A Semantic Framework for Data Analysis in Networked Systems *

Arun Viswanathan† Alefiya Hussain†‡ Jelena Mirkovic† Stephen Schwab‡ John Wroclawski†

† *USC/Information Sciences Institute* ‡ *Sparta Inc.*

{*aviswana, hussain, mirkovic, jtw*}*@isi.edu* *Stephen.Schwab@cobham.com*

Abstract

Effective analysis of raw data from networked systems requires bridging the semantic gap between the data and the user's high-level understanding of the system. The raw data represents facts about the system state and analysis involves identifying a set of semantically relevant *behaviors*, which represent "interesting" relationships between these facts. Current analysis tools, such as wireshark and splunk, restrict analysis to the low-level of individual facts and provide limited constructs to aid users in bridging the semantic gap. Our objective is to enable semantic analysis at a level closer to the user's understanding of the system or process. The key to our approach is the introduction of a logic-based formulation of high-level *behavior* abstractions as a sequence or a group of related facts. This allows treating behavior representations as fundamental analysis primitives, elevating analysis to a higher semantic-level of abstraction. In this paper, we propose a behavior-based semantic analysis framework which provides: (a) a formal language for modeling high-level assertions over networked systems data as *behavior models*, (b) an analysis engine for extracting instances of user-specified behavior models from raw data. Our approach emphasizes reuse, composibility and extensibility of abstractions. We demonstrate the effectiveness of our approach by applying it to five analyses tasks; modeling a hypothesis on traffic traces, modeling experiment behavior, modeling a security threat, modeling dynamic change and composing higher-level models. Finally, we discuss the performance of our framework in terms of behavior complexity and number of input records.

*This work is funded by the Department of Homeland Security and Space and Naval Warfare Systems Center, San Diego, under Contract No. N66001-10-C-2018. All findings and conclusions expressed in this material are those of the authors and do not reflect the views of the funding agencies.

Part of Alefiya Hussain's contributions to this paper were while she was at Sparta Inc.

1 Introduction

The ability to convert raw data into higher-level insights and understanding has become a key enabler in many fields. We approach one particular aspect of this problem, namely the analysis of data within the domain of networked and distributed systems. Such systems routinely generate a plethora of logs, trace and audit data during their operation. Users, such as researchers and system administrators, use this raw data to understand system behavior, diagnose problems, discover new behaviors, or verify hypotheses. Effective analysis of such raw data requires bridging the semantic gap between raw data and the user's high-level understanding of the analysis domain. Our experience with analysis tools reveals that this problem is ill-addressed.

A typical approach to data analysis involves the user sifting through the data using simple search and correlation constructs like boolean queries to identify relationships and infer meaning from data. For example, wireshark [19] can help identify complete or incomplete TCP flows from packet traces and splunk [16] can help identify spurious logins from a server log. Our study of four popular tools, discussed in Section 2.1, reveals that current approaches require cumbersome multi-step analyses to infer semantic relationships from data. For example, a user analyzing a network packet trace may first have to extract individual flows by specifying specific attribute values related to each flow, and then somehow manually infer relationships like concurrency between the flows. This problem is further complicated if the user has to reason and analyze over multiple types of data. This separation between the raw data and the meaning it carries constitutes the semantic gap.

In this paper, we focus on the problem of expressing analyses tasks that are meaningful and useful to the user. Specifically, given a finite, timestamped list of *facts* about the system under observation, our objective is to assist the user in expressing and modeling semantically

relevant *behaviors*, which are "interesting" relationships between these facts or sequence of facts. These relationships encompass notions of ordering, causality, dependence, or concurrency.

Our insight is that higher-level understanding in networked and distributed systems can be expressed in the form of relationships between system states, simple behaviors, and complex behaviors. For example, in most situations, a typical web-server operation is better understood as a concurrent relationship between multiple HTTP sessions to a server rather than the details of the protocols and specific values in the packet headers. Thus, our data analysis approach introduces a *behavior* as a primitive analysis construct. Behaviors can be extended or constrained to create a *behavior model*, which forms an assertion about the overall behavior of the system. A behavior model can then be rapidly applied over data to validate the assertion. We discuss complete details about specifying behavior models in Section 3, and Section 4 presents the analysis engine for extracting instances of user-specified behavior models from raw data.

The behavior models are abstract entities to capture the semantic essence of a particular relationship without focusing on unnecessary details or particular parameters that may vary between individual facts or behaviors. Incorporation of abstract behavior models as explicitly represented and manipulated constructs within our framework provides two key benefits. First, this abstraction allows users of our framework to analyze and understand the raw data at a semantically relevant level. In Section 3.4, we introduce an example of a behavior model to identify pairs of communication events where the destination IP of the second event is same as the source IP of the first. Such models can be used to analyze many different datasets without any modification. Additionally, since behavior models are primitive analysis constructs, the framework supports extensibility by composing new models from behavior models present in the knowledge base as demonstrated in Section 5.5. Thus, representing analysis expertise explicitly as behavior models formalizes the semantics for data analysis in networked systems.

The second key benefit of our work is the ability to foster sharing and reuse of knowledge embedded in explicitly represented behavior models. Our first-hand experience with existing tools suggests that in most cases knowledge inferred from analysis resides either in a domain-specific tool or a single expert's brain. This is due to a lack of an explicit representation for capturing, storing, sharing, and reusing such knowledge in a context-independent way. Many current tools are either static in nature, handling only a fixed set of analyses and record types, or may offer limited extensibility, but through some mechanism that involves significant effort. For example, wireshark [19] is easily extensible using plugins, but writing a plugin requires understanding the wireshark API and C programming skills. In contrast, a well defined shareable format for representing knowledge about networked systems data offers the prospect that many different tools can be driven by, and contribute to, a single shared knowledge base.

Beyond the basic challenge, the task of semantic-level analysis is difficult for two disparate reasons. First, the definition of "interesting" may vary widely in different situations, requiring a rich toolbox of techniques for effective analysis. We address this problem by restricting the definition of "interesting relationships" to expressing a particular set of characteristics of networked systems as discussed in Section 3.1. Second, in large scale systems, efficient and intelligent data analysis is extremely resource intensive due to the sheer volume of system events and traces. While in Section 6 we report performance results, this paper primarily discusses the fundamental aspects of defining and employing explicit behavior models as a data analysis tool. Real-time analysis of data for applications such as intrusion detection is a future goal as discussed in Section 7.

The fundamental contribution of this paper is the introduction of a behavior-based semantic analysis framework for *confirmatory* and *exploratory* analysis of multivariate, multi-type, timestamped data captured from networked systems. The main elements of the semantic framework include (a) a specialized formal language for specifying behavior models and (b) an analysis engine for extracting instances of user-specified behavior models from data. In confirmatory analysis, the user specifies a validation criteria, expected system behavior or hypothesis, by writing a specific model or through composing a high-level model from existing models contained within the knowledge base of the framework. In exploratory analysis, a user applies existing models from the knowledge base to explore data for new or unanticipated behaviors. In Section 5 we present five detailed examples of how the framework can be applied for these data analysis tasks.

2 Related Work

In this section, we set the context for our work by first studying four popular analysis tools followed by a discussion on specification-based approaches for analysis of networked systems data.

2.1 Tool Comparison

In this section, we study four popular analysis methodologies: wireshark v1.2.7 [19], splunk v4.1 [16], Simple Event Correlator (SEC) v2.5.3 [18], Bro v1.5.2 [14], and compare them with our behavior-based semantic analysis framework (SAF). Both wireshark and splunk are

	wireshark	splunk	SEC	Bro	SAF
System goals	Interactive analysis	Interactive analysis	Real-time event correlation	High-speed, real-time monitoring	Interactive analysis
Input data	Network packets	Ascii data from any source	Ascii data from files, stdin, pipes	Network packets	Any type of data (with plugin)
Specification language	Boolean logic	Boolean logic	Simple language for specifying rules	Bro scripting language	Formal language based on temporal logic, interval temporal logic and boolean logic
Primitive constructs	Boolean predicates	Boolean predicates, unix-like pipelines and commands	Boolean predicates, functions written in Perl	Events (low-level or higher-level)	Behavior (low-level or higher-level)
Semantic constructs	None	External commands can encode semantics	Perl functions can encode semantics	Network notions such as connections, IP addrs., ports, and network protocols	Temporal logic and interval temporal logic operators for defining behaviors (Section 3)
Composibility of specs	None	Queries can be recorded and then composed into other queries	Matching events can trigger creation of new high-level events	Policies can compose lower-level events to generate higher-level events	Behaviors can be composed into higher level behaviors
Abstraction	None	None	Limited	Yes	Yes

Table 1: Comparison of the behavior-based Semantic Analysis Framework (SAF) with four popular data analysis tools.

mainly interactive analysis tools while Bro and SEC are real-time monitoring tools. The behavior-based semantic analysis framework (SAF) falls in the category of interactive analysis tools. The tools are compared along seven dimensions in Table 1; (a) high-level goals, (b) input data types, (c) analysis specification language (d) primitive analysis constructs, (e) semantic analysis constructs, (f) ability to compose specifications and (g) abstraction, that is, specifications in terms of relationships between data attributes.

