Chimera: A Declarative Language for Streaming Network Traffic Analysis

Kevin Borders
National Security Agency
krborde@tycho.nsa.gov

Jonathan Springer
Reservoir Labs
springer@reservoir.com

Matthew Burnside
National Security Agency
msburns@tycho.nsa.gov

Abstract

Intrusion detection systems play a vital role in network security. Central to these systems is the language used to express policies. Ideally, this language should be powerful, implementation-agnostic, and cross-platform. Unfortunately, today's popular intrusion detection systems fall short of this goal. Each has their own policy language in which expressing complicated logic requires implementation-specific code. Database systems have adapted SQL to handle streaming data, but have yet to achieve the efficiency and flexibility required for complex intrusion detection tasks.

In this paper, we introduce Chimera, a declarative query language for network traffic processing that bridges the gap between powerful intrusion detection systems and a simple, platform-independent SQL syntax. Chimera extends streaming SQL languages to better handle network traffic by adding structured data types, first-class functions, and dynamic window boundaries. We show how these constructs can be applied to real-world scenarios, such as side-jacking detection and DNS feature extraction. Finally, we describe the implementation and evaluation of a compiler that translates Chimera queries into low-level code for the Bro event language.

1 Introduction

Intrusion detection systems (IDSs) continue to play an essential role in network security. One critical aspect of IDS design is how users express analytic tasks. In particular, *policy* should be separate from the *mechanism* [21]. This leads to simpler policies that are easier to write and easier to share because they have fewer implementation constraints. Separation also increases interoperability, which moves us closer to the goal of having a standardized language for network traffic analysis.

Unfortunately, current IDSs only partially separate policy from mechanism. They each have their own domain-specific languages, which are incompatible with one other. Snort uses a declarative rule language for defining signatures [23], which is limited in its ability to express stateful analytics. Bro [21] offers a more powerful (Turing complete) event language, but complex operations require procedural programming and direct interaction with data structures, which is cumbersome and leads to dependency between policy and mechanism.

Database systems have been attacking this problem from the opposite end. SQL is a powerful, declarative, standard language, and recent work has extended it to support streaming queries [1, 3, 19]. While these systems are typically too slow to serve as an IDS, Gigascope [7] adopts a more limited SQL-based language and has successfully applied it to packet processing. Unfortunately, Gigascope cannot express many of the complex analytic tasks that are possible in Bro.

In this paper, we introduce Chimera, a declarative query language for network traffic processing that bridges the gap between powerful intrusion detection platforms and simple, implementation-agnostic queries. The goal is to provide an SQL-like syntax while maintaining as much expressive power as possible, and without significantly impacting performance. We achieve this goal by implementing Chimera as an independent language that is compiled down into low-level policies for other platforms. For this paper, we have written a compiler[1] that translates Chimera queries into the Bro event language [21].

Chimera is similar to streaming SQL languages, but has some additional features that make it better-suited for handling network traffic. First, it supports structured data types (lists and maps). This allows rows to more closely reflect the structure of application-layer protocols, almost all of which contain structured data. Chimera also makes dealing with these types easier by introducing a SPLIT operator to break up lists into multiple rows, as well as first-class functions that can be applied to data structures. Chimera also improves upon streaming SQL by introducing dynamic windows. Instead of enforcing strict window specifications at the table level, such as “range 60 minutes, slide 1 minute", Chimera allows window boundary computation using dynamic expressions,

[1]Visit http://www.chimera-query.org for more information about obtaining the Chimera compiler source code.

such as “UNTIL count() > 10". This makes it possible to output aggregate results as soon as they are ready, which is extremely important for intrusion prevention and active response scenarios.

We motivate the design of Chimera by examining real-world scenarios where detection requires complex state tracking that is unavailable in a simple system like Snort [23]. We look at existing work on detecting side-jacking – an attack that steals a session ID from an HTTP cookie [22] – and on finding malicious domains with the EXPOSURE system [5]. We also consider two examples of detecting DNS tunnels and identifying spam/phishing servers. After describing the Chimera syntax, we present example queries for these scenarios. When compared to a previous Bro implementation [22], the query for side-jacking demonstrates how analytics in the Chimera language are very concise. The Chimera queries for extracting features used by EXPOSURE [5] led us to identify ambiguities in the original text, highlighting the need for a standard network traffic analysis language.

In final part of the paper, we describe and evaluate the implementation of a Chimera to Bro compiler. The compiler operates in two main stages: (1) it translates queries into a relational algebra, and (2) it generates Bro event language code. We compared the compiler's output to hand-optimized code for a number of queries by running each side by side on real network traffic. In the worst-case example, compiled code was 3% slower than hand-written code due to extra copying and event handlers. We plan to add optimizations to minimize these issue in the future, but our experiments show that the compiler generates code that with almost the same performance as hand-written code even it is current form.

The rest of the paper is laid out as follows. Section 2 motivates our work with examples of stateful analytics. Section 3 describes the Chimera language. Section 4 presents Chimera queries for example scenarios. Section 5 describes the Bro compiler. Section 6 evaluates teh compiler and discusses future optimizations. Finally, section 7 covers related work and section 8 concludes.

2 Motivation: Stateful Network Analytics

As attacks continue to increase in sophistication, so must analytics that detect them. Over time it is becoming more and more difficult to characterize malicious behavior with simple Snort rules [23]. As a result, many administrators rely on systems like Bro [21] that are able to perform stateful analysis on high-level protocol fields, rather than being constrained to individual packet or flow analysis.

This section outlines a number of scenarios in which simple filtering is not enough. The rest of the paper then uses these scenarios to motivate the Chimera language and its design. Keep in mind that the analytic techniques presented in this section are not necessarily bulletproof, or even practical in all situations. The point is not to assess the quality of analytics, but to provide examples of logical constructs that we would like to express in the Chimera language.

2.1 Sidejacking

Sidejacking is a term used to describe the attack where a hacker steals a session token from an unencrypted HTTP cookie and then impersonates the legitimate user. This attack is easy to pull off in a coffee-shop environment where there is a public wireless network. Countermeasures include use of HTTPS, and are discussed in work by Riley et al. [22].

Sidejacking can also be detected by monitoring network traffic. An implementation of sidejacking detection has been written for Bro [26]. This script works in the following way:

1. Group incoming HTTP requests by session ID in cookie.
2. When a new request arrives, are the client IP and User-Agent the same?
3. If not, then report sidejacking.

As you can see, the analytic logic is straightforward, but implementation requires non-trivial maintenance of client state on a per-cookie basis.

2.2 Malicious Domains

A recent research project called EXPOSURE introduced a set of sixteen features for detecting malicious domain names [5]. Some of these features could operate on a single domain name, such as the percentage of numerical characters. Many of the features, however, require state tracking across multiple DNS packets. In this paper, we examine some of EXPOSURE's stateful features. In particular, we will focus on a subset of the DNS answer- and TTL-based features:

- Number of distinct IP addresses per domain name
- Number of domains that share the same IP address
- Average TTL value
- Number of TTL value changes

These features all require parsing the DNS protocol. They also require per-domain state tracking, and the second feature needs additional per-IP state tracking.

The authors of EXPOSURE also identify time-based features that we do not discuss here. It would be possible to adapt the change point detection (CPD) algorithms used by EXPOSURE to run in the Chimera framework. However, describing the implementation of complex algorithms in a streaming model is outside of the scope of this work, and is orthogonal to the design of Chimera.

