



UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NUMBER	FILING or 371(c) DATE	GRP ART UNIT	FIL FEE REC'D	ATTY. DOCKET NO	TOT CLAIMS	IND CLAIMS
63/240,818	09/03/2021		300	LACEP040		

CONFIRMATION NO. 1130

FILING RECEIPT

182115
Lacework, Inc.
c/o Kennedy Lenart Spraggins LLP
797 Sam Bass Road #2559
ROUND ROCK, TX 78681



0000000128339698

Date Mailed: 09/14/2021

Receipt is acknowledged of this provisional patent application. It will not be examined for patentability and will become abandoned not later than twelve months after its filing date. Any correspondence concerning the application must include the following identification information: the U.S. APPLICATION NUMBER, FILING DATE, NAME OF FIRST INVENTOR, and TITLE OF INVENTION. Fees transmitted by check or draft are subject to collection.

Please verify the accuracy of the data presented on this receipt. If an error is noted on this Filing Receipt, please submit a written request for a corrected Filing Receipt identifying the requested changes, preferably by including a properly marked-up ADS showing the changes with strike-through for deletions and underlining for additions. If you received a "Notice to File Missing Parts" or other Notice requiring a response for this application, please submit any request for correction to this Filing Receipt with your reply to the Notice. When the USPTO processes the reply to the Notice, the USPTO will generate another Filing Receipt incorporating the requested corrections provided that the request is grantable.

Inventor(s)

VIKRAM KAPOOR, CUPERTINO, CA;
HARISH KUMAR BHARAT SINGH, MOUNTAIN VIEW, CA;

Applicant(s)

LACEWORK, INC., SAN JOSE, CA;

Power of Attorney:

Edward Lenart--63665

Permission to Access Application via Priority Document Exchange: Yes

Permission to Access Search Results: Yes

Applicant may provide or rescind an authorization for access using Form PTO/SB/39 or Form PTO/SB/69 as appropriate.

If Required, Foreign Filing License Granted: 09/13/2021

The country code and number of your priority application, to be used for filing abroad under the Paris Convention, is **US 63/240,818**

Projected Publication Date: None, application is not eligible for pre-grant publication

Non-Publication Request: No

Early Publication Request: No

Title

EDGE BASED DATA PLATFORM

Statement under 37 CFR 1.55 or 1.78 for AIA (First Inventor to File) Transition Applications: No**PROTECTING YOUR INVENTION OUTSIDE THE UNITED STATES**

Since the rights granted by a U.S. patent extend only throughout the territory of the United States and have no effect in a foreign country, an inventor who wishes patent protection in another country must apply for a patent in a specific country or in regional patent offices. Applicants may wish to consider the filing of an international application under the Patent Cooperation Treaty (PCT). An international (PCT) application generally has the same effect as a regular national patent application in each PCT-member country. The PCT process **simplifies** the filing of patent applications on the same invention in member countries, but **does not result** in a grant of "an international patent" and does not eliminate the need of applicants to file additional documents and fees in countries where patent protection is desired.

Almost every country has its own patent law, and a person desiring a patent in a particular country must make an application for patent in that country in accordance with its particular laws. Since the laws of many countries differ in various respects from the patent law of the United States, applicants are advised to seek guidance from specific foreign countries to ensure that patent rights are not lost prematurely.

Applicants also are advised that in the case of inventions made in the United States, the Director of the USPTO must issue a license before applicants can apply for a patent in a foreign country. The filing of a U.S. patent application serves as a request for a foreign filing license. The application's filing receipt contains further information and guidance as to the status of applicant's license for foreign filing.

Applicants may wish to consult the USPTO booklet, "General Information Concerning Patents" (specifically, the section entitled "Treaties and Foreign Patents") for more information on timeframes and deadlines for filing foreign patent applications. The guide is available either by contacting the USPTO Contact Center at 800-786-9199, or it can be viewed on the USPTO website at <http://www.uspto.gov/web/offices/pac/doc/general/index.html>.

For information on preventing theft of your intellectual property (patents, trademarks and copyrights), you may wish to consult the U.S. Government website, <http://www.stopfakes.gov>. Part of a Department of Commerce initiative, this website includes self-help "toolkits" giving innovators guidance on how to protect intellectual property in specific countries such as China, Korea and Mexico. For questions regarding patent enforcement issues, applicants may call the U.S. Government hotline at 1-866-999-HALT (1-866-999-4258).

LICENSE FOR FOREIGN FILING UNDER**Title 35, United States Code, Section 184****Title 37, Code of Federal Regulations, 5.11 & 5.15****GRANTED**

The applicant has been granted a license under 35 U.S.C. 184, if the phrase "IF REQUIRED, FOREIGN FILING LICENSE GRANTED" followed by a date appears on this form. Such licenses are issued in all applications where the conditions for issuance have been met, regardless of whether or not a license may be required as set forth in 37 CFR 5.15. The scope and limitations of this license are set forth in 37 CFR 5.15(a) unless an earlier

license has been issued under 37 CFR 5.15(b). The license is subject to revocation upon written notification. The date indicated is the effective date of the license, unless an earlier license of similar scope has been granted under 37 CFR 5.13 or 5.14.

This license is to be retained by the licensee and may be used at any time on or after the effective date thereof unless it is revoked. This license is automatically transferred to any related applications(s) filed under 37 CFR 1.53(d). This license is not retroactive.

The grant of a license does not in any way lessen the responsibility of a licensee for the security of the subject matter as imposed by any Government contract or the provisions of existing laws relating to espionage and the national security or the export of technical data. Licensees should apprise themselves of current regulations especially with respect to certain countries, of other agencies, particularly the Office of Defense Trade Controls, Department of State (with respect to Arms, Munitions and Implements of War (22 CFR 121-128)); the Bureau of Industry and Security, Department of Commerce (15 CFR parts 730-774); the Office of Foreign Assets Control, Department of Treasury (31 CFR Parts 500+) and the Department of Energy.

NOT GRANTED

No license under 35 U.S.C. 184 has been granted at this time, if the phrase "IF REQUIRED, FOREIGN FILING LICENSE GRANTED" DOES NOT appear on this form. Applicant may still petition for a license under 37 CFR 5.12, if a license is desired before the expiration of 6 months from the filing date of the application. If 6 months has lapsed from the filing date of this application and the licensee has not received any indication of a secrecy order under 35 U.S.C. 181, the licensee may foreign file the application pursuant to 37 CFR 5.15(b).

SelectUSA

The United States represents the largest, most dynamic marketplace in the world and is an unparalleled location for business investment, innovation, and commercialization of new technologies. The U.S. offers tremendous resources and advantages for those who invest and manufacture goods here. Through SelectUSA, our nation works to promote and facilitate business investment. SelectUSA provides information assistance to the international investor community; serves as an ombudsman for existing and potential investors; advocates on behalf of U.S. cities, states, and regions competing for global investment; and counsels U.S. economic development organizations on investment attraction best practices. To learn more about why the United States is the best country in the world to develop technology, manufacture products, deliver services, and grow your business, visit <http://www.SelectUSA.gov> or call +1-202-482-6800.

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it contains a valid OMB control number.

Application Data Sheet 37 CFR 1.76		Attorney Docket Number	LACEP040
		Application Number	
Title of Invention	EDGE BASED DATA PLATFORM		
The application data sheet is part of the provisional or nonprovisional application for which it is being submitted. The following form contains the bibliographic data arranged in a format specified by the United States Patent and Trademark Office as outlined in 37 CFR 1.76. This document may be completed electronically and submitted to the Office in electronic format using the Electronic Filing System (EFS) or the document may be printed and included in a paper filed application.			

Secrecy Order 37 CFR 5.2:

☐ Portions or all of the application associated with this Application Data Sheet may fall under a Secrecy Order pursuant to 37 CFR 5.2 (Paper filers only. Applications that fall under Secrecy Order may not be filed electronically.)

Inventor Information:

Inventor	1				Remove	
Legal Name						
Prefix	Given Name	Middle Name	Family Name	Suffix		
	VIKRAM		KAPOOR			
Residence Information (Select One) • US Residency Non US Residency Active US Military Service						
City	CUPERTINO	State/Province	CA	Country of Residence	US	
Mailing Address of Inventor:						
Address 1	6201 AMERICA CENTER DRIVE					
Address 2	SUITE 200					
City	SAN JOSE	State/Province	CA			
Postal Code	95002	Country	US			
Inventor	2				Remove	
Legal Name						
Prefix	Given Name	Middle Name	Family Name	Suffix		
	HARISH KUMAR BHARAT		SINGH			
Residence Information (Select One) • US Residency Non US Residency Active US Military Service						
City	MOUNTAIN VIEW	State/Province	CA	Country of Residence	US	
Mailing Address of Inventor:						
Address 1	6201 AMERICA CENTER DRIVE					
Address 2	SUITE 200					
City	SAN JOSE	State/Province	CA			
Postal Code	95002	Country	US			
All Inventors Must Be Listed - Additional Inventor Information blocks may be generated within this form by selecting the Add button.						
Add						

Correspondence Information:

Enter either Customer Number or complete the Correspondence Information section below.
For further information see 37 CFR 1.33(a).

Application Data Sheet 37 CFR 1.76		Attorney Docket Number	LACEP040
		Application Number	
Title of Invention	EDGE BASED DATA PLATFORM		

☐ An Address is being provided for the correspondence information of this application.

Customer Number	182115		
Email Address		Add Email	Remove Email

Application Information:

Title of the Invention	EDGE BASED DATA PLATFORM		
Attorney Docket Number	LACEP040	Small Entity Status Claimed	☐
Application Type	Provisional		
Subject Matter	Utility		
Total Number of Drawing Sheets (if any)	16	Suggested Figure for Publication (if any)	

Filing By Reference:

Only complete this section when filing an application by reference under 35 U.S.C. 111(c) and 37 CFR 1.57(a). Do not complete this section if application papers including a specification and any drawings are being filed. Any domestic benefit or foreign priority information must be provided in the appropriate section(s) below (i.e., "Domestic Benefit/National Stage Information" and "Foreign Priority Information").

For the purposes of a filing date under 37 CFR 1.53(b), the description and any drawings of the present application are replaced by this reference to the previously filed application, subject to conditions and requirements of 37 CFR 1.57(a).

Application number of the previously filed application	Filing date (YYYY-MM-DD)	Intellectual Property Authority or Country

Publication Information:

☐ Request Early Publication (Fee required at time of Request 37 CFR 1.219)

☐ **Request Not to Publish.** I hereby request that the attached application not be published under 35 U.S.C. 122(b) and certify that the invention disclosed in the attached application **has not and will not** be the subject of an application filed in another country, or under a multilateral international agreement, that requires publication at eighteen months after filing.

Representative Information:

Representative information should be provided for all practitioners having a power of attorney in the application. Providing this information in the Application Data Sheet does not constitute a power of attorney in the application (see 37 CFR 1.32). Either enter Customer Number or complete the Representative Name section below. If both sections are completed the customer Number will be used for the Representative Information during processing.

Please Select One:	☒ Customer Number	☐ US Patent Practitioner	☐ Limited Recognition (37 CFR 11.9)
Customer Number	182115		

Application Data Sheet 37 CFR 1.76		Attorney Docket Number	LACEP040
		Application Number	
Title of Invention	EDGE BASED DATA PLATFORM		

Domestic Benefit/National Stage Information:

This section allows for the applicant to either claim benefit under 35 U.S.C. 119(e), 120, 121, 365(c), or 386(c) or indicate National Stage entry from a PCT application. Providing benefit claim information in the Application Data Sheet constitutes the specific reference required by 35 U.S.C. 119(e) or 120, and 37 CFR 1.78.

When referring to the current application, please leave the "Application Number" field blank.

Prior Application Status		Remove	
Application Number	Continuity Type	Prior Application Number	Filing or 371(c) Date (YYYY-MM-DD)
Additional Domestic Benefit/National Stage Data may be generated within this form by selecting the Add button.			Add

Foreign Priority Information:

This section allows for the applicant to claim priority to a foreign application. Providing this information in the application data sheet constitutes the claim for priority as required by 35 U.S.C. 119(b) and 37 CFR 1.55. When priority is claimed to a foreign application that is eligible for retrieval under the priority document exchange program (PDX)ⁱ the information will be used by the Office to automatically attempt retrieval pursuant to 37 CFR 1.55(i)(1) and (2). Under the PDX program, applicant bears the ultimate responsibility for ensuring that a copy of the foreign application is received by the Office from the participating foreign intellectual property office, or a certified copy of the foreign priority application is filed, within the time period specified in 37 CFR 1.55(g)(1).

			Remove
Application Number	Country ⁱ	Filing Date (YYYY-MM-DD)	Access Code ⁱ (if applicable)
Additional Foreign Priority Data may be generated within this form by selecting the Add button.			Add

Statement under 37 CFR 1.55 or 1.78 for AIA (First Inventor to File) Transition Applications

This application (1) claims priority to or the benefit of an application filed before March 16, 2013 and (2) also contains, or contained at any time, a claim to a claimed invention that has an effective filing date on or after March 16, 2013.

☐ NOTE: By providing this statement under 37 CFR 1.55 or 1.78, this application, with a filing date on or after March 16, 2013, will be examined under the first inventor to file provisions of the AIA.

Application Data Sheet 37 CFR 1.76		Attorney Docket Number	LACEP040
		Application Number	
Title of Invention	EDGE BASED DATA PLATFORM		

Authorization or Opt-Out of Authorization to Permit Access:

When this Application Data Sheet is properly signed and filed with the application, applicant has provided written authority to permit a participating foreign intellectual property (IP) office access to the instant application-as-filed (see paragraph A in subsection 1 below) and the European Patent Office (EPO) access to any search results from the instant application (see paragraph B in subsection 1 below).

Should applicant choose not to provide an authorization identified in subsection 1 below, applicant **must opt-out** of the authorization by checking the corresponding box A or B or both in subsection 2 below.

NOTE: This section of the Application Data Sheet is **ONLY** reviewed and processed with the **INITIAL** filing of an application. After the initial filing of an application, an Application Data Sheet cannot be used to provide or rescind authorization for access by a foreign IP office(s). Instead, Form PTO/SB/39 or PTO/SB/69 must be used as appropriate.

1. Authorization to Permit Access by a Foreign Intellectual Property Office(s)

A. Priority Document Exchange (PDX) - Unless box A in subsection 2 (opt-out of authorization) is checked, the undersigned hereby **grants the USPTO authority** to provide the European Patent Office (EPO), the Japan Patent Office (JPO), the Korean Intellectual Property Office (KIPO), the State Intellectual Property Office of the People's Republic of China (SIPO), the World Intellectual Property Organization (WIPO), and any other foreign intellectual property office participating with the USPTO in a bilateral or multilateral priority document exchange agreement in which a foreign application claiming priority to the instant patent application is filed, access to: (1) the instant patent application-as-filed and its related bibliographic data, (2) any foreign or domestic application to which priority or benefit is claimed by the instant application and its related bibliographic data, and (3) the date of filing of this Authorization. See 37 CFR 1.14(h)(1).

B. Search Results from U.S. Application to EPO - Unless box B in subsection 2 (opt-out of authorization) is checked, the undersigned hereby **grants the USPTO authority** to provide the EPO access to the bibliographic data and search results from the instant patent application when a European patent application claiming priority to the instant patent application is filed. See 37 CFR 1.14(h)(2).

The applicant is reminded that the EPO's Rule 141(1) EPC (European Patent Convention) requires applicants to submit a copy of search results from the instant application without delay in a European patent application that claims priority to the instant application.

2. Opt-Out of Authorizations to Permit Access by a Foreign Intellectual Property Office(s)

☐ A. Applicant **DOES NOT** authorize the USPTO to permit a participating foreign IP office access to the instant application-as-filed. If this box is checked, the USPTO will not be providing a participating foreign IP office with any documents and information identified in subsection 1A above.

☐ B. Applicant **DOES NOT** authorize the USPTO to transmit to the EPO any search results from the instant patent application. If this box is checked, the USPTO will not be providing the EPO with search results from the instant application.

NOTE: Once the application has published or is otherwise publicly available, the USPTO may provide access to the application in accordance with 37 CFR 1.14.

Application Data Sheet 37 CFR 1.76		Attorney Docket Number	LACEP040
		Application Number	
Title of Invention	EDGE BASED DATA PLATFORM		

Applicant Information:

Providing assignment information in this section does not substitute for compliance with any requirement of part 3 of Title 37 of CFR to have an assignment recorded by the Office.			
Applicant	1 Remove		
If the applicant is the inventor (or the remaining joint inventor or inventors under 37 CFR 1.45), this section should not be completed. The information to be provided in this section is the name and address of the legal representative who is the applicant under 37 CFR 1.43; or the name and address of the assignee, person to whom the inventor is under an obligation to assign the invention, or person who otherwise shows sufficient proprietary interest in the matter who is the applicant under 37 CFR 1.46. If the applicant is an applicant under 37 CFR 1.46 (assignee, person to whom the inventor is obligated to assign, or person who otherwise shows sufficient proprietary interest) together with one or more joint inventors, then the joint inventor or inventors who are also the applicant should be identified in this section. Clear			
☐ Assignee	Legal Representative under 35 U.S.C. 117		Joint Inventor
Person to whom the inventor is obligated to assign.		Person who shows sufficient proprietary interest	
If applicant is the legal representative, indicate the authority to file the patent application, the inventor is:			
Name of the Deceased or Legally Incapacitated Inventor:			
If the Applicant is an Organization check here. ☒			
Organization Name	LACEWORK, INC.		
Mailing Address Information For Applicant:			
Address 1	6201 AMERICA CENTER DRIVE		
Address 2	SUITE 200		
City	SAN JOSE	State/Province	CA
Country	US	Postal Code	95002
Phone Number		Fax Number	
Email Address			
Additional Applicant Data may be generated within this form by selecting the Add button. Add			

Assignee Information including Non-Applicant Assignee Information:

Providing assignment information in this section does not substitute for compliance with any requirement of part 3 of Title 37 of CFR to have an assignment recorded by the Office.

Application Data Sheet 37 CFR 1.76		Attorney Docket Number	LACEP040
		Application Number	
Title of Invention	EDGE BASED DATA PLATFORM		

Assignee	1			
Complete this section if assignee information, including non-applicant assignee information, is desired to be included on the patent application publication. An assignee-applicant identified in the "Applicant Information" section will appear on the patent application publication as an applicant. For an assignee-applicant, complete this section only if identification as an assignee is also desired on the patent application publication.				
				Remove
If the Assignee or Non-Applicant Assignee is an Organization check here.				☐
Prefix	Given Name	Middle Name	Family Name	Suffix
Mailing Address Information For Assignee including Non-Applicant Assignee:				
Address 1				
Address 2				
City		State/Province		
Country ⁱ		Postal Code		
Phone Number		Fax Number		
Email Address				
Additional Assignee or Non-Applicant Assignee Data may be generated within this form by selecting the Add button.				Add

Signature:

NOTE: This Application Data Sheet must be signed in accordance with 37 CFR 1.33(b). However, if this Application Data Sheet is submitted with the **INITIAL** filing of the application and either box A or B is not checked in subsection 2 of the "Authorization or Opt-Out of Authorization to Permit Access" section, then this form must also be signed in accordance with 37 CFR 1.14(c).

This Application Data Sheet **must** be signed by a patent practitioner if one or more of the applicants is a **juristic entity** (e.g., corporation or association). If the applicant is two or more joint inventors, this form must be signed by a patent practitioner, **all** joint inventors who are the applicant, or one or more joint inventor-applicants who have been given power of attorney (e.g., see USPTO Form PTO/AIA/81) on behalf of **all** joint inventor-applicants.

See 37 CFR 1.4(d) for the manner of making signatures and certifications.

Signature	/Edward J. Lenart/		Date (YYYY-MM-DD)	2021-09-03
First Name	Edward J.	Last Name	Lenart	Registration Number
				63665
Additional Signature may be generated within this form by selecting the Add button.				Add

Application Data Sheet 37 CFR 1.76		Attorney Docket Number	LACEP040
		Application Number	
Title of Invention	EDGE BASED DATA PLATFORM		

This collection of information is required by 37 CFR 1.76. The information is required to obtain or retain a benefit by the public which is to file (and by the USPTO to process) an application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.14. This collection is estimated to take 23 minutes to complete, including gathering, preparing, and submitting the completed application data sheet form to the USPTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, P.O. Box 1450, Alexandria, VA 22313-1450. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. **SEND TO: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.**

Privacy Act Statement

The Privacy Act of 1974 (P.L. 93-579) requires that you be given certain information in connection with your submission of the attached form related to a patent application or patent. Accordingly, pursuant to the requirements of the Act, please be advised that: (1) the general authority for the collection of this information is 35 U.S.C. 2(b)(2); (2) furnishing of the information solicited is voluntary; and (3) the principal purpose for which the information is used by the U.S. Patent and Trademark Office is to process and/or examine your submission related to a patent application or patent. If you do not furnish the requested information, the U.S. Patent and Trademark Office may not be able to process and/or examine your submission, which may result in termination of proceedings or abandonment of the application or expiration of the patent.

The information provided by you in this form will be subject to the following routine uses:

1. The information on this form will be treated confidentially to the extent allowed under the Freedom of Information Act (5 U.S.C. 552) and the Privacy Act (5 U.S.C. 552a). Records from this system of records may be disclosed to the Department of Justice to determine whether the Freedom of Information Act requires disclosure of these records.
2. A record from this system of records may be disclosed, as a routine use, in the course of presenting evidence to a court, magistrate, or administrative tribunal, including disclosures to opposing counsel in the course of settlement negotiations.
3. A record in this system of records may be disclosed, as a routine use, to a Member of Congress submitting a request involving an individual, to whom the record pertains, when the individual has requested assistance from the Member with respect to the subject matter of the record.
4. A record in this system of records may be disclosed, as a routine use, to a contractor of the Agency having need for the information in order to perform a contract. Recipients of information shall be required to comply with the requirements of the Privacy Act of 1974, as amended, pursuant to 5 U.S.C. 552a(m).
5. A record related to an International Application filed under the Patent Cooperation Treaty in this system of records may be disclosed, as a routine use, to the International Bureau of the World Intellectual Property Organization, pursuant to the Patent Cooperation Treaty.
6. A record in this system of records may be disclosed, as a routine use, to another federal agency for purposes of National Security review (35 U.S.C. 181) and for review pursuant to the Atomic Energy Act (42 U.S.C. 218(c)).
7. A record from this system of records may be disclosed, as a routine use, to the Administrator, General Services, or his/her designee, during an inspection of records conducted by GSA as part of that agency's responsibility to recommend improvements in records management practices and programs, under authority of 44 U.S.C. 2904 and 2906. Such disclosure shall be made in accordance with the GSA regulations governing inspection of records for this purpose, and any other relevant (i.e., GSA or Commerce) directive. Such disclosure shall not be used to make determinations about individuals.
8. A record from this system of records may be disclosed, as a routine use, to the public after either publication of the application pursuant to 35 U.S.C. 122(b) or issuance of a patent pursuant to 35 U.S.C. 151. Further, a record may be disclosed, subject to the limitations of 37 CFR 1.14, as a routine use, to the public if the record was filed in an application which became abandoned or in which the proceedings were terminated and which application is referenced by either a published application, an application open to public inspections or an issued patent.
9. A record from this system of records may be disclosed, as a routine use, to a Federal, State, or local law enforcement agency, if the USPTO becomes aware of a violation or potential violation of law or regulation.

EDGE BASED DATA PLATFORM

BRIEF DESCRIPTION OF DRAWINGS

Figure 1 illustrates an example of an environment.
Figure 2 illustrates an example of a process to collect and report information.
Figure 3A illustrates an example of static information collected by an agent.
Figure 3B illustrates an example of variable information collected by an agent.
Figures 3C and 3D illustrate an example of histogram data.
Figure 4 illustrates a 5-tuple of data collected by an agent, physically and logically.
Figures 5-6 illustrate a portion of a polygraph.
Figure 7 illustrates an example of a communication polygraph.
Figure 8 illustrates an example of a polygraph.
Figure 9 illustrates an example of a polygraph as rendered in an interface.
Figures 10-13 illustrates portions of a polygraph as rendered in an interface.
Figure 14 illustrates an example of an insider behavior graph as rendered in an interface.
Figure 15 illustrates an example of a privilege change graph as rendered in an interface.
Figure 16 illustrates an example of a user login graph as rendered in an interface.
Figure 17 illustrates an example of a machine server graph as rendered in an interface.
Figure 18 illustrates a process for detecting anomalies in a network environment.
Figures 19A-C depicts a set of example processes communicating with other processes.
Figure 19D depicts two pairs of clusters.
Figure 20 is a representation of user actions.
Figure 21 is an alternate representation of actions occurring in Figure 20.
Figure 22 and 24 illustrate examples of a process for performing extended user tracking.
Figure 23 is a representation of user actions.
Figure 25A and 25C-F illustrates example records.
Figure 25B illustrates example output from performing an SSH connection match.
Figure 25G illustrates an adjacency relationship between two login sessions.
Figure 25H illustrates example records.
Figure 26 illustrates an example of a process for detecting anomalies.
Figure 27A illustrates a representation of an embodiment of an insider behavior graph.
Figure 27B illustrates an embodiment of a portion of an insider behavior graph.
Figure 28A-B illustrates an embodiment of a portion of an insider behavior graph.
Figure 29 illustrates a representation of an embodiment of a user login graph.
Figure 30 illustrates a representation of a process tree.
Figures 31-32 illustrates an example of a privilege change graph.
Figure 33 illustrates an example of a user interacting with a portion of an interface.
Figure 34 illustrates an example of a dossier for an event.
Figure 35 illustrates an example of a dossier for a domain.
Figure 36 illustrates an example of a card specification.
Figure 37 illustrates an example of three user tracking data sources.
Figures 38-39 depicts an example of card schema introspection.
Figure 40 depicts an example of an Entity Join Graph.
Figure 41 depicts an example introspection for to the card depicted in Figure 36.
Figure 42 depicts an example query service log for a join path search.
Figure 43 depicts an example of dynamically generated SQL.
Figure 44 illustrates an example introspection corresponding to Figure 37.
Figure 45 depicts an example query service log for a join path search.
Figure 46 depicts an example of dynamically generated SQL.
Figure 47A depicts a filter request.
Figure 47B depicts an SQL translation.
Figure 47C depicts a join.
Figure 47D depicts an SQL translation.
Figure 48 illustrates a process for dynamically generating and executing a query.
Figure 49A illustrates a portion of a query builder library.
Figure 49B illustrates three examples of SQL entities.
Figures 50A and 50B illustrate portions of a ProcessClusterFilters definition.
Figure 51 illustrates an example of an introspection for a ProcessClusterFilters request.
Figure 52A illustrates an example of a base table card.
Figure 52B and 52D illustrates examples of a filter request.
Figure 52C and 52E illustrates examples of a dynamically generated SQL query.
Figure 53A illustrates a card.
Figure 53B and 53D illustrate a request.
Figure 53C and 53E illustrate a dynamically generated SQL query.
Figure 54A illustrates a pair of virtual machines.
Figure 54B illustrates an example of one pod communicating with another.

DESCRIPTION OF EMBODIMENTS

Example methods, apparatus, and products for identifying related activity across distinct cloud environments in accordance with embodiments of the present disclosure are described with reference to the accompanying drawings, beginning with Figure 1. The invention can be implemented in numerous ways, including as a process; an apparatus; a system; a composition of matter; a computer program product embodied on a computer readable storage medium; and/or a processor, such as a processor configured to execute instructions stored on and/or provided by a memory coupled to the processor. In this specification, these implementations, or any other form that

the invention may take, may be referred to as techniques. In general, the order of the steps of disclosed processes may be altered within the scope of the invention. Unless stated otherwise, a component such as a processor or a memory described as being configured to perform a task may be implemented as a general component that is temporarily configured to perform the task at a given time or a specific component that is manufactured to perform the task. As used herein, the term "processor" refers to one or more devices, circuits, and/or processing cores configured to process data, such as computer program instructions. A detailed description of one or more embodiments of the invention is provided below along with accompanying figures that illustrate the principles of the invention. The invention is described in connection with such embodiments, but the invention is not limited to any embodiment. The scope of the invention is limited only by the claims and the invention encompasses numerous alternatives, modifications and equivalents. Numerous specific details are set forth in the following description in order to provide a thorough understanding of the invention. These details are provided for the purpose of example and the invention may be practiced according to the claims without some or all of these specific details. Technical material that is known in the technical fields related to the invention has not been described in detail so that the invention is not unnecessarily obscured.

Figure 1 illustrates an example of an environment in which activities that occur within datacenters are modeled. Using techniques described herein, a baseline of datacenter activity can be modeled, and deviations from that baseline can be identified as anomalous. Anomaly detection can be beneficial in a security context, and can also be useful in other contexts, such as devops. Two example datacenters (104 and 106) are shown in Figure 1, and belong to fictional companies ACME and BETA, respectively. A datacenter can comprise dedicated equipment. A datacenter can also comprise cloud-based resources, such as infrastructure as a service (IaaS), platform as a service (PaaS), and/or software as a service (SaaS) elements. The techniques described herein can be used in conjunction with multiple types of datacenters, including ones wholly using dedicated equipment, ones that are entirely cloud-based, and ones that use a mixture of both dedicated equipment and cloud-based resources.

Both datacenter 104 and datacenter 106 include a plurality of nodes, depicted collectively as set of nodes 108 and set of nodes 110, respectively, in Figure 1. Installed on each of the nodes are in-server / in-virtual machine (VM) / embedded in IoT device agents (as applicable), which are configured to collect data and report it to platform 102 for analysis. Agents are small, self-contained binaries that can be run on any appropriate platforms, including virtualized ones (and, as applicable, within containers). Agents monitor the nodes on which they execute for a variety of different activities, including: connection, process, user, machine, and file activities. Agents can be executed in user space, and can use a variety of kernel modules (e.g., auditd, iptables, netfilter, pcap, etc.) to collect data. Agents can be implemented in any appropriate programming language, such as C or Golang, using applicable kernel APIs.

As will be described in more detail below, agents can selectively report information to platform 102 in varying amounts of detail and/or with variable frequency. As will also be described in more detail below, the data collected by agents is used by platform 102 to create polygraphs, which are graphs of logical entities, connected by behaviors. In some embodiments, agents report information directly to platform 102. In other embodiments, at least some agents provide information to a data aggregator, such as data aggregator 114, which in turn provides information to platform 102. The functionality of a data aggregator can be implemented as a separate binary or other application (distinct from an agent binary), and can also be implemented by having an agent execute in an "aggregator mode" in which the designated aggregator node acts as a Layer 7 proxy for other agents that do not have access to platform 102. Further, a chain of multiple aggregators can be used, if applicable (e.g., with agent 112 providing data to data aggregator 114, which in turn provides data to another aggregator (not pictured) which provides data to platform 102). An example way to implement an aggregator is through a program written in an appropriate language, such as C or Golang. Use of an aggregator can be beneficial in sensitive environments where various nodes are subject to regulatory or other architectural requirements. And, use of an aggregator can help to minimize security exposure more generally. As one example, by limiting communications with platform 102 to data aggregator 114, individual nodes in group 108 need not make external network connections (e.g., via Internet 124), which can potentially expose them to compromise. Similarly, platform 102 can provide updates, configuration information, etc., to data aggregator 114 (which in turn distributes them to nodes 108), rather than requiring nodes 108 to allow incoming connections from platform 102 directly.

