Analyzing Network Traffic with Chimera

JONATHAN SPRINGER, KEVIN BORDERS, AND MATTHEW BURNSIDE

Jonathan Springer is a Managing Engineer at Reservoir Labs. Jonathan started out developing functional language implementations at the University of Illinois, where he received his PhD in computer science. After spending time working on workstation compilers at Hewlett Packard, he joined Reservoir, where he has led numerous projects developing compilers, static analysis tools, and language runtimes. springer@reservoir.com

Kevin Borders is a Computer Security Researcher and Senior Software Engineer. His research interests include large-scale stream processing and automated behavioral analysis. Kevin completed his PhD at the University of Michigan in May 2009, where his thesis topic was protecting confidential information from malicious software kevin.borders@gmail.com

Matt Burnside is a Researcher at the National Security Agency. burnside3@llnl.gov

The increasing frequency and complexity of network-based attacks is generating a correspondingly high level of interest in intrusion detection systems (IDS), which detect and filter these attacks. A variety of languages such as Snort and Bro have been developed to program an IDS to recognize specific threats, but these languages cater to specialists. We are developing a new IDS language, Chimera, that is more accessible to analysts and system administrators due to its adoption of the familiar SQL syntax.

Intrusion detection systems (IDS), operating, for example, at the switch level or as a transparent "bump on the wire," must cope with an ever-changing landscape of threats, which requires that they be very flexible. This flexibility is realized by programmability: a programming language serves to customize the IDS to look for traffic of interest. As a result, the power of an IDS is constrained by the choice of language as well as by its physical capabilities such as throughput rate.

A general-purpose programming language is not ideally suited to the task of telling IDS systems which traffic to report or filter. Core elements of the problem domain such as operating over a stream of traffic and deconstructing packets lend themselves to special syntax that gives the language user a lot of leverage. In selecting a syntactic model, the IDS programming language needs to balance a number of factors that sometimes trade off against each other. Some of these are

- Expressivity: how well properties of interest can be described in the language;
- Efficiency: how well the description can be realized with the IDS's capabilities;
- Accessibility: how easily the user can make use of the language's power.

Although the first two factors are commonly considered, less thought is often given to the third. Snort [7], for example, aims to be lightweight. Its rules are easy to write and efficient to check but are limited in their capabilities. Individual packet properties can be examined, but correlating packets to investigate properties at the level of the protocol is difficult.

Bro [6] chooses to be more expressive, able to recognize protocol-level structure and to recognize richer patterns in the traffic stream. One cost of this expressivity, however, is the additional demands on the user. Writing a Bro script is more akin to a traditional programming task (albeit aided by domain-specific support), and this can limit its audience. Furthermore, performance of these scripts is dependent on subtle implementation design decisions where small changes to a script can dramatically affect performance of the whole IDS.

Although there is overlap, the audience of programmers is fundamentally different from the audience of network operators and analysts. We would like to make the power of a language like Bro more accessible to this latter pool of people, who have the domain expertise to know what they are looking for but want better tools to express their desires. This audience needs a better programming idiom. We have selected SQL as this idiom, and created the language Chimera [2] to make use of it. We have implemented Chimera with a compiler that translates to Bro, allowing us to take advantage of Bro's expressive power and mature infrastructure.

The Chimera Language

Chimera's use of SQL structure allows it to express complex, stateful queries about data streams in a straightforward manner. We choose SQL because it is familiar to many users and uses a high-level, word-based syntax to describe the structure of data and how it is to be manipulated. Its application to processing network traffic is not exact, though, and requires some adaptation.

SQL operates over tables, in which the rows are records and the columns are fields in those records. In the network analog, packets are rows. The structure of a packet, at the IP and TCP levels as well as the application protocol level, decodes into columns. In Chimera, we do not speak in terms of packets, however, but in terms of tuples, an abstraction from relational algebra that facilitates application to generalized data flows, which may or may not be packets. Tuples are simply typed, multipart records. Unlike SQL, where table data is uniform, Chimera provides native support for variable-length records via list and map types (useful, for example, for SMTP mail headers).

The notion of a flow of data is itself a departure from SQL. Whereas we are used to thinking of an SQL query as operating over a table of fixed size, Chimera operates over streams of indeterminate length. Some SQL operations are naturally defined in terms of the cross product of all rows, an operation that doesn't make sense for a stream. Since we do not have infinite memory, we must design our operations to account for the fact that we can remember a limited number of tuples.

