due to machine learning processing.

8 Stucco 2.0 Approach

Based on our experiences with this work and the lessons learned, we have come up with alternative approaches to this problem. We are now more familiar with a data storage technology called Elasticsearch. We believe many of our data storage issues could be solved with this technology. Elasticsearch has the capabilities to handle batch writes, fast reads, and provides automatic sharding for scalability.

The issue of system complexity could be solved by developing a microservice based approach. This would involve many stand-alone projects that handle only one type of data, and an analytic fusion layer to pull all the data together for the user. This would improve the build process, reduce testing complexity, reduce the amount of alignment needed, and improve storage performance by limiting the amount of data.

9 Accomplishments

There have been some significant accomplishments during the course of the Stucco project. The current Stucco system can automatically align STIX-formatted documents with no configuration necessary.

We were able to improve data ingestion performance for the pilot Stucco instance in CPPNet. We optimized the process to transform raw structured data into a more easily aligned format and moved some of the alignment logic into the PostgreSQL data storage system. These optimizations increased performance from 3.55 seconds to 1.09 milliseconds per Geo-IP record and 20 seconds to 48 milliseconds for each Argus record. This is over 3000x and 400x improvement for Geo-IP and Argus, respectively.

We pushed NLP in cyber research forward, but it the implementation is not quite ready for production systems.

The Stucco system is a forerunner in the area of threat intelligence platforms. One of the earliest open-source projects was CIF (http://csirtgadgets.org/), which was developed by REN-ISAC and available in 2012. The data provided is not in STIX format and focuses on IPs, domains, and URLs related to malicious activity. Information Sharing and Analysis Centers (ISACs) began to appear around 2013, but they were exclusive to members within a particular industry and usually informal collections of data. Soltra Edge (https://www.soltra.com/en/) began around 2014 as a joint venture between FS-

ISAC and Depository Trust and Clearing Corporation (DTCC). It was bought by NC4 in 2016 and is now a commercial system that uses STIX and TAXII. ThreatStream (https://www.anomali.com/platform/threatstream) is a cybersecurity startup company backed by Cloudera executives and a significant amount of venture capital. The company started around 2014. It is a commercial product that can take STIX/TAXII data as input. Another commercial product is Cisco's Talos (https://www.talosintelligence.com/), which collects threat intelligence from its products at organizations, essentially crowdsourcing data collection. The intelligence it gathers is then fed into other Cisco products such as Snort, Sourcefire, and Threat Grid. There have been many organizations trying to develop systems that gather and/or synthesize threat intelligence. Stucco has significant overlap with these new threat intelligence platforms, but Stucco started earlier, is open-source, and is more comprehensive in scope.

10 Acknowledgments

This material is based on research sponsored by the Department of Homeland Security Science and Technology Directorate, Cyber Security Division via BAA 11-02; the Department of National Defence of Canada, Defence Research and Development Canada; the Dutch Ministry of Security and Justice; and the Department of Energy. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of: the Department of Homeland Security; the Department of Energy; the U.S. Government; the Department of National Defence of Canada, Defence Research and Development Canada; or the Dutch Ministry of Security and Justice.