Another benefit of an aggregator model is that network congestion can be reduced (e.g., with a single connection being made at any given time between data aggregator 114 and platform 102, rather than potentially many different connections being open between various of nodes 108 and platform 102). Similarly, network consumption can also be reduced (e.g., with the aggregator applying compression techniques/bundling data received from multiple agents). One example way that an agent (e.g., agent 112, installed on node 116) can provide information to data aggregator 114 is via a REST API, formatted using data serialization protocols such as Apache Avro. One example type of information sent by agent 112 to data aggregator 114 is status information. Status information is sent periodically and includes (in some embodiments): a. an amount of event backlog (in bytes) that has not yet been transmitted, b. configuration information, c. any data loss period for which data was dropped, d. a cumulative count of errors encountered since the agent started, e. version information for the agent binary, and f. cumulative statistics on data collection.

A second example type of information sent by agent 112 to data aggregator 114 is event data, which includes a UTC timestamp for each event. As applicable, the agent can control the amount of data that it sends to the data aggregator in each call by adjusting the amount of data sent to manage the conflicting goals of transmitting data as soon as possible, and maximizing throughput. Data can also be compressed or uncompressed by the agent (as applicable) prior to sending the data. Each data aggregator runs within a customer environment. A data aggregator (e.g., data aggregator 114) facilitates data routing from many different agents (e.g., agents executing on nodes 108) to platform 102. In some embodiments, data aggregator 114 implements a SOCKS 5 caching proxy through which agents can connect to platform 102. As applicable, data aggregator 114 can encrypt (or otherwise obfuscate) sensitive information prior to transmitting it to platform 102, and can also distribute key material to agents which can encrypt the information (as applicable). Data aggregator 114 includes a local storage, to which agents can upload data (e.g., pcap packets). The storage has a key-value interface. The local storage can also be omitted, and agents configured to upload data to a cloud storage or other storage area, as applicable. Data aggregator 114 can also cache locally and distribute software upgrades, patches, or configuration information (e.g., as received from platform 102).

In the following example, suppose that a network administrator at ACME (hereinafter "Alice") has decided to begin using the services of platform 102. In some embodiments, Alice accesses a web frontend (e.g., web app 120) using her computer (126) and enrolls (on behalf of ACME) an account with platform 102. After enrollment is complete, Alice is presented with a set of installers, pre-built and customized for the ACME environment, that she can download from platform 102 and deploy on nodes 108. Examples of such installers include: a Windows executable file; an iOS app; and a Linux package (e.g., .deb or .rpm), binary, or Docker container. When a network administrator at BETA (hereinafter "Bob") also signs up for the services of platform 102, he will similarly be presented with a set of installers that are pre-built and customized for the BETA environment. Alice deploys an appropriate installer on each of nodes 108. As applicable, the agent can be deployed in a container. Agent deployment can also be performed using one or more appropriate automation tools, such as Chef, Puppet, Salt, and Ansible. Deployment can also be performed using managed/hosted container management/orchestration frameworks such as Kubernetes, mesos, and Docker swarm. In some embodiments, the agent is installed in the user space, and the same binary is executed on each node of the same type. The primary job of an agent, such as agent 112, is to collect data (e.g., associated with node 116) and report it (e.g., to data aggregator 114). Other tasks performed by agents include data configuration and upgrades.

One approach to collecting data is to collect virtually all information available about a node (and, e.g., the processes running on it). A downside of this approach is that collecting information can be resource intensive. Another downside of this approach is that it can result in the collection of private or other sensitive information that can be difficult to filter/exclude from downstream reporting. An alternate approach is for the agent to monitor for network connections, and then begin collecting information about those processes associated with the network connections, using the presence of a network packet associated with a process as a trigger for collecting additional information about the process. As an example, if a user of node 116 executes an application, such as a calculator application, which does not typically interact with the network, no information about use of that application is collected by agent 112/sent to data aggregator 114. If, however, the user of node 116 executes an ssh command (e.g., to ssh from node 116 to node 122), agent 112 will collect information about the process and provide associated information to data aggregator 114. In some embodiments, the agent always collects/reports information about certain events, such as privilege escalation, irrespective of whether the event is associated with network activity. An approach to collecting information (e.g., by an agent) is as follows, and described in conjunction with process 200 depicted in Figure 2. An agent (e.g., agent 112) monitors its node (e.g., node 116) for network activity. One example way that agent 112 can monitor node 116 for network activity is by using a network packet capture tool (e.g., listening using libpcap). As packets are received (202), the agent obtains and maintains connection information associated with the network activity (204). Examples of such information include DNS query/response, TCP, UDP, and IP information. The agent also determines a process associated with the network connection (206). One example approach is for the agent to use a kernel network diagnostic API (e.g., netlink_diag) to obtain inode/process information

from the kernel. Another example approach is for the agent to scan using netstat to obtain sockets and relate them to processes. Information such as socket state can also be collected by the agent.

One way an agent can obtain a mapping between a given inode and a process identifier is to scan within the /proc/pid directory. For each of the processes currently running, the agent examines each of their file descriptors. If a file descriptor is a match for the inode, the agent can determine that the process associated with the file descriptor owns the inode. Once a mapping is determined between an inode and a process identifier, the mapping is cached. As additional packets are received for the connection, the cached process information is used (rather than a new search being performed). In some cases, exhaustively scanning for an inode match across every file descriptor may not be feasible. In some embodiments, searching through file descriptors is accordingly optimized. User filtering is one example of such an optimization. A given socket is owned by a user. Any processes associated with the socket will be owned by the same user as the socket. When matching an inode (identified as relating to a given socket) against processes, the agent can filter through the processes and only examine the file descriptors of processes sharing the same user owner as the socket. In some embodiments, processes owned by root are always searched against. Another example of an optimization is to prioritize searching the file descriptors of certain processes over others. One such prioritization is to search through the subdirectories of /proc/ starting with the youngest process. One approximation of such a sort order is to search through /proc/ in reverse order. Higher numbered processes are more likely to be newer, and thus more likely to be associated with new connections (i.e., ones for which inode-process mappings are not already cached). In some cases, the most recently created process may not have the highest process identifier. Another example prioritization is to query the kernel for an identification of the most recently created process and to search in a backward order through the directories in /proc/ (e.g., starting at process 400 and working backwards, then wrapping to the highest value and continuing to work backward from there). An alternate approach is for the agent to keep track of the newest process that it has reported information on (e.g., to data aggregator 114), and begin its search of /proc/ in a forward order starting from the PID of that process. Another example prioritization is to maintain, for each user actively using node 116, a list of the five most recently active processes. Those processes are more likely than other processes on node 116 to be involved with new connections, and can thus be searched first. For many processes, lower valued file descriptors tend to correspond to non-sockets. Yet another optimization is to preferentially search higher valued file descriptors (e.g., across processes) over lower valued file descriptors. In some cases, while attempting to locate a process identifier for a given inode, an agent may encounter a socket that does not correspond to the inode being matched against and is not already cached. The identity of that socket (and its corresponding inode) can be cached, once discovered, thus removing a future need to search for that pair.

In some cases, a connection may terminate before the agent is able to determine its associated process (e.g., due to a very short-lived connection, due to a backlog in agent processing, etc.). One approach to addressing such a situation is to asynchronously collect information about the connection using the audit kernel API, which streams information to user space. The information collected from the audit API (which can include PID/inode information) can be matched by the agent against pcap/inode information. In some embodiments, the audit API is always used, for all connections. However, due to CPU utilization considerations, use of the audit API can also be reserved for short/otherwise problematic connections (and/or omitted, as applicable). Once the agent has determined which process is associated with the network connection (206), the agent can then collect additional information associated with the process (208). As will be described in more detail below, some of the collected information comprises attributes of the process (e.g., a process parent hierarchy, and an identification of a binary associated with the process). As will also be described in more detail below, other of the collected information is derived (e.g., session summarization data and hash values).

The collected information is then transmitted (210), e.g., by an agent (e.g., agent 112) to a data aggregator (e.g., data aggregator 114), which in turn provides it to platform 102. In some embodiments, all information collected by an agent is transmitted (e.g., to a data aggregator and/or to platform 102). In other embodiments, the amount of data transmitted is minimized (e.g., for efficiency reasons), using various techniques. One approach to minimizing the amount of data flowing from agents (such as agents installed on nodes 108) to platform 102 is to use a technique of implicit references with unique keys. The keys can be explicitly used by platform 102 to extract/derive relationships in a data set at a later time, without impacting performance.

Depicted in Figure 3A is initial information collected by an agent about a process. Region 302 indicates the time at which the agent generated event 300. Region 304 indicates that event 300 pertains to a process. The type identifies a physical collection of related attributes. Other examples of event types that the agent can collect and report (and examples of the types of information included in such reports) are provided below (e.g., in Additional Agent Data Examples). Region 306 indicates the process ID (PID) of the process, and region 310 indicates the time the process was started. In region 308 is depicted a hash, generated by the agent to uniquely identify the particular process being reported on in event 300 (e.g., among all other processes whose data is reported to platform 102). In some embodiments, such hashes (and other hashes, such as are shown at 312-316) are generated pseudo-randomly. The hashes can be reconstructed should the agent crash and restart. An example salt for the hashes is a combination of a local process identifier, start time of the process, and one or more machine attributes. Region 318 indicates whether the process is associated with a terminal, an indicator of whether a process is interactive. Region 320 indicates a SHA-256 hash of the entire command line used to start the process, including any variable arguments, and region 322 indicates the path to the executable. Region 324 indicates the effective username of the owner of the process (which can either be the same or different from the username shown in region 328). As an example, suppose a user logs into node 116 using a first username, and then becomes a different user (e.g., using the setuid system call to inherit the permissions of a second user). Both names can be preserved in regions 324 and 328, respectively. Finally, region 326 indicates whether the process is running as a container isolation. As previously mentioned, some data collected about a process is constant and does not change over the lifetime of the process (e.g., attributes), and some data changes (e.g., statistical information and other variable information). The data depicted in Figure 3A is an example of constant data. It can be transmitted (210) once, when the agent first becomes aware of the process. And, if any changes to the data depicted in Figure 3A are detected, a refreshed version of the data depicted in Figure 3A can be transmitted (210).

Figure 3B illustrates an example of variable data that can be collected using the process shown in Figure 2. Variable data such as is shown in Figure 3B can be transmitted (210) at periodic (or other) intervals. Region 352 indicates a timestamp of when dataset 350 was collected by the agent. The hash depicted in region 354 is the same hash that was included in region 308, and is used within platform 102 to join process creation time attributes (e.g., shown in Figure 3A) with runtime attributes (e.g., shown in Figure 3B) to construct a full dataset. Region 356 indicates a thread count for the process. Region 358 indicates the total virtual memory used by the process. Region 360 indicates the total resident memory used by the process. Region 362 indicates the total time spent by the process executing in user space. Region 364 indicates the total time spent by the process executing in kernel space.

Additional examples of data that an agent, such as agent 112, can collect and provide to platform 102 include User Data, Machine Data, Network Data, Arp Data, DNS Data, Application Data, Package Data, Container Data, Container Image Data, File Data, and others. An agent 112 can be deployed in a container (e.g., a Docker container), and can also be used to collect information about containers. Collection about a container can be performed by an agent irrespective of whether the agent is itself deployed in a container or not (as the agent can be deployed in a container running in a privileged mode that allows for monitoring).

Agents can discover containers (e.g., for monitoring) by listening for container create events (e.g., provided by Docker), and can also perform periodic ordered discovery scans to determine whether containers are running on a node. When a container is discovered, the agent can obtain attributes of the container, e.g., using standard Docker API calls (e.g., to obtain IP addresses associated with the container, whether there's a server running inside, what port it is listening on, associated PIDs, etc.). Information such as the parent process that started the container can also be collected, as can information about the image (which comes from the Docker repository). In some embodiments, agents use namespaces to determine whether a process is associated with a container or not. Namespaces are a feature of the Linux kernel that can be used to isolate resources of a collection of processes. Examples of namespaces include process ID (PID) namespaces, network namespaces, and user namespaces. Given a process, the agent can perform a fast lookup to determine whether the process is part of the namespace the container claims to be its namespace. Agents can be configured to report certain types of information (e.g., attribute information) once, when the agent first becomes aware of a process. In some embodiments, such static information is not reported again, unless it changes.

In contrast to static/attribute information, certain types of data change constantly. In some embodiments, agents are configured to report a list of current connections every minute (or other time interval). In that connection list will be connections that started in that minute interval, connections that ended in that minute interval, and connections that were ongoing throughout the minute interval (e.g., a one minute slice of a one hour connection). In some embodiments, agents are configured to collect/compute statistical information about connections (e.g., at the one minute level of granularity). Examples of such information include, for the time interval, the number of bytes transferred, and in which direction. Another example of information collected by an agent about a connection is the length of time between packets.

For connections that span multiple time intervals (e.g., a seven minute connection), statistics are calculated for each minute of the connection. Such statistical information (for all connections) can be reported (e.g., to a data aggregator) once a minute.

In some embodiments, agents are also configured to maintain histogram data for a given network connection, and provide it (in the Apache Avro data exchange format) under the Connection event type data. Examples of such histograms include: 1. a packet length histogram which characterizes network packet distribution; 2. a session length histogram which characterizes a network session length; 3. a session time histogram which characterizes a network session time; and 4. a session switch time histogram which characterizes network session switch time (i.e., incoming->outgoing and vice versa). Figures 3C and 3D depict, side-by-side, examples of full outgoing (3C) and incoming (3D) network data sets, including each of the four histograms. Each of the histograms shown in Figures 3C and 3D includes the following fields: 1. count, which provides a count of the elements in the sampling; 2. sum, which provides a sum of elements in the sampling; 3. max, which provides the highest value element in the sampling; 4. std_dev, which provides the standard deviation of elements in the sampling; and 5. buckets, which provides a discrete sample bucket distribution of sampling data (if applicable).

For some protocols (e.g., HTTP), typically, a connection is opened, a string is sent, a string is received, and the connection is closed. For other protocols (e.g., NFS), both sides of the connection engage in a constant chatter. Histograms allow platform 102 to model application behavior (e.g., using machine learning techniques), for establishing baselines, and for detecting deviations. As one example, suppose that a given HTTP server typically sends/receives 1,000 bytes (in each direction) whenever a connection is made with it. If a connection generates 500 bytes of traffic, or 2,000 bytes of traffic, such connections would be considered within the typical usage pattern of the server. Suppose, however, that a connection is made that results in 10G of traffic. Such a connection is anomalous and can be flagged accordingly. Returning to Figure 1, as previously mentioned, data aggregator 114 is configured to provide information (e.g., collected from nodes 108 by agents) to platform 102. And, data aggregator 128 is similarly configured to provide information to platform 102. As shown in Figure 1, both aggregator 114 and aggregator 128 connect to a load balancer 130, which accepts connections from aggregators (and/or as applicable, agents), as well as other devices, such as computer 126 (e.g., when it communicates with web app 120), and supports fair balancing. In some embodiments, load balancer 130 is a reverse proxy that load balances accepted connections internally to various microservices, allowing for services provided by platform 102 to scale up as more agents are added to the environment and/or as more entities subscribe to services provided by platform 102. Example ways to implement load balancer 130 include using HaProxy, using nginx, and using elastic load balancing (ELB) services made available by Amazon.

Agent service 132 is a microservice that is responsible for accepting data collected from agents (e.g., provided by aggregator 114). In some embodiments, agent service 132 uses a standard secure protocol, such as HTTPS to communicate with aggregators (and as applicable agents), and receives data in an appropriate format such as Apache Avro. When agent service 132 receives an incoming connection, it can perform a variety of checks, such as to see whether the data is being provided by a current customer, and whether the data is being provided in an appropriate format. If the data is not appropriately formatted (and/or is not provided by a current customer), it is rejected. If the data is appropriately formatted, agent service 132 facilitates copying the received data to a streaming data stable storage (e.g., using Amazon Kinesis (134)). Once the ingesting into Kinesis is complete, service 132 sends an acknowledgement to the data provider (e.g., data aggregator 114). If the agent does not receive such an acknowledgement, it is configured to retry sending the data to platform 102. One way to implement agent service 132 is as a REST API server framework (e.g., Java DropWizard), configured to communicate with Kinesis (e.g., using a Kinesis library). In some embodiments, platform 102 uses one or more Kinesis streams (134) for all incoming customer data (e.g., including data provided by data aggregator 114 and data aggregator 128), and the data is sharded based on the node (also referred to herein as a "machine") that originated the data (e.g., node 116 vs. node 122), with each node having a globally unique identifier within platform 102. Multiple instances of agent service 132 can write to multiple shards. Kinesis is a streaming service with a limited period (e.g., 1-7 days). To persist data longer than a day, it is copied to long term storage (e.g., S3). S3 Loader 136 is a microservice that is responsible for picking up data from Kinesis stream 134 and persisting it in S3 (138). In one example embodiment, files collected by S3 Loader 136 from the Kinesis stream are placed into one or more buckets, and segmented using a combination of a customer identifier and time slice. Given a particular time segment, and a given customer identifier, the corresponding file (stored in S3) contains five minutes (or another appropriate time slice) of data collected at that specific customer from all of the customer's nodes. S3 Loader 136 can be implemented in any appropriate programming language, such as Java or C, and can be configured to use a Kinesis library to interface with Kinesis. In some embodiments, S3 Loader 136 uses the Amazon SQS (e.g., to alert DB Loader 140 that there is work for it to do).

DB Loader 140 is a microservice that is responsible for loading data into an appropriate database service (142), such as SnowflakeDB or Amazon Redshift, using individual per-customer databases. In particular, DB Loader 140 is configured to periodically load data into a set of raw tables from files created by S3 Loader 136 as per above. DB Loader 140 manages throughput, errors, etc., to make sure that data is loaded consistently and continuously. Further, DB Loader 140 can read incoming data and load into database 142 data that is not already present in database 142's tables. DB Loader 140 can be implemented in any appropriate programming language, such as Java or C, and an SQL framework such as jOOQ (e.g., to manage SQLs for insertion of data), and SQL/JDBC libraries. DB Loader 140 also uses Amazon S3 and Amazon Simple Queue Service (SQS) to manage files being transferred to and from the database (e.g., Snowflake). Customer data included in database 142 can be augmented with data from additional data sources, such as AWS Cloud Trail Analyzer 144, which is another microservice. AWS Cloud Trail Analyzer 144 pulls data using Amazon Cloudtrail for each applicable customer account, as soon as the data is available. AWS Cloud Trail Analyzer 144 normalizes the Cloudtrail data as applicable, so that it can be inserted into database 142 for later querying/analysis. AWS Cloud Trail Analyzer 144 can be written in any appropriate programming language, such as Java or C. AWS Cloud Trail Analyzer 144 also makes use of SQL/JDBC libraries to interact with database 142 to insert/query data.

As previously mentioned, platform 102 can model activities that occur within datacenters, such as datacenters 104 and 106. The model is stable over time, and differences, even subtle ones (e.g., between a current state of the datacenter and the model) can be surfaced. The ability to surface such anomalies can be particularly important in datacenter environments where rogue employees and/or external attackers may operate slowly (e.g., over a period of months), hoping that the elastic nature of typical resource use (e.g., virtualized servers) will help conceal their nefarious activities. Using techniques described herein, platform 102 can automatically discover entities deployed in a given datacenter. Examples of entities include workloads, applications, processes, machines, virtual machines, containers, files, IP addresses, domain names, and users. The entities are grouped together logically (into analysis groups) based on behaviors, and temporal behavior baselines can be established. In particular, using techniques described herein, periodic graphs can be constructed (also referred to herein as polygraphs), in which the nodes are applicable logical entities, and the edges represent behavioral relationships between the logical entities in the graph. Baselines can be created for every node and edge.

Communication (e.g., between applications/nodes) is one example of a behavior. A model of communications between processes is an example of a behavioral model. As another example, the launching of applications is another example of a behavior that can be modeled. The baselines are updated hourly for every entity. Deviations from the expected normal behavior can then be detected and automatically reported (e.g., as anomalies or threats detected). Such deviations may be due to a desired change, a misconfiguration, or malicious activity. As applicable, platform 102 can score the detected deviations (e.g., based on severity and threat posed). Additional examples of analysis groups include models of machine communications, models of privilege changes, and models of insider behaviors (monitoring the interactive behavior of human users as they operate within the datacenter).

Two types of information collected by agents are network level information and process level information. As previously mentioned, agents collect information about every connection involving their respective nodes. And, for each connection, information about both the server and the client is collected (e.g., using the connection-to-process identification techniques described above). DNS queries and responses are also collected. The DNS query information can be used in logical entity graphing (e.g., collapsing many different IP addresses to a single service – s3.amazonaws.com). Examples of process level information collected by agents include attributes (user ID, effective user ID, and command line). Information such as what user/application is responsible for launching a given process and the binary being executed (and its SHA-256 values) is also provided by agents.

The dataset collected by agents across a datacenter can be very large, and many resources (e.g., virtual machines, IP addresses, etc.) are recycled very quickly. For example, an IP address and port number used at a first point in time by a first process on a first virtual machine may very rapidly be used (e.g., an hour later) by a different process/virtual machine. A dataset (and elements within it) can be considered at both a physical level, and a logical level, as illustrated in Figure 4. In particular, Figure 4 illustrates an example 5-tuple of data (402) collected by an agent, represented physically (414) and logically (416). The 5-tuple includes a source address (404), a source port (406), a destination address (408), a destination port (410), and a protocol (412). In some cases, port numbers (e.g., 406, 410) may be

indicative of the nature of a connection (e.g., with certain port usage standardized). However, in many cases, and in particular in datacenters, port usage is ephemeral. For example, a Docker container can listen on an ephemeral port, which is unrelated to the service it will run. When another Docker container starts (for the same service), the port may well be different. Similarly, particularly in a virtualized environment, IP addresses may be recycled frequently (and are thus also potentially ephemeral) or could be NATed, which makes identification difficult. A physical representation of the 5-tuple is depicted in region 414. A process 418 (executing on machine 420) has opened a connection to machine 422. In particular, process 418 is in communication with process 424. Information such as the number of packets exchanged between the two machines over the respective ports can be recorded. As previously mentioned, in a datacenter environment, portions of the 5-tuple may change – potentially frequently – but still be associated with the same behavior. Namely, one application (e.g., Apache) may frequently be in communication with another application (e.g., Oracle), using ephemeral datacenter resources. Further, either/both of Apache and Oracle may be multi-homed. This can lead to potentially thousands of 5-tuples (or more) that all correspond to Apache communicating with Oracle within a datacenter. For example, Apache could be executed on a single machine, and could also be executed across fifty machines, which are variously spun up and down (with different IP addresses each time). An alternate representation of the 5-tuple shown in region 402 is depicted in region 416, and is logical. The logical representation of the 5-tuple aggregates the 5-tuple (along with other connections between Apache and Oracle having other 5-tuples) as logically representing the same connection. By aggregating data from raw physical connection information into logical connection information, using techniques described herein, a size reduction of six orders of magnitude in the data set can be achieved.

Figure 5 depicts a portion of a logical polygraph. Suppose a datacenter has seven instances of the application `update_engine` (502), executing as seven different processes on seven different machines, having seven different IP addresses, and using seven different ports. The instances of `update_engine` variously communicate with `update.core-os.net` (504), which may have a single IP address or many IP addresses itself, over the one hour time period represented in the polygraph. In the example shown in Figure 5, `update_engine` is a client, connecting to the server `update.core-os.net`, as indicated by arrow 508. Behaviors of the seven processes are clustered together, into a single summary. As indicated in region 506, statistical information about the connections is also maintained (e.g., number of connections, histogram information, etc.). A polygraph such as is depicted in Figure 5 can be used to establish a baseline of behavior (e.g., at the one-hour level), allowing for the future detection of deviations from that baseline. As one example, suppose that statistically an `update_engine` instance transmits data at 11 bytes per second. If an instance were instead to transmit data at 1000 bytes per second, such behavior would represent a deviation from the baseline and could be flagged accordingly. Similarly, changes that are within the baseline (e.g., an eighth instance of `update_engine` appears, but otherwise behaves as the other instances; or one of the seven instances disappears) are not flagged as anomalous. Further, datacenter events, such as failover, autobalancing, and A-B refresh are unlikely to trigger false alarms in a polygraph, as at the logical level, the behaviors remain the same. In some embodiments, polygraph data is maintained for every application in a datacenter, and such polygraph data can be combined to make a single datacenter view across all such applications. Figure 6 illustrates a portion of a polygraph for a service that evidences more complex behaviors than are depicted in Figure 5. In particular, Figure 6 illustrates the behaviors of S3 as a service (as used by a particular customer datacenter). Clients within the datacenter variously connect to the S3 service using one of five fully qualified domains (listed in region 602). Contact with any of the domains is aggregated as contact with S3 (as indicated in region 604). Depicted in region 606 are various containers which (as clients) connect with S3. Other containers (which do not connect with S3) are not included. As with the polygraph portion depicted in Figure 5, statistical information about the connections is known and summarized, such as the number of bytes transferred, histogram information, etc.

Figure 7 illustrates a communication polygraph for a datacenter. In particular, the polygraph indicates a one hour summary of approximately 500 virtual machines, which collectively run one million processes, and make 100 million connections in that hour. As illustrated in Figure 7, a polygraph represents a drastic reduction in size. Further, as a datacenter scales up, the polygraph for the datacenter will tend to stay the same size. As new applications are added into the datacenter, the polygraph will automatically scale to include behaviors involving those applications. In the polygraph shown in Figure 7, nodes generally correspond to workers, and edges correspond to communications the workers engage in (with connection activity being the behavior modeled in polygraph 700). Another example polygraph could model other behavior, such as application launching. The communications graphed in Figure 7 include traffic entering the datacenter, traffic exiting the datacenter, and traffic that stays wholly within the datacenter. One example of a node included in polygraph 700 is the `sshd` application, depicted as node 702. As indicated in Figure 7, 421 instances of `sshd` were executing during the one hour time period of data represented in polygraph 700. As indicated in region 704, nodes within the datacenter communicated with a total of 1349 IP addresses outside of the datacenter (and not otherwise accounted for, e.g., as belonging to a service such as Amazon AWS (706) or Slack (708)). In the following examples, suppose that Bob, an administrator of datacenter 106, is interacting with platform 102 to view visualizations of polygraphs in a web browser via web app 120. One type of polygraph Bob can view is an application-communication polygraph, which indicates, for a given one hour window, which applications communicated with which other applications. Another type of polygraph Bob can view is an application launch polygraph. Bob can also view graphs related to user behavior, such as an insider behavior graph which tracks user connections, a privilege change graph which tracks how privileges change between processes, and a user login graph, which tracks which (logical) machines a user logs into. Figure 8 illustrates an example of an application-communication polygraph for a datacenter (e.g., datacenter 106) for the one hour period of 9am-10am on June 5. The time slice currently being viewed is indicated in region 802. If Bob clicks his mouse in region 804, Bob will be shown a representation of the application-communication polygraph as generated for the following hour.

Figure 9 depicts what is shown in Bob's browser after he has clicked on region 804, and has further clicked on region 806. The selection in region 806 turns on and off the ability to compare two time intervals to one another. Bob can select from a variety of options when comparing the 9am-10am and 10am-11am time intervals. By clicking region 904, Bob will be shown the union of both graphs (i.e., any connections that were present in either time interval). By clicking region 906, Bob will be shown the intersection of both graphs (i.e., only those connections that were present in both time intervals). As shown in Figure 9, Bob has elected to click on region 908, which depicts connections that are only present in the 9am-10am polygraph in purple (910), and depicts connections that are only present in the 10am-11am polygraph in blue (912). Connections present in both polygraphs are omitted from display. As one example, in the 9am-10am polygraph (corresponding to connections made during the 9am-10am time period at datacenter 106), a connection was made by a server to `sshd` (914) and also to `systemd` (916). Both of those connections ended prior to 10am and are thus depicted in purple. As another example, in the 10am-11am polygraph (corresponding to connections made during the 10am-11am time period at datacenter 106), a connection was made from a known bad external IP to `nginx` (918). The connection was not present during the 9am-10am time slice and thus is depicted in blue. As yet another example, two different connections were made to a Slack service between 9am and 11am. However, the first was made by a first client during the 9am-10am time slice (920) and the second was made by a different client during the 10am-11am slice (922), and so the two connections are depicted respectively in purple and blue.

Returning to the polygraph depicted in Figure 8, suppose Bob enters “etcd” into the search box located in region 810. Bob will then be presented with the interface illustrated in Figure 10. As shown in Figure 10, three applications containing the term “etcd” were engaged in communications during the 9am-10am window. One application is `etcdctl`, a command line client for `etcd`. As shown in Figure 10, a total of three different `etcdctl` processes were executed during the 9am-10am window, and were clustered together (1002). Figure 10 also depicts two different clusters that are both named `etcd2`. The first cluster includes (for the 9am-10am window) five members (1004) and the second cluster includes (for the same window) eight members (1006). The reason for these two distinct clusters is that the two groups of applications behave differently (e.g., they exhibit two distinct sets of communication patterns). Specifically, the instances of `etcd2` in cluster 1004 only communicate with `locksmithctl` (1008) and other `etcd2` instances (in both clusters 1004 and 1006). The instances of `etcd2` in cluster 1006 communicate with additional entities, such as `etcdctl` and Docker containers. As desired, Bob can click on one of the clusters (e.g., cluster 1004) and be presented with summary information about the applications included in the cluster, as is shown in Figure 11 (e.g., in region 1102). He can also double click on a given cluster (e.g., cluster 1004) to see details on each of the individual members of the cluster broken out. Suppose Bob now clicks on region 812 of the interface shown in Figure 8. Bob will then be shown an application launch polygraph. Launching an application is another example of a behavior. The launch polygraph models how applications are launched by other applications. Figure 12 illustrates an example of a portion of a launch polygraph. In particular, Bob has typed “find” into region 1202, to see how the “find” application is being launched. As shown in Figure 12, in the launch polygraph for the 10am-11am time period, `find` applications (1204) are always launched by `bash` (1206), which is in turn always launched by `systemd` (1208). If `find` is launched by a different application, this would be anomalous behavior. Figure 13 illustrates another example of a portion of an application launch polygraph. In Figure 13, Bob has searched (1302) for “python ma” to see how “python marathon_lb” (1304) is launched. As shown in Figure 13, in each case (during the one hour time slice of 10am-11am), `python marathon_lb` is launched as a result of a chain of the same seven applications each time. If python

marathon_lb is ever launched in a different manner, this indicates anomalous behavior. The behavior could be indicative of malicious activities, but could also be due to other reasons, such as a misconfiguration, a performance-related issue, and/or a failure, etc. Suppose Bob now clicks on region 814 of the interface shown in Figure 8. Bob will then be shown an insider behavior graph. The insider behavior graph tracks information about behaviors such as processes started by a user interactively using protocols such as ssh or telnet, and any processes started by those processes. As one example, suppose an administrator logs into a first virtual machine in datacenter 106 (e.g., using sshd via an external connection he makes from a hotel), using a first set of credentials. From the first virtual machine, the administrator connects to a second virtual machine (e.g., using the same credentials), then uses the sudo command to change identities to those of another user, and then launches a program. Graphs built by platform 102 can be used to associate the administrator with each of his actions, including launching the program using the identity of another user. Figure 14 illustrates an example of a portion of an insider behavior graph. In particular, in Figure 14, Bob is viewing a graph that corresponds to the time slice of 3pm-4pm on June 1. Figure 14 illustrates the internal/external applications that users connected to during the one hour time slice. If a user typically communicates with particular applications, that information will become part of a baseline. If the user deviates from his baseline behavior (e.g., using new applications, or changing privilege in anomalous ways), such anomalies can be surfaced.