2.3 DNS Tunnels

The DNS protocol is designed to resolve information about domain names. However, it can also be used for covert communication by storing data in the requested domain name (e.g., <*encoded data*>.hacker.com) and sending data back to the client inside of the IP address field. While this is a low-bandwidth channel, the ubiquity of the DNS protocol makes it likely to bypass firewalls even in restricted networks.

There are many ways to detect DNS tunnels, but we will discuss a particular method here because it highlights an interesting analytic technique. In this method, the following steps are taken to find DNS tunnels:

1. Keep track of all DNS response A records, indexing by the A record IP address.
2. When a packet is seen going to an IP address, remove the corresponding DNS response record.
3. If no packet is ever sent to the A record IP (within a window), increment a counter for the client and server IP addresses from the DNS message.
4. Report tunneling for clients or servers that exceed a threshold of orphaned responses.

This analysis logic is again very straightforward. It assumes that IP address values in DNS responses from tunnels will not actually be used as IP addresses, so most of them will never see follow-up packets. Counting a threshold will eliminate false positives from command-line DNS look-ups (e.g., using the *nslookup* UNIX command) that do not have ensuing connections.

2.4 Phishing/Spam Detection

A lot of research has gone into phishing and spam detection. Some approaches look at message contents, while others look at aggregate measurements like e-mail volume and rate of sending. Here, we will consider a detector that looks for new mail transfer agents (MTAs) through which e-mail is sent to a large number of distinct recipients. The analysis happens as follows:

1. Identify SMTP messages that have a “new" MTA in their path.
2. For 24 hours after a new MTA is seen, count the number of distinct recipients in messages that traverse that MTA.
3. If the count for a new MTA exceeds a threshold, then report phishing/spam.

Though the description of this analytic is concise, implementing it requires a few complicated operations. First, there must be a data structure, such as a Bloom filter, that keeps track of whether each MTA has been seen before. That structure must have at least two windows so that it does not start emitting old values after each time it is purged. The next challenge is that the MTA path is stored in multiple headers within each SMTP message. Checking whether each MTA on the path is new either requires applying a function to each value or splitting up the SMTP message into one tuple for each MTA. When new SMTP messages arrive, checking to see if one of the MTAs is new within the past 24 hours again requires splitting the tuple prior to a join operation.

3 The Chimera Language

3.1 Query Syntax

The highest level element in the Chimera language is a query statement. Since Chimera operates passively, the only type of query allowed right now is SELECT. Chimera also includes a CREATE VIEW statement, which is effectively a macro that can be used in place of sub-queries. The syntax for a Chimera SELECT query is very similar to SQL, and can be seen in Figure 1. Many elements are shared and behave the same way, including the FROM, WHERE, and UNION. The input and output specifications are a bit different. Explicit data sources are allowed in the query, including a file (PCAP or user-defined CSV), network interface, or list of file names from standard input (the default). Similarly, output will be sent to standard output unless a file is specified. Chimera begins to differ more significantly for the GROUP BY and JOIN operations, as well as the newly introduced SPLIT, which we discuss next. It also supports an expression syntax with different data and function types, which are described in sections 3.2 and 3.4.

3.1.1 GROUP BY

The Chimera language diverges from SQL and traditional streaming database in its semantics for the GROUP BY clause. To support streaming, we have added a TABLESIZE parameter and the UNTIL keyword with an optional GLOBAL parameter and a Boolean expression. TABLESIZE specifies the maximum number of items to hold before discarding old values. (Chimera does not yet implement more intelligent QoS or load shedding like Aurora [1], but TABLESIZE effectively enables memory limits.) The UNTIL condition determines when GROUP BY will generate output. It may contain aggregate functions, such as `count` or `average`. If GLOBAL is specified, then the aggregate functions are evaluated with a single global state object, instead of separately for each key. In this case, GROUP BY will output everything in the table when the UNTIL expression becomes true. This is similar to window-based grouping in traditional streaming databases. If GLOBAL is omitted, then each item in the GROUP BY table will be evaluated and flushed independently. This allows implementation of partitioned windows, which are described by Arasu et al. [3].

```
⟨select_query⟩ ::=
    [SOURCE {STDIN | FILE ⟨fname⟩ | INTERFACE ⟨if⟩ }]
      ⟨select_body⟩
      [INTO {STDOUT | FILE ⟨fname⟩ }]
⟨create_view⟩ ::=
    CREATE VIEW ⟨alias⟩ AS ⟨select_body⟩
⟨select_body⟩ ::=
    SELECT {* | ⟨expr⟩ [AS ⟨alias⟩]
      [, ⟨expr⟩ [AS ⟨alias⟩]]* }
    FROM ⟨table_ref⟩
    [WHERE ⟨bool_expr⟩]
    [GROUP BY ⟨expr⟩ [, ⟨expr⟩]*
      UNTIL [GLOBAL] ⟨bool_expr⟩
      [TABLESIZE ⟨row_count⟩]
      [HAVING ⟨bool_expr⟩]
      [ORDER BY ⟨expr⟩ [, ⟨expr⟩]* [ASC | DESC]
        [LIMIT ⟨row_count⟩]]]
    [UNION ⟨select_body⟩]
⟨table_ref⟩ ::=
    ⟨table_instance⟩
  | ⟨table_ref⟩ [[EXCLUSIVE] {LEFT | RIGHT | FULL}
      [OUTER]] [UNORDERED] [SINGLE] JOIN
      ⟨table_instance⟩ ON ⟨expr⟩ EQUALS ⟨expr⟩
      [TABLESIZE ⟨row_count⟩]
      [WINDOW ⟨expr⟩[, ⟨expr⟩]]
  | ⟨table_ref⟩ SPLIT ⟨expr⟩ AS ⟨alias⟩, ⟨alias⟩
⟨table_instance⟩ ::=
    ⟨table_name⟩ [AS ⟨alias⟩]
  | ( ⟨select_body⟩ ) AS ⟨alias⟩
```

Figure 1: Query syntax for the Chimera language

The GROUP BY clause may also include an ORDER BY keyword that takes a sorting parameter. Because Chimera is a stream processing system, some values will inevitably be discarded. ORDER BY ensures that the highest values are kept in the GROUP BY table instead of the newest values (the default). Chimera uses a heap structure to discard rows with the lowest ORDER BY value. This allows computation of "heavy hitters" on a high-volume data stream using very little memory. LIMIT specifies how many to rows to output at the end of each window. It defaults to TABLESIZE and is only used if GLOBAL is specified.

3.1.2 JOIN

Chimera introduces a few non-standard features for joins that improve efficiency and enable new analytic semantics. The first difference is that joins are *ordered* by default. This means that the left tuple must arrive before the right tuple. This lets Chimera use only one hash table instead of two, improving efficiency. The keyword UNORDERED can be added to the JOIN clause for standard join semantics.

Because Chimera is a stream processing system, only equi-joins are supported, hence the mandatory EQ (equals) syntax. Furthermore, only one tuple is allowed per key in the join table. If a new tuple arrives on the same side with the same key, then the old one is discarded without being matched. This ensures that each new tuple will generate at most one output, keeping overhead down to $O(1)$. Support for multi-tuple joins could be added in the future, but their use could negatively affect performance.

The next feature supported by Chimera is a SINGLE JOIN, which enforces one-to-one matching between left and right tuples. Normally, a row from one side of a join is allowed to match multiple rows from the other side. When a match occurs in a SINGLE JOIN, the matching tuple is removed from the join table so that it frees up space and cannot match any other tuples. This is useful when performing an EXCLUSIVE OUTER JOIN, which is similar to a typical outer join, except that the inner part of the join is excluded, leaving only tuples that do not have a match. An EXCLUSIVE LEFT SINGLE JOIN can be used, for example, to detect ICMP ping packets that never receive a reply. Here, SINGLE effectively increases the time that can elapse before declaring a packet unmatched by removing matched packets from the table.