Each paragraph below introduces an analysis framework and the reader is directed to Table 1 for details. The corresponding features for our framework (SAF) are introduced in Table 1 and explored in future sections. We have not considered SQL-based approaches on streaming data for comparison [6], since SAF representations are at a higher-level of abstraction than database query languages. However, we further discuss how our framework could benefit by using the above SQL extensions to optimize event storage and retrieval in Section 7.

wireshark [19] is an open-source tool for interactive analysis of a large variety of network data from a packet capture file. Wireshark's design can be separated into the analysis framework and plugins. The analysis framework provides the ability to sift through large volumes of packets visually and provides a boolean query grammar for finding "interesting" relationships and statistical summaries over typical networking concepts, for example, rate, flows, bytes, and connections. The plugin architecture, on the other hand, is responsible for normalizing and presenting different types of packet data and protocol behavior to the analysis framework in a uniform way.

splunk [16] is a popular commercial framework for unified data analysis of a large variety of data. Splunk's strength comes from its ability to index various types of data, allowing the user to sift through logs by combining search queries using boolean operations, pipes and powerful statistical and aggregation functions. Splunk supports time-based, event-based, value-based correlations and also allows combining queries into higher-level queries. Splunk is extensible using apps, which allow encoding knowledge as queries for sharing and wider dissemination. However, it does not provide support for explicitly capturing domain expertise with semantic constructs. It does provide the ability to invoke external commands, thus providing an indirect way to incorporate explicit domain expertise into the analyses.

Simple Event Correlator(SEC) [18] is an open-source framework for rule-based event correlation. SEC reads the analysis specifications from a configuration file containing a set of event matching rules and corresponding actions. SEC processes data from log files, pipes and standard streams to trigger the configured actions on a match. It supports both time-based and event-based correlations and also allows specifying abstract rules that bind their values at runtime. SEC is more sophisticated than the previous two tools, it supports composing higher-level events by correlating low-level events, providing a framework for semantic understanding. Its rule-types *pair* and *pairwithwindow* capture some of the semantics of *ordering* and *duration*. However, it lacks support for inferring interval-based temporal relationships like concurrency and overlap and the analysis specification in the configuration files are not intuitive to capture

and share domain expertise in a generic way.

Bro [14] is a high-speed intrusion detection system for checking security policy violations by passively monitoring network traffic in real-time. Bro's security policies are written in the specialized Bro scripting language which is geared towards security analysis. The language supports semantic constructs such as connections, IP addresses, ports, and various network protocols along with various operators and functions to express different forms of network analyses. Bro has the ability to do time-based and event-based correlation. However, Bro mainly processes network packet data and uses a programming language-based analysis approach.

2.2 Specification-based Approaches

Specification-based approaches are particularly appealing in various areas of networked and distributed systems due to their ability to be abstract, concise, precise, and verifiable. In formal verification of distributed and concurrent systems, a system is specified in logic and then formal reasoning is applied on the specification to verify desired properties [3, 9]. In declarative networking, a specification language, Network Datalog (NDLog) [10], allows defining high-level networking specifications for rapidly specifying, modeling, implementing, and experimenting with evolving designs for network architectures. In testbed-based experimentation, a simple set of user-supplied expectations are used to validate expected behavior of an experiment [12].

The formal specification approaches have been well developed within the intrusion detection community and have been successfully applied to network and audit data for analysis. In this section we first present a brief overview of four such approaches and then compare them to SAF.

Roger et al. [15], leverage the idea that attack signatures are best expressed in simple temporal logic using temporal connectives to express ordering of events. They pose the detection problem as a model-checking problem against event logs. Naldurg et al. [13], propose another temporal-logic based approach for real-time monitoring and detection. Their language EAGLE supports parameterized recursive equations and allows specifying signatures with complex temporal event patterns along with properties involving real-time, statistics and data values. Kinder et al. [8], extend the logic CTL (Computation Tree Logic) and introduce CTPL (Computation Tree Predicate Logic) to describe malicious code as a high-level specification. Their approach allows writing specifications that capture malware variants. Ellis et al. [4], introduce a behavioral detection approach to malware by focusing on detecting patterns at higher-level of abstractions. They introduce three high-level behavioral signatures which have the ability to detect classes of worms without needing any apriori information of the worm behavior.

The SAF abstract models are comparable to the approaches of [13, 8, 4] in their use of formal logic and temporal constructs for specifications. But, in addition to providing an extended set of sophisticated intuitive operators and constructs, the behavior models presented in this paper can be generically applied to model various scenarios over a variety of data and are easily composed into semantically relevant higher-level models. This allows creating a knowledge base to explicitly capture domain expertise required for analyzing a large variety of operations encountered in networked and distributed systems as shown in Section 5. The higher-level behavioral signatures [4] based on the network-theoretic abstract communication network (ACN) are tightly bound to networking constructs like hosts, routers, sensors and links making them very restrictive in their ability to express general networked systems behaviors.

The SAF is based on a logic-based specification approach rather than a programming language-based specification approach like the one followed in Bro. Our goal is that the behavior models should be abstract but also concise and precise to support well-known knowledge representation and reasoning approaches. Logic is declarative and type-free, imparting formal semantics, abstract specifications, and efficient processing by analysis engines. The logic-based approach also enables building a knowledge base of behavior models to explicitly capture domain expertise that can be used to automatically reason and infer behavior models. However, logic-based approaches are less expressive than programming languages. The expressiveness of our approach is based on requirements derived from characteristics of networked systems as discussed in Section 3.1.

3 Behavior Models

A particular execution of a networked system or process can be captured as a sequence of *states*, where a state is a collection of attributes and their values. A *behavior* (b) is a sequence of one or more related states. A system execution is thus defined as a combination of different behaviors, and each new execution may generate a unique set of behaviors. A *behavior model* (ϕ) is a formula that makes an assertion about the overall behavior of the system.

For example, consider a simplified IP flow in networking, where a flow is a communication between two hosts identified by their IP addresses. For simplicity we assume an IP flow to be broken into two states: `ip_s2d` denotes a packet from some source to destination host and `ip_d2s` denotes a packet from a destination to source. Then, a valid IP flow behavior, `IPFLOW`, is one where `ip_s2d` and `ip_d2s` are related by their source

and destination attributes with the additional criteria that `ip_d2s` always occurs after `ip_s2d`. The behavior model (ϕ_{ipflow}) is an assertion that `IPFLOW` is valid. We discuss details of this example and extend it further in Section 3.4.

In this section, we first discuss the requirements and design choices for a language to specify behaviors followed by the formal syntax and semantics of the language.

3.1 Requirements

As discussed in Section 1, the key objective of our framework is to enable semantic-level analysis over data. A semantically expressive language for analysis over networked and distributed systems data must meet the following requirements: (a) enable analysis over multi-type, multi-variate, timestamped data, (b) express a wide variety of "interesting" relationships, (c) enable analysis over higher-level abstractions, and (d) enable composing abstractions into higher-level abstractions.

The language should express at-least the following "interesting" relationships to capture the core characteristics of networked and distributed systems: (a) causal relationships between behaviors, for example, a file being opened only if a user is authorized; (b) partial or total ordering, for example, in-order or out-of-order arrival of packets; (c) dynamic changes over time, for example, traffic between client and server drops after an attack on the server; (d) concurrency of operations, for example, simultaneous web client sessions; (e) multiple possible behaviors, for example, a polymorphic worm behavior may vary on each execution; (f) synchronous or asynchronous operations, for example, some operations need to complete within a specific time whereas others need not; (g) value dependencies between operations, for example, a TCP flow is valid only if the attribute–values contained in the individual packets are related to each other; (h) invariant operations, for example, some operations may always hold true and, (i) eventual operations, for example, some operations happen in the course of time. In addition, we need traditional mechanisms, such as boolean operators and loops, for combining these relationships into complex behaviors and mechanisms for basic counting of events and reasoning over the counts.

We do not claim completeness of the above requirements but we believe that being able to express the above classes of primitive relationships and combining them to form complex relationships would suffice for a wide range of situations, a few of which we demonstrate as case studies in Section 5.

3.2 Design

The following four design decisions realize the requirements listed above. First, our framework provides logic-based support to formulate behavior abstractions as a sequence or group of related events, where events are uniform representation of system facts as discussed later. This formulation allows treating this behavior representation as fundamental analysis primitive, elevating analyses to a higher semantic-level of abstraction.

Second, the language combines operators from Allen's interval-temporal logic [1], Lamport's Temporal Logic of Actions [9] and boolean logic. Temporal logic allows expressing the ordering of events in time without explicitly introducing time. Interval-temporal logic allows expressing relationships like concurrency, overlap and ordering between behaviors as relationships between their time-intervals. Additionally, complex behaviors are easily composed from simpler ones using boolean operators.