Figure 15 illustrates an example of a portion of a privilege change graph, which identifies how privileges are changed between processes. Typically, when a user launches a process (e.g., "ls"), the process inherits the same privileges that the user has. And, while a process can have fewer privileges than the user (i.e., go down in privilege), it is rare (and generally undesirable) for a user to escalate in privilege. Information included in the privilege change graph can be determined by examining the parent of each running process, and determining whether there is a match in privilege between the parent and the child. If the privileges are different, a privilege change has occurred (whether a change up or a change down). The application ntpd is one rare example of a scenario in which a process escalates (1502) to root, and then returns back (1504). The sudo command is another example (e.g., used by an administrator to temporarily have a higher privilege). As with the other examples, ntpd's privilege change actions, and the legitimate actions of various administrators (e.g., using sudo) will be incorporated into a baseline model by platform 102. When deviations occur, such as where a new application that is not ntpd escalates privilege, or where an individual that has not previously/does not routinely use sudo does so, such behaviors can be identified as anomalous.

Figure 16 illustrates an example of a portion of a user login graph, which identifies which users log into which logical nodes. Physical nodes (whether bare metal or virtualized) are clustered into a logical machine cluster, for example, using yet another graph, a machine-server graph, an example of which is shown in Figure 17. For each machine, a determination is made as to what type of machine it is, based on what kind(s) of workflows it runs. As one example, some machines run as master nodes and can thus be clustered as master nodes. Worker nodes are different from master nodes, for example, because they run Docker containers, and frequently change as containers move around. Worker nodes can similarly be clustered.

As previously mentioned, the polygraph depicted in Figure 7 corresponds to activities in a datacenter in which, in a given hour, approximately 500 virtual machines collectively run one million processes, and make 100 million connections in that hour. The polygraph represents a drastic reduction in size (e.g., from tracking information on 100 million connections in an hour, to a few hundred nodes and a few hundred edges). Using techniques described herein, such a polygraph can be constructed (e.g., using commercially available computing infrastructure) in less than an hour (e.g., within a few minutes). Thus, ongoing hourly snapshots of a datacenter can be created within a two hour moving window (i.e., collecting data for the time period 8am-9am, while also generating a snapshot for the time previous time period 7am-8am). The following section describes various example infrastructure that can be used in polygraph construction, and also describes various techniques that can be used to construct polygraphs.

Returning to Figure 1, embodiments of platform 102 are built using AWS infrastructure as a service (IaaS). For example, platform 102 can use Simple Storage Service (S3) for data storage, Key Management Service (KMS) for managing secrets, Simple Queue Service (SQS) for managing messaging between applications, Simple Email Service (SES) for sending emails, and Route 53 for managing DNS. Other infrastructure tools can also be used. Examples include: orchestration tools (e.g., Kubernetes or Mesos/Marathon), service discovery tools (e.g., Mesos-DNS), service load balancing tools (e.g., marathon-LB), container tools (e.g., Docker or rkt), log/metric tools (e.g., collectd, fluentd, kibana, etc.), big data processing systems (e.g., Spark, Hadoop, AWS Redshift, Snowflake etc.), and distributed key value stores (e.g., Apache Zookeeper or etcd2).

As previously mentioned, in some embodiments, platform 102 makes use of a collection of microservices. Each microservice can have multiple instances, and is configured to recover from failure, scale, and distribute work amongst various such instances, as applicable. For example, microservices are auto-balancing for new instances, and can distribute workload if new instances are started or existing instances are terminated. In some embodiments, microservices are deployed as self-contained Docker containers. A Mesos-Marathon or Spark framework can be used to deploy the microservices (e.g., with Marathon monitoring and restarting failed instances of microservices as needed). The service etcd2 can be used by microservice instances to discover how many peer instances are running, and used for calculating a hash-based scheme for workload distribution. Microservices are configured to publish various health/status metrics to either an SQS queue, or etcd2, as applicable. And, Amazon DynamoDB can be used for state management.

GraphGen (146) is a microservice that is responsible for generating raw behavior graphs on a per customer basis periodically (e.g., once an hour). In particular, GraphGen 146 generates graphs of entities (as the nodes in the graph) and activities between entities (as the edges). In some embodiments, GraphGen 146 also performs other functions, such as aggregation, enrichment (e.g., geolocation and threat), reverse DNS resolution, TF-IDF based command line analysis for command type extraction, parent process tracking, etc. GraphGen 146 performs joins on data collected by the agents, so that both sides of a behavior are linked. For example, suppose a first process on a first virtual machine (e.g., having a first IP address) communicates with a second process on a second virtual machine (e.g., having a second IP address). Respective agents on the first and second virtual machines will each report information on their view of the communication (e.g., the PID of their respective processes, the amount of data exchanged and in which direction, etc.). When GraphGen performs a join on the data provided by both agents, the graph will include a node for each of the processes, and an edge indicating communication between them (as well as other information, such as the directionality of the communication – i.e., which process acted as the server and which as the client in the communication). In some cases, connections are process to process. In other cases, a process may be in communication with a node (e.g., outside of ACME) which does not have an agent deployed upon it. As one example, a node within ACME might be in communication with node 172, outside of ACME. In such a scenario, communications with node 172 are modeled (e.g., by GraphGen 146) using the IP address of node 172. Similarly, where a node within ACME does not have an agent deployed upon it, the IP address of the node can be used by GraphGen in modeling. Graphs created by GraphGen 146 are written to database 142 and cached for further processing. The graph is a summary of all activity that happened in the time interval. As each graph corresponds to a distinct period of time, different rows can be aggregated to find summary information over a larger timestamp. And, picking two different graphs from two different timestamps can be used to compare different periods. If necessary, GraphGen can parallelize its workload and be implemented in any appropriate programming language, such as Java or C, and machine learning libraries, such as Spark's MLLib. Example ways that GraphGen computations can be implemented include using SQL or Map-R, using Spark or Hadoop.

SshTracker (148) is a microservice that is responsible for following ssh connections and process parent hierarchies to determine trails of user ssh activity. Identified ssh trails are placed by the sshTracker into database 142 and cached for further processing. SshTracker 148 can be implemented in any appropriate programming language, such as Java or C, and machine libraries, such as Spark's MLLib. Example ways that sshTracker computations can be implemented include using SQL or Map-R, using Spark or Hadoop. ThreatAggr (150) is a microservice that is responsible for obtaining third party threat information from various applicable sources, and making it available to other micro-services. Examples of such information include reverse DNS information, GeoIP information, lists of known bad domains/IP addresses, lists of known bad files etc. As applicable, the threat information is normalized before insertion into database 142. ThreatAggr can be implemented in any appropriate programming language, such as Java or C, using SQL/JDBC libraries to interact with database 142 (e.g., for insertions and queries). Hawkeye (152) is a microservice that acts as a scheduler and can run arbitrary jobs organized as a directed graph. Hawkeye ensures that all jobs for all customers are able to run every hour. It handles errors and retrying for failed jobs, tracks dependencies, manages appropriate resource levels, and scales jobs as needed. Hawkeye can be implemented in any appropriate programming language, such as Java or C. A variety of components can also be used, such as open source scheduler frameworks (e.g., Airflow), or AWS services (e.g., the AWS Data pipeline) which can be used for managing schedules.

Graph Behavior Modeler (GBM) (154) is a microservice that computes Polygraphs. In particular, the GBM can be used to find clusters of nodes in a graph that should be considered similar based on some set of their properties and relationships to other nodes. As will be described in more detail below, the clusters and their relationships can be used to provide visibility into a datacenter environment without requiring user specified labels. The GBM tracks such clusters over time persistently, allowing for changes to be detected and alerts to be generated. The GBM takes as input a raw graph, nodes are actors of a behavior, and edges are the behavior relationship itself. For example, in the case of communication, example actors include processes, which communicate with other processes. The GBM clusters the raw graph based on behaviors of actors and produces a summary (the Polygraph). The Polygraph summarizes behavior at a datacenter level. The GBM also produces "observations" that represent changes detected in the datacenter. Such observations are based on differences in cumulative behavior (e.g., the baseline) of the datacenter with its current behavior. The GBM can be implemented in any appropriate programming language, such as Java, C, or Golang, using appropriate libraries (as applicable) to handle distributed graph computations (handling large amounts of data analysis in a short amount of time). Apache Spark is another example tool that can be used to compute Polygraphs. The GBM can also take feedback from users and adjust the model according to that feedback. For example, if a user is interested in relearning behavior for an entity, the GBM can be instructed to "forget" the implicated part of the polygraph.

GBM Runner (156) is a microservice that is responsible for interfacing with GBM 154 and providing GBM 154 with raw graphs (e.g., using SQL to push any computations it can to database 142). GBM Runner 156 also inserts Polygraph output from GBM 154 to database 142. GBM Runner can be implemented in any appropriate programming language, such as Java or C, using SQL/JDBC libraries to interact with database 142 to insert and query data.

EventGen (158) is a microservice that is responsible for generating alerts. It examines observations (e.g., produced by GBM 154) in aggregate, deduplicates them, and scores them. Alerts are generated for observations with a score exceeding a threshold. EventGen 158 also computes (or retrieves, as applicable) data that a customer (e.g., Alice or Bob) might need when reviewing the alert. Examples of events that can be detected by platform 102 (and alerted on by EventGen 158) include an that is created the first time a user (e.g., of node 116) is first observed by an agent within a datacenter, an event generated when an interactive user launches an application for the first time, an event generated when user privileges are escalated and a new application is run, an event generated when an application or container is seen for the first time, an event generated when a connection to an external IP/domain is made from a new application, an event generated when a new external host or IP is involved in a connection with a datacenter, an event generated when a connection between internal-only applications is seen for the first time, an event generated when a new external connection is seen for an application which typically does not have external connections, an event generated when an application is launched by a different parent, an event if an environment makes a connection to a known bad IP or domain, an event will be generated, an event generated when a successful connection to a datacenter from a known bad IP is observed by platform 102, and others.

EventGen can be implemented in any appropriate programming language, such as Java or C, using SQL/JDBC libraries to interact with database 142 to insert and query data. In some embodiments, EventGen also uses one or more machine learning libraries, such as Spark's MLlib (e.g., to compute scoring of various observations). EventGen can also take feedback from users about which kinds of events are of interest and which to suppress.

QsJobServer (160) is a microservice that looks at all the data produced by platform 102 for an hour, and compiles a materialized view (MV) out of the data to make queries faster. The MV helps make sure that the queries customers most frequently run, and data that they search for, can be easily queried and answered. QsJobServer 160 also precomputes and caches a variety of different metrics so that they can quickly be provided as answers at query time. QsJobServer 160 can be implemented using any appropriate programming language, such as Java or C, using SQL/JDBC libraries. The QsJobServer is able to compute an MV efficiently at scale, where there could be a large number of joins. An SQL engine can be used to execute the SQL.

Alert notifier 162 is a microservice that takes alerts produced by EventGen 158 and sends the alerts to customers' integrated SIEM products (e.g., Splunk, Slack etc.). Alert notifier 162 can be implemented using any appropriate programming language, such as Java or C. Alert notifier 162 can be configured to use an email service (e.g., AWS SES or pagerduty) to send emails. Alert notifier 162 also provides templating support (e.g., Velocity or Moustache) to manage templates and structured notifications to SIEM products.

Reporting module 164 is a microservice responsible for creating reports out of customer data (e.g., daily summaries of events, etc.) and providing those reports to customers (e.g., via email). Reporting module 164 can be implemented using any appropriate programming language, such as Java or C. Reporting module 164 can be configured to use an email service (e.g., AWS SES or pagerduty) to send emails. Reporting module 164 also provides templating support (e.g., Velocity or Moustache) for templates (e.g., for constructing HTML-based email).

Web app 120 is a microservice that provides a user interface to data collected and processed on platform 102. Web app 120 provides login, authentication, query, data visualization, etc. features. Web app 120 includes both client and server elements. Example ways the server elements can be implemented are using Java DropWizard or Node.js to serve business logic, and a combination of JSON/HTTP to manage the service. Example ways the client elements can be implemented are using frameworks such as React, Angular, or Backbone. JSON, jQuery, and JavaScript libraries (e.g., underscore) can also be used.

Query service 166 is a microservice that manages all database access for web app 120. Query service 166 abstracts out data obtained from database 142 and provides a JSON-based REST API service to web app 120. Query service 166 generates SQL queries for the REST APIs that it receives at run time. Query service 166 can be implemented using any appropriate programming language, such as Java or C and SQL/JDBC libraries, or an SQL framework such as jOOQ. Query service 166 can internally make use of a variety of types of databases, including postgres, AWS Aurora (168) or Snowflake (142) to manage data for customers. Examples of tables that query service 166 manages are OLTP tables and data warehousing tables. Redis (170) is an open source project which provides a key-value store. Platform 102 can use Redis as a cache to keep information for frontend services about users. Examples of such information include valid tokens for a customer, valid customer cookies, a time a customer tried to login, etc.

Figure 18 illustrates an example of a process for detecting anomalies in a network environment. In some embodiments, process 1800 is performed by platform 102. The process begins at 1802 when data associated with activities occurring in a network environment (such as ACME's datacenter) is received. One example of such data that can be received at 1802 is agent-collected data described above (e.g., in conjunction with process 200). At 1804, a logical graph model is generated, using at least a portion of the monitored activities. A variety of approaches can be used to generate such logical graph models, and a variety of logical graphs can be generated (whether using the same, or different approaches). The following is one example of how data received at 1802 can be used to generate and maintain a model.

During bootstrap, platform 102 creates an aggregate graph of physical connections (also referred to herein as an aggregated physical graph) by matching connections that occurred in the first hour into communication pairs. Clustering is then performed on the communication pairs. Examples of such clustering, described in more detail below, include performing Matching Neighbor clustering and similarity (e.g., SimRank) clustering. Additional processing can also be performed, such as by splitting clusters based on application type, and annotating nodes with DNS query information. The resulting graph (also referred to herein as a base graph or common graph) can be used to generate a variety of models, where a subset of node and edge types and their properties are considered in a given model. One example of a model is a UID to UID model (a.k.a., a Uid2Uid model) which clusters together processes that share a username and show similar privilege change behavior. Another example of a model is a CType model, which clusters together processes that share command line similarity. Yet another example of a model is a PType model, which clusters together processes that share behaviors over time.

Each hour after bootstrap, a new snapshot is taken and information from the new snapshot is merged with existing data to create and maintain a cumulative graph. The cumulative graph (a.k.a., a cumulative PType graph and a polygraph) is a running model of how processes behave over time. Nodes in the cumulative graph are PType nodes, and provide information such as a list of all active processes and PIDs in the last hour, the number of historic total processes, the average number of active processes per hour, the application type of the process (e.g., the CType of the PType), and historic CType information/frequency. Edges in the cumulative graph can represent connectivity and provide information such as connectivity frequency. The edges can be weighted (e.g., based on number of connections, number of bytes exchanged, etc.). Edges in the cumulative graph (and snapshots) can also represent transitions. One approach to merging a snapshot of the activity of the last hour into a cumulative graph is as follows. An aggregate graph of physical connections is made for the connections included in the snapshot. And, clustering/splitting is similarly performed on the snapshot's aggregate graph. Next, PType clusters in the snapshot's graph are compared against PType clusters in the cumulative graph to identify commonality.

One approach to determining commonality is, for any two nodes that are members of a given CmdType, comparing internal neighbors and calculating a set membership Jaccard distance. The pairs of nodes are then ordered by decreasing similarity (i.e., with the most similar sets first). For nodes with a threshold amount of commonality (e.g., at least 66% members in common), any new nodes (i.e., appearing in the snapshot's graph but not the cumulative graph) are assigned the same PType identifier as is assigned to the corresponding node in the cumulative graph. For each node that is not classified (i.e., has not been assigned a PType identifier), a network signature is generated (i.e., indicative of the kinds of network connections the node makes, who the node communicates with, etc.). The following processing is then performed until convergence. If a match of the network signature is found in the cumulative graph, the unclassified node is assigned the PType identifier of the corresponding node in the cumulative graph. Any nodes which remain unclassified after convergence are new PTypes and are assigned new identifiers and added to the cumulative graph as new. As applicable, the detection of a new PType can be used to generate an alert. If the new PType has a new CmdType, a severity of the alert can be increased. If any surviving nodes (i.e., present in both the cumulative graph and the snapshot graph) change PTypes, such change is noted as a transition, and an alert can be generated. Further, if a surviving node changes PType and also changes CmdType, a severity of the alert can be increased.

Changes to the cumulative graph (e.g., a new PType or a new edge between two PTypes) can be used (e.g., at 1806) to detect anomalies. Two example kinds of anomalies that can be detected by platform 102 include security anomalies (e.g., a user or process behaving in an unexpected manner) and devops/root cause anomalies (e.g., network congestion, application failure, etc.). Detected anomalies can be recorded and surfaced (e.g., to administrators, auditors, etc.), such as through alerts which are generated at 1808 based on anomaly detection. Additional detail regarding processing performed, by various components depicted in Figure 1 (whether performed individually or in combination), in conjunction with model/polygraph construction (e.g., as performed at 1804) are provided below.

As explained above, an aggregated physical graph can be generated on a per customer basis periodically (e.g., once an hour) from raw physical graph information, by matching connections (e.g., between two processes on two virtual machines). In some embodiments, a deterministic fixed approach is used to cluster nodes in the aggregated physical graph (e.g., representing processes and their communications). As one example, Matching Neighbors Clustering (MNC) can be performed on the aggregated physical graph to determine which entities exhibit identical behavior and cluster such entities together.

Figure 19A depicts a set of example processes (p1, p2, p3, and p4) communicating with other processes (p10 and p11). Figure 19A is a graphical representation of a small portion of an aggregated physical graph showing (for a given hour) which processes in a datacenter communicate with which other processes. Using MNC, processes p1, p2, and p3 will be clustered together (1902), as they exhibit identical behavior (they communicate with p10 and only p10). Process p4, which communicates with both p10 and p11, will be clustered separately.

In MNC, only those processes exhibiting identical (communication) behavior will be clustered. In some embodiments, an alternate clustering approach can also/instead be used, which uses a similarity measure (e.g., constrained by a threshold value, such as a 60% similarity) to cluster items. In some embodiments, the output of MNC is used as input to SimRank, in other embodiments, MNC is omitted. Figure 19B depicts a set of example processes (p4, p5, p6) communicating with other processes (p7, p8, p9). As illustrated, most of nodes p4, p5, and p6 communicate with most of nodes p7, p8, and p9 (as indicated in Figure 19B with solid connection lines). As one example, process p4 communicates with process p7 (1952), process p8 (1954), and process p9 (1956). An exception is process p6, which communicates with processes p7 and p8, but does not communicate with process p9 (as indicated by dashed line 1958). If MNC were applied in Figure 19B, only p4 and p5 would be clustered.

One approach to similarity clustering is to use SimRank. In an embodiment of the SimRank approach, for a given node v in a directed graph, $I(v)$ and $O(v)$ denote the respective set of in-neighbors and out-neighbors of v . Individual in-neighbors are denoted as $li(v)$, for $1 \leq i \leq |I(v)|$, and individual out-neighbors are denoted as $Oi(v)$, for $1 \leq i \leq |O(v)|$. The similarity between two objects a and b can be denoted by $s(a,b) \in [1,0]$. A recursive equation (hereinafter "the SimRank equation") can be written for $s(a,b)$, where, if $a=b$, then $s(a,b)$ is defined as 1, otherwise, $s(a,b) = \frac{C}{|I(a)||I(b)|} \sum_{i=1}^{|I(a)|} \sum_{j=1}^{|I(b)|} s(I_i(a), I_j(b))$ where C is a constant between 0 and 1. One example value for the decay factor C is 0.8 (and a fixed number of iterations such as five). Another example value for the decay factor C is 0.6 (and/or a different number of iterations). In the event that a or b has no in-neighbors, similarity is set to $s(a,b)=0$, so the summation is defined to be 0 when $I(a)=\emptyset$ or $I(b)=\emptyset$. The SimRank equations for a graph G can be solved by iteration to a fixed point. Suppose n is the number of nodes in G . For each iteration k , n^2 entries $sk(*,*)$ are kept, where $sk(a,b)$ gives the score between a and b on iteration k . Successive computations of $s_{k+1}(*,*)$ are made based on $sk(*,*)$. Starting with $s0(*,*)$, where each $s0(a,b)$ is a lower bound on the actual SimRank score $s(a,b)$: $s_0(a,b) = \begin{cases} 1, & \text{if } a = b, \\ 0, & \text{if } a \neq b. \end{cases}$

The SimRank equation can be used to compute $s_{k+1}(a,b)$ from $sk(*,*)$ with $s_{k+1}(a,b) = \frac{C}{|I(a)||I(b)|} \sum_{i=1}^{|I(a)|} \sum_{j=1}^{|I(b)|} s_k(I_i(a), I_j(b))$ for $a \neq b$, and $s_{k+1}(a,b)=1$ for $a=b$. On

each iteration $k+1$, the similarity of (a,b) is updated using the similarity scores of the neighbors of (a,b) from the previous iteration k according to the SimRank equation. The values $sk(*,*)$ are nondecreasing as k increases. Returning to Figure 19B, while MNC would cluster nodes p4 and p5 together (and not include node p6 in their cluster), application of SimRank would cluster nodes p4-p6 into one cluster (1960) and also cluster nodes p7-p9 into another cluster (1962).

Figure 19C depicts a set of processes, and in particular server processes s1 and s2, and client processes c1, c2, c3, c4, c5, and c6. Suppose only nodes s1, s2, c1, and c2 are present in the graph depicted in Figure 19C (and the other nodes depicted are omitted from consideration). Using MNC, nodes s1 and s2 would be clustered together, as would nodes c1 and c2. SimRank clustering as described above would also result in those two clusters (s1 and s2, and c1 and c2). As previously mentioned, in MNC, identical behavior is required. Thus, if node c3 were now also present in the graph, MNC would not include c3 in a cluster with c2 and c1 because node c3 only communicates with node s2 and not node s1. In contrast, a SimRank clustering of a graph that includes nodes s1, s2, c1, c2, and c3 would result (based, e.g., on an applicable selected decay value and number of iterations) in a first cluster comprising nodes s1 and s2, and a second cluster of c1, c2, and c3. As an increasing number of nodes which communicate with server process s2, and do not also communicate with server process s1, are included in the graph (e.g., as c4, c5, and c6 are added), under SimRank, nodes s1 and s2 will become increasingly similar.

In some embodiments, SimRank is modified (from what is described above) to accommodate differences between the asymmetry of client and server connections. As one example, SimRank can be modified to use different thresholds for client communications (e.g., an 80% match among nodes c1-c6) and for server communications (e.g., a 60% match among nodes s1 and s2). Such modification can also help achieve convergence in situations such as where a server process dies on one node and restarts on another node.

The application of MNC/SimRank to an aggregated physical graph results in a smaller graph, in which processes which are determined to be sufficiently similar are clustered together. Typically, clusters generated as output of MNC will be underinclusive. For example, for the nodes depicted in Figure 19B, process p6 will not be included in a cluster with processes p4 and p5, despite substantial similarity in their communication behaviors. The application of SimRank (e.g., to the output of MNC) helps mitigate the underinclusiveness of MNC, but can result in overly inclusive clusters. As one example, suppose (returning to the nodes depicted in Figure 19A) that as a result of applying SimRank to the depicted nodes, nodes p1-p4 are all included in a single cluster. Both MNC and SimRank operate agnostically of which application a given process belongs to. Suppose processes p1-p3 each correspond to a first application (e.g., an update engine), and process p4 corresponds to a second application (e.g., sshd). Further suppose process p10 corresponds to contact with AWS. Clustering all four of the processes together (e.g., as a result of SimRank) could be problematic, particularly in a security context.

As previously mentioned, platform 102 maintains a mapping between processes and the applications to which they belong. In some embodiments, the output of SimRank (e.g., SimRank clusters) is split based on the applications to which cluster members belong (such a split is also referred to herein as a "CmdType split"). If all cluster members share a common application, the cluster remains. If different cluster members originate from different applications, the cluster members are split along application-type (CmdType) lines. Using the nodes depicted in Figure 19C as an example, suppose that nodes c1, c2, c3, and c5 all share "update engine" as the type of application to which they belong (sharing a CmdType). Suppose that node c4 belongs to "ssh," and suppose that node c6 belongs to "bash." As a result of SimRank, all six nodes (c1-c6) might be clustered into a single cluster. After a CmdType split is performed on the cluster, however, the single cluster will be broken into three clusters (c1, c2, c3, c5; c4; and c6). Specifically, the resulting clusters comprise processes associated with the same type of application, which exhibit similar behaviors (e.g., communication behaviors). Each of the three clusters resulting from the CmdType split represents, respectively, a node (also referred to herein as a PType) of a particular CmdType. Each PType is given a persistent identifier and stored as a cumulative graph.

A variety of approaches can be used to determine a CmdType for a given process. As one example, for some applications (e.g., sshd), a one-to-one mapping exists between the CmdType and the application/binary name. Thus, processes corresponding to the execution of sshd will be classified using a CmdType of sshd. In some embodiments, a list of common application/binary names (e.g., sshd, apache, etc.) is maintained by platform 102 and manually curated as applicable. Other types of applications (e.g., Java, Python, and Ruby) are multi-homed, meaning that several very different applications may all execute using the binary name, "java." For these types of applications, information such as command line / execution path information can be used in determining a CmdType. In particular, the subapplication can be used as the CmdType of the application, and/or term frequency analysis (e.g., TF/IDF) can be used on command line information to group, for example, any marathon related applications together (e.g., as a python.marathon CmdType) and separately from other Python applications (e.g., as a python.airflow CmdType). In some embodiments, machine learning techniques are used to determine a CmdType. The CmdType model is constrained such that the execution path for each CmdType is unique. One example approach to making a CmdType model is a random forest based approach. An initial CmdType model is bootstrapped using process parameters (e.g., available within one minute of process startup) obtained using one hour of information for a given customer (e.g., ACME). Examples of such parameters include the command line of the process, the command line of the process's parent(s) (if applicable), the uptime of the process, UID/EUID and any change information, TTY and any change information, listening ports, and children (if any). Another approach is to perform term frequency clustering over command line information to convert command lines into cluster identifiers.

The random forest model can be used (e.g., in subsequent hours) to predict a CmdType for a process (e.g., based on features of the process). If a match is found, the process can be assigned the matching CmdType. If a match is not found, a comparison between features of the process and its nearest CmdType (e.g., as determined using a Levenstein distance) can be performed. The existing CmdType can be expanded to include the process, or, as applicable, a new CmdType can be created (and other actions taken, such as generating an alert). Another approach to handling processes which do not match an existing CmdType is to designate such processes as unclassified, and once an hour, create a new random forest seeded with process information from a sampling of classified processes (e.g., 10 or 100 processes per CmdType) and the new processes. If a given new process winds up in an existing set, the process is given the corresponding CmdType. If a new cluster is created, a new CmdType can be created.

Conceptually, a polygraph represents the smallest possible graph of clusters that preserve a set of rules (e.g., in which nodes included in the cluster must share a CmdType and behavior). As a result of performing MNC, SimRank, and cluster splitting (e.g., CmdType splitting) many processes are clustered together based on commonality of behavior (e.g., communication behavior) and commonality of application type. Such clustering represents a significant reduction in graph size (e.g., compared to the original raw physical graph). Nonetheless, further clustering can be performed (e.g., by iterating on the graph data using the GBM to achieve such a polygraph). As more information within the graph is correlated, more nodes can be clustered, reducing the graph size, until convergence is reached and no further clustering is possible.

Figure 19D depicts two pairs of clusters. In particular, cluster 1964 represents a set of client processes sharing the same CmdType ("a1"), communicating (collectively) with a server process having a CmdType ("a2"). Cluster 1968 also represents a set of client processes having a CmdType a1 communicating with a server process having a CmdType a2. The nodes in clusters 1964 and 1968 (and similarly nodes in 1966 and 1970) remain separately clustered (as depicted) after MNC/SimRank/CmdType splitting – isolated islands. One reason this could occur is where server process 1966 corresponds to processes executing on a first machine (having an IP address of 1.1.1.1). The machine fails and a new server process 1970 starts, on a second machine (having an IP address of 2.2.2.2) and takes over for process 1966.

Communications between a cluster of nodes (e.g., nodes 1964) and the first IP address can be considered different behavior from communications between the same set of nodes and the second IP address, and thus communications 1972 and 1974 will not be combined by MNC/SimRank in some embodiments. Nonetheless, it could be desirable for nodes 1964/1968 to be combined (into cluster 1976), and for nodes 1966/1970 to be combined (into cluster 1978), as representing (collectively) communications between a1 and a2. One task that can be performed by platform 102 is to use DNS query information to map IP addresses to logical entities. As will be described in more detail below, GBM 154 can make use of the DNS query information to determine that graph nodes 1964 and graph nodes 1968 both made DNS queries for "appserverabc.example.com," which first resolved to 1.1.1.1 and then to 2.2.2.2, and to combine nodes 1964/1968 and 1966/1970 together into a single pair of nodes (1976 communicating with 1978). In some embodiments, GBM 154 operates in a batch manner in which it receives as input the nodes and edges of a graph for a particular time period along with its previous state, and generates as output clustered nodes, cluster membership edges, cluster-to-cluster edges, events, and its next state. GBM 154 does not try to consider all types of entities and their relationships that may be available in a conceptual common graph all at once. Instead, GBM uses a concept of models where a subset of node and edge types and their properties are considered in a given model. Such an approach is helpful for scalability, and also to help preserve detailed information (of particular importance in a security context) – as clustering entities in a more complex and larger graph could result in less useful results. In particular, such an approach allows for different types of relationships between entities to be preserved/more easily analyzed. While GBM 154 can be used with different models corresponding to different subgraphs, core abstractions remain the same across types of models:

- * Each node type in a GBM model is considered to belong to a class. The class can be thought of as a way for the GBM to split nodes based on the criteria it uses for the model. The class for a node is represented as a string whose value is derived from the node's key and properties depending on the GBM Model. Note that different GBM models may create different class values for the same node. For each node type in a given GBM model, GBM 154 can generate clusters of nodes for that type. A GBM generated cluster for a given member node type cannot span more than one class for that node type. GBM 154 generates edges between clusters that have the same types as the edges between source and destination cluster node types.

- * The processes described herein as being used for a particular model can be used (can be the same) across models, and different models can also be configured with different settings.

- * The node types and the edge types may correspond to existing types in the common graph node and edge tables but this is not necessary. Even when there is a correspondence, the properties provided to GBM 154 are not limited to the properties that are stored in the corresponding graph table entries. They can be enriched with additional information before being passed to GBM 154.

Logically, the input for a GBM model can be characterized in a manner that is similar to other graphs. Edge triplets can be expressed, for example, as an array of source node type, edge type, and destination node type. And, each node type is associated with node properties, and each edge type is associated with edge properties. Other edge triplets can also be used (and/or edge triplets can be extended) in accordance with some embodiments. Note that the physical input to the GBM model need not (and does not, in some embodiments) conform to the logical input. For example, the edges in the PtypeConn model correspond to edges between Matching Neighbors (MN) clusters, where each process node has an MN cluster identifier property. In the User ID to User ID model (also referred to herein as the Uid2Uid model), edges are not explicitly provided separately from nodes (as the cuid array in the node properties serves the same purpose). In both cases, however, the physical information provides the applicable information necessary for the logical input. The state input for a particular GBM model can be stored in a file, a database, or other appropriate storage. The state file (from a previous run) is provided, along with graph data, except for when the first run for a given model is performed, or the model is reset. In some cases, no data may be available for a particular model in a given time period, and GBM may not be run for that time period. As data becomes available at a future time, GBM can run using the latest state file as input. GBM 154 outputs cluster nodes, cluster membership edges, and inter-cluster relationship edges that are stored (in some embodiments) in the graph node tables: node_c, node_cm, and node_icr, respectively. The type names of nodes and edges conform to the following rules: a given node type can be used in multiple different GBM models. The type names of the cluster nodes generated by two such models for that node type will be different. For instance, process type nodes will appear in both PtypeConn and Uid2Uid models, but their cluster nodes will have different type names; the membership edge type name is "MemberOf"; The edge type names for cluster-to-cluster edges will be the same as the edge type names in the underlying node-to-node edges in the input. The following are example events GBM 154 can generate: new class, new cluster, new edge from class to class, split class (the notion that GBM 154 considers all nodes of a given type and class to be in the same cluster initially and if GBM 154 splits them into multiple clusters, it is splitting a class), new edge from cluster and class, new edge between cluster and cluster, new edge from class to cluster, and others. One underlying node or edge in the logical input can cause multiple types of events to be generated. Conversely, one event can correspond to multiple nodes or edges in the input. Not every model generates every event type. Additional information regarding examples of data structures/models that can be used in conjunction with models used by platform 102 is provided in this section.