The maximum number of elements stored in the JOIN table can be set with TABLESIZE, just as with GROUP BY, which guarantees a limit on memory utilization. In addition to a size-based limit, JOIN also supports a conditional WINDOW clause, which allows it to selectively age off old tuples from the window. The conditional expression for the WINDOW clause is evaluated in a special context where the oldest tuple is assigned the name `old` in the root object, and the newest tuple given the name `new`. For each new tuple, it and the oldest tuple are used to evaluate the WINDOW expression. If the expression is false, then the old tuple is removed and the expression is re-evaluated against the next oldest tuple. For example, `[new].[time] - [old].[time] < 60` enforces a 60 second time window. There can be two window conditions if the join is UNORDERED, which are applied to incoming left and right tuples, respectively.

3.1.3 SPLIT

Chimera includes the SPLIT keyword to its query language to make it easier to handle structured data types. There are some cases where it makes more sense to process a list or map structure as a single object (e.g., looking up a value at an index), but others where it is better to split the list and handle each item in its own tuple (e.g., examining DNS resource records). The SPLIT keyword takes an expression that evaluates to a structured data type (list or map, discussed in section 3.2) as an argument, as well as an alias name for each individual item, and an alias for the item index (which cannot be derived if there are duplicate items). When a split occurs, Chimera creates a new tuple for each item in the object argument. These tuples have references to all of the original data, including the structured object, but also contain the individual item (map items are emitted as two-value [key, value] lists) and its index as extra values. If the SPLIT object is empty, then Chimera will emit one tuple with NULL values for both the item and the index.

3.2 Data Types

The Chimera language has several data types that it uses to represent message fields in network traffic. Chimera takes a minimalist approach to typing modeled after the types used in JSON [8]. This makes data manipulation much simpler by reducing the number of functions and operators that are required.

Chimera supports six primitive data types: `Integer`, `Float`, `String`, `Bool`, `Null`, and `IPAddress`. The first five correspond to the four primitive types in JSON, with the additional distinction between integer and floating-point numbers. The `Integer` type does not have any constraint on its size. It will be expanded as necessary if it overflows the bounds of a 32- or 64-bit integer. `Float` types are all double precision. All `String` types are binary strings, which is appropriate for network traffic analysis. The `Bool` and `Null` types are self-explanatory. The remaining data type, `IPAddress`, could have been encapsulated in a `String` or `Integer`. Its existence is not necessary, but it is frequently used in network traffic analysis so we decided to add it out of convenience.

Chimera also supports two structured types: a `List`, and a `Map`. The `List` type directly corresponds to an array in JSON. The `Map` type is similar to maps in other languages, but it also supports ordering and duplicate keys. This makes it better-suited for network protocols that contain map-like structures. The ordering of map elements in a network message may have significance. Keys can also be repeated, both for legitimate and malicious purposes. Internally, `Map` objects are implemented by hash tables when they are created by assigning to a key, and by lists or numerically-indexed tables when they are created by appending key-value pairs. Iterating through a map will yield list objects with two items: a key and a value. The objects will be in the original insertion order if the map was created by appending items.

3.3 Naming and Schemata

One core part of the Chimera query language is the set of available schemata. In general, Chimera is not tied to any specific schemata or naming system. In fact, it supports CSV file input with user-defined column names. In this mode, Chimera reads the column names from the first line of a CSV file and applies them to each row.

When dealing with network traffic instead of user-defined meta-data, it is important to have a common naming scheme that is the same across all platforms. Right now, the only platform supported by Chimera is Bro. We could have just used the Bro names exactly, but they contain some implementation artifacts. Instead, we opted to create our own protocol schemata and write a Bro translation function for each one. This way, the naming and structure is more closely tied to actual protocol messages than to implementation choices specific to Bro.

Table 1: The schema for HTTP requests in Chimera

Name	Type
packets	List(tcp_packet)
method	String
path	String
version	String
headers	Map(String→String)
body	String

We will not enumerate the schema of every protocol here due to space constraints, but provide an example of the schema for HTTP requests in table 1. This schema is simple and corresponds directly to the protocol structure. In addition, there is a list of `packets` in the schema. All top-level protocols in Chimera have this field, which refers to the original packets that make up the message. This allows you to retrieve original IP addresses, port numbers, etc. It also allows more flexible handling of time because each individual packet's arrival time is exposed in the schema.

Another important aspect of the HTTP request schema is that it does not expose any anomalies or low-level parsing details. For example, we assume that the parser strips out any chunked-encoding headers from the `body` field. Once these are gone, we do not know whether the body was split up into many one-byte chunks, one hundred-byte chunks, or any other chunk sizes. If there were anomalies in a chunk header, such as only having a newline character instead of a carriage return and a newline, then the parser will either fail altogether or discard the information and continue silently. The problem becomes even more serious for DNS messages where hiding data in slack space is a well-known technique. This is a systemic problem that affects all protocol parsers and is orthogonal to the design of Chimera. The problem could be addressed by adding more fields to the parser that contain raw bytes. If these fields were added to low-level parsers, it would be easy to extend the Chimera naming scheme to include them.

3.4 Functions

In Chimera, functions are essential building blocks used for data manipulation and extraction. Chimera supports four different types of functions, which are described in this section. Functions can be defined by the user in the target language (Bro in this case). The set of available functions and their definitions are considered outside of the core Chimera language, with the exception of functions for which there are syntactic shortcuts. Examples of other specific functions are given later in section 4, which provides example Chimera queries for analysis scenarios.

3.4.1 Methods

The first function type that Chimera supports is a method. Methods operate on objects and can be chained together using a dot syntax (Example: `<object>.a().b().c()`). Each method function can operate on one or more types of input data, and can generate multiple output types. If any function in a method chain generates a NULL output, then evaluation stops and later functions in the chain are not called.

Within an expression in a Chimera query statement, methods may be called without an explicit base object. In this case, Chimera uses the implicit default object, which is a `Map` representing a tuple in the current schema. Chimera also supports a square-bracket syntax: `[<field>]`. This is syntactic sugar for calling the get function `get('<field>')`, which will retrieve the first value in the map that has a key matching the given input string. If the get function or bracket syntax is used on a `List` object, then Chimera assumes that the list consists of `Map` objects and will add an implicit iterator over the list, returning the first object that is not NULL. Such "apply" functions are discussed more later in this section.

In the Chimera language, arguments to method functions must be literals and cannot be derived from the default tuple object. Functions that need to manipulate multiple elements in the default tuple must be written as static functions (described in the next section) instead of method functions. This was a choice that we made based on readability and it does not effect expressiveness.

3.4.2 Static Functions and Operators

Chimera supports static functions that can operate on multiple objects (Example: `concat(<string1>, <string2>, ...)`). The arguments to static functions can be literals or chains of method functions. Chimera also has a number of basic operators. These operators are essentially syntactic sugar for static function calls, though they may be compiled down to the same operator in the target language if it exists and has the same semantics. Chimera currently support most of the C operators, including:

- Arithmetic: $+$, $-$ (subtraction and unary), $*$, $/$, % (modulo)
- Comparison: $==$, $!=$, $<$, $>$, $<=$, $>=$
- Logical: ! (NOT), && (AND), || (OR)
- Bitwise: $\sim$ (NOT), & (AND), | (OR), ^(XOR), $<<$ (Left Shift), $>>$ (Right Shift)

For arithmetic and comparison operators between integers and floats, integers are promoted to floats. Bitwise operations are only allowed on integers, and left shifting an integer will never truncate bits that are set. Instead, it will be expanded so that it can hold the value. For strings and IP addresses, only the comparison operators are supported. For Boolean values, only the equality, inequality, and logical operators are supported.