Third, the framework enables specifying dependency relationships between event attributes while leaving the values to be dynamically populated at runtime. Late binding enables abstract specifications that enrich the knowledge base as they can be directly applied to a wide variety of data-sets. This also enables parametrization of models during complex model composition as discussed in Section 5.5.

Lastly, the framework introduces the notion of a domain-independent *event* as a uniform representation of multi-type, multi-variate, timestamped data. Specifically, an *event* (e) is a representation of system state and is given by a 4-tuple $\langle o, c, t, av \rangle$ where o is the event-origin (for example, the host IP), c is the event-type (for example, PKT_TCP or APP_HTTPD), t is the event timestamp and $av = \{ \langle a_i, v_i \rangle \,|\, a_i \in A, v_i \in \mathsf{Strings}, 1 \leq i \leq D_c \}$ are the attribute-value pairs contained in the event. A is the set of attribute labels, for example, $sip, dip, etype$. D_c is the number of attributes in an event of type c. This normalization of data to events ensures that the analysis algorithms are independent of the input domain.

We believe these design decisions ensure developing abstract behavior models as first-order primitives for capturing, storing, and reusing domain expertise for the analysis of networked systems. Next we discuss the syntax of such a language.

3.3 Syntax

The language grammar for defining a behavior model ϕ as a formula, consists of five key elements as shown in Figure 1: state propositions S as atomic formulae; grouping operators '(' and ')' to define sub-formulae; logical operators and temporal operators for relating sub-formulae or atomic-formulae; the optional behavior constraints $bcon$ and operator constraints $opcon$ written within '[' and ']'; and the relational operators $relop$.

A *state proposition, S,* is an atomic formula for capturing events that satisfy specified relations between at-

$$
\begin{array}{rcll}
\phi & ::= & \text{`(' } S \mid \phi \text{ `)'} \{ bcon \} & \\
 & | & \texttt{not}\ \phi & (negation) \\
 & | & \phi\ \texttt{and}\ \phi & (logical\ and) \\
 & | & \phi\ \texttt{or}\ \phi & (logical\ or) \\
 & | & \phi\ \texttt{xor}\ \phi & (logical\ xor) \\
 & | & \phi \rightsquigarrow_{(opcon)} \phi & (leadsto) \\
 & | & \Box_{(opcon)}\ \phi & (always) \\
 & | & \phi\ \texttt{olap}_{(opcon)}\ \phi & (overlaps) \\
 & | & \phi\ \texttt{dur}_{(opcon)}\ \phi & (during) \\
 & | & \phi\ \texttt{sw}_{(opcon)}\ \phi & (startswith) \\
 & | & \phi\ \texttt{ew}_{(opcon)}\ \phi & (endswith) \\
 & | & \phi\ \texttt{eq}_{(opcon)}\ \phi & (equals) \\
bcon & ::= & \text{`[' } \{ tc \mid cc \} \text{ `]'} & \\
tc & ::= & \{\texttt{at} \mid \texttt{duration} \mid \texttt{end}\}\ relop\ t\{: t\} & \\
cc & ::= & \{\texttt{icount} \mid \texttt{bcount} \mid \texttt{rate}\}\ relop\ c\{: c\} & \\
opcon & ::= & \text{`[' } relop\ t\{: t\} \text{ `]'} & \\
relop & ::= & \{ > \mid < \mid = \mid \geq \mid \leq \mid \neq \} & \\
t & ::= & [0-9]+\ \{s|ms\} & \\
c & ::= & [0-9]+ &
\end{array}
$$

Figure 1: The grammar for specifying a behavior model ϕ.

tributes and their values. In essence, S captures states of a system or process and is the basic element of a behavior model. The most trivial behavior model is one with a single state proposition. Formally, S is represented as a finite collection of related attribute-value tuples as:

$$S = \{(a_i, r_i, v_i) \mid i \in \mathbb{N}, a_i \in A, v_i \in V, \\ r_i \in (=, >, <, \geq, \leq, \neq)\}$$

A is a set of string labels, such as `sip`, `dip`, `etype` and V is a set of string constants, such as `10.1.1.2`, `/bin/sh`, along with two special strings: (a) strings prefixed with '$', as in `$$`, `$s2.dst` (b) strings with the wild-card character '*', as in `/etc/pas*`. Considering our previous example of `IPFLOW`, the state propositions `ip_s2d` and `ip_d2s` are written as:

```
ip_s2d = {etype=PKT_IP, sip=$$,dip=$$}
ip_d2s = {etype=PKT_IP,sip=$ip_s2d.dip,
          dip=$ip_s2d.sip}
```

State proposition `ip_s2d` contains three attributes `etype`, `sip` and `dip`. `etype` has a constant value `PKT_IP`, while `sip` and `dip` attributes use the '$' prefixed special variables which are dynamically bound at runtime. State proposition `ip_d2s` defines the values of its `sip` and `dip` attributes as being dependent on values of state `ip_s2d`. Dependent attributes along with dynamic binding of values allows leaving out details like the actual IP addresses from the specification.

The *temporal operators* allow expressing temporal relationships like ordering and concurrency between one-or-more behaviors. The linear-time temporal operator $\rightsquigarrow$ (*leadsto*), written as $\sim>$, is used to express causal relationships between behaviors. The interval temporal logic operators express concurrent relationships between behaviors as either relationships: (a) between their starttimes using `sw` (*startswith*), (b) between their endtimes using `ew` (*endswith*) or (c) between their durations using `olap` (*overlap*), `eq` (*equals*) and `dur` (*during*). The $\Box$ (*always*) operator, written as $[\,]$, allows expressing invariant behaviors. The *logical operators* `not`, `and`, `or`, `xor` are supported for logical operations over behaviors and for creating complex behaviors.

Behavior constraints allow placing additional constraints on the matching behavior instances and are specified immediately following the behavior within square brackets. Constraints and their values are related using the standard relational operators. The six behavior constraints are divided as time constraints tc and count constraints cc. Time constraints allow constraining behavior starttime using `at`, behavior endtime using `end` and behavior duration using `duration`. The time value, t, for the constraint can be specified as a single positive value or as a range. Additionally, the values can be suffixed with either 's' or 'ms' to indicate seconds or milliseconds respectively. The count constraints allow constraining number of matching behavior instances using `icount`, the size of each behavior instance using `bcount` and rate of events within a behavior instance using `rate`. *Operator constraints* allow specifying time bounds over the temporal operators thus allowing their semantics to be slightly modified. The operator constraint values are specified as a single value or a range along with a relational operator. Table 2 presents detailed semantics of operators along with behavior and operator constraints.

Expressing a behavior in the language constitutes writing sub-formulae. Behaviors are always enclosed within parenthesis '(' and ')'. Simple behaviors are constructed by relating one-or-more state propositions using operators, while complex behaviors are constructed by relating one-or-more behaviors. The grammar also allows expressing complex behaviors using recursion and we present an example in Section 5.3. Recursive definitions allow expressing looping behavior for which the loop bounds can be optionally specified using the `bcount` behavior constraint. The current grammar does not support existential and universal quantification since such a need is not clear. We explore these language extensions as part of our future work.

Writing behavior models in the framework involves additional syntax such as namespaces, headers and variables which are discussed along with the case-studies in Section 5.1 and Section 5.2. Next section presents the formal semantics of the language.

3.4 Semantics

We first define two concepts important for understanding the semantics. A *sequential log* (L) is a finite sequence of timestamped events $L = e_1, e_2, e_3, \ldots, e_n$ such that $e_i.t \leq e_j.t\ , \forall\, i < j$. A *behavior instance* B_ϕ for a behavior model ϕ is sequence or groups of events satisfying