Beyond the principal differences just described, the SQL model and lower-level features such as its expression language fit our problem domain well. We illustrate this and introduce the language details through examples below.

Basic Queries

While there are several top-level commands in SQL, including those to create and update a data set, almost the only one of interest to Chimera is the query operation, introduced by the SELECT construct.

```
SELECT ⟨exp⟩ AS ⟨name⟩ [, AS ⟨name⟩]* [modifiers]
```

A variety of the familiar SQL clauses may be used in a SELECT query, and we survey those in the next section.

As a first taste of Chimera, consider the program in Listing 1.

```
SELECT
  $.get('packets').first().get('srcip') AS srcip,
  $.get('headers').first().get('User-Agent') AS agent
FROM http
```

Listing 1: A basic Chimera query

This informational query consists of one SELECT with a FROM clause to indicate what data stream to process. There are a number of protocol parsers built into Chimera; these cover HTML, SMTP, DNS, and other common protocols. All that is needed to access the parsed stream of objects is to refer to the correct predefined stream. Each is a stream of tuples, all of which conform to a record structure with a specific set of named fields.

The main body of the SELECT—the lines beginning with `$.get` in this example—are a comma-separated list of data items that are returned as the result of the select query. These data items can optionally be named with an AS clause, with these names used in other clauses attached to the SELECT (though none exist in this case).

Each of the two data items is constructed by code drawing from Chimera's rich expression syntax. In this case, the code performs a sequence of operations, evaluated from left to right. The stream produced by the protocol parser is accessed by referring to the special token $. The first function call, the method `get('packets')` operates on the stream to obtain the raw list of packets. The result of this operation is a list, from which we pick off the first item via a second function call `first()`. The object we obtain is a Map, which maps names to values as in a tuple from the stream. We pick out the source IP address with another "get" call, `get('srcip')`. The second data item is constructed in just the same way, except by referring to the list of HTTP headers provided by the protocol parser and picking out the "User-Agent" header.

The "get" operation is so common that Chimera supports a shorter equivalent, [*field*]. An expression `[srcip]` will perform a `get('srcip')` function call. In addition, if the object it operates on turns out to be a list rather than a record, it applies a `first()` operation. Finally, if a method is not applied to any object (no dot operator), it is treated as implicitly referring to the stream as with $. Thus, our example can be rewritten more concisely, as in Listing 2.

```
SELECT
  [packets].[srcip] AS srcip,
  [headers].[User-Agent] AS agent
FROM http
```

Listing 2: Variant form of first Chimera query

Arranging Information

With only the SELECT construct, we cannot do much data processing beyond retrieving structured data from the network packet stream. Often we want to filter and rearrange a stream to get a more concise or pertinent result. This can be done with additional modifier clauses supported in conjunction with a SELECT.

WHERE { boolexp }

The WHERE clause can be used to filter the result according to a Boolean expression. The { *boolexp* } is evaluated for each tuple in the stream, and only those for which the result is true are retained.

GROUP BY { exp } UNTIL { boolexp }

The GROUP BY clause operates a little differently from its SQL counterpart. Because we have a stream of input data, we cannot process an entire table at once and must consider when exactly to bundle an incoming group as a unit for processing. Controlling this "window" of processing is key to keeping execution efficient and timely. The GROUP BY clause combines like tuples according to { *exp* } until { *boolexp* } becomes true, at which point it emits the group of tuples and starts another.

Listing 3 shows an example of a query that uses the additional features discussed above.

```
SELECT count_distinct([aip]), [name]
 FROM dns_rr
 WHERE [aip] != NULL
 GROUP BY [name]
  UNTIL GLOBAL
    ([packets].first().timestamp() -
     [packets].last().timestamp()) > 86400
```

Listing 3: Query to list distinct IP addresses per domain name

The goal of this query is to list the distinct IP addresses per domain. It starts with the DNS protocol stream; a special form that has been decoded into individual columns (or tuples) by Chimera is provided by the `dns_rr` token. Tuples without an IP address are dropped by the WHERE clause. The tuples are grouped by like domain names by the GROUP BY clause, and chunked to a 24-hour window (the GLOBAL keyword indicates that the boolexp refers to the global stream rather than the element being processed). When the window of the GROUP BY has expired, the name and a count of the distinct IP addresses are constructed (utilizing a call to a built-in function `count_distinct`) and returned.