3.4.3 Aggregate Functions

The next type of function available in Chimera is an aggregate function. Aggregate functions are used to compute some result over multiple data items. Aggregate functions are typically seen in expressions that are part of the SELECT, HAVING, or UNTIL clauses in a statement that uses GROUP BY. In these places, a different aggregate value will be computed for each unique GROUP BY key (each key has a different state). Aggregate functions may also be used in WHERE clauses or statements without GROUP BY, but they will have a single global state in these cases.

The syntax for an aggregate function is exactly the same as for static functions. However, the definition must specify four routines, which are shown in Table 2. These routines are similar to those for defining an aggregate function in a standard relational database.

Table 2: User-defined aggregate function routines

	Arguments	**Returns**	**When Called**
Initialization	None	State	Before the first input
Iteration	State, Inputs	State	For each new input
Evaluation	State	Outputs	To read current output
Termination	State	State	At end of each window

Each of the routines in an aggregate function deals with a state object. This state object is returned from calls to aggregate routines (except evaluation, which does not update the state), stored, and then passed back to the next aggregate routine call. This state object is opaque to the rest of the system and can contain anything.

The termination function works a bit differently than in a traditional database due to the streaming nature of Chimera. This function will be called at the end of each window as specified in an UNTIL clause in an aggregate statement. The state object that it returns will be passed back to the next iteration call for the first item in the next window. This allows aggregate functions to maintain state across multiple windows.

Some traditional databases also support a *Merge* routine for user-defined aggregates. This allows intermediate results to be merged together, which allows parallel computation. Chimera does not yet support merging, but could be extended to do so in the future.

3.4.4 Apply Functions

The final type of function available in Chimera is an `apply` function. An apply function is a method on a structured object that takes another function as a first-class object and applies it to items in the structured object. How the argument is applied depends on the particular function. Apply functions can take normal arguments in parentheses, but use a curly bracket syntax for their function argument (e.g., `[list].apply(<args>){<fnarg>}`) to clearly differentiate them from other function types. Arguments can be passed to inner functions using the symbols `$`, `$2`, `$3`, etc. ("1" omitted from first argument for brevity). This lets user-defined apply functions pass an arbitrary number of arguments. It also allows the inner functions to be methods, static functions, or aggregate functions (e.g., `[list].apply{$.strlen()}` or `[list].apply{count($)}`). This syntax is slightly different from other languages like Javascript, but we felt it to be more concise and easy to read in this context.

Chimera does support multiple levels of apply functions. When there are multiple levels, however, inner functions *cannot* directly reference parent arguments. First-class functions in Chimera are not full closures.

Apply functions can be defined by the user, but a few examples are provided here to illustrate the concept. Note that when iterating over a map instead of a list, each key-value pair is represented as two-item [key, value] list.

- **foreach** – Apply the function to each item in the list and update it with the output value. Example: `[list].foreach{$.substr(3)}`
- **foridx**(*index*) – Apply the function to the item in the list at the given index and update it with the output value. Example: `[map].foreach{$.foridx(0){$.substr(3)}}`
- **iter** – Iteratively apply the function to each item in the list and return the first value that is not NULL. Example: `[list].iter{$.match('as.*df')}`
- **iterall** – Apply the function to all items in the list and return the last output value. Example: `[list].iterall{count($)}`
- **filter** – Apply the function to each item in the list and discard items for which it evaluates to false or NULL. Example: `[map].filter{$.first().strlen() > 3}`
- **find** – Apply the function to each item in the list and return the first item for which it does not evaluate to false or NULL. Example: `[map].find{$.first() == 'A'}`

4 Implementing Analytics in Chimera

In section 2, we introduced several attack scenarios that require advanced analysis capabilities to detect. Now that we have presented the Chimera language, we show here how it can be used to implement analytics for these scenarios. While these scenarios demonstrate many of Chimera's features and capabilities, they are by no means a complete exposition of its power. The goal here is to provide examples of how the language can be used in practice that serve as a starting point for future work.

4.1 Sidejacking

As you may recall, sidejacking involves searching for multiple clients that are using the same session identifier for a web service. For simplicity, clients can be represented as an IP address and User-Agent pair. Now that we have an understanding of Chimera's query model, we can break down the analysis task into some key facts:

- This query requires aggregation using the session ID as the GROUP BY key.
- The session ID is inside of a key-value list in the "Cookie" header and will need to be broken out of the list.
- Detection requires counting more than one distinct client. This will be the UNTIL trigger condition.

This leads us to the following query, which cleanly implements sidejacking detection and is much more shorter than the previous Bro implementation [26] (though the Bro implementation contains a few more additional features not included here):

```
SELECT
  list_agg(distinct(concat(
   [packets].[srcip], ':',
    [headers].[User-Agent])))
    AS clientlist
  [headers].[Cookie].split(';').
    foreach{$.split('=')}.
    find{$.first() == 'SID'}.last()
    AS sessionid
FROM http
WHERE [sessionid] != NULL
GROUP BY [sessionid]
UNTIL [clientlist].size() > 1
```

The first expression in the SELECT statement extracts the source IP address from the first packet in the connection (HTTP messages are comprised of one or more packets), finds the value of the "User-Agent" header (or NULL if it is missing), and concatenates the two together to form a client identifier string. Because [packets] is a list of map objects, the bracket operator that follows includes an implicit iteration, thus extracting [srcip] from the first packet in the list. The query then passes this string to the aggregate function `distinct`, which will check each incoming value to see if it has occurred before. If not, it will pass through the value, otherwise it will output NULL. The `distinct` function can be implemented with a Bloom filter, or with a hash table if more accuracy is desired. Our implementation of `distinct` in the Bro language currently uses a hash table. Finally, the `list_agg` aggregate function will take each non-NULL input item and append it to a list.

The next expression in the SELECT statement pulls out the session ID from the “Cookie" header. If there is more than one “Cookie" header, then the implicit call to `get()` made by the square brackets will just grab the first one. The expression then splits the the cookie header value up into a list of strings separated by the ’;’ character. Next, it iterates over this list with `foreach`, further splitting each string using the ’=’ character into a key-value list. Finally, the `find` function extracts the first pair in the list where its first item is the string ’SID’, and `last` pulls out the corresponding value. If at any point during this chain of functions there is a NULL value, then processing will stop and the result will be NULL.

The remainder of the query is pretty straightforward. The WHERE clause filters out only HTTP messages that have Cookie headers and session IDs. GROUP BY aggregates based on the session ID, and UNTIL will trigger an output whenever it sees more than one client using the same session ID.