Behavior model ψ	**Meaning of ψ**	**L satisfies ψ ($L \models \psi$) iff**
(ϕ)	ϕ is a behavior.	$\exists B_\phi \subseteq L$ and $\lvert B_\phi \rvert > 0$
S	S is a state proposition defined as $S = \{(a_1, r_1, v_1) \ldots, (a_d, r_d, v_d)\}$.	(a) $\lvert B_S \rvert > 0$, (b) $\forall\, e \in B_S$, $\forall\, i \in \{1, \ldots, d\}$, $e.a_i$ is defined and values $e.v_i$ and $S.v_i$ satisfy relation r_i.
$(\texttt{neg}\ \phi)$	Negation of behavior is true.	$L \not\models \phi$, that is, $\lvert B_\phi \rvert = 0$
$(\phi_1\ \texttt{and}\ \phi_2)$	Both ϕ_1 and ϕ_2 are true.	$L \models \phi_1$ and $L \models \phi_2$
$(\phi_1\ \texttt{or}\ \phi_2)$	ϕ_1 and ϕ_2 are not both false simultaneously.	$L \models \phi_1$ or $L \models \phi_1$ or satisfies both ϕ_1 and ϕ_2
$(\phi_1\ \texttt{xor}\ \phi_2)$	Either of ϕ_1 or ϕ_2 are true but not both.	$L \models \phi_1$ or $L \models \phi_2$ but not both
$(\phi_1 \rightsquigarrow \phi_2)$	ϕ_1 leadsto ϕ_2, that is, whenever ϕ_1 is satisfied ϕ_2 will eventually be satisfied.	(a) $L \models \phi_1$ and $L \models \phi_2$, (b) $B_{\phi_1}[1] \neq B_{\phi_2}[1]$, (c) $B_{\phi_2}.starttime \geq B_{\phi_1}.endtime$
$(\phi_1 \rightsquigarrow [\leq t]\, \phi_2)$	Whenever ϕ_1 is satisfied ϕ_2 will be satisfied within t time units.	(a) $L \models (\phi_1 \rightsquigarrow \phi_2)$, (b) $B_{\phi_2}.starttime \leq (B_{\phi_1}.endtime + t)$
$(\Box\, \phi)$	ϕ is always satisfied, that is, satisfied by each event.	$\forall\, e \in L, e \models \phi$
$(\Box[= t]\, \phi)$	ϕ is always satisfied within every consecutive interval(epoch) of t time units.	$t > 0$ and for all consecutive intervals t, $l_t \subseteq L$ and $l_t \models \phi$
$(\phi_1\ \texttt{sw}\ \phi_2)$	ϕ_1 starts with ϕ_2.	(a) $L \models \phi_1$ and $L \models \phi_2$, (b) $B_{\phi_1}[1] \neq B_{\phi_2}[1]$, (c) $B_{\phi_1}.starttime = B_{\phi_2}.starttime$
$(\phi_1\ \texttt{sw}[\geq t]\, \phi_2)$	ϕ_1 starts t time units after ϕ_2.	(a) $L \models (\phi_1\ \texttt{sw}\ \phi_2)$, (b) $B_{\phi_1}.starttime \geq (B_{\phi_2}.starttime + t)$
$(\phi_1\ \texttt{ew}\ \phi_2)$	ϕ_1 ends with ϕ_2.	(a) $L \models \phi_1$ and $L \models \phi_2$, (b) $B_{\phi_1}[1] \neq B_{\phi_2}[1]$, (c) $B_{\phi_1}.endtime = B_{\phi_2}.endtime$
$(\phi_1\ \texttt{ew}[= t]\, \phi_2)$	ϕ_1 ends t time units after ϕ_2.	(a) $L \models (\phi_1\ \texttt{ew}\ \phi_2)$, (b) $B_{\phi_1}.endtime = (B_{\phi_2}.endtime + t)$
$(\phi_1\ \texttt{olap}\ \phi_2)$	ϕ_1 overlaps ϕ_2, that is, ϕ_1 starts after ϕ_2 starts but before ϕ_2 ends and ends after ϕ_2 ends.	(a) $L \models \phi_1$ and $L \models \phi_2$, (b) $B_{\phi_1}[1] \neq B_{\phi_2}[1]$, (c) $(B_{\phi_2}.starttime < B_{\phi_1}.starttime < B_{\phi_2}.endtime)$ and $(B_{\phi_1}.endtime > B_{\phi_2}.endtime)$
$(\phi_1\ \texttt{olap}[> t]\, \phi_2)$	ϕ_1 overlaps ϕ_2 and the overlapping region is greater than t time units.	(a) $L \models (\phi_1\ \texttt{olap}\ \phi_2)$, (b) the overlap $(B_{\phi_2}.endtime - B_{\phi_1}.starttime) > t$
$(\phi_1\ \texttt{eq}\ \phi_2)$	ϕ_1 equals ϕ_2 in duration.	(a) $L \models \phi_1$ and $L \models \phi_2$, (b) $B_{\phi_1}[1] \neq B_{\phi_2}[1]$, (c) $B_{\phi_1}.duration = B_{\phi_2}.duration$
$(\phi_1\ \texttt{eq}[= t]\, \phi_2)$	ϕ_1 and ϕ_2 are both of duration t.	(a) $L \models (\phi_1\ \texttt{eq}\ \phi_2)$, (b) $B_{\phi_1}.duration = B_{\phi_2}.duration = t$
$(\phi_1\ \texttt{dur}\ \phi_2)$	ϕ_1 occurs during ϕ_2, that is, ϕ_1 starts after ϕ_2 and ends before ϕ_2 ends.	(a) $L \models \phi_1$ and $L \models \phi_2$, (b) $B_{\phi_1}[1] \neq B_{\phi_2}[1]$, (c) $(B_{\phi_1}.starttime > B_{\phi_2}.starttime)$ and $(B_{\phi_1}.endtime < B_{\phi_2}.endtime)$
$(\phi_1\ \texttt{dur}[= t_1 : t_2]\, \phi_2)$	ϕ_1 occurs during ϕ_2 with duration between t_1 and t_2.	(a) $L \models (\phi_1\ \texttt{dur}\ \phi_2)$, (b) $(t_1 \leq B_{\phi_1}.duration \leq t_2)$
$(\phi)[\texttt{icount} = c]$	The number of behavior instances satisfying ϕ is c.	(a) $L \models \phi$, (b) there exist distinct $B^1_\phi \ldots B^c_\phi \subseteq L$
$(\phi)[\texttt{bcount} = c]$	Behavior instances satisfying ϕ are of size c.	(a) $L \models \phi$, (b) $B_\phi.bcount = c$
$(\phi)[\texttt{rate} > c]$	Behavior instances satisfying ϕ have a rate, defined as (behavior size / behavior duration) greater than c.	(a) $L \models \phi$, (b) $(B_\phi.bcount / B_\phi.duration) > c$ and $B_\phi.duration > 0$
$(\phi)[\texttt{at} < t]$	Starting time of behavior instances satisfying ϕ must be less than absolute time t.	(a) $L \models \phi$, (b) $B_\phi.starttime < t$
$(\phi)[\texttt{end} \geq t]$	Behavior instances satisfying ϕ have endtime greater than absolute time t.	(a) $L \models \phi$, (b) $B_\phi.endtime \geq t$
$(\phi)[\texttt{duration} \neq t]$	Behavior instances satisfying ϕ are of duration $\neq t$.	(a) $L \models \phi$, (b) $B_\phi.duration \neq t$

Table 2: Semantics of operators, behavior constraints and operator constraints in our logic. We describe semantics for constraints considering only a single relational operator and refer the reader to the framework webpage [17] for details.

the behavior model ϕ.

$$B_\phi = \langle starttime, endtime, bcount, (b_1, b_2, \ldots, b_k) \rangle$$

where $(b_1, b_2, \ldots, b_k) \subseteq L$ could be an individual event e or another behavior-instance B_{ϕ_i}. $starttime = b_1.starttime$ is the starting time of the behavior as defined by its first element and $endtime = b_k.endtime$ is the ending time of the behavior as defined by its last element. $bcount = k$ is the total number of elements in the behavior instance. All b_i's are in increasing time-order of their $starttime$. Additionally, let $B_\phi.duration$ = $(B_\phi.endtime - B_\phi.startime)$ be the duration of the behavior instance and $\lvert B_\phi \rvert = B_\phi.bcount$ represent the size of behavior instance. If ϕ is a simple behavior, such as a state proposition S, then

$$B_s = \langle e_{i_1}.t, e_{i_k}.t, k, (e_{i_1}, \ldots, e_{i_k}) \rangle$$

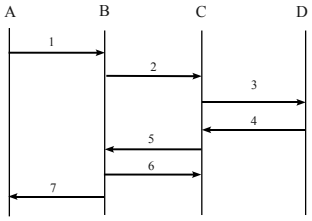


Figure 2: Sequence diagram of IP-interaction between four nodes. $\rightarrow$ or $\leftarrow$ represent an IP packet between a source (s) and destination (d). An IP flow is a packet pair between s and d.

where $(e_{i_1}, \ldots, e_{i_k}) \subseteq L$.

Given a finite sequential log L and a user-defined behavior model ϕ, goal of the analysis is to find all behavior instances $(B^1_\phi, B^2_\phi, \ldots)$ from L that satisfy the behavior model, where *satisfiability* is defined as follows:

$$L \models \phi \text{ iff } \exists B_\phi \subseteq L \text{ and } |B_\phi| > 0$$

That is, the log L satisfies ($\models$) the behavior model ϕ iff there exists a behavior instance B_ϕ in L of finite length $|B_\phi|$. Since ϕ is a composite formula created using many sub-formulas, the satisfiability of ϕ is determined as a function of satisfiability of its sub-formulae. Table 2 defines the satisfiability criteria for sub-formulae formed using the operators and constraints. We next explain the key language ideas by defining simple models and applying them to a fictitious data set.

Assume a packet trace of seven IP packets representing an interaction between four nodes A, B, C and D as shown in Figure 2. Let the sequential log of corresponding events be $e_1, e_2, \ldots, e_7$.