PTypeConn Model: This model clusters nodes of the same class that have similar connectivity relationships. For example, if two processes had similar incoming neighbors of the same class and outgoing neighbors of the same class, they could be clustered. The node input to the PTypeConn model for a given time period includes non-interactive (i.e., not associated with tty) process nodes that had connections in the time period and the base graph nodes of other types (IP Service Endpoint (IPSep) comprising an IP address and a port), DNS Service Endpoint (DNSSep) and IPAddress) that have been involved in those connections. The base relationship is the connectivity relationship for the following type triplets: Process, ConnectedTo, Process; Process, ConnectedTo, IP Service Endpoint (IPSep); Process, ConnectedTo, DNS Service Endpoint (DNSSep); IPAddress, ConnectedTo, Process; Process, ConnectedTo, Process.

The edge inputs to this model are the ConnectedTo edges from the MN cluster; instead of individual node-to-node ConnectedTo edges from the base graph. The membership edges created by this model refer to the base graph node type provided in the input. The class values of nodes are determined as follows depending on the node type (e.g., Process nodes, IPSep nodes, DNSSep nodes, and IP Address nodes). A new class event in this model for a process node is equivalent to seeing a new CType being involved in a connection for the first time. Note that this does not mean the CType was not seen before. It is possible that it was previously seen but did not make a connection at that time. A new class event in this model for an IPSep node with IP_internal = 0 is equivalent to seeing a connection to a new external IP address for the first time. A new class event in this model for a DNSSep node is equivalent to seeing a connection to a new domain for the first time. A new class event in this model for an IPAddress node with IP_internal = 0 and severity = 0 is equivalent to seeing a connection from any external IP address for the first time. A new class event in this model for an IPAddress node with IP_internal = 0 and severity > 0 is equivalent to seeing a connection from any bad external IP address for the first time. A new class to class edge from a class for a process node to a class for a process node is equivalent to seeing a communication from the source CType making a connection to the destination CType for the first time. A new class to class edge from a class for a process node to a class for a DNSSep node is equivalent to seeing a communication from the source CType making a connection to the destination domain name for the first time. The IntPConn Model is similar to the PTypeConn Model, except that connection edges between parent/child processes and connections between processes where both sides are not interactive are filtered out.

The Uid2Uid Model clusters processes with the same username that show similar privilege change behavior. For instance, if two processes with the same username had similar effective user values, launched processes with similar usernames, and were launched by processes with similar usernames, then they could be clustered. An edge between a source cluster and destination cluster generated by this model means that all of the processes in the source cluster had a privilege change relationship to at least one process in the destination cluster. The node input to this model for a given time period includes process nodes that are running in that period. The value of a class of process nodes is "<username>". The base relationship that is used for clustering is privilege change, either by the process changing its effective user ID, or by launching a child process which runs with a different user. The physical input for this model includes process nodes (only), with the caveat that the complete ancestor hierarchy of process nodes active (i.e., running) for a given time period is provided as input even if an ancestor is not active in that time period. Note that effective user IDs of a process are represented as an array in the process node properties, and launch relationships are available from ppid_hash fields in the properties as well. A new class event in this model is equivalent to seeing a user for the first time. A new class to class edge event is equivalent to seeing the source user making a privilege change to the destination user for the first time.

The Ct2Ct Model clusters processes with the same CType that show similar launch behavior. For instance, if two processes with the same CType have launched processes with similar CTypes, then they could be clustered. The node input to this model for a given time period includes process nodes that are running in that period. The value class of process nodes is CType (similar to how it is created for the PTypeConn Model). The base relationship that is used for clustering is a parent process with a given CType launching a child process with another given destination CType. The physical input for this model includes process nodes (only) with the caveat that the complete ancestor hierarchy active process nodes (i.e., that are running) for a given time period is provided as input even if an ancestor is not active in that time period. Note that launch relationships are available from ppid_hash fields in the process node properties. An edge between a source cluster and destination cluster generated by this model means that all of the processes in the source cluster launched at least one process in the destination cluster. A new class event in this model is equivalent to seeing a CType for the first time. Note that the same type of event will be generated by the PTypeConn Model as well. A new class to class edge event is equivalent to seeing the source CType launching the destination CType for the first time.

The MTypeConn Model clusters nodes of the same class that have similar connectivity relationships. For example, if two machines had similar incoming neighbors of the same class and outgoing neighbors of the same class, they could be clustered. A new class event in this model will be generated for external IP addresses or (as applicable) domain names seen for the first time. Note that a new class to class edge Machine, class to class for an IPSep or DNSName node will also be generated at the same time. The membership edges generated by this model will refer to Machine, IPAddress, DNSName, and IPSep nodes in the base graph. Though the nodes provided to this model are IPAddress nodes instead of IPSep nodes, the membership edges it generates will refer to IPSep type nodes. Alternatively, the base graph can generate edges between Machine and IPSep node types. Note that the Machine to IPAddress edges have tcp_dst_ports/udp_dst_ports properties that can be used for this purpose.

The node input to this model for a given time period includes machine nodes that had connections in the time period and the base graph nodes of other types (IPAddress and DNSName) that were involved in those connections. The base relationship is the connectivity relationship for the following type triplets: Machine, ConnectedTo, Machine; Machine, ConnectedTo, IPAddress; Machine, ConnectedTo, DNSName; IPAddress, ConnectedTo, Machine, DNS, ConnectedTo, Machine. For a given model and time period, the following example types of events can represent multiple changes in the underlying GBM cluster level graph in terms of multiple new clusters or multiple new edges between clusters: NewClass, NewEdgeClassToClass, NewEdgeNodeToClass, NewEdgeClassToNode, and others. Multiple NewClass events with the same model and class can be output if there are multiple clusters in that new class. Multiple NewEdgeClassToClass events with the same model and class pair can be output if there are multiple new cluster edges within that class pair. Multiple NewEdgeNodeToClass events with the same model and destination class can be output if there are multiple new edges from the source cluster to the destination clusters in that destination class (the first time seeing this class as a source cluster class for the source cluster). Multiple NewEdgeClassToNode events with the same model and source class can be output if there are multiple new edges from source clusters to the destination clusters in that source class (the first time seeing this class as a source cluster class for the destination cluster). These events may be combined at the class level and treated as a single event when it is desirable to view changes at the class level, e.g., when one wants to know when there is a new CType. Also note that different models may have partial overlap in the types of nodes they use from the base graph. Therefore, they can generate NewClass type events for the same class. NewClass events can also be combined across models when it is desirable to view changes at the class level.

Using techniques herein, actions can be associated with processes and (e.g., by associating processes with users) actions can thus also be associated with extended user sessions. Such information can be used to track user behavior correctly, even where a malicious user attempts to hide his trail by changing user identities (e.g., through lateral movement). Extended user session tracking can also be useful in operational use cases without malicious intent, e.g., where users make original logins with distinct usernames (e.g., "charlie" or "dave") but then perform actions under a common username (e.g., "admin" or "support"). One such example is where multiple users with administrator privileges exist, and they need to gain superuser privilege to perform a particular type of maintenance. It may be desirable to know which operations are performed (as the superuser) by which original user when debugging issues. In the following examples describing extended user session tracking, reference is generally made to using the secure shell (ssh) protocol as implemented by openssh (on the server side) as the mechanism for logins. However, extended user session tracking is not limited to the ssh protocol or a particular limitation and the techniques described herein can be extended to other login mechanisms. On any given machine, there will be a process that listens for and accepts ssh connections on a given port. This process can run the openssh server program running in daemon mode or it could be running another program (e.g., initd on a Linux system). In either case, a new process running openssh will be created for every new ssh login session and this process can be used to identify an ssh session on that machine. This process is called the "privileged" process in openssh. After authentication of the ssh session, when an ssh client requests a shell or any other program to be run under that ssh session, a new process that runs that program will be created under (i.e., as a child of) the associated privileged process. If an ssh client requests port forwarding to be performed, the connections will be associated with the privileged process. In modern operating systems such as Linux and Windows, each process has a parent process (except for the very first process) and when a new process is created the parent process is known. By tracking the parent-child hierarchy of processes, one can determine if a particular process is a descendant of a privileged openssh process and thus if it is associated with an ssh login session. For user session tracking across machines (or on a single machine with multiple logins) in a distributed

environment, it is established when two login sessions have a parent-child relationship. After that, the “original” login session, if any, for any given login session can be determined by following the parent relationship recursively.

Figure 20 is a representation of a user logging into a first machine and then into a second machine from the first machine, as well as information associated with such actions. In the example of Figure 20, a user, Charlie, logs into Machine A (2002) from a first IP address (2004). As part of the login process, he provides a username (2006). Once connected to Machine A, an openssh privileged process (2008) is created to handle the connection for the user, and a terminal session is created and a bash process (2010) is created as a child. Charlie launches an ssh client (2012) from the shell, and uses it to connect (2014) to Machine B (2016). As with the connection he makes to Machine A, Charlie’s connection to Machine B will have an associated incoming IP address (2018), in this case, the IP address of Machine A. And, as part of the login process with Machine B, Charlie provides a username (2020) which need not be the same as username 2006. An openssh privileged process (2022) is created to handle the connection, and a terminal session and child bash process (2024) will be created. From the command line of Machine B, Charlie launches a curl command (2026), which opens an HTTP connection (2028) to an external Machine C (2030). Figure 21 is an alternate representation of actions occurring in Figure 20, where events occurring on Machine A are indicated along line 2102, and events occurring on Machine B are indicated along line 2104. As shown in Figure 21, an incoming ssh connection is received at Machine A (2106). Charlie logs in (as user “x”) and an ssh privileged process is created to handle Charlie’s connection (2108). A terminal session is created and a bash process is created (2110) as a child of process 2108. Charlie wants to ssh to Machine B, and so executes an ssh client on Machine A (2112), providing credentials (as user “y”) at 2114. Charlie logs into Machine B, and an ssh privileged process is created to handle Charlie’s connection (2116). A terminal session is created and a bash process is created (2118) as a child of process 2116. Charlie then executes curl (2120) to download content from an external domain (via connection 2122). The external domain could be a malicious domain, or it could be benign. Suppose the external domain is malicious (and, e.g., Charlie has malicious intent). It would be advantageous (e.g., for security reasons) to be able to trace the contact with the external domain back to Machine A, and then back to Charlie’s IP address. Using techniques described herein (e.g., by correlating process information collected by various agents), such tracking of Charlie’s activities back to his original login (2000) can be accomplished. In particular, an extended user session can be tracked that associates Charlie’s ssh processes together with a single original login and thus original user.

As previously explained, software agents (such as agent 112) run on machines (such as machine 116) and detect new connections, processes, and logins. As also previously explained, such agents send associated records to platform 102 which includes one or more datastores (e.g., database 142) for persistently storing such data. Such data can be modeled using logical tables, also persisted in datastores (e.g., in a relational database that provides an SQL interface), allowing for querying of the data. Other datastores such as graph oriented databases and/or hybrid schemes can also be used. An ssh login session can be identified uniquely by an (MID, PID, hash) tuple. The MID is a machine identifier that is unique to each machine, whether physical or virtual, across time and space. Operating systems use numbers called process identifiers (PIDs) to identify processes running at a given time. Over time processes may die and new processes may be started on a machine or the machine itself may restart. The PID is not necessarily unique across time in that the same PID value can be reused for different processes at different times. In order to track process descendants across time, one should therefore account for time as well. In order to be able to identify a process on a machine uniquely across time, another number called a PID_hash is generated for the process. In some embodiments, the PID_hash is generated using a collision-resistant hash function that takes the PID, start time, and (in some embodiments, as applicable) other properties of a process. Input data collected by agents comprises the input data model and is represented by at least the following logical tables: A Connections Table that maintains records of TCP/IP connections observed on each machine; a Processes Table that maintains records of processes observed on each machine; a Logins Table that maintains records of logins to the machines. The output data generated by session tracking is represented with the following logical tables: login-local-descendant, login-connection, and login-lineage. Using data in these tables, it is possible to determine descendant processes of a given ssh login session across the environment (i.e., spanning machines). Conversely, given a process, it is possible to determine if it is an ssh login descendant as well as the original ssh login session for it if so.

Figure 22 illustrates an example of a process for performing extended user tracking. In some embodiments, process 2200 is performed by platform 102. The process begins at 2202 when data associated with activities occurring in a network environment (such as ACME’s datacenter) is received. One example of such data that can be received at 2202 is agent-collected data described above (e.g., in conjunction with process 200). At 2204, the received network activity is used to identify user login activity. And, at 2206, a logical graph that links the user login activity to at least one user and at least one process is generated (or updated, as applicable). Additional detail regarding process 2200, and in particular, portions 2204 and 2206 of process 2200 are described in more detail below (e.g., in conjunction with discussion of Figure 24).

Figure 23 depicts a representation of a user logging into a first machine, then into a second machine from the first machine, and then making an external connection. The scenario depicted in Figure 23 is used to describe an example of processing that can be performed on data collected by agents to generate extended user session tracking information. Figure 23 is an alternate depiction of the information shown in Figures 20 and 21. At time t1 (2302), a first ssh connection is made to Machine A (2304) from an external source (2306) by a user having a username of “X.” In the following example, suppose the external source has an IP address of 1.1.1.10 and uses source port 10000 to connect to Machine A (which has an IP address of 2.2.2.20 and a destination port 22). External source 2306 is considered an external source because its IP address is outside of the environment being monitored (e.g., is a node outside of ACME’s datacenter, connecting to a node inside of ACME’s datacenter). A first ssh login session LS1 is created on machine A for user X. The privileged openssh process for this login is A1 (2308). Under the login session LS1, the user creates a bash shell process with PID_hash A2 (2310). At time t2 (2312), inside the bash shell process A2, the user runs an ssh program under a new process A3 (2314) to log in to machine B (2316) with a different username (“Y”). In particular, an ssh connection is made from source IP address 2.2.2.20 and source port 10001 (Machine A’s source information) to destination IP address 2.2.2.21 and destination port 22 (Machine B’s destination information). A second ssh login session LS2 is created on machine B for user Y. The privileged openssh process for this login is B1 (2318). Under the login session LS2, the user creates a bash shell process with PID_hash B2 (2320). At time t3 (2324), inside the bash shell process B2, the user runs a curl command under a new process B3 (2326) to download a file from an external destination (2328). In particular, an HTTPS connection is made from source IP address 2.2.2.21 and source port 10002 (Machine B’s source information) to external destination IP address 3.3.3.30 and destination port 443 (the external destination’s information). Using techniques described herein, it is possible to determine the original user who initiated the connection to external destination 2328, which in this example is a user having the username X on machine A (where the extended user session can be determined to start with ssh login session LS1). Based on local descendant tracking, the following determinations can be on machine A and B without yet having performed additional processing: A3 is a descendant of A1 and thus associated with LS1; the connection to the external domain from machine B is initiated by B3; B3 is a descendant of B1 and is thus associated with LS2; the connection to the external domain is thus associated with LS2. An association between A3 and LS2 can be established based on the fact that LS2 was created based on an ssh connection initiated from A3. Accordingly, it can be determined that LS2 is a child of LS1. To determine the user responsible for making the connection to the external destination (e.g., if it were a known bad destination), first, the process that made the connection would be traced, i.e., from B3 to LS2. Then LS2 would be traced to LS1 (i.e., LS1 is the origin login session for LS2). Thus the user for this connection is the user for LS1, i.e., X. As represented in Figure 23, one can visualize the tracing by following the links (in the reverse direction of arrows) from external source 2328 to A1 (2308). In the example scenario, it is assumed that both ssh connections occur in the same analysis period. However, the approaches described herein will also work for connections and processes that are created in different time periods.

Figure 24 illustrates an example of a process for performing extended user tracking. In some embodiments, process 2400 is performed periodically (e.g., once an hour in a batch fashion) by ssh tracker 148 to generate new output data. In general, batch processing allows for efficient analysis of large volumes of data. However, the approach can be adapted, as applicable, to process input data on a record-by-record fashion while maintaining the same logical data processing flow. As applicable the results of a given portion of process 2400 are stored for use in a subsequent portion. The process begins at 2402 when new ssh connection records are identified. In particular, new ssh connections started during the current time period are identified by querying the connections table. The query uses filters on the start_time and dst_port columns. The values of the range filter on the start_time column are based on the current time period. The dst_port column is checked against ssh listening port(s). By default, the ssh listening port number is 22. However, as this could vary across environments, the port(s) that openssh servers are listening to in the environment can be determined by data collection agents dynamically and used as the filter value for the dst_port as applicable. In the scenario depicted in Figure 23, the query result will generate the records shown in Figure 25A. Note that for the connection between machine A and B, the two machines are likely to report start_time

values that are not exactly the same but close enough to be considered matching (e.g., within one minute or another appropriate amount of time). In the above table, they are shown to be the same for simplicity.

At 2404, ssh connection records reported from source and destination sides of the same connection are matched. The ssh connection records (e.g., returned from the query at 2402) are matched based on the following criteria: the five tuples (src_IP, dst_IP, IP_prot, src_port, dst_port) of the connection records must match; the delta between the start times of the connections must be within a limit that would account for the worst case clock difference expected between two machines in the environment and typical connection setup latency; and if there are multiple matches possible, then the match with the smallest time delta is chosen. Note that record 2502 from machine A for the incoming connection from the external source cannot be matched with another record as there is an agent only on the destination side for this connection. Example output of portion 2404 of process 2400 is shown in Figure 25B. The values in the dst_PID_hash column (2504) are that of the sshd privileged process associated with ssh logins. At 2406, new logins during the current time period are identified by querying the logins table. The query uses a range filter on the login_time column with values based on the current time period. In the example depicted in Figure 23, the query result will generate the records depicted in Figure 25C. At 2408, matched ssh connection records created at 2404 and new login records created at 2406 are joined to create new records that will eventually be stored in the login-connection table. The join condition is that dst_MID of the matched connection record is equal to the MID of the login record and the dst_PID_hash of the matched connection record is equal to the sshd_PID_hash of the login record. In the example depicted in Figure 23, the processing performed at 2408 will generate the records depicted in Figure 25D.

At 2410, login-local-descendant records in the lookback time period are identified. It is possible that a process that is created in a previous time period makes an ssh connection in the current analysis batch period. Although not depicted in the example illustrated in Figure 23, consider a case where bash process A2 does not create ssh process A3 right away but instead that the ssh connection A3 later makes to machine B is processed in a subsequent time period than the one where A2 was processed. While processing this subsequent time period in which processes A3 and B1 are seen, knowledge of A2 would be useful in establishing that B1 is associated with A3 (via ssh connection) which is associated with A2 (via process parentage) which in turn would be useful in establishing that the parent of the second ssh login is the first ssh login. The time period for which look back is performed can be limited to reduce the amount of historical data that is considered. However, this is not a requirement (and the amount of look back can be determined, e.g., based on available processing resources). The login local descendants in the lookback time period can be identified by querying the login-local-descendant table. The query uses a range filter on the login_time column where the range is from start_time_of_current_period - lookback_time to start_time_of_current_period. (No records as a result of performing 2410 on the scenario depicted in Figure 23 are obtained, as only a single time period is applicable in the example scenario.)

At 2412, new processes that are started in the current time period are identified by querying the processes table. The query uses a range filter on the start_time column with values based on the current time period. In the example depicted in Figure 23, the processing performed at 2412 will generate the records depicted in Figure 25E. At 2414, new login-local-descendant records are identified. The purpose is to determine whether any of the new processes in the current time period are descendants of an ssh login process and if so to create records that will be stored in the login-local-descendant table for them. In order to do so, the parent-child relationships between the processes are recursively followed. Either a top down or bottom up approach can be used. In a top down approach, the ssh local descendants in the lookback period identified at 2410, along with new ssh login processes in the current period identified at 2408 are considered as possible ancestors for the new processes in the current period identified at 2412.

Conceptually, the recursive approach can be considered to include multiple sub-steps where new processes that are identified to be ssh local descendants in the current sub-step are considered as ancestors for the next step. In the example scenario depicted in Figure 23, the following descendancy relationships will be established in two sub-steps: Sub-step 1: Process A2 is a local descendant of LS1 (i.e., MID = A, sshd_PID_hash = A1) because it is a child of process A1 which is the login process for LS1. Process B2 is a local descendant of LS2 (i.e., MID = B, sshd_PID_hash = B1) because it is a child of process B1 which is the login process for LS2. Sub-step 2: Process A3 is a local descendant of LS1 because it is a child of process A2 which is associated to LS1 in sub-step 1. Process B3 is a local descendant of LS2 because it is a child of process B1 which is associated to LS2 in sub-step 1. Implementation portion 2414 can use a datastore that supports recursive query capabilities, or, queries can be constructed to process multiple conceptual sub-steps at once. In the example depicted in Figure 23, the processing performed at 2414 will generate the records depicted in Figure 25F. Note that the ssh privileged processes associated with the logins are also included as they are part of the login session. At 2416, the lineage of new ssh logins created in the current time period is determined by associating their ssh connections to source processes that may be descendants of other ssh logins (which may have been created in the current period or previous time periods). In order to do so, first an attempt is made to join the new ssh login connections in the current period (identified at 2408) with the combination of the login local descendants in the lookback period (identified at 2410) and the login local descendants in the current time period (identified at 2412). This will create adjacency relationships between child and parent logins. In the example depicted in Figure 23, the second ssh login connection will be associated with process A3 and an adjacency relationship between the two login sessions will be created (as illustrated in Figure 25G). Next, the adjacency relationships are used to find the original login sessions. While not shown in the sample scenario, there could be multiple ssh logins in a chain in the current time period, in which case a recursive approach (as in 2414) could be used. At the conclusion of portion 2416, the login lineage records depicted in Figure 25H will be generated. Finally, at 2418, output data is generated. In particular, the new login-connection, login-local-descendant, and login-lineage records generated at 2408, 2414, and 2416 are inserted into their respective output tables (e.g., in a transaction manner).

An alternate approach to matching TCP connections between machines running an agent is for the client to generate a connection GUID and send it in the connection request (e.g., the SYN packet) it sends and for the server to extract the GUID from the request. If two connection records from two machines have the same GUID, they are for the same connection. Both the client and server will store the GUID (if it exists) in the connection records they maintain and report. On the client side, the agent can configure the network stack (e.g., using IP tables functionality on Linux) to intercept an outgoing TCP SYN packet and modify it to add the generated GUID as a TCP option. On the server side, the agent already extracts TCP SYN packets and thus can look for this option and extract the GUID if it exists.

Administrators and other users of network environments (e.g., ACME's datacenter 104) often change roles to perform tasks. As one example, suppose that at the start of a workday, an administrator (hereinafter "Joe Smith") logs in to a console, using an individualized account (e.g., username=joe.smith). Joe performs various tasks as himself (e.g., answering emails, generating status reports, writing code, etc.). For other tasks (e.g., performing updates), Joe may require different/additional permission than his individual account has (e.g., root privileges). One way Joe can gain access to such permissions is by using sudo, which will allow Joe to run a single command with root privileges. Another way Joe can gain access to such permissions is by su or otherwise logging into a shell as root. After gaining root privileges, another thing that Joe can do is switch identities. As one example, to perform administrative tasks, Joe may use "su help" or "su database-admin" to become (respectively) the help user or the database-admin user on a system. He may also connect from one machine to another, potentially changing identities along the way (e.g., logging in as joe.smith at a first console, and connecting to a database server as database-admin). When he's completed various administrative tasks, Joe can relinquish his root privileges by closing out of any additional shells created, reverting back to a shell created for user joe.smith. While there are many legitimate reasons for Joe to change his identity throughout the day, such changes may also correspond to nefarious activity. Joe himself may be nefarious, or Joe's account (joe.smith) may have been compromised by a third party (whether an "outsider" outside of ACME's network, or an "insider"). Using techniques described herein, the behavior of users of the environment can be tracked (including across multiple accounts and/or multiple machines) and modeled (e.g., using various graphs described herein). Such models can be used to generate alerts (e.g., to anomalous user behavior). Such models can also be used forensically, e.g., helping an investigator visualize various aspects of a network and activities that have occurred, and to attribute particular types of actions (e.g., network connections or file accesses) to specific users. In a typical day in a datacenter, a user (e.g., Joe Smith) will log in, run various processes, and (optionally) log out. The user will typically log in from the same set of IP addresses, from IP addresses within the same geographical area (e.g., city or country), or from historically known IP addresses/geographical areas (i.e., ones the user has previously/occasionally used). A deviation from the user's typical (or historical) behavior indicates a change in login behavior. However, it does not necessarily mean that a breach has occurred. Once logged into a datacenter, a user may take a variety of actions. As a first example, a user might execute a binary/script. Such binary/script might communicate with other nodes in the datacenter, or outside of the datacenter, and transfer data to the user (e.g., executing "curl" to obtain data from a service external to the datacenter). As a second example, the user can similarly transfer data (e.g., out of the datacenter), such as by using POST. As a third example, a user might change privilege (one or more times), at which point the user can send/receive data as per above. As a fourth example, a user might connect to a different

machine within the datacenter (one or more times), at which point the user can send/receive data as per the above. In some embodiments, the above information associated with user behavior is broken into four tiers. The tiers represent example types of information that platform 102 can use in modeling user behavior: 1. The user's entry point (e.g., domains, IP addresses, and/or geolocation information such as country/city) from which a user logs in; 2. The login user and machine class; 3. Binaries, executables, processes, etc. a user launches; 4. Internal servers with which the user (or any of the user's processes, child processes, etc.) communicates, and external contacts (e.g., domains, IP addresses, and/or geolocation information such as country/city) with which the user communicates (i.e., transfers data). In the event of a security breach, being able to concretely answer questions about such information can be very important. And, collectively, such information is useful in providing an end-to-end path (e.g., for performing investigations).

In the following example, suppose a user ("UserA") logs into a machine ("Machine01") from a first IP address ("IP01"). Machine01 is inside a datacenter. UserA then launches a script ("runnable.sh") on Machine01. From Machine01, UserA next logs into a second machine ("Machine02") via ssh, also as UserA, also within the datacenter. On Machine02, UserA again launches a script ("new_runnable.sh"). On Machine02, UserA then changes privilege, becoming root on Machine02. From Machine02, UserA (now as root) logs into a third machine ("Machine03") in the datacenter via ssh, as root on Machine03. As root on Machine03, the user executes a script ("collect_data.sh") on Machine03. The script internally communicates (as root) to a MySQL-based service internal to the datacenter, and downloads data from the MySQL-based service. Finally, as root on Machine03, the user externally communicates with a server outside the datacenter ("External01"), using a POST command. To summarize what has occurred, in this example, the source/entry point is IP01. Data is transferred to an external server External01. The machine performing the transfer to External01 is Machine03. The user transferring the data is "root" (on Machine03), while the actual user (hiding behind root) is UserA. In the above scenario, the "original user" (ultimately responsible for transmitting data to External01) is UserA, who logged in from IP01. Each of the processes ultimately started by UserA, whether started at the command line (tty) such as "runnable.sh" or started after an ssh connection such as "new_runnable.sh," and whether as UserA, or as a subsequent identity, are all examples of child processes which can be arranged into a process hierarchy.

As previously mentioned, machines can be clustered together logically into machine clusters. One approach to clustering is to classify machines based on information such as the types of services they provide/binaries they have installed upon them/processes they execute. Machines sharing a given machine class (as they share common binaries/services/etc.) will behave similarly to one another. Each machine in a datacenter can be assigned to a machine cluster, and each machine cluster can be assigned an identifier (also referred to herein as a machine class). One or more tags can also be assigned to a given machine class (e.g., database_servers_west or prod_web_frontend). One approach to assigning a tag to a machine class is to apply term frequency analysis (e.g., TF/IDF) to the applications run by a given machine class, selecting as tags those most unique to the class. Platform 102 can use behavioral baselines taken for a class of machines to identify deviations from the baseline (e.g., by a particular machine in the class).

Figure 26 illustrates an example of a process for detecting anomalies. In some embodiments, process 2600 is performed by platform 102. As explained above, a given session will have an original user. And, each action taken by the original user can be tied back to the original user, despite privilege changes and/or lateral movement throughout a datacenter. Process 2600 begins at 2602 when log data associated with a user session (and thus an original user) is received. At 2604, a logical graph is generated, using at least a portion of the collected data. When an anomaly is detected (2606), it can be recorded, and as applicable, an alert is generated (2608). The following are examples of graphs that can be generated (e.g., at 2604), with corresponding examples of anomalies that can be detected (2606) and alerted upon (2608). Figure 27A illustrates a representation of an insider behavior graph. In the example of Figure 27A, each node in the graph can be: (1) a cluster of users; (2) a cluster of launched processes; (3) a cluster of processes/servers running on a machine class; (4) a cluster of external IP addresses (of incoming clients); or (5) a cluster of external servers based on DNS/IP/etc. As depicted in Figure 27A, graph data is vertically tiered into four tiers. Tier 0 (2752) corresponds to entry point information (e.g., domains, IP addresses, and/or geolocation information) associated with a client entering the datacenter from an external entry point. Entry points are clustered together based on such information. Tier 1 (2754) corresponds to a user on a machine class, with a given user on a given machine class represented as a node. Tier 2 (2756) corresponds to launched processes, child processes, and/or interactive processes. Processes for a given user and having similar connectivity (e.g., sharing the processes they launch and the machines with which they communicate) are grouped into nodes. Finally, Tier 3 (2758) corresponds to the services/servers/domains/IP addresses with which processes communicate. A relationship between the tiers can be stated as follows: Tier 0 nodes log in to tier 1 nodes. Tier 1 nodes launch tier 2 nodes. Tier 2 nodes connect to tier 3 nodes. The inclusion of an original user in both Tier 1 and Tier 2 allows for horizontal tiering. Such horizontal tiering ensures that there is no overlap between any two users in Tier 1 and Tier 2. Such lack of overlap provides for faster searching of an end-to-end path (e.g., one starting with a Tier 0 node and terminating at a Tier 3 node). Horizontal tiering also helps in establishing baseline insider behavior. For example, by building an hourly insider behavior graph, new edges/changes in edges between nodes in Tier 1 and Tier 2 can be identified. Any such changes correspond to a change associated with the original user. And, any such changes can be surfaced as anomalous and alerts can be generated. As explained above, Tier 1 corresponds to a user (e.g., user "U") logging into a machine having a particular machine class (e.g., machine class "M"). Tier 2 is a cluster of processes having command line similarity (e.g., CType "C"), having an original user "U," and running as a particular effective user (e.g., user "U1"). The value of U1 may be the same as U (e.g., joe.smith in both cases), or the value of U1 may be different (e.g., U=joe.smith and U1=root). Thus, while an edge may be present from a Tier 1 node to a Tier 2 node, the effective user in the Tier 2 node may or may not match the original user (while the original user in the Tier 2 node will match the original user in the Tier 1 node). As a reminder, a change from a user U into a user U1 can take place in a variety of ways. Examples include where U becomes U1 on the same machine (e.g., via su), and also where U sshes to other machine(s). In both situations, U can perform multiple changes, and can combine approaches. For example, U can become U1 on a first machine, ssh to a second machine (as U1), become U2 on the second machine, and ssh to a third machine (whether as user U2 or user U3). In some embodiments, the complexity of how user U ultimately becomes U3 (or U5, etc.) is hidden from a viewer of an insider behavior graph, and only an original user (e.g., U) and the effective user of a given node (e.g., U5) are depicted. As applicable (e.g., if desired by a viewer of the insider behavior graph), additional detail about the path (e.g., an end-to-end path of edges from user U to user U5) can be surfaced (e.g., via user interactions with nodes).