4.2 Malicious Domains

There were several DNS features presented earlier in section 2.2. These features each perform some aggregate computation on DNS responses. In the Chimera language, lists of objects can be split into one tuple for each item using the SPLIT command. DNS responses contain lists of answer records in a single DNS packet, which can be split up into individual answer records. However, Chimera also includes a schema for individual resource records (essentially pre-split) that corresponds to resource record events in the Bro language. The queries below use the DNS resource record schema out of convenience, but could use the DNS schema and SPLIT as well. Here are queries for each of the listed features:

4.2.1 Number of distinct IP addresses per domain

```
SELECT count_distinct([aip]), [name]
FROM dns_rr
WHERE [aip] != NULL
GROUP BY [name]
UNTIL GLOBAL
  nextwindow([packets].[time], 86400)
```

One thing to note about this query is the `count_distinct` function. Counting the number of distinct items can be done more efficiently than by keeping a list and computing its size. This query also uses an aggregate function `nextwindow` to compute when the packet timestamp has transitioned into the next 86400-second (one day) time window. It essentially performs integer division and change detection. When this occurs, the entire table will be flushed and the computation will restart.

4.2.2 Number of domains that share the same IP

```
SELECT [name], [ip], [count]
FROM (
  SELECT
    [aip] AS ip
    list_agg(distinct([name])) AS names
    count_distinct([name]) AS count
  FROM dns_rr
  WHERE [aip] != NULL
  GROUP BY [aip]
  UNTIL GLOBAL
    nextwindow([packets].[time], 86400)
) SPLIT names AS name, nameidx
```

This query will keep a list of domains for each IP address, maintain a count of its size, and then output each domain along with an IP address and count every day. As you may have noticed, this query does not precisely quantify the "number of domains that share the same IP address" because a domain name can have multiple IPs, and the original EXPOSURE paper was not clear about whether all the domains on all the IPs should be counted [5]. This query will actually output multiple counts for each domain name, one for each IP address that it uses. This is an example where having a common query language would make explicit analytic descriptions much easier, allowing researchers to more precisely describe their techniques.

4.2.3 Average TTL value

```
SELECT avg([ttl]), [name]
FROM dns_rr
WHERE [ttl] != NULL
GROUP BY [name]
UNTIL GLOBAL
  nextwindow([packets].[time], 86400)
```

This query is very similar to the first, except that it employs the avg (average) aggregate function instead of `count_distinct`. Another point of ambiguity in EXPOSURE is whether the TTL values should be counted for all types of resource records (as is done above), or just for A records.

4.2.4 Number of TTL value changes

```
SELECT count(), [name]
FROM (
  SELECT [name]
  FROM dns_rr
  WHERE [ttl] != NULL
  GROUP BY [name]
  UNTIL
    last([ttl]) != last([ttl], 2, true) &&
    last([ttl], 2, true) != NULL
)
GROUP BY [name]
UNTIL GLOBAL
  nextwindow([packets].[time], 86400)
```

This query uses a nested statement with two instances of the aggregate function `last`. In its first form, `last` just outputs the current tuple value. The second call to `last([ttl], 2, true)` actually outputs the second-to-last value (2 parameter) and persists across windows (`true` parameter). For the sequence {A, B, A}, the UNTIL statement will become true and flush the result after B arrives. Because the second call to `last` persists, it will hold on to the B value and output another change when the next A arrives. This is an example of aggregate functions that maintain state across windows.

4.3 DNS Tunnels

The DNS tunnel detection algorithm described in section 2.3 works by identifying responses that never have follow-up connections. Here are some key facts about this analytic:

- DNS responses may contain several A records, but only the first one will be likely to receive a connection. It is thus better to use the whole-message DNS schema rather the individual resource record schema.
- We only want to count responses that do *not* have matching packets, so we need to use an EXCLUSIVE LEFT SINGLE JOIN.
- Because individual false positives may occur, we should apply a per-client threshold to unmatched responses, which will require a GROUP BY using the client as the key.

Here is a query that implements DNS tunnel detection:

```
SELECT
  [dns].[packets].[dstip] AS client,
  last([dns].[packets].[time]) AS start,
  first([dns].[packets].[time]) AS end
FROM dns EXCLUSIVE LEFT SINGLE JOIN ip_packet
  ON [answers].[aip] EQUALS [dstip]
  WINDOW [new].[packets].[time] -
         [old].[packets].[time] < 300
WHERE [dns].[answers].[aip] != NULL
GROUP BY [client]
UNTIL count() > 100
HAVING [end] - [start] < 3600
```

This query counts the number of DNS answers with an A-record IP address that have no matching packets within a five-minute time window. It then groups those unmatched responses by their destination IP (the client who made the request) and applies a threshold of 100 responses. Note that the threshold is applied in an UNTIL clause. This makes it so that detection happens immediately when the threshold is reached, instead of having to wait for the end of a time window. The timestamps of the first and last responses can then be checked in the HAVING clause to make sure they occurred within some reasonable amount of time (one hour in this case). This query demonstrates the latency benefit from using UNTIL instead of a time- or count-based window like in existing streaming databases.

4.4 Phishing/Spam Detection

Section 2.4 describes a method for detecting spam and phishing e-mails based on filtering SMTP messages with "new" mail transfer agents (MTAs) and then counting the number of recipients to which the new MTAs send e-mail in the first 24 hours. Here is a Chimera query that implements this analytic:

```
CREATE VIEW mtasmtp AS
  SELECT *
  FROM smtp SPLIT [headers].
    filter{$.first() == 'Received'}.
    foreach{$.second().regex_extract
    ('.*by ([^ ]*)')} AS mta, midx;
SELECT
  merge([b].[headers].[To].split(','),
    [b].[headers].[Cc].split(','),
    [b].[headers].[Bcc].split(',')).
    iterall{count_distinct($.strip())}
    AS recipient_count,
  [a].[mta] AS mta
FROM (
    SELECT *
    FROM mtasmtp
    WHERE unique([mta])
  ) AS a JOIN mtasmtp AS b
  ON [mta] EQUALS [mta]
WHERE [b].[packets].[time] -
      [a].[packets].[time] < 86400
GROUP BY [a].[mta]
UNTIL [recipient_count] > 50
```

This query contains a number of more complicated operations to achieve the desired result. The CREATE VIEW statement is used for the first time to set up a table of SMTP messages that are split by MTAs. The MTAs are extracted from "Received" headers in the SMTP message using a regular expression that searches for the string "by " and pulls out the following word.

In the first part of the select statement that follows, all of the destination e-mail addresses are extracted by splitting the "To", "Cc", and "Bcc" headers by commas, and then merging them into one list. The apply function `iterall` is then used to pass each recipient through the aggregate `count_distinct` function to count the number of unique recipients for each MTA.

The sub-query in the left part of the join uses a stateful function `unique` in a WHERE clause. This means that it will use one global state instead of having a different state for each aggregate key. Furthermore, the `unique` function will accumulate values indefinitely. This function is different from `distinct` in a subtle way; it is designed to only output "new" values. It will silently add items to a Bloom filter during a learning phase at start-up, and then start generating output once a certain percentage of its inputs have already been seen. As the Bloom filter fills up, `distinct` will stop adding to it and create a new one. Once the new filter becomes full, the old one will be discarded and process will continue so that there are always two Bloom filters in use. With a Bloom filter it is possible to falsely label new items as not unique a small percentage of the time. This trade-off buys reduced memory utilization. The false match rate of a Bloom filter will depend on its size and the number of insertions that are made before rolling it over.

The final part of the query joins new MTAs with future e-mails that contain those MTAs, using the WHERE clause to cut off the count after 24 hours. The UNTIL clause will trigger as soon as the unique recipient count exceeds 50 and generate a final query output.

5 Bro Compiler Implementation

For this paper, we implemented a Chimera compiler that produces policies for the Bro event language [21]. While we have only implemented one specific target, it would be possible to extended the compiler to target other languages. The work that we describe in section 5.1 on translating a declarative query to an intermediate relational algebra will be applicable for all targets. The code generation phase, which is described in section 5.2, will depend on the target language.