Using the states `ip_s2d` and `ip_d2s` defined earlier in Section 3.3, IP flow behavior is written as a causal relationship between the state propositions `ip_s2d` and `ip_d2s` as `IPFLOW=(ip_s2d ⇝ ip_d2s)`. There are three IP flow instances in Figure 2 that satisfy `IPFLOW`, that is, $icount = 3$ with $bcount = 2$ for each instance:

$$B^1_{ipflow} = (e_1, e_7)$$
$$B^2_{ipflow} = (e_2, e_5)$$
$$B^3_{ipflow} = (e_3, e_4)$$

Extending the example, a complex behavior for pairs of overlapping IP flows can now be written as `IPFLOW_PAIRS=(IPFLOW olap IPFLOW)`. There are in all three instances of overlapping `IPFLOW` pairs from Figure 2. That is,

$$B^1_{ipflow\_pairs} = ((e_1, e_7), (e_2, e_5))$$
$$B^2_{ipflow\_pairs} = ((e_1, e_7), (e_3, e_4))$$
$$B^3_{ipflow\_pairs} = ((e_2, e_5), (e_3, e_4))$$

Again, $icount = 3$ and for each instance $bcount = 2$, since $bcount$ counts the number of `IPFLOW` occurrences and not individual events.

We can additionally define a bad IP flow behavior `BAD_IPFLOW` as one for which there was no matching response from the destination. That is, `BAD_IPFLOW=(ip_s2d ⇝ (not ip_d2s))`. Event e_6 matches `BAD_IPFLOW` model since it has no matching response. That is, $B^1_{bad\_ipflow} = (e_6)$, with $bcount = 1$.

The next section describes the architecture of the analysis framework.

4 Semantic Analysis Framework

Given our objective of semantic-level data analysis, we require the analysis framework to support (a) analysis of multi-type, multi-variate, timestamped data, (b) defining new models by composing existing models, and (c) storage, retrieval and extensibility of domain-specific behavior models. The framework has five components as shown in Figure 3; the knowledge base, a data normalizer, an event storage system, an analysis engine and a presentation engine. The decoupling of behavior model specification, the input processing and the analysis algorithms, allows the framework to be directly applied across several different domains. Subsequent sections discuss the details of each component.

4.1 Knowledge Base

The knowledge base provides a namespace-based storage mechanism to store behavior models and is central in providing an extensible framework. For example, our networking domain currently defines models for *ipflow*, *tcpflow*, *icmpflow* and *udpflow*. These behavior models capture common domain information and allow a user to rapidly compose higher-level models by reusing existing behavior models. Reusing a behavior model from the knowledge base constitutes importing it using its namespace and name. For example, referring to the behavior model in Figure 4(a), line 5 imports the `IPFLOW` model from the `NET.BASE_PROTO` domain. The namespace allows categorization of models into domain-specific areas while allowing composition of models across domains. We implement namespaces similar to Java namespaces, that is, each component in the namespace corresponds to a directory name on the filesystem. This simple design ensures that the knowledge base is easily customizable and extensible.

4.2 Data Normalizer

The data normalizer maps a data record to the event format defined in Section 3.2. Raw data accepted by the normalizer can be in the form or trace files, packet dumps, audit logs, security logs, syslogs, kernel logs or script output with the only requirement that each data record have a timestamp and a message field. Specialized plugins in the normalizer convert each type of raw data into corresponding events. Figure 3(b) shows a possible event

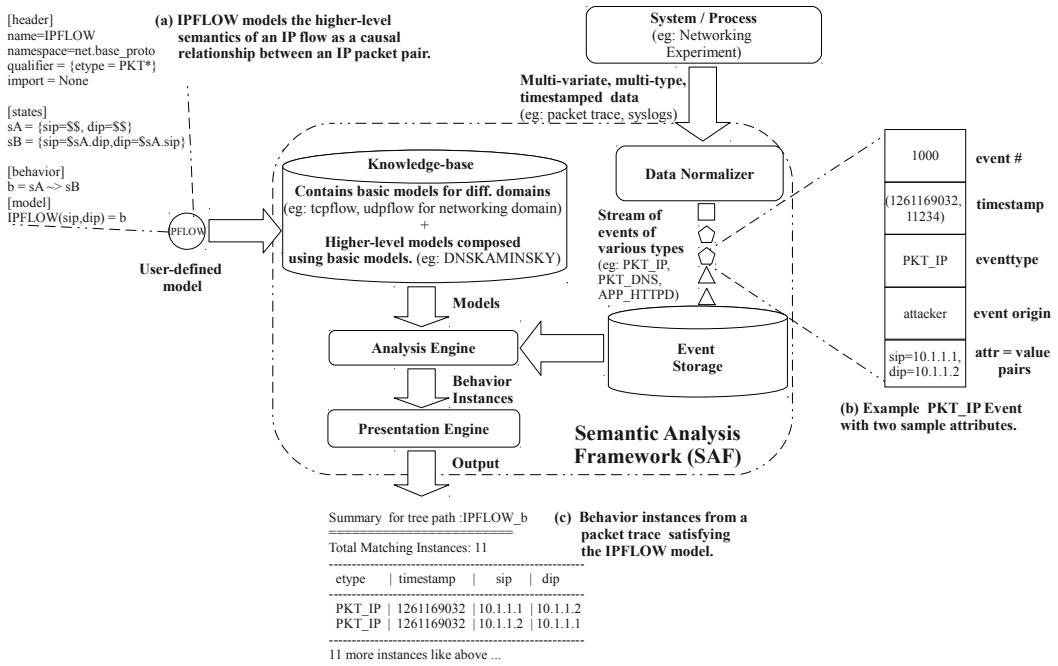


Figure 3: The semantic analysis framework (SAF) captures a user's higher-level analysis intent as (a) a behavior model, applies the model over (b) a finite stream of events normalized from raw data, and (c) outputs events satisfying the behavior model.

format for an IP packet from a packet dump. The current normalizer supports a C-based plugin API for writing new specialized plugins. The framework includes plugins for the basic packet-types of IP, TCP, UDP, ICMP, DNS along with plugins for parsing syslog, auth and server logs.

4.3 Event Storage

The event storage component is responsible for storing the events from the data normalizer into a database. Every event-type has a separate table, the columns of the tables correspond to the event attributes and each row describes an event. The current implementation stores all events into a SQLite database for two reasons: (a) it provides a standard and ready-to-use interface for storing and fetching events and (b) its server-less operation and open-source nature ensures portability on commodity systems. Our experience suggests that SQLite performs reasonably well for a large number of situations but presents challenges for complex analysis as the volume of events increases. Our future work includes investigating the scale and efficiency challenges involved in storage and retrieval of events.

4.4 Analysis and Presentation Engine

Given a finite sequential log L and a user-defined behavior model ϕ, goal of the analysis engine is to find all behavior instances $(B^1_\phi, B^2_\phi, \ldots)$ from L that satisfy the behavior model. Let the events in L be stored internally in the event storage database E_{db}. We discuss only the key ideas behind the analysis process by describing extraction of behavior instances satisfying the `IPFLOW` model defined in Section 3.4 from the sample data in Figure 2.

The behavior model ϕ is first internally represented in a manner similar to a compiler expression-tree and is then evaluated left-to-right in a post-order fashion. The satisfiability of the behavior model is determined as a function of satisfiability of each of the component behaviors according to the semantics defined in Table 2. For the `IPFLOW` model, the state proposition `ip_s2d={etype=PKT_IP,sip=$$,dip=$$}` is evaluated first. Since it does not have any dependent attributes, its expression is converted to the following query `{etype=PKT_IP,sip=*,dip=*}` and is used to fetch all events in E_{db} matching the query. All events $(e_1, e_2, e_3, e_4, e_5, e_6, e_7)$ match the state `ip_s2d`.

Next, the proposition `ip_d2s={etype=PKT_IP, sip=$ip_s2d.dip, dip=$ip_s2d.sip}` is evaluated. The attributes depend on the attributes of state `ip_s2d`. So, using each event that matched `ip_s2d`, a corresponding query is generated by resolving the values of `sip` and `dip` using the values from the matched events. From Figure 2, e_1 matches e_7, e_2 matches e_5, e_3 matches e_4. e_5 and e_6 are also possible candidates but since e_5 already matched e_2, it is not paired with e_6. Finally, the operator $\rightsquigarrow$ is evaluated, where the satisfiability criteria described in Table 2 is applied and any specified operator constraints are checked. The three instances satisfying the criteria (e_1,e_7), (e_2, e_5), and (e_3,e_4) are returned.

The *presentation engine* is responsible for extracting the output from the analysis stage and presenting it in a summarized format. We currently support printing the output in a tabular format as shown in Figure 3(c). We next present a brief analysis of the algorithm.

Algorithm Analysis As described in Section 3.3, state propositions could either contain constant attribute-values (*cStates*), such as `10.1.1.2`; dependent values (*dStates*), such as `$s1.dip`; or dynamic values (*iStates*), such as `$$`. A simple behavior consists of a combination of these states using one or more combinations of operators and constraints. We assume a constant processing time for all operators and constraints. Then, given an input of N events, processing a state proposition can involve two important operations which influence the runtime: (i) querying using the state expression and (ii) processing the results of the query if any. In the case of *cStates* and *iStates*, there is exactly one query made, and it generates at most N responses. Thus, the worst case for processing those N responses is $O(N)$. In the case of a *dstate*, given N events, there are N queries to be made and in the worst case every query may return $O(N)$ results that have to be processed. Thus, processing dependent states involves a worst case of $O(N^2)$ operations. We present our performance results in Section 6.