Figure 27B illustrates an example of a portion of an insider behavior graph (e.g., as rendered in a web browser). In the example shown, node 2702 (the external IP address, 52.32.40.231) is an example of a Tier 0 node, and represents an entry point into a datacenter. As indicated by directional arrows 2704 and 2706, two users, "aruneli_prod" and "harish_prod," both made use of the source IP 52.32.40.231 when logging in between 5pm and 6pm on Sunday July 30 (2708). Nodes 2710 and 2712 are examples of Tier 1 nodes, having aruneli_prod and harish_prod as associated respective original users. As previously mentioned, Tier 1 nodes correspond to a combination of a user and a machine class. In the example depicted in Figure 27B, the machine class associated with nodes 2710 and 2712 is hidden from view to simplify visualization, but can be surfaced to a viewer of interface 2700 (e.g., when the user clicks on node 2710 or 2712). Nodes 2720-2738 are examples of Tier 2 nodes – processes that are launched by users in Tier 1 and their child, grandchild, etc. processes. Note that also depicted in Figure 27B is a Tier 1 node 2714 that corresponds to a user, "root," that logged in to a machine cluster from within the datacenter (i.e., has an entry point within the datacenter). Nodes 2742-2744 are examples of Tier 3 nodes – internal/external IP addresses, servers, etc., with which Tier 2 nodes communicate. In the example shown in Figure 27B, a viewer of interface 2700 has clicked on node 2738. As indicated in region 2746, the user running the marathon container is "root." However, by following the directional arrows in the graph backwards from node 2738 (i.e., from right to left), the viewer can determine that the original user, responsible for node 2738, is "aruneli_prod," who logged into the datacenter from IP 52.32.40.231.

The following are examples of changes that can be tracked using an insider behavior graph model: A user logs in from a new IP address; A user logs in from a geolocation not previously used by that user; A user logs into a new machine class; A user launches a process not previously used by that user; A user connects to an internal server to which the user has not previously connected; An original user communicates with an external server (or external server known to be malicious) with which that user has not previously communicated; A user communicates with an external server which has a geolocation not previously used by that user.

Such changes can be surfaced as alerts, e.g., to help an administrator determine when/what anomalous behavior occurs within a datacenter. Further, the behavior graph model can be used (e.g., during forensic analysis) to answer questions helpful during an investigation. Examples of such questions include: Was there any new login activity (Tier 0) in the timeframe being investigated? As one example, has a user logged in from an IP address with unknown geolocation information? Similarly, has a user started communicating externally with a new Tier 3 node (e.g., one with unknown geolocation information)? Has there been any suspicious login activity (Tier 0)

in the timeframe being investigated? Similarly, has there been any suspicious Tier 3 activity? Were any anomalous connections made within the datacenter during the timeframe being investigated? Who is (the user) responsible for running a particular process?

An example of an insider behavior graph being used in an investigation is depicted in Figures 28A and 28B. Figure 28A depicts a baseline of behavior for a user, “Bill.” As shown in Figure 28A, Bill typically logs into a datacenter from the IP address, 71.198.44.40 (2802). He typically makes use of ssh (2804), and sudo (2806), makes use of a set of typical applications (2808) and connects (as root) with the external service, api.lacework.net (2810). Suppose Bill’s credentials are compromised by a nefarious outsider (“Eve”). Figure 28B depicts an embodiment of how the graph depicted in Figure 28A would appear once Eve begins exfiltrating data from the datacenter. Eve logs into the datacenter (using Bill’s credentials) from 52.5.66.8 (2852). As Bill, Eve escalates her privilege to root (e.g., via su), and then becomes a different user, Alex (e.g., via su alex). As Alex, Eve executes a script, “sneak.sh” (2854), which launches another script, “post.sh” (2856), which contacts external server 2858 which has an IP address of 52.5.66.7, and transmits data to it. Edges 2860-2866 each represent changes in Bill’s behavior. As previously mentioned, such changes can be detected as anomalies and associated alerts can be generated. As a first example, Bill logging in from an IP address he has not previously logged in from (2860) can generate an alert. As a second example, while Bill does typically make use of sudo (2806), he has not previously executed sneak.sh (2854) or post.sh (2856) and the execution of those scripts can generate alerts as well. As a third example, Bill has not previously communicated with server 2858, and an alert can be generated when he does so (2866). Considered individually, each of edges 2860-2866 may indicate nefarious behavior, or may be benign. As an example of a benign edge, suppose Bill begins working from a home office two days a week. The first time he logs in from his home office (i.e., from an IP address that is not 71.198.44.40), an alert can be generated that he has logged in from a new location. Over time, however, as Bill continues to log in from his home office but otherwise engages in typical activities, Bill’s graph will evolve to include logins from both 71.198.44.40 and his home office as baseline behavior. Similarly, if Bill begins using a new tool in his job, an alert can be generated the first time he executes the tool, but over time will become part of his baseline. In some cases, a single edge can indicate a serious threat. For example, if server 2852 (or 2858) is included in a known bad IP listing, edge 2860 (or 2866) indicates compromise. An alert that includes an appropriate severity level (e.g., “threat level high”) can be generated. In other cases, a combination of edges could indicate a threat (where a single edge might otherwise result in a lesser warning). In the example shown in Figure 28B, the presence of multiple new edges is indicative of a serious threat. Of note, even though “sneak.sh” and “post.sh” were executed by Alex, because platform 102 also keeps track of an original user, the compromise of Bob’s account will be discovered.

Figure 29 illustrates a representation of an embodiment of a user login graph. In the example of Figure 29, tier 0 (2902) clusters source IP addresses as belonging to a particular country (including an “unknown” country) or as a known bad IP. Tier 1 (2904) clusters user logins, and tier 2 (2906) clusters type of machine class into which a user is logging in. The user login graph tracks the typical login behavior of users. By interacting with a representation of the graph, answers to questions such as the following can be obtained: Where is a user logging in from? Have any users logged in from a known bad address? Have any non-developer users accessed development machines? Which machines does a particular user access? Examples of alerts that can be generated using the user login graph include: A user logs in from a known bad IP address; A user logs in from a new country for the first time; A new user logs into the datacenter for the first time; A user accesses a machine class that the user has not previously accessed. One way to track privilege changes in a datacenter is by monitoring a process hierarchy of processes. To help filter out noisy commands/processes such as “su -u,” the hierarchy of processes can be constrained to those associated with network activity. In a *nix system, each process has two identifiers assigned to it, a process identifier (PID) and a parent process identifier (PPID). When such a system starts, the initial process is assigned a PID 0. Each user process has a corresponding parent process. Figure 30 illustrates a representation of a process tree. In the example shown in Figure 30, PIDs have been replaced with an effective user running a given process. Thus, a designation of “root” (3002) indicates the user running the process is root, and a designation of “avahi” (3004) indicates the user running the process is avahi. Further, in the example shown in Figure 30, processes depicted to the right of other processes are child processes. In line 3006, the user avahi became root and ran the process “padac run,” whose parent is “avahi-daemon.” This represents a privilege change (from avahi to root). Using techniques described herein, a graph can be constructed (also referred to herein as a privilege change graph) which models privilege changes. In particular, a graph can be constructed which identifies where a process P1 launches a process P2, where P1 and P2 each have an associated user U1 and U2, with U1 being an original user, and U2 being an effective user. In the graph, each node is a cluster of processes (sharing a CType) executed by a particular (original) user. As all the processes in the cluster belong to the same user, a label that can be used for the cluster is the user’s username. An edge in the graph, from a first node to a second node, indicates that a user of the first node changed its privilege to the user of the second node.

Figure 31 illustrates an example of a privilege change graph. In the example shown in Figure 31, each node (e.g., nodes 3102 and 3104) represents a user. Privilege changes are indicated by edges, such as edge 3106. As with other graphs, anomalies in graph 3100 can be used to generate alerts. Three examples of such alerts are as follows: New user entering the datacenter; Privilege change; Privilege escalation. An example of an anomalous privilege change and an example of an anomalous privilege escalation are each depicted in graph 3200 of Figure 32. In particular, as indicated in region 3202, two alerts for the time period 2pm-3pm on June 8 were generated (corresponding to the detection of the two anomalous events). In region 3204, root has changed privilege to the user “daemon,” which root has not previously done. This anomaly is indicated to the user by highlighting the daemon node (e.g., outlining it in the color red). As indicated by edge 3206, Bill has escalated his privilege to the user root (which can similarly be highlighted in region 3208). This action by Bill represents a privilege escalation.

As described throughout this Specification, datacenters are highly dynamic environments. And, different customers of platform 102 (e.g., ACME vs. BETA) may have different/dispersive needs/requirements of platform 102, e.g., due to having different types of assets, different applications, etc. Further, as time progresses, new software tools will be developed, new types of anomalous behavior will be possible (and should be detectable), etc. In some embodiments, platform 102 makes use of predefined relational schema (including by having different predefined relational schema for different customers). However, the complexity and cost of maintaining/updating such predefined relational schema can rapidly become problematic – particularly where the schema includes a mix of relational, nested, and hierarchical (graph) datasets. In other embodiments, the data models and filtering applications used by platform 102 are extensible. As will be described in more detail below, in some embodiments, platform 102 supports dynamic query generation by automatic discovery of join relations via static or dynamic filtering key specifications among composable data sets. This allows a user of platform 102 to be agnostic to modifications made to existing data sets as well as creation of new data sets. The extensible query interface also provides a declarative and configurable specification for optimizing internal data generation and derivations. As will also be described in more detail below, platform 102 is configured to dynamically translate user interactions (e.g., received via web 120) into SQL queries (and without the user needing to know how to write queries). Such queries can then be performed (e.g., by query service 166) against any compatible backend (e.g., database 142).

Figure 33 illustrates an example of a user interacting with a portion of an interface. When a user visits platform 102 (e.g., via web app 120 using a browser), data is extracted from database 142 as needed (e.g., by query service 166), to provide the user with information, such as the visualizations depicted variously throughout the Specification (e.g., in Figures 9 and 33). As the user continues to interact with such visualizations, such interactions act as triggers that cause query service 166 to continue to obtain information from database 142 as needed (and as described in more detail below). In the example shown in Figure 33, ACME administrator Alice is viewing a dashboard that provides various information about ACME users (3302), during the time period March 2 at midnight – March 25 at 7pm (which she selected by interacting with region 3304). Various statistical information is presented to Alice in region 3306. Region 3308 presents a timeline of events that occurred during the selected time period. Alice has opted to list only the critical, high, and medium events during the time period by clicking on the associated boxes (3310-3314). A total of 55 low severity, and 155 info-only events also occurred during the time period. Each time Alice interacts with an element in Figure 33 (e.g., clicks on box 3314, clicks on link 3320, or clicks on tab 3322), her actions are translated/formalized into filters on the data set and used to dynamically generate SQL queries. The SQL queries are generated transparently to Alice. Alice notes in the timeline (3316) that a user, Harish, connected to a known bad server (examplebad.com) using wget, an event that has a critical severity level. As will be described in more detail below, a dossier is a template for a collection of visualizations. As shown in interface 3400, the event of Harish using wget to contact examplebad.com on March 16 was assigned an event ID of 9291 by platform 102 (3402). For convenience to Alice, the event is also added to her dashboard in region 3420 as a bookmark (3404). A summary of the event is depicted in region 3406. By interacting with boxes shown in region 3408, Alice can see a timeline of related events.

Data stored in database 142 can be internally organized as an activity graph. In the activity graph, nodes are also referred to as Entities. Activities generated by Entities are modeled as directional edges between nodes. Thus, each edge is an activity between two Entities. One example of an Activity is a “login” Activity, in which a user Entity logs into a machine Entity (with a directed edge from the user to the machine). A second example of an Activity is a “launch” Activity, in which a parent process

launches a child process (with a directed edge from the parent to the child). A third example of an Activity is a “DNS query” Activity, in which either a process or a machine performs a query (with a directed edge from the requestor to the answer, e.g., an edge from a process to `www.example.com`). A fourth example of an Activity is a network “connected to” Activity, in which processes, IP addresses, and listen ports can connect to each other (with a directed edge from the initiator to the server). As will be described in more detail below, query service 166 provides either relational views or graph views on top of data stored in database 142. Typically, a user will want to see data filtered using the activity graph. For example, if an entity was not involved in an activity in a given time period, that entity should be filtered out of query results. Thus, a request to show “all machines” in a given time frame will be interpreted as “show distinct machines that were active” during the time frame. Query service 166 relies on three main data model elements: fields, entities, and filters. As used herein, a field is a collection of values with the same type (logical and physical). As viewed by query service 166, an entity is a collection of fields that describe a data set. Some fields are common for all entities. One example of such a field is a “first observed” timestamp (when first use of the entity was detected). A second example of such a field is the entity classification type (e.g., one of: 1. Machine (on which an agent is installed), 2. Process, 3. Binary, 4. UID, 5. IP, 6. DNS Information, 7. ListenPort, and 8. PType). A third example of such a field is a “last observed” timestamp. A filter is an operator that: 1. takes an entity and field values as inputs, 2. a valid SQL expression with specific reference(s) of entity fields, or 3. is a conjunct/disjunct of filters. As will be described in more detail below, filters can be used to filter data in various ways, and limit data returned by query service 166 without changing the associated data set.

A dossier is a template for a collection of visualizations. Each visualization (e.g., the box including chart 3514) has a corresponding card, which identifies particular target information needed (e.g., from database 142) to generate the visualization. In some embodiments, platform 102 maintains a global set of dossiers/cards. Users of platform 102 such as Alice can build their own dashboard interfaces using preexisting dossiers/cards as components, and/or they can make use of a default dashboard (which incorporates various of such dossiers/cards). As an example of a Card Specification, a JSON file can be used to store multiple cards. A particular card is represented by a JSON object with a unique name as a field name.

Figure 36 illustrates an example of a card specification. The data set associated with the card shows a list of binary information running by users within the period of [StartTimeRange (3602), EndTimeRange (3604)]. The card depicted in Figure 36 is implemented by extending an existing (base) card as shown in sources field 3606 with explicit filters on ENTITY_TYPE (as shown in filters field 3608). It also adds dynamic filters via fkeys PID_KEY (3610) and implicit join. The ProcessClusterFilters Entity is a global external filter data set comprising composable internal data sets (which use the same syntax as cards such as the card depicted in Figure 36). The ProcessClusterFilters Entity defines a logical group of entities, the primary purpose of which is to form the scope of filters. Physical grouping will be determined at runtime by filters provided by users. The filters and grouping are referred to herein as implicit filters and joins, respectively. Similar to other derived entities, ProcessClusterFilters has the following standard JSON fields, with extended definitions: an array of references to entities, an array of external filter names that represent the schema of ProcessClusterFilters, an array of FilterKey(s), an array of implicit joins, and others. Returning to Figure 36, the dynamic application of filters via implicit join illustrates that the definition of the card (referred to as “Card 189” in the query service catalog) is transparent to future changes in ProcessClusterFilters. A second extensibility scenario is one in which a FilterKey in the filter data set is extended (i.e., existing template functions are used to define a new data set). As also mentioned above, data sets used by platform 102 are composable/reusable/extensible, irrespective of whether the data sets are relational or graph data sets. One example data set is the User Tracking polygraph, which is generated as a graph data set (comprising nodes and edges). Like other polygraphs, User Tracking is an external data set that can be visualized both as a graph (via the nodes and edges) and can also be used as a filter data set for other cards, via the cluster identifier (CID) field. An example of this scenario is depicted in Figure 37, in which three new user tracking data sets are added as new data sources for the ProcessClusterFilters Entity. Corresponding CID fields (UserTrackingIPAddress_CID, UserTrackingUser_CID, and UserTrackingDNSSep_CID) are also added as external filters to ProcessClusterFilters. Note that while changes will be made to ProcessClusterFilters, due to the extensibility techniques used herein, such changes will not impact card 3600.

As users such as Alice navigate through/interact with interfaces provided by platform 102 (e.g., as shown in Figure 33), such interactions trigger query service 166 to generate and perform queries against database 142. Dynamic composition of filter datasets can be implemented using FilterKeys and FilterKey Types. A FilterKey can be defined as a list of columns and/or fields in a nested structure (e.g., JSON). Instances of the same FilterKey Type can be formed as an Implicit Join Group. The same instance of a FilterKey can participate in different Implicit Join Groups. A list of relationships among all possible Implicit Join Groups is represented as a Join Graph for the entire search space to create a final data filter set by traversing edges and producing Join Path(s). Each card (e.g., as stored in the query service catalog and used in a dossier) can be introspected by a /card/describe/CardID REST request. Figures 38 and 39 both depict examples of card schema introspection. In particular, Figure 38 depicts introspection of a non-graph schema, while Figure 39 depicts introspection of a graph schema. In both cases, a request is made (via the REST API) for information pertinent to a particular card, in accordance with the specified schema, and subject to the specified filters.

At runtime (e.g., whenever it receives a request from web frontend 120), query service 166 parses the list of implicit joins and creates a Join Graph to manifest relationships of FilterKeys among Entities. A Join Graph (an example of which is depicted in Figure 40) comprises a list of Join Link(s). A Join Link represents each implicit join group by the same FilterKey type. A Join Link maintains a reverse map (Entity-to-FilterKey) of FilterKeys and their Entities. As previously mentioned, Entities can have more than one FilterKey defined. The reverse map guarantees one FilterKey per Entity can be used for each JoinLink. Each JoinLink also maintains a list of entities for the priority order of joins. Each JoinLink is also responsible for creating and adding directional edge(s) to graphs. An edge represents a possible join between two Entities. At runtime, each Implicit Join uses the Join Graph to find all possible join paths. The search of possible join paths starts with the outer FilterKey of an implicit join. One approach is to use a shortest path approach, with breadth first traversal and subject to the following criteria: use the priority order list of Join Links for all entities in the same implicit join group; stop when a node (Entity) is reached which has local filter(s); include all join paths at the same level (depth); exclude join paths based on the predefined rules (path of edges).

Figure 41 depicts an example introspection corresponding to the card depicted in Figure 36. In particular, in Figure 41, the card is filtered by an external DNS name. Figure 42 depicts an example query service log for a Join Path search (in accordance with the breadth first approach described above) corresponding to Figure 41. The shortest path is indicated at line 4202. Finally, the SQL generated by query service 166, using the result of the Join Path search, is depicted in Figure 43. Figure 44 depicts an example introspection corresponding to Figure 37. In particular, in Figure 44, filtering on Card 189 is performed by the User Tracking cluster. Figure 45 depicts an example query service log for a Join Path search corresponding to Figure 44. Finally, the SQL generated by query service 166, using the result of the Join Path search, is depicted in Figure 46. Examples of translating a filter and a join, respectively, to an SQL predicate are depicted in Figures 47A and 47B, and in Figures 47C and 47D.

Figure 48 illustrates an example of a process for dynamically generating and executing a query. In some embodiments, process 4800 is performed by platform 102. The process begins at 4802 when a request is received to filter information associated with activities within a network environment. One example of such a request occurs in response to Alice clicking on tab 3322. Another example of such a request occurs in response to Alice clicking on icon 3320. Yet another example of such a request occurs in response to Alice clicking on icon 3324 and selecting (e.g., from a dropdown) an option to filter based on specific criteria that she provides. At 4804, a query is generated based on an implicit join. One example of processing that can be performed at 4804 is as follows. As explained above, one way dynamic composition of filter datasets can be implemented is by using FilterKeys and FilterKey Types. And, instances of the same FilterKey Type can be formed as an Implicit Join Group. A Join Graph for the entire search space can be constructed from a list of all relationships among all possible Join Groups. And, a final data filter set can be created by traversing edges and producing one or more Join Paths. Finally, the shortest path in the join paths is used to generate an SQL query string.

One approach to generating an SQL query string is to use a query building library (authored in an appropriate language such as Java). Figure 49A illustrates a code excerpt of a common interface “sqlGen”. An example way that sqlGen can be used in conjunction with process 4800 is as follows. First, a card/entity is composed by a list of input cards/entities, where each input card recursively is composed by its own list of input cards. This nested structure can be visualized as a tree of query blocks(SELECT) in standard SQL constructs. SQL generation can be performed as the traversal of the tree from root to leaf entities (top-down), calling the sqlGen of each entity. Each entity can be treated as a subclass of the Java class(Entity). An implicit join filter (EntityFilter) is implemented as a subclass of Entity, similar to the right hand side of a SQL semi-join operator. Unlike the static SQL semi-join construct, it is conditionally and recursively generated even if it is specified in the input sources of the JSON specification. Another recursive interface can also be used in conjunction with process 4800, preSQLGen, which is primarily the entry point for

EntityFilter to run a search and generate nested implicit join filters. During preSQLGen recursive invocations, the applicability of implicit join filters is examined and pushed down to its input subquery list. Another top-down traversal, pullUpCachable, can be used to pull up common sub-query blocks, including those dynamically generated by preSQLGen, such that SELECT statements of those cacheable blocks are generated only once at top-level WITH clauses. A recursive interface, sqlWith, is used to generate nested subqueries inside WITH clauses. The recursive calls of a sqlWith function can generate nested WITH clauses as well. An sqlFrom function can be used to generate SQL FROM clauses by referencing those subquery blocks in the WITH clauses. It also produces INNER/OUTER join operators based on the joins in the specification. Another recursive interface, sqlWhere, can be used to generate conjuncts and disjuncts of local predicates and semi-join predicates based on implicit join transformations. Further, sqlProject, sqlGroupBy, sqlOrderBy, and sqlLimitOffset can respectively be used to translate the corresponding directives in JSON spec to SQL SELECT list, GROUP BY, ORDER BY, and LIMIT/OFFSET clauses. Returning to process 4800, at 4806, the query (generated at 4804) is used to respond to the request. As one example of the processing performed at 4806, the generated query is used to query database 142 and provide (e.g., to web app 120) fact data formatted in accordance with a schema (e.g., as associated with a card associated with the request received at 4802).

Figure 49B illustrates three examples of SQL entities. An SQL entity is composable using other entities in its statement, as well as input to other entities. Figures 50A and 50B illustrate, collectively, example portions of an embodiment of a ProcessClusterFilters definition. Figure 51 illustrates an example of an introspection corresponding to a ProcessClusterFilters request. Introspection can be performed by a /card/describe/CardID REST request – a dynamic card parsed and created by query service 166. The sources of the request are a target card (e.g., Card26) and other filter activity cards referenced by the fields in the filters. Figure 52A illustrates an example of a base table card. Figure 52B illustrates an example of a filter request, and Figure 52C illustrates an example of an SQL query (e.g., generated by query service 166 in response to the request). Figure 52D illustrates an additional example of a filter request, and Figure 52E illustrates a corresponding dynamically generated SQL query. Figures 53A, 53B, and 53C illustrate, respectively, a card, a request, and a generated SQL query corresponding to a join. Figures 53D and 53E illustrate, respectively, a request and a generated SQL query corresponding to an explicit filter join.

Figure 54A illustrates a pair of virtual machines: VM0 (5402) and VM1 (5404). In the example shown, the virtual machines are Linux-based and use the Kubernetes platform for managing containerized workloads and services. Other types of systems (including ones using other containerization techniques) can also be used in conjunction with techniques described herein. Included in virtual machines VM0 and VM1 are respective pods (with pods 5406 and 5408 on VM0 and pods 5410 and 5412 on VM1). A pod is an abstraction of a set of one or more containers deployed together on the same host. In multi-container pods, a data sharing model is typically used so that containers can exchange information with each other. Pods are managed by a master node (also referred to as an orchestrator), which is responsible for tasks such as vertical and horizontal scaling, and handling failover. The orchestrator is also responsible for managing communications between pods, including pods on the same virtual machine (e.g., communications between pods 5406 and 5436) and pods on different virtual machines (e.g., communications between pods 5406 and 5410). Different types of orchestration infrastructure use different approaches to handling such communications. As one example, in a Kubernetes environment, an overlay network is often used (e.g., Calico for Kubernetes or other appropriate overlay networking technology). One example of a pod is an app server pod that includes three different containers: the app server, a monitoring adapter, and a logging adapter. Another example of a pod includes an nginx webserver, a MySQL database server. In this example, the nginx container needs information to respond to requests. The data is provided to nginx by the Java application container, which obtains data from the MySQL container. Pods 5408 and 5412 are also referred to as “router pods.” Router pods are a system service and one router pod runs on each virtual machine. Also shown in Figure 54A are two respective agent pods (5430 and 5432) responsible for executing embodiments of agent 112.

Kubernetes makes use of a flat networking model. In this model, when one pod (e.g., pod 5406) needs to communicate with another pod, it does so by requesting access to a destination port (e.g., port 1031) without supplying a destination IP address. The orchestrator determines which particular pod on which particular virtual machine should be contacted and facilitates the connection. The orchestrator may select a pod collocated on the same virtual machine as the requesting pod or may select one on an entirely different machine. Various networking approaches may be used in different containerized environments. For flexibility in deployment, agent pods may be configured to dynamically determine in which type of environment they are executing and adjust their networking assumptions as applicable (e.g., to expose frames that would otherwise be hidden from the agent due to the network overlay).

Figure 54B illustrates an example one pod communicating with another. When pod 5406 needs to communicate with another pod, it sends a frame (5420) to router pod 5408 that includes a destination port. The orchestrator provides router pod 5408 with routing information (e.g., as a routing table that indicates how to route packets to particular service endpoints), and router pod 5408 determines whether the frame should be sent locally (e.g., to pod 5436) or remotely (e.g., to pod 5410). When the frame is to be sent locally, agent pod 5430 (listening to connections, e.g., via pcap as discussed above) can determine that, for example, pod 5406 is communicating with pod 5436, and provide the information as telemetry or other data to platform 102. In particular, and similar to techniques described above, agent pod 5430 is able to extract the source port, destination port, source IP, destination IP, and protocol out of the frame destined for pod 5436 from pod 5406. When a frame is to be sent remotely (5428), router pod 5408 encapsulates the frame (e.g., using the IP in IP tunneling protocol) and provides the encapsulated frame to interface 5414. The outer header of the encapsulated frame will include an IP address of virtual machine 5404. Router pod 5412 will extract the inner frame and provide it to pod 5410 (5426) based on the inner header of the encapsulated frame, which includes information usable by router pod 5412 to deliver the traffic to pod 5410. Unfortunately, the additional encapsulation can pose a challenge to agent pod 5430's ability to extract information about the communication. The inner header information will include a foreign IP address that is unknown to agent pod 5430, abstracting out information such as that pod 5406 is communicating with pod 5410 (5434) difficult (e.g., using techniques described in Section III. Above). Similarly, when router pod 5412 de-encapsulates the packet, the originating address in the inner header information will include a foreign IP address that is unknown to agent pod 5432. As a result, containerization can make alerting and/or other security enforcement techniques described herein difficult. One approach to addressing the situation is for agent pod 5430 to attempt to obtain routing information from the orchestrator master node. However, this may not be practical, or possible. Even were such information made available, it could be out of date and lead to false attribution or misattributions of which pods are communicating with which pods. In an alternate approach, agents 5430 and 5432 can be configured to track each hop of every frame (e.g., from pod 5406 to pod 5408 to interface 5414 to interface 5416 (5428) to pod 5412 to pod 5410) and provide the information to platform 102 to match up the connections. This approach may also not be practical or possible. As one example, significant resource overhead may be needlessly expended. Accordingly, in some embodiments, agent pods 5430 and 5432, working in cooperation with platform 102, use yet another alternate approach (using techniques described herein) that can efficiently determine and thus abstract out when pods communicate.

Although the examples described herein largely relate to embodiments where data is collected from agents and ultimately stored in a data store such as those provided by Snowflake, in other embodiments data that is collected from agents and other sources may be stored in different ways. For example, data that is collected from agents and other sources may be stored in a data warehouse, data lake, data mart, and/or any other data store.

A data warehouse may be embodied as an analytic database (e.g., a relational database) that is created from two or more data sources. Such a data warehouse may be leveraged to store historical data, often on the scale of petabytes. Data warehouses may have compute and memory resources for running complicated queries and generating reports. Data warehouses may be the data sources for business intelligence (“BI”) systems, machine learning applications, and/or other applications. By leveraging a data warehouse, data that has been copied into the data warehouse may be indexed for good analytic query performance, without affecting the write performance of a database (e.g., an Online Transaction Processing (“OLTP”) database). Data warehouses also enable the joining data from multiple sources for analysis. For example, a sales OLTP application probably has no need to know about the weather at various sales locations, but sales predictions could take advantage of that data. By adding historical weather data to a data warehouse, it would be possible to factor it into models of historical sales data.

Data lakes, which store files of data in their native format, may be considered as “schema on read” resources. As such, any application that reads data from the lake may impose its own types and relationships on the data. Data warehouses, on the other hand, are “schema on write,” meaning that data types, indexes, and relationships are imposed on the data as it is stored in the EDW. “Schema on read” resources may be beneficial for data that may be used in several contexts and poses little risk of losing data. “Schema on write” resources may be beneficial for data that has a specific purpose, and good for data that must relate properly to data from other sources. Such data stores may include data that is encrypted using homomorphic encryption, data encrypted using privacy-preserving encryption, smart contracts, non-fungible tokens, decentralized finance, and other techniques.

Data marts may contain data oriented towards a specific business line whereas data warehouses contain enterprise-wide data. Data marts may be dependent on a data

warehouse, independent of the data warehouse (e.g., drawn from an operational database or external source), or a hybrid of the two. In embodiments described herein, different types of data stores (including combinations thereof) may be leveraged. Such data stores may be proprietary or may be embodied as vendor provided products or services such as, for example, Google BigQuery, Druid, Amazon Redshift, IBM Db2, Dremio, Databricks Lakehouse Platform, Cloudera, Azure Synapse Analytics, and others.

The deployments (e.g., a customer's cloud deployment) that are analyzed, monitored, evaluated, or otherwise observed by the systems described herein (e.g., systems that include components such as the platform 102 of Figure 1, the data collection agents described herein, and/or other components) may be provisioned, deployed, and/or managed using infrastructure as code ('IaC'). IaC involves the managing and/or provisioning of infrastructure through code instead of through manual processes. With IaC, configuration files may be created that include infrastructure specifications. IaC can be beneficial as configurations may be edited and distributed, while also ensuring that environments are provisioned in a consistent manner. IaC approaches may be enabled in a variety of ways including, for example, using IaC software tools such as Terraform by HashiCorp. Through the usage of such tools, users may define and provide data center infrastructure using JavaScript Object Notation ('JSON'), YAML, proprietary formats, or some other format. In some embodiments, the configuration files may be used to emulate a cloud deployment for the purposes of analyzing the emulated cloud deployment using the systems described herein. Likewise, the configuration files themselves may be used as inputs to the systems described herein, such that the configuration files may be inspected to identify vulnerabilities, misconfigurations, violations of regulatory requirements, or other issues. In fact, configuration files for multiple cloud deployments may even be used by the systems described herein to identify best practices, to identify configuration files that deviate from typical configuration files, to identify configuration files with similarities to deployments that have been determined to be deficient in some way, or the configuration files may be leveraged in some other ways to detect vulnerabilities, misconfigurations, violations of regulatory requirements, or other issues prior to deploying an infrastructure that is described in the configuration files. In some embodiments the techniques described herein may be use in multi-cloud, multi-tenant, cross-cloud, cross-tenant, cross-user, industry cloud, digital platform, and other scenarios depending on specific need or situation.