5.1 Translation to Relational Algebra

Because Chimera is very similar to SQL, we begin the compilation process in the same way as traditional database systems: by parsing the query and translating it into an intermediate relational algebra. We used a simple YACC parser [16] and the syntax from section 3 to convert the original query into an abstract syntax tree (AST) representation. From there, the compiler translates the AST into a data-flow representation that loosely corresponds to relational algebra, which we call the *Chimera Core*. The Chimera Core operators are shown in figure 2. This step is performed using syntax-directed translation [2], wherein syntactic elements are converted into data-flow operators as shown in figure 3. During this process, the compiler uses a symbol table to map aliases to locations in the data-flow graph, but does not need to perform full data-flow analysis because all data-flow connections are explicit in the Chimera syntax.

source(*source*)
parser(*parser*)
split($expr_{list}$, $alias_{item}$)
projection($expr_1, alias_1, ..., expr_n, alias_n$)
selection(*expr*)
rename(*newlabel*)
join($label_{left}$, $label_{right}$, $expr_{left}$, $expr_{right}$,
    $expr_{window}$, *joinkind*, *tablesize*)
group($expr_{groupby}$, $expr_{until}$, *options*, *tablesize*,
    $aggexpr_1, alias_1, ..., aggexpr_n, alias_n$)
output(dest)

Figure 2: Chimera Core language constructs

`CREATE VIEW`	→	add alias to symbol table
`SOURCE`	→	**source**
`<proto-name>`	→	**parser**
`SPLIT`	→	**split**
`<table> AS ...`	→	**rename**
`JOIN`	→	**join**
`WHERE`	→	**selection**
`GROUP BY...UNTIL...ORDER BY...LIMIT`	→	**group**
`HAVING`	→	**selection**
`SELECT`	→	**projection**
`INTO`	→	**output**

Figure 3: Summary of translation to Chimera Core

To illustrate translation from a Chimera query to the Chimera Core language, consider the following example:

```
SOURCE STDIN
SELECT avg([b].[z]) AS avgz
FROM dns AS a JOIN smtp AS b ON [x] EQ [y]
WHERE [a].[x] > 5
GROUP BY [a].[x]
UNTIL avgz > 3
INTO STDOUT
```

Figure 4 shows the data-flow graph that results from this example query. Using top-down syntax-directed translation, the first node emitted is a **source** node corresponding to SOURCE STDIN. The FROM statement is processed next. Because there is a JOIN, the compiler first translates the left and right tables, adding **parser** nodes to the **source**. The parser outputs are then fed through **rename** operators so that they can be referenced in the **join** operator, which combines them into a single data flow. Next, the data flows through a **selection**

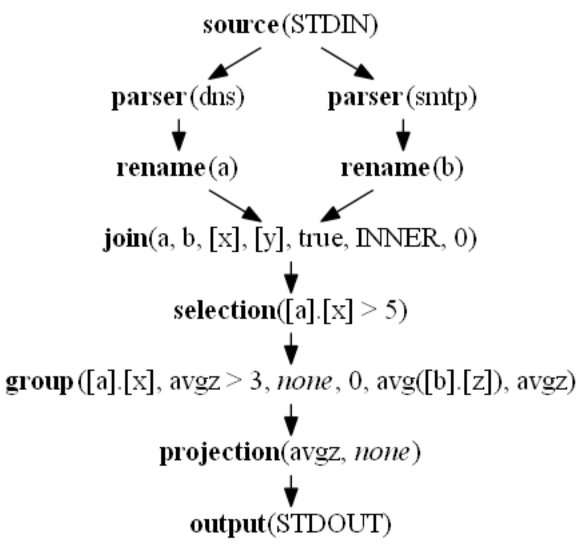

Figure 4: Chimera Core data-flow graph for example

operator that filters tuples using the WHERE expression. The tuples are then aggregated with a **group** operator, which also computes and adds aggregate expressions from HAVING and SELECT clauses to the data flow. Finally, expressions in the SELECT clause are extracted with the **projection** operator, and **output** sends data to standard output.

5.2 Code Generation

The next step in compilation is to translate the data-flow graph into Bro code. This process happens in two main stages: (1) type computation, and (2) event code generation. The event code generation step further depends on the implementation of user-defined functions, which written natively in the Bro language. Also note that data sources in Bro are specified on the command line, so the **source** operator is emitted as a shell script wrapper and not as part of the Bro language.

Type computation involves visiting each edge in the data-flow graph, determining the contents of tuples that flow through that edge, and then creating a record type for those tuples. Edges coming from operators that do not change the data – **selection** and **rename** – can be ignored during this pass. It would have been possible to use a table of the *any* type in Bro for tuples, or to create another dynamic data structure. We chose to use custom record types instead because they are better-documented and do not require modifying Bro internals.

After types have been defined for each input and output tuple, the compiler generates code for each node in the data flow graph in the form of an event handler:

- **parser** – This node adds a Bro protocol parser at the beginning of the file (if it does not yet exist) and defines an event handler that converts Bro protocol events into output tuples.
- **split** – This node takes a tuple with a list expression and outputs a new tuple for each item in the list, which also includes all the original tuple items.
- **projection** – This node outputs an event handler that executes one or more expressions on each input tuple and assigns their results to an output tuple.
- **selection** – This node evaluates an expression on each input tuple and passes that tuple as output if the expression is true.
- **rename** – This node passes tuples through unchanged, but renames the event.
- **join** – This node stores tuples in a hash table keyed on their join expression values and later matches them against tuples from the other side of the join. When there is a match (or no match for OUTER joins), this node will generate a new output tuple with one or both elements. To support the WINDOW expression, we extended the Bro table data structure to expose its oldest element.
- **group** – This node maintains a hash table keyed on the GROUP BY expression value. The table contains state objects for each aggregate function, all of which have their *Iteration* routine called for each new tuple. When the UNTIL expression becomes true, this node calls each aggregate function's *Evaluation* routine, adds the results to an output tuple, and then calls the aggregate *Termination* routines to flush the state objects.
- **output** – This special-purpose node outputs tuples in CSV format, or, if the tuple only has one packet, sends output to a PCAP file.

Some of the operator nodes take function expressions as arguments. As mentioned before, each function is written natively in Bro. A few functions, such as those that use a Bloom filter, also required some implementation in the Bro internal function (BIF) language. When a function is encountered during code generation, its definition is included in the Bro code and it is called with a standard Bro expression. Bro does not support method-style function calls using a syntax like $x(arg1).y(arg2)$, so these are re-written as $y(x(arg1), arg2)$. Apply functions are implemented by generating inline anonymous first-class function definitions, which are supported by Bro.