5 Case Studies

In this section, we evaluate the utility of our semantic framework by applying it to five different analysis scenarios: (a) confirming a hypothesis on collected network traces, (b) specifying expected system behavior during network experimentation, (c) modeling worm behavior as an example security threat, (d) modeling dynamic change, and (e) rapidly composing models to create higher-level behaviors. We present detailed explanation of input, the behavior model and analysis output for the first two cases. Due to space constraints, we briefly discuss the remaining three cases with their corresponding behavior models, demonstrating features of our semantic analysis framework.

5.1 Modeling Hypothesis

Researchers frequently need to validate hypothesis or test results presented by other researchers. We emulate one such scenario by validating the results presented by Hussain et al. [5] to demonstrate how behavior models can be rapidly created to reproduce results. We also discuss the syntax involved in writing a complete behavior model.

In the above referenced paper, a threshold-based heuristic was presented to identify DDoS attacks in traces captured at an ISP. Attacks on a victim were identified by testing for two thresholds on anonymized traces: (a) the number of sources that connect to the same destination within one second exceeds 60, or (b) the traffic rate exceeds 40,000 packets/sec. We demonstrate the advantages of behavior model-based analysis by defining a model to test for the two heuristics listed above using 10 seconds of the trace file containing the start of an attack. We normalize the packet traces to 142,530 `PKT_IP` events.

```
1. [header]
2. NAMESPACE=NET.ATTACKS
3. NAME=DDOS_HYP
4. QUALIFIER={}
5. IMPORT=NET.BASE_PROTO.IPFLOW

6. [states]
7. sA=IPFLOW.ip_s2d()
8. sB=IPFLOW.ip_s2d(dip=$sA.dip)

9. [behavior]
10.hyp_1=(sA)[bcount=1] ~>[<=1s] (sB)[bcount>=59]
11.hyp_2=(sA)[rate > 40000]

12.[model]
13.DDOS_HYP(timestamp,sip,dip,etype)= (hyp_1 or hyp_2)
```

(a) DDOS_HYP models two thresholds for detecting DDoS attacks.

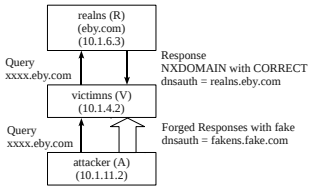
```
Summary : DDOS_HYP_hyp1
========================
Total Matching Instances: 2
Instance : 1 of 2 (Total Event Count: 60)
------------------------------------------------------------
  timestamp   |   sip        |   dip        |  etype
------------------------------------------------------------
                State Definition: sA
  1025390156  |201.199.184.56|87.231.216.115| PKT_ICMP

State Definition: ~> [<= 1 s ] sB [ ecount >= 59 ]
  1025390156  |201.199.184.56|87.231.216.115| PKT_ICMP
  1025390156  |201.199.184.56|87.231.216.115| PKT_ICMP
<truncated output containing remaining 57 events>

Instance : 2 of 2 (Total Event Count: 60)
------------------------------------------------------------
  timestamp   |   sip        |   dip        |  etype
------------------------------------------------------------
                State Definition: sA
  1025390157  |53.232.170.113|87.134.184.48 | PKT_ICMP

State Definition: ~> [<= 1 s ] sB [ ecount >= 59 ]
  1025390157  |33.138.213.170|87.134.184.48 | PKT_ICMP
  1025390157  |33.138.213.181|87.134.184.48 | PKT_ICMP
<truncated output containing remaining 57 events>
```

(b) Behavior instances satisfying the DDOS_HYP model.

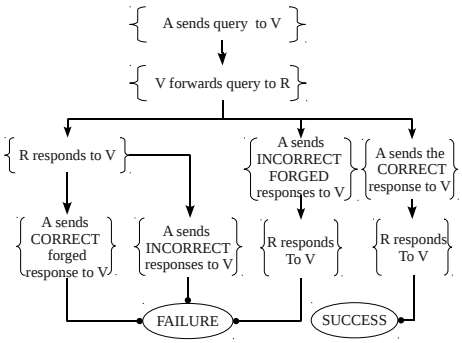
Figure 4: Behavior model for confirming a hypothesis and corresponding behavior instances from network traces satisfying the model.

Referring to the model script shown in Figure 4(a), lines 2–5 define the model header. Line 4 does not specify any qualifying conditions, that is, filters, for the events it can process. Line 5 imports the `IPFLOW` model from the knowledge base. Lines 7–8 define the necessary state propositions. Line 7 defines `sA`, a simple state which just captures an IP packet from some source to destination. Line 8 defines a state `sB` with a dependency that its `dip` has to be equal to the `dip` in `sA`. State `sA` thus provides a context for `sB`.

Line 10 expresses the first hypothesis that there should be more than 60 sources connecting to the same destination for an attack. We apply the $\rightsquigarrow$ operator to denote that we expect `sA` to occur before `sB`. The behavior constraint *bcount* (refer Section 3.4) applied to `sA` limits number of events returned to 1, whereas it is applied to `sB` so that atleast 59 events should occur since the event matching `sA` occurred. Additionally, the operator constraint `[<=1s]`



(a) DNS Kaminsky experiment setup.



(b) Set of possible experiment behaviors.

```
1. [header]
2. NAMESPACE = NET.ATTACKS
3. NAME = DNSKAMINSKY
4. QUALIFIER = {etype='PKT_DNS'}
5. IMPORT = NET.APP_PROTO.DNSREQRES

6. [states]
7. # Attacker to victim query
8. AtoV_query = DNSREQRES.dns_req()

9. # Victim to real ns query
10. VtoR_query= DNSREQRES.dns_req(sip=$AtoV_query.dip,
            dnsquesname=$AtoV_query.dnsquesname)

11.# Real NS to victim real response
12.RtoV_resp = DNSREQRES.dns_res($VtoR_query,
                                 dnsauth=fakens.fake.com)

13.# Attacker to victim CORRECT fake response
14.AtoV_resp = DNSREQRES.dns_res($VtoR_query,
                    dnsauth=realns.eby.com) [bcount>=1]

15.# Attacker to victim INCORRECT response case
16.AtoV_noresp = DNSREQRES.dns_res($VtoR_query,
17.                dnsid != $VtoR_query.dnsid) [bcount>=1]


18.[behavior]
19.initial_query = (AtoV_query ~> VtoR_query)
20.b_1 = initial_query~>RtoV_resp ~> (AtoV_resp xor
                                      AtoV_noresp)
21.b_2 = initial_query ~> AtoV_noresp ~> RtoV_resp
22.b_3 = initial_query ~> AtoV_resp ~> RtoV_resp

23.[model]
24.FAILURE(sip,dip,sport,dport,dnsid,dnsauth) = b_1 or b_2
25.SUCCESS(sip,dip,sport,dport,dnsid,dnsauth) = b_3
```

(c) DNSKAMINSKY models complete experiment behavior.

Figure 5: Experiment setup, possible set of behaviors and corresponding behavior model for validating a networked experiment.

binds `sA` and `sB` to occur within a second in the order specified.

Line 11 defines the second hypothesis that requires that the packet rate be $\geq$ 40,000 by using the `rate` constraint on state proposition `sA`. Lastly, line 13 defines the behavior model `DDOS_HYP` which asserts that either `hyp_1` or `hyp_2` or both are valid. The four attributes `timestamp,sip,dip,etype` are reported in the final output.

When the model is applied to the packet trace, it produces an output as shown Figure 4(b). We see that there are two instances reported matching hypothesis `hyp_1` both with 60 events within a 1 second interval. The output also shows the corresponding state or behavior definitions matching the following events. The two destination IPs that are under attack are 87.231.216.115 and 87.134.184.48. This output is consistent with the findings reported in the original paper [5].

This example clearly demonstrates the ease with which simple hypotheses could be modeled and validated. The original authors wrote about 2,000 lines of C code to identify attacks. The same validation was expressed in about five lines as a behavior model. Additionally, this model can now be shared and easily modified and extended.

5.2 Modeling Experiment Behavior

Running experiments on a testbed, such as DETER [2], is challenging since it is hard to ascertain the validity of the experiment manually. With our framework, a model can be used to capture the "definition of validity" which includes possible successful and failed behaviors for an experiment and then confirmatory analysis can verify if it was met. Such a model can also be easily shared with other experimenters promoting sharing and reuse of experiments.

We present an experiment emulating Dan Kaminsky's popular DNS attack [7] using the metasploit [11] framework. Referring to Figure 5(a), the attackers objective is to poison the cache of the *victimns* so that any requests to *eby.com* are redirected to a fake nameserver (*fakens*) instead of the real nameserver (*realns*). We refer the reader to [7] for a detailed understanding of the attack. Since the attack exploits a race condition, our experiment setup has to permit successful occurrences as well as failed occurrences of the attack.