In some embodiments, the deployments (e.g., a customer's cloud deployment) that are analyzed, monitored, evaluated, or otherwise observed by the systems described herein (e.g., systems that include components such as the platform 102 of Figure 1, the data collection agents described herein, and/or other components) may also be provisioned, deployed, and/or managed using security as code ('SaC'). SaC extends IaC concepts by defining cybersecurity policies and/or standards programmatically, so that the policies and/or standards can be referenced automatically in the configuration scripts used to provision cloud deployments. Stated differently, SaC can automate policy implementation and cloud deployments may even be compared with the policies to prevent "drift." For example, if a policy is created where all personally identifiable information ('PII') or personal health information ('PHI') must be encrypted when it is stored, that policy is translated into a process that is automatically launched whenever a developer submits code, and code that violates the policy may be automatically rejected.

In some embodiments, SaC may be implemented by initially classifying workloads (e.g., by sensitivity, by criticality, by deployment model, by segment). Policies that can be instantiated as code may subsequently be designed. For example, compute-related policies may be designed, access-related policies may be designed, application-related policies may be designed, network-related policies may be designed, data-related policies may be designed, and so on. Security as code may then be instantiated through architecture and automation, as successful implementation of SaC can benefit from making key architectural-design decisions and executing the right automation capabilities. Next, operating model protections may be built and supported. For example, an operating model may "shift left" to maximize self-service and achieve full-life-cycle security automation (e.g., by standardizing common development toolchains, CI/CD pipelines, and the like). In such an example, security policies and access controls may be part of the pipeline, automatic code review and bug/defect detection may be performed, automated build processes may be performed, vulnerability scanning may be performed, checks against a risk-control framework may be made, and other tasks may be performed all before deploying an infrastructure or components thereof.

As described in other portions of the present disclosure, the systems described herein may be used to analyze, monitor, evaluate, or otherwise observe a customer's cloud deployment. While securing traditional datacenters requires managing and securing an IP-based perimeter with networks and firewalls, hardware security modules ('HSMs'), security information and event management ('SIEM') technologies, and other physical access restrictions, such solutions are not particularly useful when applied to cloud deployments. As such, the systems described herein may be configured to interact with and even monitor other solutions that are appropriate for cloud deployments such as, for example, "zero trust" solutions.

A zero trust security model (a.k.a., zero trust architecture) describes an approach to the design and implementation of IT systems. A primary concept behind zero trust is that devices should not be trusted by default, even if they are connected to a managed corporate network such as the corporate LAN and even if they were previously verified. Zero trust security models help prevent successful breaches by eliminating the concept of trust from an organization's network architecture. Zero trust security models can include multiple forms of authentication and authorization (e.g., machine authentication and authorization, human/user authentication and authorization) and can also be used to control multiple types of accesses or interactions (e.g., machine-to-machine access, human-to-machine access).

In some embodiments, the systems described herein may be configured to interact with zero trust solutions in a variety of ways. For example, agents that collect input data for the systems described herein (or other components of such systems) may be configured to access various machines, applications, data sources, or other entity through a zero trust solution, especially where local instances of the systems described herein are deployed at edge locations. Likewise, given that zero trust solutions may be part of a customer's cloud deployment, the zero trust solution itself may be monitored to identify vulnerabilities, anomalies, and so on. For example, network traffic to and from the zero trust solution may be analyzed, the zero trust solution may be monitored to detect unusual interactions, log files generated by the zero trust solution may be gathered and analyzed, and so on.

In some embodiments, the systems described herein may leverage various tools and mechanisms in the process of performing its primary tasks (e.g., monitoring a cloud deployment). For example, Linux eBPF is mechanism for writing code to be executed in the Linux kernel space. Through the usage of eBPF, user mode processes can hook into specific trace points in the kernel and access data structures and other information. For example, eBPF may be used to gather information that enables the systems described herein to attribute the utilization of networking resources or network traffic to specific processes. This may be useful in analyzing the behavior of a particular process, which may be important for observability/SIEM.

The systems described may be configured to collect security event logs (or any other type of log or similar record of activity) and telemetry in real time for threat detection, for analyzing compliance requirements, or for other purposes. In such embodiments, the systems described herein may analyze telemetry in real time (or near real time), as well as historical telemetry, to detect attacks or other activities of interest. The attacks or activities of interest may be analyzed to determine their potential severity and impact on an organization. In fact, the attacks or activities of interest may be reported, and relevant events, logs, or other information may be stored for subsequent examination.

In one embodiment, systems described herein may be configured to collect security event logs (or any other type of log or similar record of activity) and telemetry in real time to provide customers with a SIEM or SIEM-like solution. SIEM technology aggregates event data produced by security devices, network infrastructure, systems, applications, or other source. In some embodiments the primary data source is log data, but other forms of data, such as network telemetry data (flows and packets) may also be utilized. In some embodiments, event data can be combined with contextual information about users, assets, threats, vulnerabilities, and so on, for the purposes of scoring, prioritization and expediting investigations. In some embodiments, input data may be normalized, so that events, data, contextual information, or other information from disparate sources can be analyzed more efficiently for specific purposes (e.g., network security event monitoring, user activity monitoring, compliance reporting). The embodiments described here offer real-time analysis of events for security monitoring, advanced analysis of user and entity behaviors, querying and long-range analytics for historical analysis, other support for incident investigation and management, reporting (for compliance requirements, for example), and other functionality.

In some embodiments, the systems described herein may be part of an application performance monitoring ('APM') solution. APM software and tools enable the observation of application behavior, observation of its infrastructure dependencies, observation of users and business key performance indicators ('KPIs') throughout the application's life cycle, and more. The applications being observed may be developed internally, as packaged applications, as software as a service ('SaaS'), or embodied in some other ways. In such embodiments, the systems described herein may provide one or more of the following capabilities:

- The ability to operate as an analytics platform that ingests, analyzes, and builds context from traces, metrics, logs, and other sources.

- Automated discovery and mapping of an application and its infrastructure components.
- Observation of an application's complete transactional behavior, including interactions over a data communications network.
- Monitoring of applications running on mobile (native and browser) and desktop devices.
- Identification of probable root causes of an application's performance problems and their impact on business outcomes.
- Integration capabilities with automation and service management tools.
- Analysis of business KPIs and user journeys (for example, login to check-out).
- Domain-agnostic analytics capabilities for integrating data from third-party sources.
- Endpoint monitoring to understand the user experience and its impact on business outcomes.
- Support for virtual desktop infrastructure ('VDI') monitoring.

In embodiments where the systems described herein are used for APM, some components of the system may be modified, other components may be added, some components may be removed, and other components may remain the same. In such an example, similar mechanisms as described elsewhere in this disclosure may be used to collect information from the applications, network resources used by the application, and so on. The graph based modelling techniques may also be leveraged to perform some of the functions mentioned above, or other functions as needed.

In some embodiments, the systems described herein may be part of a solution for developing and/or managing artificial intelligence ('AI') or machine learning ('ML') applications. For example, the systems described herein may be part of an AutoML tool that automate the tasks associated with developing and deploying ML models. In such an example, the systems described herein may perform various functions as part of an AutoML tool such as, for example, monitoring the performance of a series of processes, microservices, and so on that are used to collectively form the AutoML tool. In other embodiments, the systems described herein may perform other functions as part of an AutoML tool or may be used to monitor, analyze, or otherwise observe an environment that the AutoML tool is deployed within.

In some embodiments, the systems described herein may be used to manage, analyze, or otherwise observe deployments that include other forms of AI/ML tools. For example, the systems described herein may manage, analyze, or otherwise observe deployments that include AI services. AI services are, like other resources in an as-a-service model, ready-made models and AI applications that are consumable as services and made available through APIs. In such an example, rather than using their own data to build and train models for common activities, organizations may access pre-trained models that accomplish specific tasks. Whether an organization needs natural language processing ('NLP'), automatic speech recognition ('ASR'), image recognition, or some other capability, AI services simply plug-and-play into an application through an API. Likewise, the systems described herein may be used to manage, analyze, or otherwise observe deployments that include other forms of AI/ML tools such as Amazon SageMaker (or other cloud machine-learning platform that enables developers to create, train, and deploy ML models) and related services such as Data Wrangler (a service to accelerate data prep for ML) and Pipelines (a CI/CD service for ML).

In some embodiments, the systems described herein may be used to manage, analyze, or otherwise observe deployments that include various data services. For example, data services may include secure data sharing services, data marketplace services, private data exchanges services, and others. Secure data sharing services can allow access to live data from its original location, where those who are granted access to the data simply reference the data in a controlled and secure manner, without latency or contention from concurrent users. Because changes to data are made to a single version, data remains up-to-date for all consumers, which ensures data models are always using the latest version of such data. Data marketplace services operate as a single location to access live, ready-to-query data (or data that is otherwise ready for some other use). A data marketplace can even include a "feature stores," which can allow data scientists to repurpose existing work. For example, once a data scientist has converted raw data into a metric (e.g., costs of goods sold), this universal metric can be found quickly and used by other data scientists for quick analysis against that data.

In some embodiments, the systems described herein may be used to manage, analyze, or otherwise observe deployments that include distributed training engines or similar mechanisms such as, for example, such as tools built on Dask. Dask is an open source library for parallel computing that is written in Python. Dask is designed to enable data scientists to improve model accuracy faster, as Dask enables data scientists can do everything in Python end-to-end, which means that they no longer need to convert their code to execute in environments like Apache Spark. The result is reduced complexity and increased efficiency. The systems described herein may also be used to manage, analyze, or otherwise observe deployments that include technologies such as RAPIDS (an open source Python framework which is built on top of Dask). RAPIDS optimizes compute time and speed by providing data pipelines and executing data science code entirely on graphics processing units (GPUs) rather than CPUs. Multi-cluster, shared data architecture, DataFrames, Java user-defined functions (UDF) are supported to enable trained models to run within a data warehouse.

In some embodiments, the systems described herein may be leveraged for the specific use case of detecting and/or remediating ransomware attacks and/or other malicious action taken with respect to data, systems, and/or other resources associated with one or more entities. Ransomware is a type of malware from cryptovirology that threatens to publish the victim's data or perpetually block access to such data unless a ransom is paid. In such embodiments, ransomware attacks may be carried out in a manner such that patterns (e.g., specific process-to-process communications, specific data access patterns, unusual amounts of encryption/re-encryption activities) emerge, where the systems described herein may monitor for such patterns. Alternatively, ransomware attacks may involve behavior that deviates from normal behavior of a cloud deployment that is not experiencing a ransomware attack, such that the mere presence of unusual activity may trigger the systems described herein to generate alerts or take some other action, even without explicit knowledge that the unusual activity is associated with a ransomware attack.

In some embodiments, particular policies may be put in place the systems described herein may be configured to enforce such policies as part of an effort to thwart ransomware attacks. For example, particular network sharing protocols (e.g., Common Internet File System ('CIFS'), Network File System ('NFS')) may be avoided when implementing storage for backup data, policies that protect backup systems may be implemented and enforced to ensure that usable backups are always available, multifactor authentication for particular accounts may be utilized and accounts may be configured with the minimum privilege required to function, isolated recovery environments may be created and isolation may be monitored and enforced to ensure the integrity of the recovery environment, and so on. As described in the present disclosure, the systems described herein may be configured to explicitly enforce such policies or may be configured to detect unusual activity that represents a violation of such policies, such that the mere presence of unusual activity may trigger the systems described herein to generate alerts or take some other action, even without explicit knowledge that the unusual activity is associated with a violation of a particular policy.

Readers will appreciate that ransomware attacks are often deployed as part of a larger attack that may involve, for example:

- Penetration of the network through means such as, for example, stolen credentials and remote access malware.
- Stealing of credentials for critical system accounts, including subverting critical administrative accounts that control systems such as backup, Active Directory ('AD'), DNS, storage admin consoles, and/or other key systems.
- Attacks on a backup administration console to turn off or modify backup jobs, change retention policies, or even provide a roadmap to where sensitive application data is stored.
- Data theft attacks.

As a result of the many aspects that are part of a ransomware attack, embodiments of the present disclosure may be configured as follows:

- The systems may include one or more components that detect malicious activity based on the behavior of a process.
- The systems may include one or more components that store indicator of compromise ('IOC') or indicator of attack ('IOA') data for retrospective analysis.
- The systems may include one or more components that detect and block fileless malware attacks.
- The systems may include one or more components that remove malware automatically when detected.
- The systems may include a cloud-based, SaaS-style, multitenant infrastructure.
- The systems may include one or more components that identify changes made by malware and provide the recommended remediation steps or a rollback capability.
- The systems may include one or more components that detect various application vulnerabilities and memory exploit techniques.

- The systems may include one or more components that continue to collect suspicious event data even when a managed endpoint is outside of an organization's network.
- The systems may include one or more components that perform static, on-demand malware detection scans of folders, drives, devices, or other entities.
- The systems may include data loss prevention (DLP) functionality.

In some embodiments, the systems described herein may manage, analyze, or otherwise observe deployments that include deception technologies. Deception technologies allow for the use of decoys that may be generated based on scans of true network areas and data. Such decoys may be deployed as mock networks running on the same infrastructure as the real networks, but when an intruder attempts to enter the real network, they are directed to the false network and security is immediately notified. Such technologies may be useful for detecting and stopping various types of cyber threats such as, for example, Advanced Persistent Threats ("APT's"), malware, ransomware, credential dumping, lateral movement and malicious insiders. To continue to outsmart increasingly sophisticated attackers, these solutions may continuously deploy, support, refresh and respond to deception alerts.

In some embodiments, the systems described herein may manage, analyze, or otherwise observe deployments that include various authentication technologies, such as multi-factor authentication and role-based authentication. In fact, the authentication technologies may be included in the set of resources that are managed, analyzed, or otherwise observed as interactions with the authentication technologies may be monitored. Likewise, log files or other information retained by the authentication technologies may be gathered by one or more agents and used as input to the systems described herein.

In some embodiments, the systems described herein may be leveraged for the specific use case of detecting supply chain attacks. More specifically, the systems described herein may be used to monitor a deployment that includes software components, virtualized hardware components, and other components of an organization's supply chain such that interactions with an outside partner or provider with access to an organization's systems and data can be monitored. In such embodiments, supply chain attacks may be carried out in a manner such that patterns (e.g., specific interactions between internal and external systems) emerge, where the systems described herein may monitor for such patterns. Alternatively, supply chain attacks may involve behavior that deviates from normal behavior of a cloud deployment that is not experiencing a supply chain attack, such that the mere presence of unusual activity may trigger the systems described herein to generate alerts or take some other action, even without explicit knowledge that the unusual activity is associated with a supply chain attack.

In some embodiments, the systems described herein may be leveraged for other specific use cases such as, for example, detecting the presence of (or preventing infiltration from) cryptocurrency miners (e.g., bitcoin miners), token miners, hashing activity, non-fungible token activity, other viruses, other malware, and so on. As described in the present disclosure, the systems described herein may monitor for such threats using known patterns or by detecting unusual activity, such that the mere presence of unusual activity may trigger the systems described herein to generate alerts or take some other action, even without explicit knowledge that the unusual activity is associated with a particular type of threat, intrusion, vulnerability, and so on.

The systems described herein may also be leveraged for endpoint protection, such the systems described herein form all of or part of an endpoint protection platform. In such an embodiment, agents, sensors, or similar mechanisms may be deployed on or near managed endpoints such as computers, servers, virtualized hardware, internet of things ("IoT") devices, mobile devices, phones, tablets, watches, other personal digital devices, storage devices, thumb drives, secure data storage cards, or some other entity. In such an example, the endpoint protection platform may provide functionality such as:

- Prevention and protection against security threats including malware that uses file-based and fileless exploits.
- The ability to apply control (allow/block) to access of software, scripts, processes, microservices, and so on.
- The ability to detect and prevent threats using behavioral analysis of device activity, application activity, user activity, and/or other data.
- The ability for facilities to investigate incidents further and/or obtain guidance for remediation when exploits evade protection controls
- The ability to collect and report on inventory, configuration and policy management of the endpoints.
- The ability to manage and report on operating system security control status for the monitored endpoints.
- The ability to scan systems for vulnerabilities and report/manage the installation of security patches.
- The ability to report on internet, network and/or application activity to derive additional indications of potentially malicious activity.

Example embodiments are described in which policy enforcement, threat detection, or some other function is carried out by the systems described herein by detecting unusual activity, such that the mere presence of unusual activity may trigger the systems described herein to generate alerts or take some other action, even without explicit knowledge that the unusual activity is associated with a particular type of threat, intrusion, vulnerability, and so on. Although these examples are largely described in terms of identifying unusual activity, in these examples the systems described herein may be configured to learn what constitutes 'normal activity' – where 'normal activity' is activity observed, modeled, or otherwise identified in the absence of a particular type of threat, intrusion, vulnerability, and so on. As such, detecting 'unusual activity' may alternatively be viewed as detecting a deviation from 'normal activity' such that 'unusual activity' does not need to be identified and sought out. Instead, deviations from 'normal activity' may be assumed to be 'unusual activity'.

Readers will appreciate that while specific examples of the functionality that the systems described herein can provide are included in the present disclosure, such examples are not to be interpreted as limitations as to the functionality that the systems described herein can provide. Other functionality may be provided by the systems described herein, all of which are within the scope of the present disclosure. For the purposes of illustration and not as a limitation, additional examples can include governance, risk, and compliance ("GRC"), threat detection and incident response, identity and access management, network and infrastructure security, data protection and privacy, identity and access management ("IAM"), and many others.

In order to provide the functionality described above, the systems described herein or the deployments that are monitored by such systems may implement a variety of techniques. For example, the systems described herein or the deployments that are monitored by such systems may tag data and logs to provide meaning or context, persistent monitoring techniques may be used to monitor a deployment at all times and in real time, custom alerts may be generated based on rules, tags, and/or known baselines from one or more polygraphs, and so on.

Although examples are described above where data may be collected from one or more agents, in some embodiments other methods and mechanisms for obtaining data may be utilized. For example, some embodiments may utilize agentless deployments where no agent (or similar mechanism) is deployed on one or more customer devices, deployed within a customer's cloud deployment, or deployed at another location that is external to the data platform. In such embodiments, the data platform may acquire data through one or more APIs such as the APIs that are available through various cloud services. For example, one or more APIs that enable a user to access data captured by Amazon CloudTrail may be utilized by the data platform to obtain data from a customer's cloud deployment without the use of an agent that is deployed on the customer's resources. In some embodiments, agents may be deployed as part of a data acquisition service or tool that does not utilize a customer's resources or environment. In some embodiments, agents (deployed on a customer's resources or elsewhere) and mechanisms in the data platform that can be used to obtain data from through one or more APIs such as the APIs that are available through various cloud services may be utilized. In some embodiments, one or more cloud services themselves may be configured to push data to some entity (deployed anywhere), which may or may not be an agent. In some embodiments, other data acquisition techniques may be utilized, including combinations and variations of the techniques described above, each of which is within the scope of the present disclosure.

Readers will appreciate that while specific examples of the cloud deployments that may be monitored, analyzed, or otherwise observed by the systems described herein have been provided, such examples are not to be interpreted as limitations as to the types of deployments that may be monitored, analyzed, or otherwise observed by the systems described herein. Other deployments may be provided by the systems described herein, all of which are within the scope of the present disclosure. For the purposes of illustration and not as a limitation, additional examples can include multi-cloud deployments, on-premises environments, hybrid cloud environments, sovereign cloud environments, heterogeneous environments, DevOps environments, DevSecOps environments, GitOps environments, quantum computing environments, data fabrics, composable applications, composable networks, decentralized applications, and many others.

Readers will appreciate that while specific examples of the types of data that may be collected, transformed, stored, and/or analyzed by the systems described herein have been provided, such examples are not to be interpreted as limitations as to the types of data that may be collected, transformed, stored, and/or analyzed by the systems described herein. Other types of data can include, for example, data collected from different tools (e.g., DevOps tools, DevSecOps, GitOps tools), different

forms of network data (e.g., routing data, network translation data, message payload data, Wifi data, Bluetooth data, personal area networking data, payment device data, near field communication data, metadata describing interactions carried out over a network, and many others), data describing processes executing in a container, lambda, EC2 instance, virtual machine, or other execution environment), information describing the execution environment itself, and many other types of data.

The systems described above may be deployed in a variety of ways. In one embodiment, for example, the systems described above may be deployed as a Secure Access Service Edge ('SASE') or SASE-like solution. In such an embodiment, one or more portions of the systems described above may be deployed at edge sites.

For example, one or more portions of the systems described above may be at or near each population center in a given geographic region. The one or more portions of the systems described above may be deployed, for example, on infrastructure provided by a cloud services provider that is in a particular region, availability zone, local zone, or similar location that is proximate to a particular population center that a particular deployment is intended to serve.

Readers will appreciate that by deploying the systems described above as a SASE or SASE-like solution, a variety of benefits can be achieved. Consider an example in which a particular business organization desires to provide access to one of their internal applications (e.g., an order tracking application) to one or more users. In traditional examples, such an application may be deployed on one or more servers that are reachable via an internal network. In such an example, users may be required to access such an application via a virtual private network ('VPN') or similar mechanism. In such an example, however, once the user has VPN access, the user may have access to many applications, data stores, machines, and other entities that are reachable via the VPN. Furthermore, if the application is deployed in a cloud computing environment, traditional mechanisms like a VPN aren't available. Embodiments of the present disclosure resolve these shortcomings by: 1) providing more granular access to particular resources (rather than providing access to networks that include the particular resources and possibly other resources), and 2) providing security services for resources that are deployed in a cloud environment, where traditional mechanisms like a VPN are not useful.

In some embodiments, one or more portions of the systems described above may be deployed to implement a zero trust architecture rather than a VPN or similar mechanism, such that devices are not to be trusted by default, even if they are connected to a managed network such as a corporate LAN or a VPN, or even if the devices were previously verified. Furthermore, in some embodiments a variety of capabilities may be offered that are not associated with a VPN or similar mechanism. For example, SaaS application security features may be incorporated in embodiments of the present disclosure such that applications may be monitored to detect anomalous or suspicious behavior (e.g., someone is downloading all files available via a document sharing application that executes in a cloud environment). In addition, DNS security features may be incorporated in embodiments of the present disclosure such that attempts to access suspicious DNS can be identified. Likewise, SSL inspection security features may be incorporated in embodiments of the present disclosure such that SSL can be cracked open and inspected to support improved malware protection, document loss prevention, and so on.

Some embodiments of the present disclosure can use the systems described above, where one or more portions of such systems are deployed in a geographically distributed manner, to perform continuous risk behavior based security. For example, the following may be performed:

- User anomaly detection
- Risk adjustment events

In some embodiments, the systems described above (where one or more portions of such systems are deployed in a geographically distributed manner) may be utilized as a middle layer between devices and applications that are to be accessed by the devices. Consider an example in which the set of devices that will access applications via this middle layer include laptops, smartphones, desktop computer, and many other types of devices that are utilized by personnel within a particular business organization. In such an example, each of these distinct devices may be used to access at least three types of applications:

- a. Internal, private applications: These are applications whose access should generally be restricted to the business organization's employees. These applications can be a diverse set of applications such as, for example, a company's inventory system, a company's HR database, a company's internal automation servers for assisting with the development and integration of software applications, and many other types of applications.
- b. SaaS applications: These are applications that, while publicly available, may be configured for a particular organization with access that is generally limited to the business organization's employees, although other organizations may also utilize their own instances of these applications. Such applications can include, for example, a software productivity suite such as Microsoft Office365, Gsuite, financial tools such as Netsuite, Oracle, SAP, human resources tools such as Workday, Greenhouse, Paylocity, a company's CRM platform as provided by companies like Salesforce, ServiceNow, a marketing tool such as Marketo, and many others.
- c. Open Web applications: These are applications that are publicly available accessible via tools such as, for example, a web browser.

In embodiment of the present disclosure, the middle layer described above may offer access to each type of application in distinct ways. For example, for the first type of application (Internal, private applications), the first type of application may be configured with a 'dial out' connector such that the first type of application attempts to connect to the middle layer via a zero trust application that runs in the middle layer to that can verify the first type of application or perform similar steps. Because the connector operates in a dial out mode, steps like opening a firewall, connecting to VPNs, and similar steps do not need to occur. In such an example, a user device that is attempting to access these applications can also connect to the zero trust application in that is running in the middle layer that verify the device or perform similar steps. In such an example, communications between the device and the first type of application can subsequently be routed through the middle layer. In this example, the middle layer will also provide many of the functions and capabilities described above with respect to the accessing devices and the first type of application such as, for example, monitoring for anomalies, monitoring user and application behavior, and performing all of the other security related tasks, compliance related tasks, and so on.

Continuing with this example, when a device attempts to access the second type of application, a different process may be carried out by the middle layer. In such an example, the middle layer can include a SaaS security application that a device can utilize to access the SaaS software. This SaaS security application can perform steps such as, for example, ensuring that users are authenticated before they can access the second type of application. As was the case with respect to the first type of application, the middle layer will also provide many of the functions and capabilities described above with respect to the accessing devices and the second type of application such as, for example, monitoring for anomalies, monitoring user and application behavior, and performing all of the other security related tasks, compliance related tasks, and so on.

Continuing with this example, when a device attempts to access the third type of application, a different process may be carried out by the middle layer. In such an example, the middle layer can include an inspection application that performs tasks such as DNS and SSL inspection. As was the case with respect to the first and second types of application, the middle layer will also provide many of the functions and capabilities described above with respect to the accessing devices and the third type of application such as, for example, monitoring for anomalies, monitoring user and application behavior, and performing all of the other security related tasks, compliance related tasks, and so on. In this example, things like malware detection/protection, device comprise detection, and similar features aimed at protecting the devices (with little or no concern about protecting the third type of application) can be performed.

The middle layer may also include real-time visibility and policy enforcement software. The ability to perform real-time visibility and policy enforcement may span accesses to all of the types of applications described above. Performing real-time visibility and policy enforcement may be carried out, for example, such that any policy that is created may be immediately distributed to each geographic location where one or more modules of the systems described above are being executed such that the policy may be immediately enforced for any applications or devices that are connected to the instance of the system that is executing in a proximate location. The middle layer may also include a user behavior and anomaly detection layer software, as well as events, workflows, and automatic risk management software.

Readers will appreciate that multiple instances of the middle layer may be running at various geographic locations. In such an example, each device that is associated with the business organization can access applications using the closest middle layer instance, such that all of the functionality of the systems described above can be pushed to a network edge rather than being deployed as a centralized service. In such an example, information such as PII may be maintained in a such a way that complies with relevant regulations (e.g., EU GDPR, California CCPA, Canadian PIPEDA, Brazilian LGPD) such that data isn't transport across boundaries in violation of one or more regulatory requirements. In such embodiments, the agents that are similar to those described above may be running on the user device and collecting information as described above.

The embodiments described above may enable a variety of security related functions. For example, users with risky behavior can be identified, over provisioned privileges may be identified (e.g., user has privileges that they don't need or use), user geofencing can be enforced, device compromise can be detected, phished or lost credentials may be identified, malicious SaaS application usage may be detected, malicious insiders may be detected, malicious document accesses may be detected, and so on. Once particular events or conditions are detected, events may be managed automatically and workflows may be initiated to involve parties (e.g., administrators, executives) that can implement or approved remedial actions via one or more alerts, remedial actions may be automatically initiated, users may be alerted to their own behavior, and so on.

Enterprise employees rely heavily on the network to do their work. Starting all the way from being a remote and global workforce, increasing use of SaaS and all the personal content flowing in from the internet. This also fits with current attack models. Most attacks are delivered over the network, be it email Phishing, or Browser download, or drive by attacks. USB, or physical threat to laptops and employees has diminished over time. As such, network security as a service (NSaaS) focuses on:

- A new innovative way to deliver security at the network level.
- NSaaS has some advantages
- Security needed by the corporate workforce. I.e. NSaaS is between employees and devices and services they need access to, to do their work.

Security and user experience (UX) often are at odds with each other and it is important for us to decide how to break the use cases. In general to make things secure, sometimes the user's workflow has to be changed or becomes a little less inconvenient. In some embodiments, the UX adheres to one or more of the following:

- No change to user's workflow or setup
- E.g. Asking users to change their applications or their IP space etc.
- Auto-discover or not require a discovery as much as possible. Users do not enter hard to gather information
- Administrators are able to use the product out of the box with good security

Zero trust is a much better way to access private applications vs VPN. We want to adhere to zero trust as a first principle and design the product to be as secure as architecture allows. This implies no laptop is able to discover or connect to any service that policy does not allow for. Since we do not want to change user applications, we expect to do this in a transparent way. Zero trust is also a very important feature for us for initial insertion into our ICP.

Data loss protection (DLP) can involve many features from many different markets and vendors. On the network, content is accessed. In some embodiments, the content is checked. In some embodiments, a data polygraph is used to identify, based on a time series of data, temporal anomalies. In some embodiments, document usage is tagged with user tags to find anomalous activity. In some embodiments, a hash based server-side tracks content.

In some embodiments, DLP meets one or more regulations (e.g., HIPAA). In some embodiments, storage (e.g., Amazon S3) accessibility changes are monitored (e.g., non-public to public), finds sensitive info in buckets (e.g., SSN, keys), and allows searching/filtering of findings. In some embodiments, NSaaS includes compliance, discover risky activity, alert through ML, log analysis. Scans disks and other storage, prevent digital right files from saved to USB, network and web monitor, protects data movement between cloud apps. Image recognition, data fingerprinting. In some embodiments, NSaaS includes activity visibility and search, granular contextual policies such as "No downloads of PII onto personal devices", supports 500+ files types, 3000+ data identifiers, fingerprinting, exact match; Advanced date types such as resume. Policy actions include encrypt, block, quarantine; Exfiltration protection; Usage anomaly detection through rule + ML. In some embodiments, NSaaS includes activity analysis and dashboards. In some embodiments, NSaaS includes activity visibility to uncover shadow IT, find patterns, policies and alerts through conditions and actions, enforce permission, apply custom policies on sensitive data types.

In some embodiments, user privacy adheres to one or more of the following:

- A user is able to view her own data
- An administrator may only be allowed to view aggregated views and data
- There are 'Super-admins' who are allowed to view individual user data
 - This comes with a strict Audit log capturing activity monitored.
 - User needs to enter a comment to describe the reason activity is being viewed.
 - It is hard to accidentally investigate. I.e. workflow may make it explicit investigation is in progress
- We may minimize side channel information leak.

Cloud Access Security Broker (CASB) – can be divided into two parts:

- **Inline** - Proxy based inspection and enforcement.
In this mode CASB products typically act as a proxy and can do content inspection, or policy enforcement. Policies can range from FQDN, or action based policies.
- **Offline** - Access Log based analysis
In this mode, SaaS Access logs can be analyzed, in an offline manner, to find anomalous behavior. This is more suited for a SIEM or Analytics based approach

In some embodiments, inline is broken down to:

- Proxy content inspection
- Proxy Visibility
- Proxy Policy enforcement

Risk - Most people use risk to prioritize the security activity. In some embodiments, Risk is used as a way to give user visibility and take meaningful actions. Every Entity (User, Device, Application, SaaS, Document) has an associated risk based on their behavior and relationship with other entities. Risk is learned in some embodiments, using Polygraphs and apply it to create a unique and differentiated view for Administrators.

Visibility/Analytics – In some embodiments, NSaaS is distributed across multiple regions. In some embodiments, visibility and analytics are implemented.

NACL - Many applications like AWS, Snowflake use NACL as a way to limit accessibility and thus risk of application compromise. In some embodiments, NSaaS uses a well known set of Ips to allow for whitelisting in other applications as needed. To prevent the use of blacklisted IPs, the use of public IPs is managed.

UX for customers is improved in a meaningful way by supporting multiple K8s clusters in a seamless way, and provide a single consistent place to apply policy and get visibility. K8s and CSPs provide a way to be able to access every container, or VM under their purview via a control channel (E.g kubectl).