Working with Multiple Streams

So far, we have the ability to do detailed inspection and manipulation of a single stream. Often, however, we want to be able to correlate information learned across streams, or perform multiple manipulations of the same stream. Chimera supports this through JOIN and CREATE VIEW syntax.

JOIN { stream } ON { exp } EQUALS { exp }

A JOIN combines two streams into one. Chimera joins are required to be equi-joins, meaning { exp } expressions may compare for equality only. There are still many different ways to perform the combination. Chimera understands the standard LEFT/RIGHT/FULL, EXCLUSIVE, and OUTER dimensions. Note that not all combinations of these modifiers are supported in the current implementation.

Additionally, Chimera makes an efficiency-related distinction relevant to streams. When matching elements from the left and right streams, storing them is necessary (Chimera uses a hash table for this purpose). By default, Chimera orders the join so that left-side tuples will only match later right-side tuples, meaning only left-side ones need to be stored. The UNORDERED keyword can be used to get the traditional, symmetric behavior (at the cost of also storing right-side tuples).

CREATE VIEW { name } AS { select }

Unlike the above constructs, CREATE VIEW is not a modifier to a SELECT, but rather a top-level construct in its own right. The purpose is simply to save the results of some query by assigning a name to it.

Now we have the tools to construct complex queries that correlate across multiple streams. Consider the problem of spam detection. One way to approach this would be to write an analytic that keeps an eye out for new mail transfer agents (MTAs), and if one is seen that transmits a large amount of mail in a small amount of time, report it. We can write a query that operates over the SMTP-parsed stream, looking for MTAs in the "Received" header. For 24 hours after a new one is seen, keep a count of the number of distinct recipients from that MTA. If the amount exceeds some threshold (say 50), emit a tuple reporting this.

This query is complex in that it requires not only understanding the protocol, but keeping state on the history of traffic and correlating the new MTAs with the recipient count. Listing 4 gives an implementation in Chimera.

```
CREATE VIEW mtasmtp
  AS (SELECT headers AS headers,
             [packets].timestamp() AS time0,
             [headers].find('RECEIVED').sub_regex('^.*by +', '')
                                       .sub_regex(' .*$', '') AS mta
      FROM smtp
      WHERE [headers].find('RECEIVED') == /.*by .*/ );
CREATE VIEW mtasmtp_unique
  AS (SELECT headers, mta AS mta, time0 AS time0
      FROM mtasmtp
      WHERE unique([mta]) );
SELECT
```

```
  merge([b].[headers].find('TO').split_regex(', '),
      [b].[headers].find('CC').split_regex(', '),
      [b].[headers].find('BCC').split_regex(', ')
      ).iterall{count_distinct($)}
    AS recipient_count,
  [a].[mta]
FROM
  mtasmtp_unique AS a  JOIN  mtasmtp AS b
    ON [mta] EQUALS [mta]
WHERE [b].[time0] - [a].[time0] < 86400
GROUP BY [a].[mta]
  UNTIL [recipient_count] > 50
```

Listing 4: Spam detection Chimera script

To start, we create two subsidiary queries with the CREATE VIEW construct. The first creates a stream `mtasmtp`, which is a view of `smtp` in which we have extracted the MTA from the "Received" line as well as a timestamp and the headers. The second view is created by filtering `mtasmtp` down to unique MTAs using a Chimera built-in function `unique()` on the `mta` field that we constructed in the previous CREATE VIEW. With these two views, we are ready to construct the core query via SELECT. The two views are joined, performing the key correlation between MTA and recipients mentioned above. Only tuples within the one-day window are retained. We then extract specific recipients from all relevant headers (To, Cc, and Bcc) and feed those into a total count. This is used to trigger a new group, leading to the query output.

Related Work

We are not the first to combine SQL with a streaming data model, nor even to apply this to network traffic analysis. STREAM [5] and Aurora [1] are seminal works in this area. Research into windowed querying [4] and load shedding [8] has also been done. These efforts informed the present work, and Chimera builds on them in a few ways. Chimera adds support for structured datatypes, and operations such as SPLIT mediate between structured values in the expression language and the domain of tuples manipulated by the query language. Chimera also innovates in its support for windows, offering the UNTIL trigger for aggregates and the WINDOW condition for joins. Finally, of course, Chimera provides a translation to an external framework, Bro.