5.3 Example

Here we demonstrate Bro code generation with a simple example. In the interest of space, the example does not include **join** and **group** operators. Consider the following Chimera query:

```
SELECT [path]
FROM http-request
WHERE [method] == "GET"
```

This query extracts the path from all HTTP GET requests. It translates to the following data-flow graph, where each operator sends data to the next:

l0: **source**(STDIN)
l1: **parser**(http-request)
l2: **selection**([method] == "GET")
l3: **projection**([path], none)
l4: **output**()

Finally, this is compiled down to the following Bro script. Note that Bro splits up the HTTP headers and body into multiple events. To have everything available in one tuple, we also add an event handler for `http_all_headers` that saves the headers in the session table, which is omitted here to save space.

```
@http-reply
type http_request_type: record {
  method: string;
  path: string;
  headers: listmap;
  body: string;
  packetlist: packetlist_type;
};
type l3_type: record {
  v1: string;
};
event l3(t: l3_type) {
  print t$v1;
}
event l2(t: http_request_type) {
  local out: l3_type;
  out$v1 = t$path;
  event l3(t);
}
event l1(t: http_request_type) {
  if (!(t$method == "GET")) return;
  event l2(t);
}
event http_message_done(c: connection, ...) {
  local t = http_request_translate(c);
  event l1(t);
}
```

6 Evaluation & Future Optimizations

We have presented the implementation of a compiler that translates Chimera queries into the Bro event language. Because functionality was our primary focus, we have not yet implemented any performance optimizations. However, there are many areas that have potential for optimization. Here we evaluate the compiler's processing performance in its current unoptimized form and discuss opportunities for future performance optimization. This section does not evaluate memory utilization because it is highly dependent on the particular query, desired window size, and data rate of the connection. Windows for JOIN and GROUP BY operations can be scaled according to the operating environment and analytic needs.

6.1 Performance Measurement

The performance measurements in this section were taken by processing a 2 GB PCAP file (stored on a ram disk) with Bro and recording the execution time. The PCAP file was generated by capturing traffic at a U.S. government network gateway, so it includes data from a variety of protocols. It contains approximately 81k HTTP, 58k SMTP, and 32k DNS messages.

To test the compiler's performance, we compare the Bro event code generated by the Chimera compiler to hand-written Bro code that implements the same functionality. For example, the Bro code in section 5.3 could be written by hand as follows:

```
@http-reply
event http_request(c: connection,
    method: string, original_URI: string,
    unescaped_URI: string, version: string) {
  if (method == "GET")
    print original_URI;
}
```

This shorter implementation has three optimizations:

1. Data is not copied into new record types.
2. Events with only one handler are evaluated inline.
3. An earlier event handler (http_request) is used because the headers and body are not needed.

Our first experiment tests the effect of each optimization by applying them one-by-one to the section 5.3 example. We ran each configuration 30 times against the test data. Table 3 summarizes our results. Bypassing data copying saves about 1.5% execution time. Inlining event code makes no significant difference in this case. Switching to a single earlier handler saves another 1.5%, for a 3.0% overall speed-up. While the difference between current compiled code and hand-written code is noticeable, it does not have a major impact.

Table 3: Execution times with different optimizations

Configuration	Base	Opt-1	Opt-1+2	Opt-1+2+3
Avg. Time (s)	14.21	14.00	14.01	13.79
Std. Dev. σ (s)	0.084	0.083	0.074	0.081
Speed-up (%)	-	1.5%	1.4%	3.0%

For the next part of our evaluation, we tested a selection of more complicated queries from sections 4.1, 4.2.1, and 4.4. We ran the queries as they were compiled to Bro code, and after they were optimized by hand by eliminating unnecessary copying and event handlers. Because a Bro implementation of side-jacking was already available for query 4.1 [26], we used that as a basis for

Table 4: Execution times for different queries, with and without hand-optimization

Query	Base Time (s)	Optimized (s)	Speed-up (%)
4.1	15.64 ($\sigma = 0.081$)	15.48 ($\sigma = 0.067$)	1.1%
4.2.1	8.81 ($\sigma = 0.085$)	8.72 ($\sigma = 0.021$)	0.96%
4.4.	2.77 ($\sigma = 0.027$)	2.75 ($\sigma = 0.019$)	0.79%

comparison. We optimized the other two queries ourselves. Table 4 shows the results averaged across 30 runs for each measurement. Much like the first experiment, the overhead added by extra copying and event handlers only has a minor impact on overall performance, increasing running time by about 1%. Though the Bro code generator could benefit from some optimizations, in its current form it generates code that is almost exactly equivalent to hand-written code for these real-world scenarios.

6.2 Other Optimizations

The previous section discussed optimizations in the code generator related to event and data handling. There are also opportunities for optimization at the relational algebra level before any code is generated, and in the analysis logic. Prior work on query optimization for databases [11, 15, 25] is directly applicable here because it operates on relational algebra that is almost exactly the same as the Chimera Core language. One common trick is to break up selection operators into sub-expressions and put the cheapest one with the greatest data reduction first. Similarly, selection operators that occur after joins can have sub-expressions that do not depend on both join sides pushed before the join, thus reducing the number of items in the join table. Finally, any nodes that duplicate one another, including parsers, can be merged together. We plan to incorporate all of these optimizations in future versions of the Chimera compiler.

Another area of optimization that we plan to explore is improving the actual analysis logic. For example, an ordered EXCLUSIVE RIGHT JOIN is effectively an existence check; there is no need to actually store left tuples in the join table because they will never be emitted as output. Going further down this route, an existence check can be approximated efficiently using a Bloom filter. For analytics where complete precision is not necessary, an exclusive right join could be implemented with a windowed bloom filter.

Finally, queries in the Chimera language lend themselves well to parallel processing using a map-reduce model. Tuples can be mapped to a processing node using their join or group key right before each join or group operator in the data-flow graph. Each node will then execute the operator to perform the reduction. Global aggregates can be computed by extending aggregate functions to have a merge routine that combines partial answers as discussed in section 3.4.3 (though not all aggregates can be merged efficiently). We plan to extend the Chimera compiler in the future to automatically produce code that can run in a parallel environment.

7 Related Work

There has been a lot of prior research on streaming database systems. STREAM [3, 19] and Aurora [1] were pioneers in this area. Following initial work, others have developed improved techniques for windowed query evaluation [17] and load shedding [24]. An effort has also been make to create a standard for streaming SQL [14] that accounts for semantic differences between various systems. Others have focused on window specification semantics for streaming queries [6, 20].

Streaming database research is useful and serves as a basis for ideas in this paper, but Chimera goes beyond what has been done in prior work. It is the first language designed to translate into external intrusion detection frameworks like Bro. Chimera also adds two new capabilities that are very important for handling network traffic. The first is support for structured data types, which includes the new SPLIT operator and apply functions. The second major contribution is the addition of dynamic window conditions using the UNTIL trigger for aggregates, and the WINDOW condition for joins. This gives the query writer full control over window boundaries, allowing for immediate response after a detection threshold has been reached, rather than having to wait until the window expires as with traditional fixed window specifications.

One project that is related to Chimera is Gigascope [7]. Gigascope is a platform for performing network traffic analysis that uses an SQL query language. However, Gigascope is different from Chimera in a few key ways. First, it is a vertically integrated query language and platform for performing analysis. Its language is therefore tied to the implementation and has not been adapted to target other platforms. Chimera, on the other hand, is designed to be implementation-agnostic and serve as a general-purpose language for network processing. Furthermore, Gigascope's SQL query language has the same limitations as traditional streaming database systems. As far as we are aware, it only supports flat schemata, which prevents it from properly handling structured data. It also uses standard window specifications instead of dynamic window boundaries, which limits flexibility for join and aggregate queries.

IBM's Stream Processing Language (SPL) [13] is also related to Chimera. Unlike Chimera, SPL is not entirely declarative. Its *logic* clause uses procedural code and one must specify data flow paths to define analysis logic. SPL does support dynamic window boundaries using a *punct* type of *tumbling* window in which boundaries are set by messages from upstream operators. These operators can use arbitrarily complex logic to generate *punct* messages, which in theory provides the same power as dynamic window conditions in Chimera, but in a less concise manner. We view SPL as largely analogous to the Bro event language, except that it is data-flow-based rather than event-based. It is a powerful lower-level language that provides greater control, but suffers from the same problems of being less concise and more complicated than Chimera. We imagine that it would be possible to adapt the Chimera compiler to generate code for SPL in the future.