Global Security View – In some embodiments, NSaaS creates a global security view:

- Ability to see, analyze and understand **all user activity** for entire corporation
- Policy documents that can be written once and applied at various levels.
 - Policy documents that have access to context from multiple different security services
- Evaluate Risk based on temporal access of every user, device and be able to use it to evaluate and write policies on it

Policy Engine – a policy engine is applied in a stateless way across every transaction. In other words, all the context needed to apply a policy is cached server side and, as a transaction is in progress, the policy is applied on the transaction. In some embodiments, distribution affinity is implemented to facilitate a large context cache,

In some embodiments, NSaaS is used across one or more of the following use cases:

Employee protection- For all user activity, a context-rich data is maintained on the user and the activity. It also maintains a past history and using both creates a Risk profile for the user. All new activity is judged against this score.

SSL Inspection- NSaaS can act as a MITM proxy and decrypt all user traffic. Security checks are applied to the decrypted traffic. PKI Certs are automatically managed by EG for all sites and customers.

Malicious downloads - User downloads from the www are checked for malicious content. We check against Third party Intelligence feeds as well as generated feeds.

Malicious URLs - URLs visited by users are checked for maliciousness against 3rd party feeds like VirusTotal and ReversingLabs.

DNS Security - FQDNs visited by users are checked for maliciousness. For every FQDN, a risk score is maintained based on the DNS name, how recently it was registered and usage by others around the world. If DNS is deemed risky, automatic actions may be taken on it, including blocking it.

Media protection - Administrators can decide to block content based on media type as needed. E.g. If Java plugins are not desired, NSaaS can block them at the service level. Or a branch office may have limited bandwidth and customers might want to block youtube. Or rate limit docker uploads/downloads on a per-user basis.

Automated Polygraph alerts - NSaaS uses Polygraphs to detect anomalous activity automatically. When anomalous activity is detected, the user is automatically notified to confirm the activity is valid via MFA. If not confirmed, activity can be blocked.

Total security visibility - Lots of existing security technologies have a view of 1 session or packet at a time. EG creates a global view via information available from multiple regions. Using the global view allows for much better security posture and response.

Data Protection - NSaaS can help administrators keep company confidential information secure. Any HTTP traffic going through a proxy may be checked for DLP policies. Because DLP is quite extensive in its own right (e.g. Symantec scans local disks), it may be preferable to focus on SASE. We may focus on features that are SASE related, and features that deliver real value to the customers who are not buying other DLP products. During all HTTP transactions, content can be checked for policies and action taken.

Content check - All documents served by NSaaS can be checked for various confidential information in the content. We need to specify the types of information we can identify, and allow users to write policies to say which kinds they care about.

- User-defined dictionary to match against document metadata (e.g. Office, pdf metadata) and data.
- Types of the doc, e.g. movies, ppt.
- Financial information like credit cards, ABA routing number, SWIFT code etc.
- PII - SSN, various national IDs, passport number, phone number, email address, name, address etc.
- Sensitive cybersecurity info: e.g. private keys, username/passwords
- Generic types of data: source code, resume, design doc, internal presentation and charts.

Automated DLP - corporate document behavior polygraph

- MD5 based server-side workflow.
 - Used for Logging and creating polygraphs.
- Can be used to find documents used by different groups
 - E.g Sales team accessing financial reports.
 - HR → HR docs
 - Finance → Finance docs
- Classify docs as : private and public.
 - Based on domain names where they are used
 - Can be used to detect mismatch between domain and docs
 - E.g. HR upload employee list to social media sites?
- Which docs are important?
 - Based on (MD5, URL)

SaaS Applications - For SaaS applications such as Salesforce or Workday with confidential information, NSaaS can:

Monitor All Access to the Applications:

- To make sure all traffic goes through EG, a NACL needs to be set for SaaS Apps
 - EG has a set of well known EIPs that can be used to limit access
- All users accessing the application are checked for malicious behavior
 - If malicious activity is detected, a policy can be applied to it
 - Behavior checks are both Rules and Polygraphs.
- Unparalleled visibility and search capability on SaaS usage

Enforce Policies on SaaS Access

- Block certain actions on some sites as needed
 - E.g. restrict download of certain size from salesforce

Private Applications - Zero Trust Access ensures:

- No need to trust the network.
 - No access to anything, just because the device is on the network
- Every user has to authenticate with EG and is allowed access to only apps approved by admins
 - EG MUST not require a change to private apps for broad adaptability.
 - IF EG uses SAML as authentication, then SAML single sign-on story may work
 - EG MUST support http/https/ftp etc.
- IP/port scanning may not be possible for an attacker
 - Usage is allowed directly to app and not to the network
 - Including ping
- No need to sign in to a VPN.
 - Always on or background operation
- Works seamlessly with changes in the network and has single authentication.

Automated polygraph alerts detect anomalous activity automatically:

- Resource Type → TFIDF(Resource Content, URL.)
 - Type → ppt, doc, videos
- Site Type → Algo(FQDN)
- User Type → Dept Tag grouping
- Polygraph → User Type, Site Type, Private public. Action, Resource Type

Devops productivity - Cloud VMs (Virtual Machines) or K8 (Kubernetes) containers may be logged into without having to maintain private keys. Use cases include developers having access to keys that are:

- Temporal in nature
- Can be revoked easily.
- Does not require provisioning for every VM, but uses SAML as a way to provision

Visibility/Reporting/Analytics - Polygraphs can provide visibility into their activity for all employees and devices. To maintain privacy, administrators can see anonymized or aggregated data by have an option to see per-user activity.

Admin visibility - an admin console provides visibility for admins into various aspects of the system.

- Private Apps accessed and their stats
- Users activity summary and risk
 - Kinds of sites visited and how often.

- Device risk
- Time of day usage
- Geo location stats on where in the world service is being more often

Auto-Discovery of Apps – can discover:

- SaaS Apps being used by employees
- Common software being used
- Private apps running in the datacenter via packet inspection
 - All discovered connections → DNS/IP:port.

Polygraphs include user SaaS usage polygraph and user document usage polygraph.

A comprehensive risk score may be calculated for every user, device, and activity. It may do it by looking at all activity, also comparing activity between employees to reduce false positives, and looking at various security functions comprehensively. Risk may be also computed for various Groups and Device groups.

Alerts - All the alerts produced, by policy or polygraphs. Alerts can be analyzed to understand what is causing alerts and which users, and which activity is responsible. Alerts allow for various workflows to allow for changing policies to adjust the alert levels. Alerts can also be sent to various channels like slack, pager duty etc for security teams to easily interact with.

Compliance – includes various compliance policies like SOC and ISO.

Reports – Various reports can be delivered daily or weekly, monthly that report on summary of security posture. Security posture calculates how employees, devices and activity is trending compared to previous periods.

Policies – policies may be restricted based on context such as:

- User name, group, location or other properties
 - Geo location
- Applications and properties
 - Local Application.
 - Headers
- Device or Device properties like OS version, or patch level
- Connection properties
 - Security, protocol, versions
- Document size
- Time control
- Risk Score
 - Dimension
 - Limits

IF Man in the Middle (MITM):

- FQDN, or TLD+1, IPs or IP subnets
 - Or properties like registered name, Risk score, or Site type
- Site actions
 - Like POST, GET, PUT etc.
 - Upload, Download etc
- Media type

Actions – policies may trigger a log/Report and/or one or more of the following actions:

- Allow
- Hard block - the traffic is not allowed through. An error message is shown to the user, and an alert is sent to the user (EG app, SMS or email).
 - Possibly after a limit. E.g. A Number on bandwidth and/or number of sessions.
- Soft block - we send a notification to the user, asking the user to confirm the action. The notification may be SMS or push notification to the EG app.
- Alert - the action is allowed, but we send an alert to the user, letting him know that the action may violate policy and asks him to double check. This can help naive users to self correct. This also gives us an opportunity to point the user to the best practice training material, which is a vital part of DLP practice.
- Deprioritize/Slow down or rate limit
 - E.g. for Docker uploads

Policies – example policy types may include:

- SaaS access/usage policy
Defines how admins want to manage Sites and actions allowed, disallowed on a site
- Private Apps define/access/usage policy
Defines how admins want to manage Sites and actions allowed, disallowed on a site
- Media policies
Defines how admins want to manage medias
- SSL Inspection policy
Which sites to intercept for full SSL inspection
- Data loss protection policy
Dictionary/Expressions to search for in content for DLP
- Application policies
Policies for Applications (on end points). E.g. certain apps can be blocked
- User policy
Defines what is allowed for users
- Threat Intelligence
Customers can define their own blacklist/whitelist for IP/DNS
- IP based device policy
For some devices like printers, it is not possible to authenticate the user. These policies can be written on IP level. All traffic coming from an IP device is screened for the policy
- TOR Policy- ability to connect from known TOR IPs.

Logs – one or more of the following logs may be collected:

- User activity
 - along with user, device, app context
 - DNS, FQDN, Content properties observed
 - Policies applied and its versions
 - Security module input/output where possible

- Anonymized...
- Control plane Admin actions (Audit log)
 - In UI too.
- Endpoint logs
 - There is a mechanism to upload End point logs in case of errors
 - Logs need to be anonymized.
 - Client HTTP logs.
- Logs from services are cross referenced with this log to ensure no out of band access is occurring.
 - Diff (Cloudtrail - our logs)

Threat intelligence – example threat services include:

- DB of DNS site reputation and category of content
 - Commercial corp.
 - E.g. Facebook as social, ebay as auction etc.
 - Alexa Score
 - Cloud WebRisk API : (Safe browsing)
- Known DBs of malicious DNS, IP, MD5, URLs
 - MD5 : Malware Signatures
 - ReversingLab, VirusTotal, Proofpoint, Cisco Talos, SafeBrowsing API
- List of suspicious or insecure applications behaviour
 - Telnet, bittorrent, bitcoin
 - Goes into risk score
- Use of SSL and MFA.
 - Risk score
- Most common password checks
 - Most common passwords : 'Password'
 - Shared passwords
 - Reuse detection
- Vulnerability datasets
 - This can be used in risk score too

User Experience – may include:

Dashboard

EG Regions

- Global map of user count
- EG regions in use and stats
 - Number of users connecting globally to each region
- Private app locations
- Top Apps, Users, Alerts, Policies

Apps

- List of apps (Internal or SaaS)
 - FQDN or IP, ports, users connected, bytes transferred, usage chart
 - Anonymized usage pattern
 - Allow regexp based filtering
 - Risk score of every app
 - Computed based on Total content and blocked content.
 - Category for every app
- Discover
 - List of Apps discovered automatically.
 - Need to do a port scan to discover these
 - Need to also discover devices like printers
- Define new apps
 - User are able to define new applications (IP-port or FQDN-port)
- Top Apps by
 - bytes
 - Users connected
 - Risk
- Inactive Apps
 - By usage
- ZTA Connectors deployed
 - Health Metrics
 - Bandwidth, user metrics

Users

- List of users
 - Last logged in time, location
 - Behavior risk score
 - Polygraph of their usage
- Policies that apply to a user
 - Based on tags, we need to calculate this
 - Least Priv
 - Actions allowed that user is not using
 - Policy recommendations for the group
- Currently logged users
 - Jump to user details
 - Block the user real time

- View sessions
- Top users by
 - Bytes
 - Risk
- Devices (with no users)

Groups

Customers are able to create 'Groups' based on tags. I.e. if tags match, user or app belongs to that group.

- Create Groups
 - There needs to be a '*' group for every tag used. I.e. users have to land in some place
 - We may include with some default groups. E.g. 'All users', 'All apps'
 - Multiple tags can be combined together to create higher level groups
 - E.g. QA, Dev, Ops can be combined in 1 'Eng' group
- List Groups
 - Top Groups by policies defined
 - Top Groups by policies enforced
 - Users with no groups
 - For the same tag, if there are values for which no policy exists.
 - There might be a need for an action here.
- Delete Groups
- Modify
 - The tag query that defines the group
 - Name of the group

Devices

- List of Devices
 - Health metrics
 - Cert metrics
 - User
 - Risk score
 - Location
- Devices with no user
 - Type of device
 - Firmware etc.
 - Risk score

Policies – example policy user actions include:

- Create different kind of policies
 - Any kind of policy supported by EdgeGuard
 - All policies are essentially a set of (TAG-VALUE) regexp match
 - Name the policy
- View them
 - And all old versions
 - Version of policy in effect
 - Last update time, last update user
 - Diff any 2 versions
 - Policy version currently getting enforced in regions.
 - Top policies
 - By enforced block.
 - For every block, detailed log message on why it was blocked
 - By eval
 - By global distro
- Edit them
 - Edit may create a new version not yet enforced.
 - Multiple edits can be done till customer is ready to enforce
- Delete a version
 - Only if policy is not enforced
- Enforce the policy
 - 1 button click to enforce worldwide
 - Feedback loop till policy is enforced in secs
 - Pick which regions to enforce first.
 - If policy enforcement has any error, it shows.

Example alerts include:

Blocked Content

- All content blocked in past time period
 - Intel source
 - User involved
 - App involved
 - Malware information - if we have it
 - Blocked because of policy
 - Top Detailed Log on every block
 - Block Action (2FA or something else)
- Block stats
 - Number of users
 - Reason

- Session Risk
- Detailed Log of blocks

Example health data includes real time and historic data, such as:

- RTT for every user
 - Calculated from Proxy transactions
- Response time averages for Apps
 - Both Private and SaaS
- Users connecting in last 24 hours
- Remote/Mobile users

User Application Polygraph:

User-App graph

- For every user
 - which apps they use,
 - What are the tags on these apps as defined in groups
 - When do they use it
 - From which location
 - Data transferred to the apps to/fro
 - URL params used
 - User-Agent used
 - Browser-fingerprinting hash
- Can be aggregated for per group

Example datasets to ingest may include, for example:

From Client Device (via an Agent that may be running on the client device in some embodiments):

- Device Posture Details
 - OS (os, os_version, is_patched)
 - Applications (apps installed, version)
 - Device Posture details from customer vendor
 - CrowdStrike/CarbonBlack/other MDM (Mobile Device Management) integration:
 - Indications of Compromise ('IOC') on device
 - Browser History
 - Files under special directories (eg: /tmp)
 - Mutexes

Access Logs (collected at the edge sites):

- Timestamp, Client_ip, session (user id, device id), src_application/src_file, dst_host, dst_port, bytesIn/out, duration

Threat Intel (from integrations with 3rd party tools and other tools):

- Webroot:
 - Ip/Dns Reputation
 - Web Categories
 - File Reputation
- WhoIs:
 - Domain Age
 - Registration Details
- GeoLocation
- CrowdStrike/Carbon Black IOCs

Customer's SaaS Application Logs (collected from the customer's applications)

- Office 365, Okta, Google Access logs

Email Security Alerts from customer used vendors

User behavior may be modeled in two primary ways: without man in the middle (MITM) and with MITM. Without MITM, a user location may be tracked. For example, a user (belonging to an Org) can be working from different locations (home, office, cafe, travelling etc). The locations can be a well-known Org location (eg: corporate location, headquarters, branch office). If the org is using VPN/proxy, a user's traffic may originate from the IP address of the VPN. Location detection may be performed by learning from users' traffic and detecting whether the traffic is coming from a known location: 1) New/Unknown/Rare/Suspicious Location: Based on user behavior, it can be detected when traffic is originating from an unknown location. 2) Impossible travel/impossible concurrent access: Based on the user's location changes, it can be detected if the user is travelling. Based on learned corporate Vpn's locations and users past locations, it can be detected whether the travelling behavior shown by the user is Suspicious. An example algorithm includes:

```

Algo:
For (user+device):
  Say_Current_location = L
  If current_location NOT in most frequent locations (note: consider ip address for vpns) by tenant (this eliminates VPN traffic) :
    Last_location = L1
  Last_time_seen for L1 = t1
  Current_time = t2
  Distance = compute_distance(L1, L)
  Time taken to travel = t2 - t1
  Speed = Distance/time

```

If Speed > Speed of airplane:

//Detect Impossible travel

Polygraph based detections: User Group App Access; User Location App Access; User Time App Access. Ingested data may include: user-groups, application-groups, user-application assignment (directly or via group) from an Org's tools. Traffic originating from an Org's users attempting to access an Org's application is monitored. Polygraph use cases include:

Least Privilege Access - The usage pattern of all users may be learned to identify over privileged users. A user may be identified as over-privileged if:

- The user is assigned to groups that have access to apps which are not used by this user.

- b. The user is provided access to an app that they normally do not use. The user may be require the access rarely and once we detect that, we can provide the access on-demand and restrict this on-demand access for some time-interval

Least privilege access policy, given group membership as (Integer LP) optimization problem: Assume that there are M groups and N applications. We can represent the access policy to be a $M \times N$, $[0, 1]$ policy matrix P . where the columns represent the application and rows represent the groups. P_{ij} is the i - j th entry of P , if $P_{ij} = 1$, then group i has access to app j . For each user k , we can have a user vector u_k representing the membership of user k , this is a M dimensional vector, where $u_{k,m}$ is 1 if only if user k belongs to group m . So the apps that user k has access to are given by $u_k^T * P$ which is a N dimensional vector. Given a history (7 days) of the user, we can optimize the matrix P to obtain some P^* , which is the least privilege policy set. Note that the apps a user accessed over the last 7 days, can be represented by v_k , just like the case of u_k . This becomes an integer optimization problem, i.e.:

$$\text{Min } \|P\| + \text{Sum} (\|u_k^T * P\|)$$

$$\text{St. } u_k^T * P > v_k$$

Least privilege access policy, given group membership and static group policy as LP program - Following from above, If we have the matrix P' , representing the existing access policy matrix, we can instead optimize the matrix difference between P' and target policy matrix P^* . Let Q be a matrix where $Q_{ij} = 0$ for all i, j s.t. $P_{ij} = 0$. We only need to consider the part of Q_{ij} where $P_{ij} \neq 0$. More specifically, the target policy $P = P' - Q$, for each $Q_{ij} = 1$, indicates that the access grant from group i to application j is unnecessary for the current access pattern. The objective function is then to maximize the matrix norm of Q , i.e. the sum of all Q_{ij} , i.e:

$$\text{Max } \|Q\|$$

$$\text{St. } u_k^T * (P' - Q) > v_k$$

If P' is sparse, then the number of variables of the integer LP reduces significantly.

Elastic least privilege access policy given access pattern: This does not assume any user group or existing static policy. But assuming we have a history of users' application access patterns (this can also be established as enriched PolyGraphs). The general idea is to perform user clustering, and if necessary, iterate between user clustering and application clustering. There are two stages:

- Establish features for users. This can be done through:
 - User node embedding from a graph representing users' application access pattern, using any graph/node embedding techniques, such as deepWalk, Node2Vec, etc
 - Tf-idf on user's application access pattern, constructed into documents.
 - Note that appropriate dimension reduction techniques are used to denoise the user node embeddings.
- Clustering of the users based on their feature embeddings, using techniques such as kmeans, DBSCAN on similarity or distance measures established in the users' embedding space.
- Labels User as groups, reverse the procedure for applications

If other information, such as user tags, application tags available, such information could potentially enter either the feature set, or be used to explain the user/application grouping for better generalization. For example, if a set of tags serve well in distinguishing a cluster of users from others, we can define a user group using the set of tags and subsequently a new user with similar tagging can automatically enter the group and enjoy the same access policy without observing his usage pattern. Similarly, a new deployed app with proper tagging could automatically get accessed from the intended set of users.

Unexplained/Undefined group accessing an App: Based on the behavior exhibited by all users, each access from a user to an application is grouped into these buckets:

- Known group accessing a known app:
 - If more than 80% of group's users accessing App A
OR
 - 90% of the access for A1 is from the users of group G AND more than 25% (or 25 users) of org's users accessing the app
OR
 - Static policy defined by customer
- Unknown Group accessing a Popular Org app
 - If more than 25% (or 25 users) are using the app, then the app can be labelled as a popular app
- Unknown Group accessing unpopular app
 - If access does not belong to (a) and does not belong to (b)

Application accessed from an unusual location: The location from which an application is accessed by a user is monitored. For every application access, the originating location is classified as:

- Known user location:
 - If the location was seen in the past and has > 1% of traffic
 - If the location seen in the last 30 days
- Known Tenant Location:
 - If more than 10% users seen from a location in the last 30 days
 - If more than 10% of traffic seen from this location
- Unknown location:
 - If not (a) and not (b)

When modeling user behaviors and looking for anomalies or unusual user behaviors that could indicate possible security threats, we use various different approaches and techniques to examine different categories of features or signals.

- Numeric features
 - Time dependent features (e.g. timestamps of first login of the day)
 - Frequency-dependent features (e.g. daily activity counts, connections, bytes downloads)
 - Statistical features (e.g means, media, variance of other numeric features, etc)
- Categorical features
 - Entities
 - Actions

Application accessed by a user at an unusual time: These models aim to establish daily (or weekly) user networking activity profiles (per user). This model primarily examine statistical anomalies on numeric features. The main features may be:

- Timestamps (ideally local to user's physical location) of user triggered traffic activities, in particular we pay attention to the following derived features
 - First access of the day
 - Last access of the day
 - Duration between first and last network access
 - Gaps between connections, especially http, browser activities
- Traffic based features:

- a. Total download bytes
- b. Total upload bytes

More granularized categories may also be used (e.g., per application, per device). The feature set of each user may be profiled according to one or more of: 1) Gaussian mixtures and 2) PCA components.

Cohort analysis: User's profile vector => Autoencoder / PCA or other dimension reduction => Clustering => measure of cluster separation.

Unusual user behavior from multiple factors event history: Through ingested datasets and the construction of a polygraph, we are able to recover the 4W (who, where, when, what) event sequences of each user. The users' behaviors are studied by an Analytic Engine. As most of the events consist of entities and actions, the model mainly examines categorical features. However, as the model explores the correlations between the events and entities, it implicitly considers certain frequency based metrics. Furthermore, certain numeric features could be bucketed and enter the model.

Learning User Events:

1. Construct corpus of user events for all the users of a tenant
2. The event corpus is then used to train document-specific event language models using deep learning techniques such as Word2Vec for FastText models.

Note:

- We can also derive a knowledge graph for the events collected and construct contextual language corpus from the graph
- Word2Vec is just an example to embed events
- Embedding can happen at different levels too for user, user group, app, app groups etc.

3. Every user event can be embedded with the tenant specific event language model. The relationship among entities (4W and beyond) can be represented by the similarity among the embedding vectors of the entities and events.

Detecting anomalous/unusual events:

1. Embedded user events are used to train appropriate sized AutoEncoders.
2. Autoencoders perform dimension reduction on the event matrices. The reconstruction errors from autoencoders are recorded.
3. Due to the dimension reduction, only prevailing characteristics of each user's historical events are preserved in autoencoders. Therefore, for common events (thus behaviors) of a user, the reconstruction errors of autoencoders may be small.
4. Separate validation tests of data may be used to tune the threshold of the event autoencoder reconstruction errors.
5. If any event results in reconstruction errors (e.g measured in MSE) greater than certain threshold, the event is labeled unusual and further examination is suggested to the admins.

Note:

- Essentially, autoencoder works as a simple dimension reduction algorithm to capture the main/common features of events in a high dimensional space. Despite different implementations, it is similar to clustering / PCA as dimension reduction algorithms and finding anomalies based on distance from clusters.
- Alternatively, with embedded events, users, etc, we can do clustering, or rely on algorithms like isolation forest to identify anomalies.
- Another option is to look at the user behavior of the day, collecting per user features/embeddings, and train multi-class classifiers to identify the user. For testing data, if the classifier output is different from the actual user, we identify an anomaly. This approach may not be as robust, as new users may force the model to be retrained. However, the problem can be mitigated by training only the last classification layer. So the model may still be manageable. However, we need to put more thought into the representations of user behavior in high dimensional space (e.g. some existing literature convert user's daily activity into time-domain heatmaps and train CNN classifiers using the heatmap image).

User Cohort/Level Detection

1. For the same user, if multiple events are detected as unusual in the same period (hourly, daily, weekly), the user higher risk score may be used and may alert the admin.
2. Similar analysis can be performed to cohorts of any of the 4W entities (or more). Events/User may be unusual compared to the user's past behavior but common or less unusual compared to cohort of the users or entities. Such user events may be identified to reduce false positives.

In some embodiments, Events marked as unusual could be illustrated on PolyGraph for much easier examinations, especially when there are commonalities among certain sub-groups of events. E.g., a new application is seen in the system. The admin / security team could easily green light and mark things benign in batches from PolyGraph.

Botnet/C&C (Command and Control) Detection:

DGA Detection - General availability of DGA data sets allow us to train relatively good performing (accuracy > 98%) models (based on CNN and RNN). We can monitor the availability of new DGA dataset to retrain models if necessary. As an example, a model may include: 1) 1D Convolutional layer -> MaxPooling, and 2) 64 Bidirectional LSTM.

In order to keep up with the new types of DGA algorithms, we obtain new training datasets and update the model periodically while cross checking against threat intel providers such as webroot. Common domain names, such top 1 millions web domains from alexa or other benign domain dataset may also be used to improve the false positive rate of the algorithms.

Known C& Server Traffic: Using threat intel, it can be determined if a user device is communicating to a malicious C&C server.

Beacon detection: while monitoring the traffic and providing access to the user, the following features may be generated:

- a. For each src, dst (Device -> dst_domain)
 - i. Time series of all the sessions ordered by time
 - ii. Find time-delta between two successive sessions
 - iii. Find bytes transferred in each session
- b. For each Domain: WHOIS (Age, country, registrar)
- c. Popularity score of domain for this tenant
 - i. How popular is the domain. A Domain is popular if it is visited by many users of this tenant.

Port Scanning features: number of domains visited, number of distinct ports, number of newly observed destinations, number of resets/failed connections.

Multiple suspicious failed login attempts – features – number of resets or failed connections.

Malicious employee downloads/uploads – numbers of the following: applications/domains visited, categories visited, bytes in, bytes out, connections, distinct destination ports, newly observed destinations, risky connections based on threat intelligence.

With MITM:

Polygraph based detections: one or more of the following polygraphs may be generated: Document access polygraph; User content type polygraph.

Malware Detection/Protection –

Third party detection - the hostnames and URLs that a user is visiting are checked against 3rd party threat systems to determine if the user is visiting sites that have malicious content. These could be integrations that are managed by the NSaaS system itself, or they could be integrations with 3rd party systems that the customer subscribes to (e.g. a "bring your own" model).

Static detection - Content inspection rules that look for malicious behaviors may be used. This is beyond the usual 3rd party detection systems which are usually just matching specific URLs or hashes. Instead the static detection may match on patterns in the content itself, for example an exploit kit that uses a particular pattern for fetching a dropper. The static detection would have access to augmented metadata, for example information from whois about the domain registrant, or information from the certificates used. The static detection could be executed either on the content as seen by the user, or on a separately fetched copy. By doing the scans directly against the content as seen by the user, it makes it more difficult to evade detection using geo-fencing, source-ip filtering, or time- and usage-based mechanisms to selectively deliver clean or malicious content.

Dynamic detection - In some embodiments, content inspection that loads content into simulated browsing environments (e.g. sandbox) to observe the behaviors is used. Static detection can also be applied against the results of the dynamic execution of the content, for example viewing the DOM of the rendered page to look for exploit kits or phishing content.

Ensemble detection - In some embodiments, multiple detection mechanisms are used, each specializing in certain types of detections. There may be an orchestration component that looks at the results from different detection engines and applies machine learning techniques to decide the probability of the different mechanisms being correct. Human analysts may examine the decisions that different systems make to help train the system. When there is a decision whether a particular resource (e.g. URL, file) may be submitted to the various detection systems, accuracy and cost of the various mechanisms is considered. Mechanisms that are more likely to give an accurate classification are trusted to send suspicious results to further mechanisms for confirmation that there is malware.

Detection on client (examining history or system) - For users that are running the installed client, the client may be able to scan the local system for evidence of compromise. This could be examining the browser history on the computer to look for URLs that are known to be malicious (e.g. phishing) even though there is no otherwise visible indicator of compromise (e.g. file hash on a dropper). In some embodiments, traditional indicators of compromise are identified by scanning temporary and recent download folders for files that have hashes recognized to be malicious, as well as mutexes that are used by malicious code to avoid repeatedly infecting the same machine.

When content is possibly or definitely malicious, in some embodiments, the content is blocked. However to reduce the cost of a false positive, in some embodiments, the content is wrapped to make it safe for the user. That could involve things like removing all scripts from a page, or wrapping those scripts with code that intercepts and blocks what the malware is trying to do. The wrapping could also be at a horizontal level where the entire page is returned to the user as a server-rendered page, with no local execution at all (sometimes called "remote browsing").

Response/Remediation

Additional user authentication: When a suspicious activity is detected, in some embodiments, an alert is generated. Better is to involve the user directly in the decision making process since in many cases the user is not the attacker, and is in the best position to determine what activities are anomalous. By interacting with the user, through either the client application or through a second factor, it may be confirmed that the observed behavior is legitimate and desired.

Workflows (user, manager, admin) - When there is a pattern of suspicious behavior, using a workflow to confirm and grant access ensures that there is an auditable trail of decisions. Those decisions might need to involve others besides the user, to deal with the case that the user is behaving maliciously (or not responsibly). The workflows could have time-based permissions tied to them, to grant access for a limited amount of time based on approval. The approval could come from the user, a manager, or a security admin. Routing and escalating the approval requests is part of setting up and managing the workflow. Access policies would be tied to the successful execution of these workflows.

Polygraphs:

User Group App Access Polygraph

Tracks and baselines the group associated for a user accessing a pvt.app

[user] —> [group/undefined group] —> [pvt app]

Events:

Node Events:

- New Pvt App
- New User Group

Edge Events:

- Edge: undefined group -> app: User Accessing App through an Undefined group association
- Edge: Group -> app: User Accessing App through a new Group association

User Location App Access Polygraph

Tracks and baselines the Location from which a user accesses a pvt.app

[user] —> [Ip/location] --[known/unknown/new] —> [pvt app]

Events:

Node Events:

- New Location: Traffic from a New Geo accessing a pvt app
- New Bad Ip/Location: A known Bad Ip address in Geo accessing an app

Edge Events:

- Edge: unknown location -> app: User Accessing App through an unknown location
- Edge: Location -> app: User Accessing App through a new Geo

User Location Time Access Polygraph

Tracks and baselines the Location from which a user accesses a pvt.app

[user] —> [Ip/location] —> [known/unknown/new] -> [time] -> [normal/unusual time]-> [pvt app]

Events:

Edge Events:

- Edge: unusual time -> app: User Accessing App through a location at an unusual time

App Content View Polygraph

It monitors what kind of content is consumed by who and from where?

[Location] —> [User Group] —> [client app/exe_path/file-hash] + [Content Action + content-type]
 —> [Pvt App] —> Error Code(Allowed | Disallowed)

For an local exe_path accessing an App for a user, we can compare the access with all the accesses to that app across the Org and label the access as well-known/new/rare access.

LACEP040

Events:

Node Events:

- Bad file-hash: App accessed using a known Bad file

Edge Events:

- Edge: exe_path+content_type -> app: User Accessing App through using a new file

Document Access Polygraph

Location (logical location) → User Group

→ [Doc Action + Doc Type] → DestinationDocument → Allowed | Disallowed

Location → *logical Location (is most frequent)*

User Group → *user groups obtained from okta*

Doc Action → *Read/Write (default: Access)*

Doc Type → *pdf/ppt/doc*

DestinationDocument → *Document Id/Folder/Shared Drive*

Events:

Edge Events:

- UserGroup accessed a document for the first time
 - Condition: If the document is not New.

Risk Profiling – score users/devices based on their behavior and find high-risk users. In some embodiments, administrators may create policies to control a risky user's access (e.g., limit applications that the user may access, block access to all apps, etc.)