Another project that aims to support network traffic analysis using an SQL query language is Gigascope [3]. Gigascope is a vertically integrated platform where the query language is tied to the implementation platform. Chimera is designed to be platform-agnostic, and we are developing implementation targets other than Bro as well as stream sources other than network traffic. Gigascope's query language also shares the limitations of the streaming SQL work noted above with respect to windows and to structured data.

Looking Forward

We have covered just the core features of Chimera, but there is more in the query language, the expression language, and the built-in library of functions and protocol parsers. Additional details are provided in our symposium paper [2]. We have also set up a site, www.chimera-query.org, which tracks the latest news and updates to the language and implementation.

Chimera is in its early stages yet. More experience is needed at the language level in order to assess it from a practical usability standpoint. There is no substitute for people writing queries to determine what works well and what weaknesses need to be addressed. On the implementation side, while we have a preliminary compiler to Bro, there are still missing features and much more testing needs to be done.

Our goal is to release the implementation under an OSI-approved license. We believe that this software will be especially attractive to those who use or might consider Bro, as the two can coexist, allowing different interfaces to a common installation. Our hope is to foster an ecosystem around Chimera so that the power of the IDS can be utilized more readily by system administrators and analysts.

References

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik, "Aurora: A New Model and Architecture for Data Stream Management," 2003.

[2] Kevin Borders, Jonathan Springer, and Matthew Burnside, "Chimera: A Declarative Language for Streaming Network Traffic Analysis," Proceedings of the 21st USENIX Security Symposium, 2012.

[3] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk, "Gigascope: A Stream Database for Network Applications," *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data,* 2003, pp. 647-651.

[4] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker, "No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams," 2005.

[5] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma, "Query Processing, Resource Management, and Approximation in a Data Stream Management System," Technical Report 2002-41, Stanford InfoLab, 2002.

[6] Vern Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," 1999.

[7] Martin Roesch, "Snort-Lightweight Intrusion Detection for Networks," *Proceedings of LISA '99: 13th Systems Administration Conference,* USENIX, 1999.

[8] Nesime Tatbul and Stan Zdonik, "Window-Aware Load Shedding for Aggregation Queries over Data Streams," 32nd International Conference on Very Large Data Bases, 2009.

Who We Are

Since 1975, the USENIX Association has brought together the community of engineers, system administrators, scientists, and technicians working on the cutting edge of the computing world. Our mission is to:

- Foster technical excellence and innovation
- Support and disseminate research with a practical bias
- Provide a neutral forum for discussion of technical issues
- Encourage computing outreach into the community at large

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

What We Do

We offer services to the advanced computing systems community in the following ways:

- Conferences (technology sharing, community building, and educational training)
- Academic Programs & Good Works (for students, academics, and the community)
- Publications (journals, proceedings, and books)
- Membership (provides benefits and participation in shaping the industry)
- LISA (a SIG for system administrators)

Why Students Should Join

USENIX offers programs tailored especially for students, including:

- **Student Discounts:** We keep membership dues and conference registration fees at an affordable low rate for full-time students. Students can join USENIX for $50 a year and LISA, the SIG for system administrators, for $30 a year.
- **Conference Grants:** USENIX works with corporate partners to provide financial assistance to students to attend USENIX conferences, covering registration and helping with expenses.
- **Unparalleled Networking Opportunities:** Students who attend USENIX conferences and contribute to mailing lists have the chance to mingle with leaders in their field. Take the opportunity to chat with industry experts during the many conference Guru Is In sessions, the "hallway track," and evening events.
- **Publish Your Work:** A must-have for technology students wanting to stay ahead of the curve. CiteSeer ranks our proceedings among the top ten in highest impact for computer science.
  Proceedings are published for each event and are immediately available to student members. USENIX also offers a Best Student Paper Award at many events. The awards are cash prizes awarded to the best paper for which a student is the lead author at the USENIX event. Keep an eye out for our Calls for Papers (CFPs), and feel free to submit a paper!
- **Special Discounts:** USENIX offers its members discounts on everything from *Linux Journal* to No Starch Press and O'Reilly books, and more.

www.usenix.org/students