There are a number of procedural language extensions for traditional relational databases, including PL/SQL[10], Transact-SQL [9], and PL/pgSQL [18]. These procedural languages offer powerful constructs like conditional statements and looping. PL/SQL also offers array data types, and arrays can be simulated with delimiter-separated strings. These languages do not directly offer apply functions or SPLIT operations, but the same result can be achieved (albeit not as elegantly) with nested queries. While it is possible to express Chimera queries and data types in these procedural programming languages (they are Turing complete), we believe that the Chimera language is more intuitive for processing structured network protocol traffic. Chimera also goes further by running in a streaming environment and translating to the Bro event language.

The idea of having a high-level language that translates into low-level policy has been applied previously to other areas. One particularly relevant example is for router and firewall configurations [4, 12]. Low-level firewall policies precisely describe the mechanism for filtering traffic in a level of detail that goes beyond the high-level goals behind them. This makes firewall configuration policies difficult to read and error-prone. Previous work by Guttman et al. and Bartal et al. distills out the underlying security goals into a high-level language, and then translates that into low-level policies, thus eliminating the need for administrators to write those low-level policies. Chimera is applying the same idea of separating policy from mechanism, but for a much different different domain.

8 Conclusion

In this paper, we introduced Chimera, a new query language for processing network traffic. Chimera effectively separates policy from mechanism, leading to concise queries that are independent of implementation. Chimera is based on a streaming SQL syntax, which it extends by adding structured data, first-class functions, and dynamic window boundaries. These additional features allow Chimera to better handle complex network traffic analysis tasks.

This paper looks at example scenarios to motivate Chimera's design and demonstrate its utility. Two of the examples – side-jacking and DNS feature extraction – are taken from prior work. Writing Chimera queries for these examples showed how they are more compact than lower-level Bro event code and more precise than human language descriptions. The other two scenarios – detecting DNS tunnels and identifying spam/phishing e-mail – demonstrated some of Chimera's more advanced capabilities and showed how it can be used to express complex analysis logic with concise delcarative queries.

Finally, we presented the design and implementation of a compiler that translates Chimera queries into the Bro event language. This compiler works in two phases by first transforming an abstract syntax tree into a data flow representation, and then translating that representation into Bro event code. We tested the compiler's output against hand-optimized code for several queries and showed that it is only 3% slower in the worst case. This experiment highlighted opportunities for optimization by eliminating unnecessary copying and event handlers, but also showed that the Compiler generates code that is almost as efficient as hand-written code in its current form. In the future, we hope to implement these optimizations and also incorporate optimizations at the relational algebra level so that Chimera obviates the need to write low-level code for network analysis logic.

References

[1] Abadi, D. J., Carney, D., ÃĞetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. Aurora: A new model and architecture for data stream management. *The VLDB Journal 12*, 2 (2003).

[2] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* 1986.

[3] Arasu, A., Babu, S., and Widom, J. CQL: A language for continuous queries over streams and relations. *Lecture Notes in Computer Science 2921*, 123–124 (2004).

[4] Bartal, Y., Mayer, A., Nissim, K., and Wool, A. Firmato: A novel firewall management toolkit. In *IEEE Symposium on Security and Privacy* (1999).

[5] Bilge, L., Kirda, E., Kruegel, C., and Balduzzi, M. Exposure: Finding malicious domains

using passive dns analysis. In *Network and Distributed System Security Symposium* (2011).

[6] BOTAN, I., DERAKHSHAN, R., DINDAR, N., HAAS, L., MILLER, R. J., AND TATBUL, N. SECRET: A model for analysis of the execution semantics of stream processing systems. *Proceedings of the VLDB Endowment 3*, 1–2 (2010).

[7] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: A stream database for network applications. In *2003 ACM SIGMOD International Conference on Management of Data* (2003).

[8] CROCKFORD, D. The application/json media type for javascript object notation (json). RFC 4627, Internet Engineering Task Force, July 2006.

[9] DARNOVSKY, M., AND BOWMAN, G. Transact-sql user's guide. Tech. Rep. 3231-21, Sybase, Inc., 1987.

[10] FEUERSTEIN, S. *Oracle PL/SQL Programming*, third ed. O'Reilly & Associates, Sebastapol, CA, 2002.

[11] GRAEFE, G. The volcano optimizer generator: Extensibility and efficient search. In *ICDE* (1993), pp. 209–218.

[12] GUTTMAN, J. Filtering postures: Local enforcement for global policies. In *IEEE Symposium on Security and Privacy* (1997).

[13] HIRZEL, M., ANDRADE, H., GEDIK, B., KUMAR, V., LOSA, G., MENDELL, M., NASGAARD, H., SOULÃĽ, R., AND WU, K.-L. Streams processing language (spl). Tech. Rep. RC24897, IBM, 2009.

[14] JAIN, N., MISHRA, S., SRINIVASAN, A., GEHRKE, J., WIDOM, J., BALAKRISHNAN, H., ÃĞETINTEMEL, U., CHERNIACK, M., TIBBETTS, R., AND ZDONIK, S. Towards a streaming SQL standard. *Proceedings of the VLDB Endowment 1*, 2 (2008).

[15] JARKE, M., AND KOCH, J. Query optimization in database systems. *ACM Computing Surveys*, 2 (1984), 111–152.

[16] JOHNSON, S. C. Yacc: Yet another compiler-compiler. Tech. Rep. 32, Bell Laboratories, 1975.

[17] LI, J., MAIER, D., TUFTE, K., PAPADIMOS, V., AND TUCKER, P. A. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *Information Systsems 34*, 1 (2005).

[18] MONJIAN, B. *PostgreSQL: Introduction and Concepts*. Addison-Wesley, Boston, MA, 2000.

[19] MOTWANI, R., WIDOM, J., ARASU, A., BABCOCK, B., BABU, S., DATAR, M., MANKU, G., OLSTON, C., ROSENSTEIN, J., AND VARMA, R. Query processing, resource management, and approximation in a data stream management system. Technical Report 2002-41, Stanford InfoLab, 2002.

[20] PATROUMPAS, K., AND SELLIS, T. Maintaining consistent results of continuous queries under diverse window specifications. *Information Systsems 36*, 1 (2011), 42–61.

[21] PAXSON, V. Bro: a system for detecting network intruders in real-time. *Computer Networks 31*, 23–24 (1999), 2435–2463.

[22] RILEY, R. D., ALI, N. M., AL-SENAIDI, K. S., AND AL-KUWARI, A. L. Empowering users against sidejacking attacks. In *ACM SIGCOMM 2010 conference* (2010).

[23] ROESCH, M. Snort – lightweight intrusion detection for networks. In *USENIX LISA âĂŹ99 Conference* (1999).

[24] TATBUL, N., AND ZDONIK, S. Window-aware load shedding for aggregation queries over data streams. In *32nd International Conference on Very Large Data Bases* (2009).

[25] V. MARKL, G. M. LOHMAN, V. R. Leo: An autonomic query optimizer for db2. *IBM Systems Journal 42*, 1 (2003).

[26] VALLENTIN, M. Taming the sheep: Detecting sidejacking with bro. http://matthias.vallentin.net/blog/2010/10/taming-the-sheep-detecting-sidejacking-with-bro/, 2010.