Figure 5(b) captures the experiment behavior as a tree of possibilities where the nodes are the experiment states and the paths connecting the states are possible experiment behaviors. These states are not exhaustive but sufficient to capture most of the semantics of the experiment. Specifically, we see that there are three possible behaviors that can lead to failures and one behavior that can lead to success.

The behavior model script is shown in Figure 5(c). Lines 2–4 define the model as `DNSKAMINSKY` over events of type `PKT_DNS`. Line 5 imports the `DNSREQRES` model that already defines states and behaviors relevant to the DNS protocol.

Lines 7–17 define five different states that are relevant to the experiment. Line 8 defines the first DNS query from attacker to victim and provides a context for further

```
Summary : DNSKAMINSKY_SUCESS
========================
Total Matching Instances: 1
-------------------------------------------------------------------------------------------
  etype    | timestamp  | sip        |  dip       | sport | dport | dnsid |    dnsauth
-------------------------------------------------------------------------------------------
  PKT_DNS | 1275515488 | 10.1.11.2 | 10.1.4.2 | 38323 |  53   | 59439 |
  PKT_DNS | 1275515488 | 10.1.4.2  | 10.1.6.3 | 32778 |  53   | 59439 |
  PKT_DNS | 1275515488 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 59439 |fakens.fakeeby.com
  PKT_DNS | 1275515488 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 59439 |realns.eby.com
-------------------------------------------------------------------------------------------
Summary : DNSKAMINSKY_FAILURE
========================
Total Matching Instances: 622

<truncated output>
-------------------------------------------------------------------------------------------
  etype    | timestamp  | sip        | dip        | sport | dport | dnsid |    dnsauth
-------------------------------------------------------------------------------------------
  PKT_DNS | 1275515486 | 10.1.11.2 | 10.1.4.2 |  6916 |  53   | 47217 |
  PKT_DNS | 1275515486 | 10.1.4.2  | 10.1.6.3 | 32778 |  53   | 15578 |
  PKT_DNS | 1275515486 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 15578 |realns.eby.com
  PKT_DNS | 1275515486 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 47217 |fakens.fakeeby.com
  PKT_DNS | 1275515486 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 47217 |fakens.fakeeby.com
  PKT_DNS | 1275515486 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 47217 |fakens.fakeeby.com
  PKT_DNS | 1275515486 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 47217 |fakens.fakeeby.com
  PKT_DNS | 1275515486 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 47217 |fakens.fakeeby.com
  PKT_DNS | 1275515486 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 47217 |fakens.fakeeby.com
  PKT_DNS | 1275515486 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 47217 |fakens.fakeeby.com
-------------------------------------------------------------------------------------------
  PKT_DNS | 1275515486 | 10.1.11.2 | 10.1.4.2 | 28902 |  53   | 50921 |
<truncated output>
```

Figure 6: Behavior instances satisfying the DNSKAMINSKY model.

states. Line 10 defines a query from the victim to real nameserver by requiring that the source IP address of this query be same as the destination IP address of the previous query and the DNS questions of both states be identical. This makes sure that the forwarded query by the victim nameserver is the same as the one received. Line 12 defines the response from the real nameserver to the victim nameserver. The response is related to the request in line 10 by using the state identifier of the query state VtoR_query. To specifically distinguish this response from the attacker's response, we mention the value of the dnsauth attribute that is expected in the response. There are two cases for specifying the attacker's response. Line 14 defines the attacker's response same as the real nameserver response except that we mention the fake nameserver as value of the dnsauth attribute. Line 16 defines the case where the attacker's response is incorrect due to a wrongly guessed DNS transaction id. The bcount constraint specifies that any number of responses can be matched since the attacker can send multiple forged responses. Attribute values not defined in the above states default to their definitions in DNSREQRES.

Lines 19–22 specify four possible behaviors corresponding to the four different paths in Figure 5(b). Line 20 uses the xor operator to merge two behavior paths. The other behaviors use the ⇝ operator to capture the causation between the states. Finally, the behavior model is defined in the model section using FAILURE and SUCCESS behaviors. Referring to Figure 5(b), we see that b_1 and b_2, where b_1 is a composite of two behaviors, lead to FAILURE and b_3 leads to SUCCESS. By default, the framework composes the final model by or'ing the behaviors specified in the model section.

After running the experiment and capturing DNS packets, we normalize the last 10,000 packets to PKT_DNS events since they contain a successful attack along with failures representative of rest of the capture. The framework outputs one SUCCESS instance and 622 FAILURE instances as shown in Figure 6.

```
1. scan_A     = {etype=SCAN, src=$infect_A.host, dst=$$}
2. infect_A   = {etype=INFECT, host=$scan_A.dst}
3. single_spread  = (scan_A ~> infect_A)
4. spread_chain   = (single_spread ~> spread_chain)
5. WORMSPREAD(host) = (spread_chain)
```

(a) Modeling the worm infection chain over IDS alerts.

```
1. IMPORT = NET.APP_PROTO.HTTP
2. http_pkt     = HTTP.HTTP_PKT(sip=$$, dip=$$)
3. attack_event = {etype=DOSATTACK,src=$$,dst=http_pkt.dip}
4. http_stream_at100 = ((http_pkt)[rate=100])
5. http_stream_below50  = ((http_pkt)[rate=0:50])
6. attack_start=(http_stream_at100 ew[<= 5s](attack_event))
7. DYNAMIC_CHANGE = (attack_start ~> http_stream_below50)
```

(b) Modeling change in rate of packet streams.

```
1. IMPORT = NET.ATTACKS.DNSKAMINSKY,NET.ATTACKS.WORMSPREAD
2. worm_attack= WORMSPREAD.single_spread(host=$$)
3. dns_attack = DNSKAMINSKY.SUCCESS(sip=$worm_attack.host)
4. COMBINED_ATTACK = (worm_attack ~> (dns_attack))
```

(c) Modeling an attack by composing WORMSPREAD and DNSKAMINSKY models.

Figure 7: Excerpts from behavior models for (a) modeling a security threat, (b) modeling a dynamic change and (c) composing higher-level models. We refer the reader to the framework webpage [17] for details.

This case study demonstrates the ease with which the full system behavior was semantically modeled at the level of user's understanding. Additionally, the model was composed using existing models from the knowledge base, extended with user's context-specific values for attributes and then validated.

5.3 Modeling a Security Threat

In this case study, we define a behavior model of a typical worm spread detected by IDS alerts collected from multiple hosts. Assume a network with IDSes on each host reporting two types of timestamped alerts: a SCAN alert when a scan is detected by a host and an INFECT alert when the host is found infected. Assume an event log created by normalizing the alerts to two types of events with their corresponding attributes. Given the event log, our objective here is to define a behavior model to extract all possible infection chains of any length and report the hosts involved.

We model the worm spread behavior as shown in Figure 7(a) in two stages; by first defining a single_spread behavior using events from a single host and then defining the spread_chain as a chain of related single_spread occurrences. The single_spread behavior, concerning a vulnerable host A, is a sequence of two dependent and casual events: (a) a scan_A event with its src attribute pointing to an earlier infected host, followed by (b) an infect_A event with its host attribute the same as scan_A.dst. A worm spread chain (spread_chain) is then simply defined by a recursive occurrence of related single_spread behaviors. Referring to the model, the forward-dependent attribute src in the definition of scan_A connects successive single_spread behaviors by requiring the src

of the next scan to be the same as the previously infected host. The forward-dependent attribute `src` is initialized automatically the first time `single_spread` is parsed by considering it to be a dynamic ($$) variable. The next iteration over `spread_chain` then uses the values as determined dynamically by `single_spread`.

5.4 Modeling Dynamic Change

Dynamic changes are a fundamental characteristic of networked and distributed environments. One example of a dynamic change is the change in rate of a stream of packets due to an anomalous condition such as a DoS attack. Our objective in this case study is to model an expected reduction in the rate of legitimate HTTP traffic due to DoS attack on a server. Our raw data consists of IDS DoS attack alerts and HTTP packets.

The `DYNAMIC_CHANGE` model, containing only the relevant aspects is described in Figure 7(b). Line 2 defines a state capturing a HTTP packet between a source and destination. Line 3 defines a state capturing a DoS attack alert, additionally requiring the destination to be same as the destination in the HTTP packet. Lines 4 and 5 describe the HTTP packet stream rates before and after the attack respectively. The change boundary is defined by the `attack_event` that is triggered once the attack starts. Since `attack_event` represents a single event, it has the same starttime and endtime. Line 6 use the `ew` (endswith) operator to define the `attack_start` condition, which specifies that the `http_stream_at100` behavior end within five seconds of the `attack_event`. The `DYNAMIC_CHANGE` model is then an assertion that the HTTP stream rate reduces following the attack.

5.5 Composing Models

Our final case study demonstrates the ease of composing and extending existing models to define semantically relevant higher-level behavior.

We combine our previously defined models `DNSKAMINSKY` and `WORMSPREAD` to create a `COMBINED_ATTACK` scenario as shown in Figure 7(c). Line 2 captures the behavior where a worm infects a host machine and scans and infects another host. Line 3 describes the behavior where the worm launches a DNS Kaminsky attack on some DNS server from the last infected host. We do not specify any server for the DNS Kaminsky attack due to the abstractness of the `DNSKAMINSKY` model which infers the destination dynamically. Line 4 is the final behavior model combining both the attacks. In line 3, we only constrain the `sip` and leave other attributes unspecified. This demonstrates the ability to extend the imported models with only the desired attribute values while leaving the others as defined in the imported model.

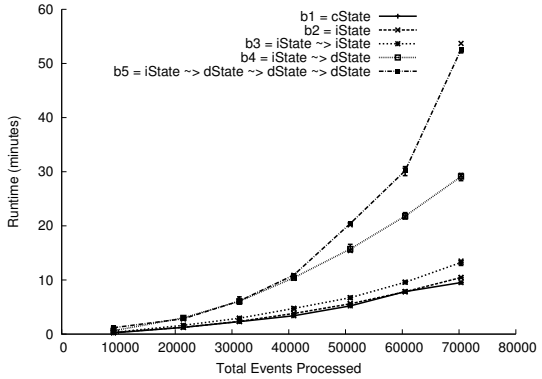


Figure 8: Plot of runtime against number of events for five types of behavior complexity. Behaviors containing dependent value states (dStates) result in quadratic complexity.

6 Performance Analysis

A common approach for semantic-level analysis involves use of custom scripts or tools encoding context-specific semantics. Since custom scripts and tools can be written using a variety of programming and optimization techniques, any evaluation of our generic framework against them would be very subjective and thus flawed. Instead, we choose to report the raw runtime performance of our prototype implementation on five basic analyses tasks over event datasets of increasing size.

The runtime performance of the framework depends on the language constructs, input data, analysis algorithm and implementation mechanisms used. Since our primary focus in this paper is on enabling semantic functionality, we prototyped the framework in Python using a SQLite database as backend for storing events. The input events used were `PKT_DNS` events collected for the case study in Section 5.2. The performance analysis was conducted on a laptop with an Intel Pentium-M processor running at 1.86 GHz and with a memory of 2 GB.

We measure runtime as a function of two variables: (a) the number of events input to the algorithm, (b) the *behavior complexity*, defined as the processing complexity of state propositions in a behavior formula. As discussed in Section 3.3, there are three types of state propositions based on attribute assignments; constant value attributes denoted as *cState*, dependent value attributes denoted as *dState*, and dynamic attribute values denoted as *iState*. These states can be combined to form five basic behaviors, each representing a basic semantic analysis task: b1 = (cState), represents extracting events with known attributes and values; b2 = (iState), represents extracting events with particular attributes but unknown values; b3 = (iState $\rightsquigarrow$ iState), represents extracting causally correlated yet value-independent events; b4 = (iState $\rightsquigarrow$ dState), represents extracting causally correlated and value-dependent events; and b5 = (iState $\rightsquigarrow$

dState $\rightsquigarrow$ dState $\rightsquigarrow$ dState), represents extracting a long chain of causal events. Although we limit our analysis to the $\rightsquigarrow$ operator, all operators incur uniform processing overhead in the algorithm, thus resulting in similar performance results. The chosen event set along with the behaviors are representative of a worst-case input to the framework. We measure the performance using above behaviors over event sets in increments of 10,000 events. We stop at the event set when runtime exceeds 60 minutes.

The results are averaged over three runs and are shown in Figure 8. The plots for behaviors consisting of *cStates* and *iStates* b1, b2 and b3 tend to be linear as discussed in Section 4.4. One would expect that behavior b5, containing three *dStates* would show significantly higher runtime than behavior b4 containing only one *dState*. Both show quadratic performance, since, in a chain of dependent states, the states further in the chain process lesser events than states in front of the chain. We thus see that runtime quickly becomes quadratic given a worst-case set of events and behaviors containing dependent state propositions. The current Python and SQLite-based implementation also add penalty to the framework runtime. We investigate these issues as part of our future work.

7 Conclusion and Future Work

In this paper, we presented a behavior-based semantic analysis framework that allows the user to analyze data at a higher-level of abstraction. Typically, system experts rely on their intuition and experience to manually analyze and categorize scenarios and then hand-craft rules and patterns for analysis. Hence due to the manual and ad-hoc nature of this analysis process, there is limited extensibility and composibility of analysis strategies. In this paper we show that our approach is more systematic, can retain expert knowledge, and supports composing behaviors from existing models. We evaluated the utility of our framework against five analyses scenarios which demonstrated the ease with which a user's higher-level understanding of system operation was expressed as behavior models over data.

Our future work includes investigating the scale and efficiency issues that arise during processing large volumes of data in both offline and real-time settings like intrusion detection. We will investigate stream-based SQL query extensions [6] to improve performance. We will also investigate extending our logic with existential and universal quantifiers. Currently, our framework requires a user to either manually specify behavior models or use existing models from the knowledge base to explore data. To further exploratory analysis, we would need to alert users to interesting unanticipated behaviors. We are exploring data mining algorithms to automatically discover and compose behavior models from data.

The fundamental goal of the behavior-based semantic analysis framework is to introduce a semantic approach to data analysis in networked and distributed systems research and operations. We hope that this paper serves as a catalyst for further research on semantic data analysis.

References

[1] ALLEN, J. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM 26*, 11 (Nov. 1983), 832–843.

[2] BENZEL, T., BRADEN, R., KIM, D., NEUMAN, C., JOSEPH, A., SKLOWER, K., OSTRENGA, R., AND SCHWAB, S. Experience with DETER: A Testbed for Security Research. In *2nd Intl. Conf. on Testbeds and Research Infrastructures for the Devel. of Networks and Communities - TRIDENTCOM* (2006), p. 10.

[3] BÉRARD, B. *Systems and Software Verification: Model-checking Techniques and Tools*. Springer, 2001.

[4] ELLIS, D. R., AIKEN, J. G., ATTWOOD, K. S., AND TENAGLIA, S. D. A Behavioral Approach to Worm Detection. In *Proc. of the ACM workshop on Rapid malcode* (2004), pp. 43–53.

[5] HUSSAIN, A., HEIDEMANN, J., AND PAPADOPOULOS, C. A Framework For Classifying Denial of Service Attacks. *Proc. of the Conf. on Applications, Technologies, Architectures, and Protocols for Comp. Comm. - SIGCOMM* (2003), 99.

[6] JAIN, N., MISHRA, S., SRINIVASAN, A., GEHRKE, J., WIDOM, J., BALAKRISHNAN, H., ÇETINTEMEL, U., CHERNIACK, M., TIBBETTS, R., AND ZDONIK, S. Towards a Streaming SQL Standard. *Proc. VLDB Endow. 1* (August 2008), 1379–1390.

[7] KAMINSKY, D. Multiple DNS Implementations Vulnerable to Cache Poisoning. http://www.kb.cert.org/vuls/id/800113, 2008.

[8] KINDER, J., KATZENBEISSER, S., SCHALLHART, C., AND VEITH, H. Detecting Malicious Code by Model Checking. In *Intrusion and Malware Detection and Vuln. Assessment*, K. Julisch and C. Kruegel, Eds., vol. 3548 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, pp. 174–187.

[9] LAMPORT, L. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst. 16*, 3 (1994), 872–923.

[10] LOO, B. T., CONDIE, T., GAROFALAKIS, M., GAY, D. E., HELLERSTEIN, J. M., MANIATIS, P., RAMAKRISHNAN, R., ROSCOE, T., AND STOICA, I. Declarative Networking: Language, Execution and Optimization. In *Proc. of ACM SIGMOD* (2006), pp. 97–108.

[11] Metasploit Framework Website. http://www.metasploit.com/.

[12] MIRKOVIC, J., SOLLINS, K., AND WROCLAWSKI, J. Managing the Health of Security Experiments. In *Proc. of the conf. on Cyber Security Experimentation and Test* (2008), USENIX, pp. 7:1–7:6.

[13] NALDURG, P., SEN, K., AND THATI, P. A Temporal Logic Based Framework for Intrusion Detection. In *Proc. of the 24th IFIP Intl. Conf. on Formal Tech. for Net. & Dist. Sys.* (2004).

[14] PAXSON, V. Bro: A System for Detecting Network Intruders in Real-time. *Comput. Networks 31*, 23-24 (1999), 2435–2463.

[15] ROGER, M., AND GOUBAULT-LARRECQ, J. Log Auditing through Model-Checking. In *Proc. of the 14th IEEE Computer Security Foundations Workshop* (2001), pp. 220–236.

[16] Splunk Website. http://www.splunk.com/.

[17] Semantic Analysis Framework Website. http://thirdeye.isi.deterlab.net/.

[18] VAARANDI, R. SEC - A Lightweight Event Correlation Tool. *IEEE Workshop on IP Operations and Management* (2002), 111 – 115.

[19] Wireshark Website. http://www.wireshark.org/.