Detect unmanaged popular applications (e.g., applications not managed by IT or other internal entities) – From the 'user group app access polygraph' it may be detected that the applications that are accessed by users and are strongly correlated to users of at least one user group. These applications may not be managed by IT. Looking at the output of the polygraph, administrators are supported with a workflow to detect new applications and manage them.

Content-inspection policies without MITM – browser (e.g., Chrome) extensions apply policies and model user behavior without breaking open an SSL or other encrypted connection.

Privacy-conscious policy - With a browser extension, we can inspect content but export telemetry that preserves user-privacy, but applies policy that the admin cares about.

Noise reduction in user behavior anomaly detection - User behavior has randomness to it, which makes noise reduction important in any user behavior based anomaly detection system.

Using behavior based anomaly detection to discover zero-day exploits - Zero-day exploits by definition can't be discovered through any signature based techniques.

Existing industry practices have low efficacy in discovering zero-days. Zero-day exploits are discovered through behavior based anomaly detection. Behavior based anomaly detection may also be used to discover malicious insiders.

High precision and high recall security system workflow - It's hard to achieve both high precision and high recall in security systems where user behavior noise is high.

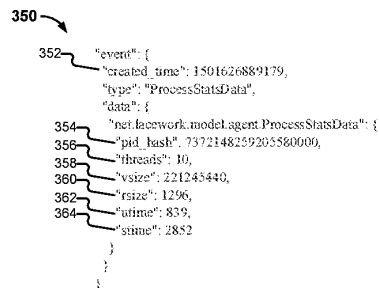
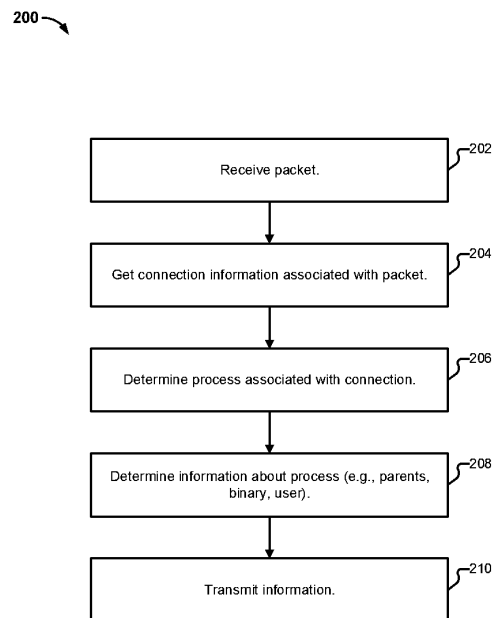
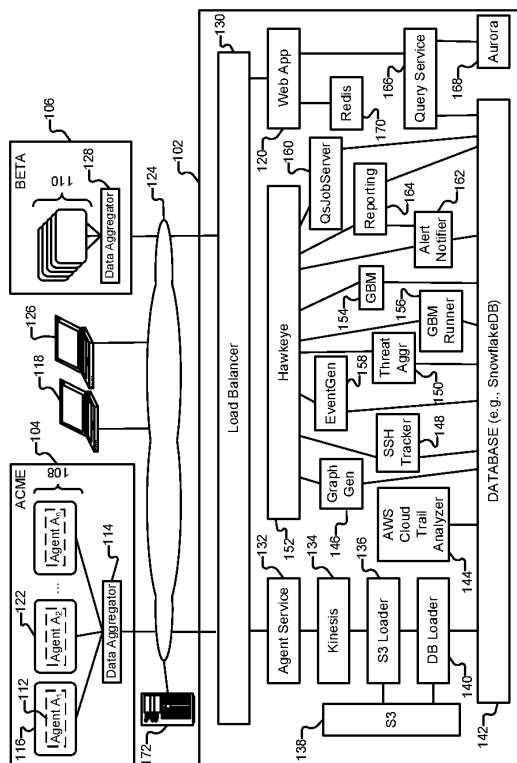
Usually a system can have only one of the two. We achieve both by using algorithms that are high precision but low recall, or high recall but low precision. We design workflows behind them so that high precision signals are acted upon automatically, and high recall low precision signals are filtered by users first.

Bootstrapping – in some embodiments, past behaviors of users belonging to a particular group may be used to bootstrap new users of that group. As an example, particular applications detected as being used by users of a particular group may be automatically installed for a user belonging to that group.

CLAIMS

What is claimed is:

1. A method comprising any of the features described herein.
2. A system comprising any of the features described herein.
3. A non-transitory computer-readable medium storing instructions that, when executed, direct a processor of a computing device to perform any of the features described herein.



```

"outgoing": {
  "net.lacework.model.agent.UniDirectionData": {
    "bytes": 7524648,
    "compress_ratio": 0,
    "compress_samples": 0,
    "packet_len_hist": {
      "net.lacework.model.agent.Histogram": {
        "count": 471,
        "sum": 7524648,
        "max": 31856,
        "std_dev": 0,
        "buckets": null
      }
    },
    "session_len_hist": {
      "net.lacework.model.agent.Histogram": {
        "count": 4,
        "sum": 7524648,
        "max": 2532151,
        "std_dev": 0,
        "buckets": null
      }
    },
    "session_time_hist": {
      "net.lacework.model.agent.Histogram": {
        "count": 4,
        "sum": 152452,
        "max": 52039,
        "std_dev": 0,
        "buckets": null
      }
    },
    "session_switch_time_hist": {
      "net.lacework.model.agent.Histogram": {
        "count": 3,
        "sum": 544171,
        "max": 223693,
        "std_dev": 0,
        "buckets": null
      }
    }
  },
  "incoming": {
    "net.lacework.model.agent.UniDirectionData": {
      "bytes": 1041,
      "compress_ratio": 0,
      "compress_samples": 0,
      "packet_len_hist": {
        "net.lacework.model.agent.Histogram": {
          "count": 6,
          "sum": 1041,
          "max": 298,
          "std_dev": 0,
          "buckets": null
        }
      },
      "session_len_hist": {
        "net.lacework.model.agent.Histogram": {
          "count": 3,
          "sum": 1041,
          "max": 347,
          "std_dev": 0,
          "buckets": null
        }
      },
      "session_time_hist": {
        "net.lacework.model.agent.Histogram": {
          "count": 3,
          "sum": 87,
          "max": 43,
          "std_dev": 0,
          "buckets": null
        }
      },
      "session_switch_time_hist": {
        "net.lacework.model.agent.Histogram": {
          "count": 3,
          "sum": 1750190,
          "max": 1180754,
          "std_dev": 0,
          "buckets": null
        }
      }
    }
  }
}

```

FIG. 3C

FIG. 3D

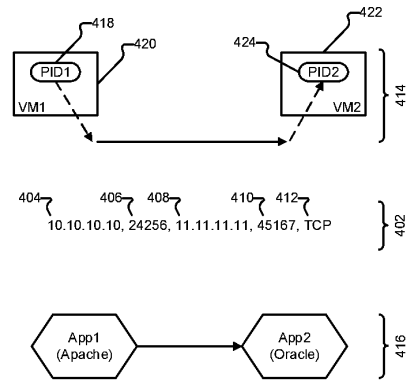


FIG. 4

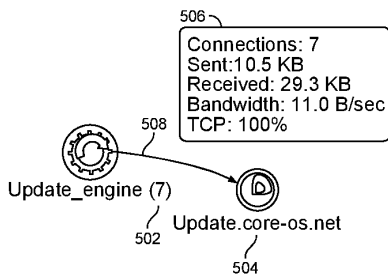


FIG. 5

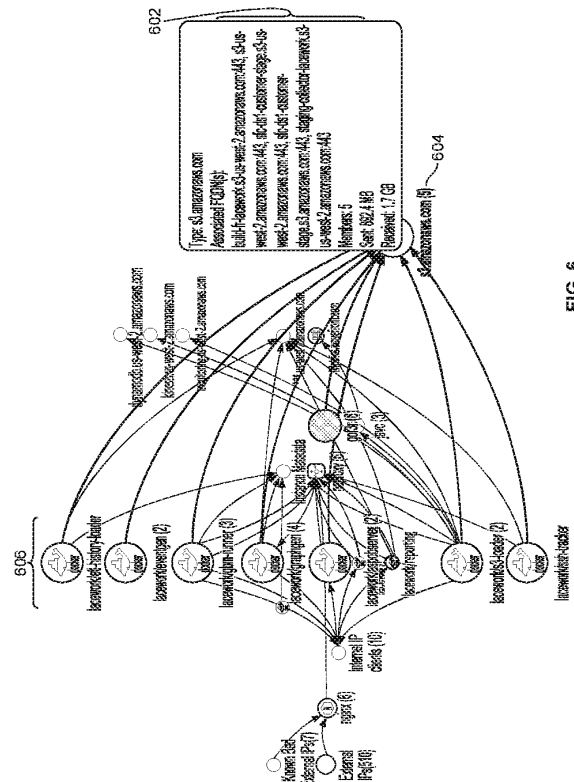


FIG. 6

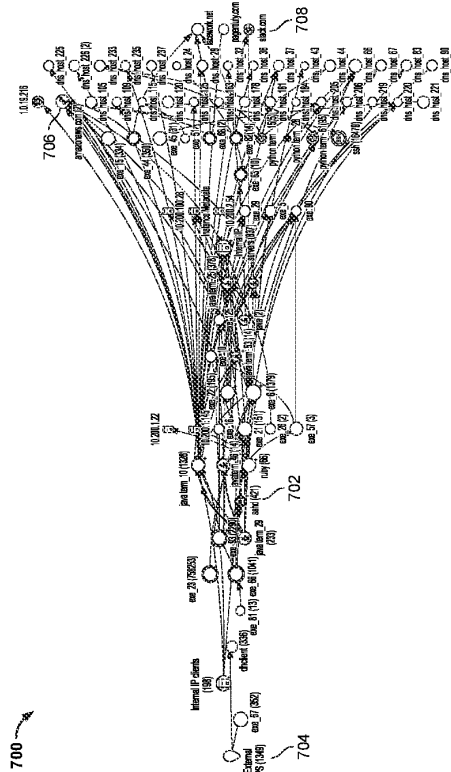


FIG. 7

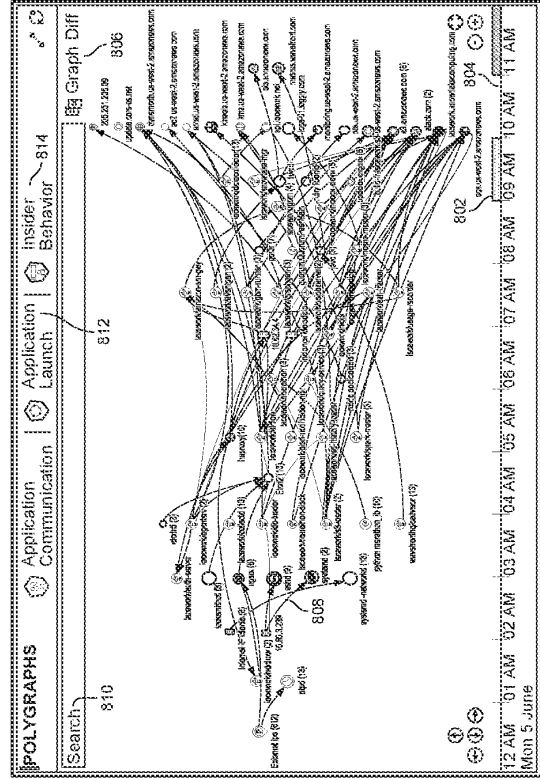


FIG. 8

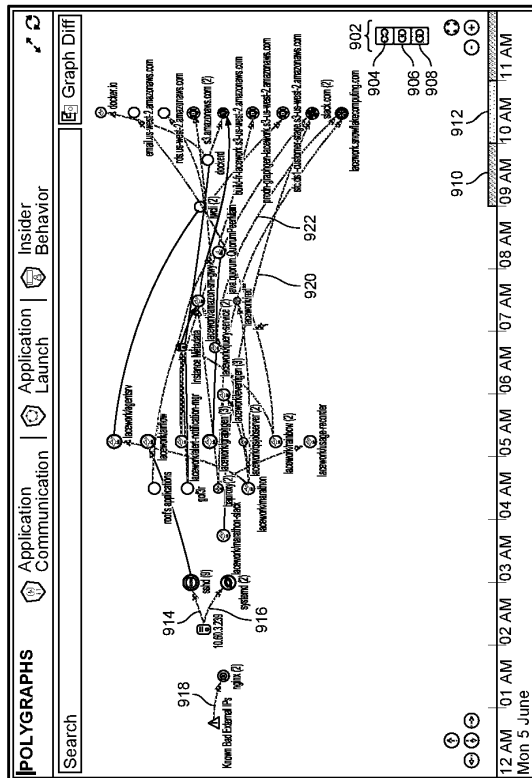


FIG. 9

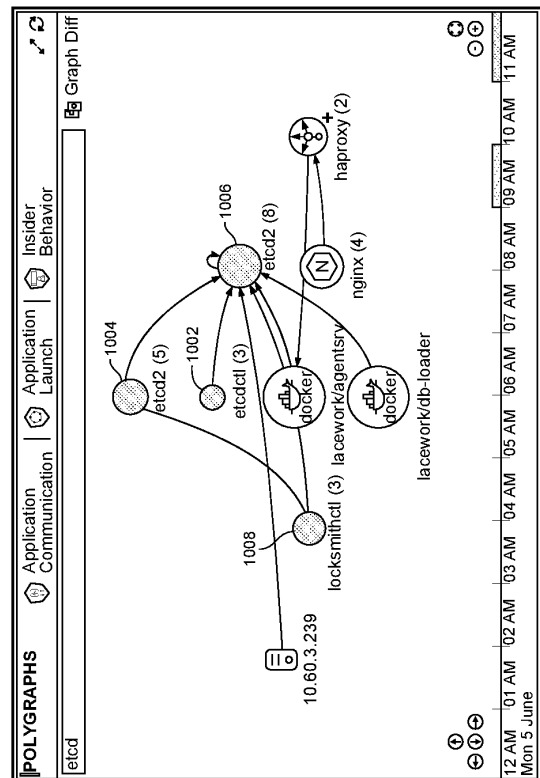


FIG. 10

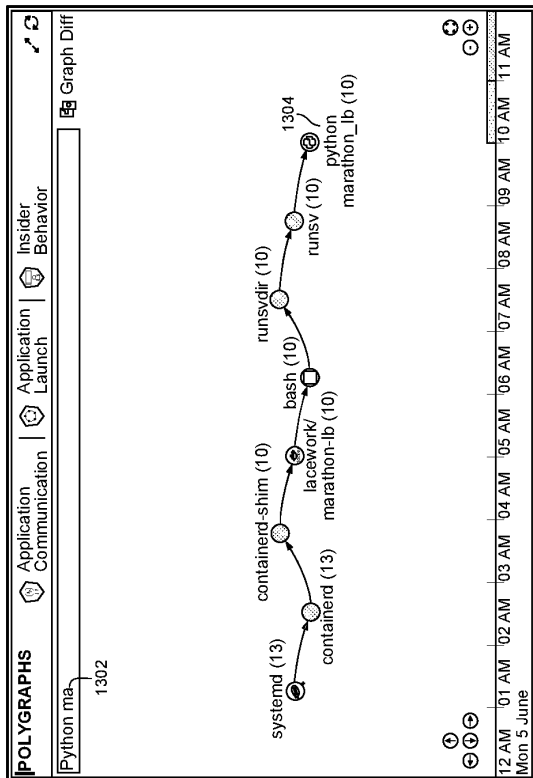


FIG. 13

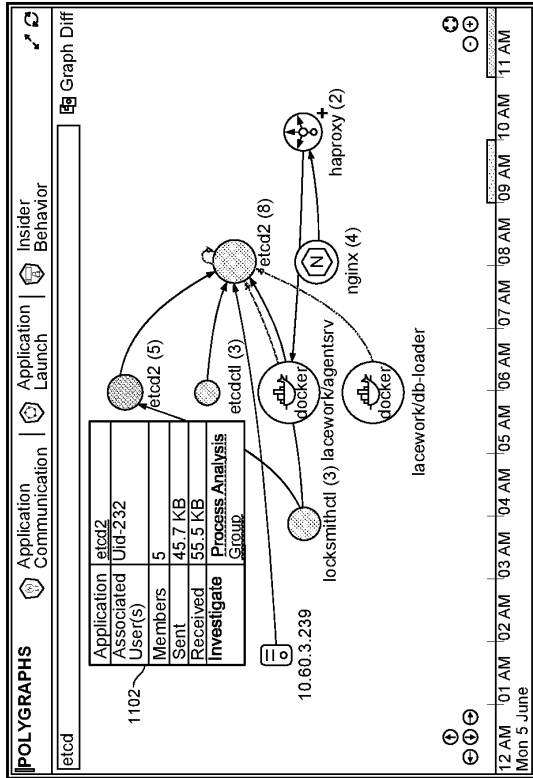


FIG. 11

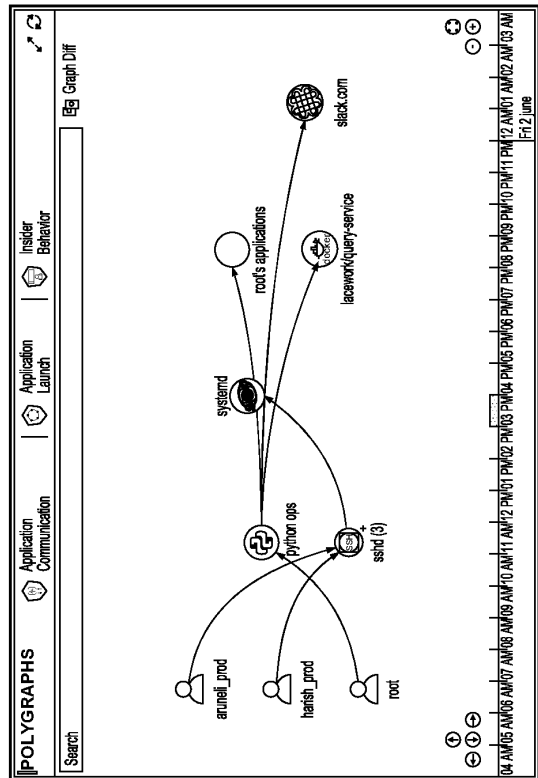


FIG. 14

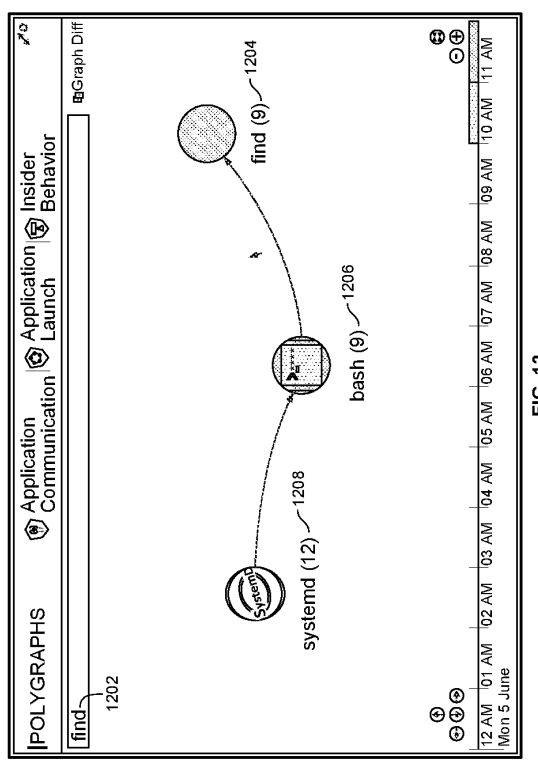
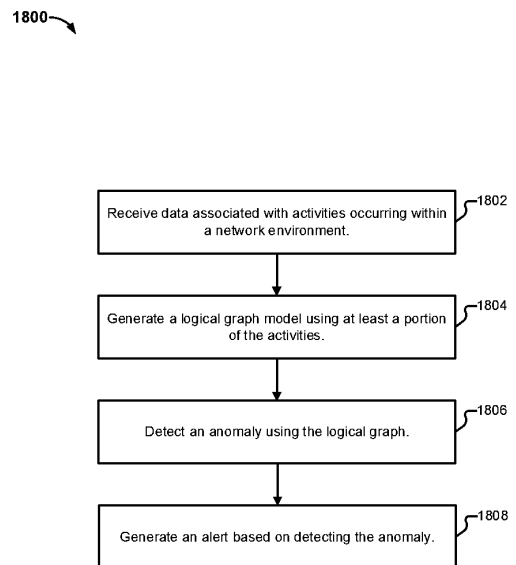
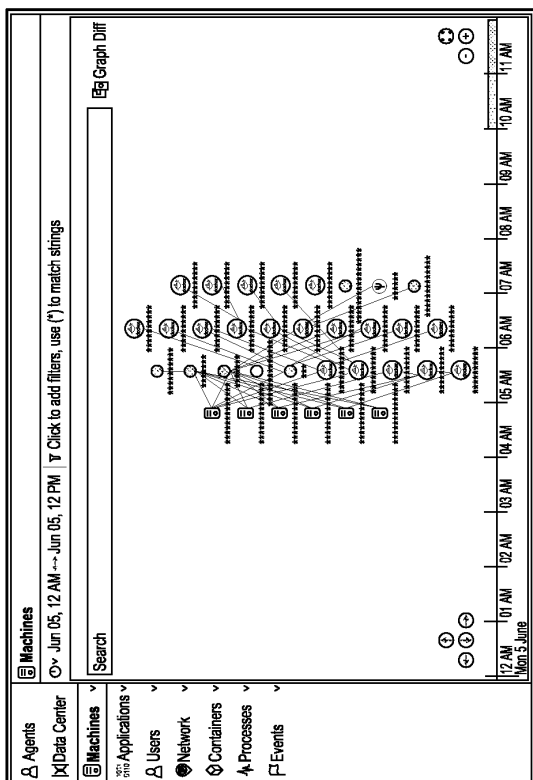
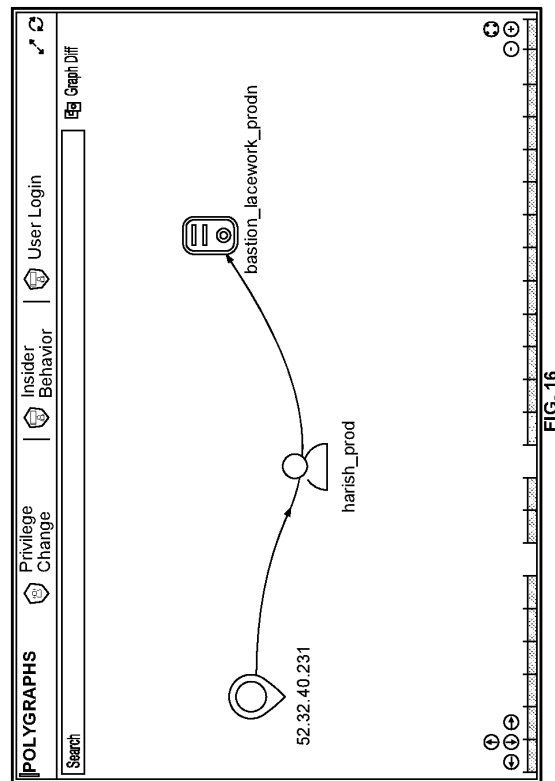
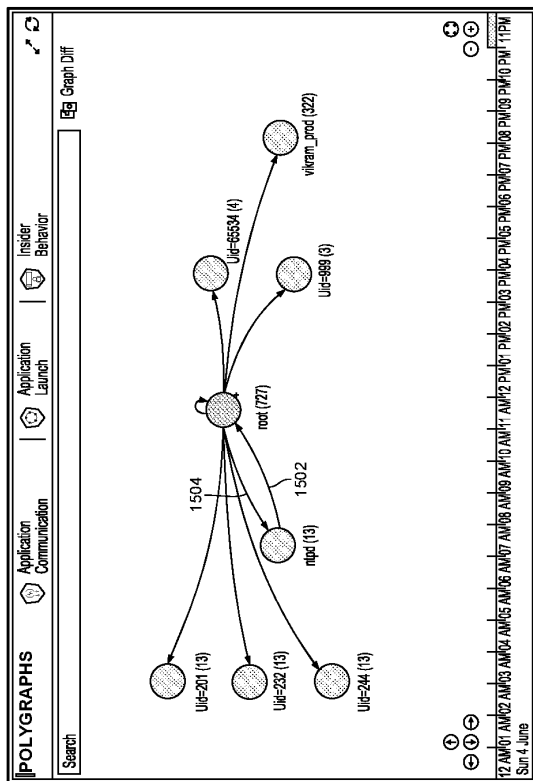


FIG. 12



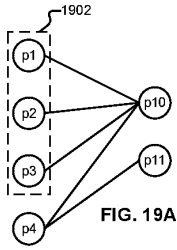


FIG. 19A

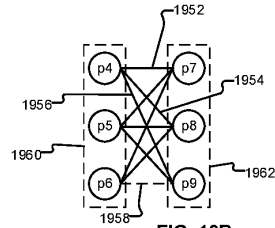


FIG. 19B

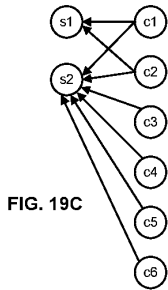


FIG. 19C

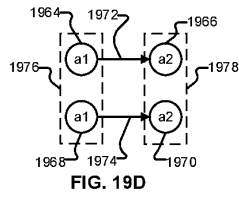


FIG. 19D

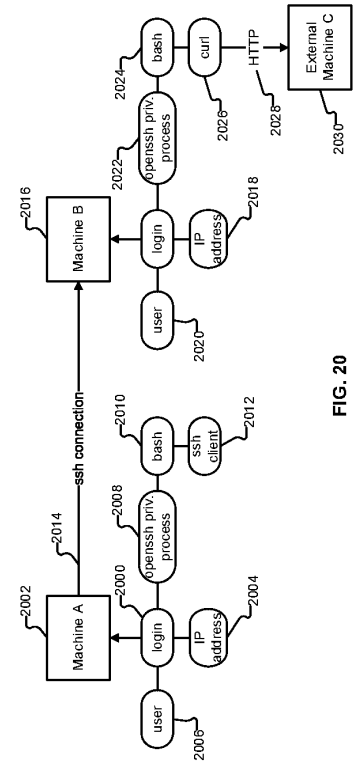


FIG. 20

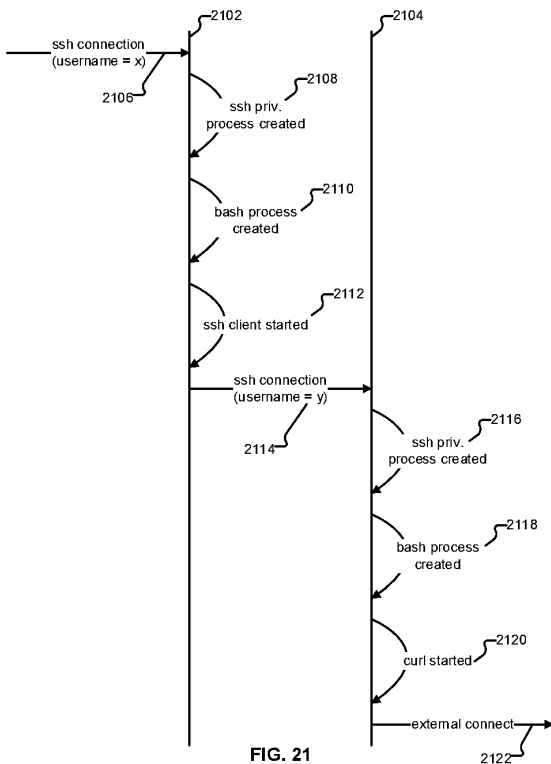


FIG. 21

2200

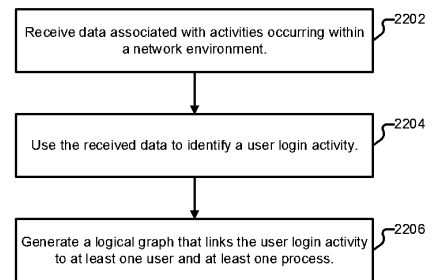


FIG. 22

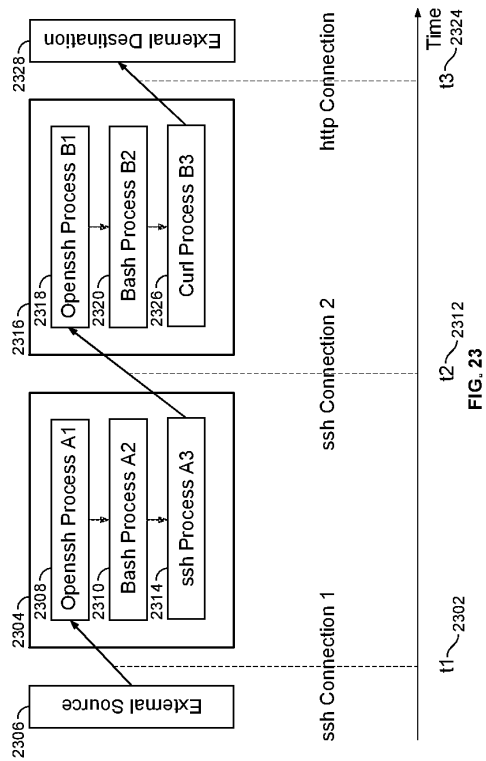


FIG. 23

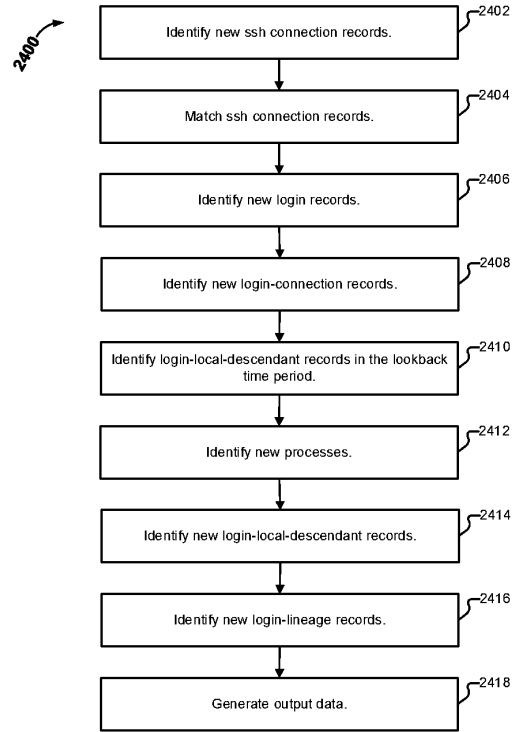


FIG. 24

MID	start_time	FID_hash	src_ip_addr	src_port	dst_ip_addr	dst_port	prot	dir
A	t1	A1	1.1.1.10	10000	2.2.2.20	22	TCP	Incoming
A	t2	A3	2.2.2.20	10001	2.2.2.21	22	TCP	Outgoing
B	t2	B1	2.2.2.20	10001	2.2.2.21	22	TCP	Incoming

FIG. 25A

MID	start_time	FID_hash	exe_path	parent_FID_hash
A	t1	A1	/usr/sbin/sshd	A0
A	t1	A2	/bin/bash	A1
A	t2	A3	/usr/bin/sash	A2
B	t2	B1	/usr/sbin/sshd	B0
B	t2	B2	/bin/bash	B1
B	t3	B3	/usr/bin/curl	B2

FIG. 25E

src_MID	src_FID_hash	dst_MID	dst_FID_hash	dst_start_time	src_ip_addr	src_port	dst_ip_addr	dst_port
null	null	A	A1	t1	1.1.1.10	10000	2.2.2.20	22
A	A3	B	B1	t2	2.2.2.20	10001	2.2.2.21	22

FIG. 25B

MID	sshd_FID_hash	FID_hash
A	A1	A1
A	A1	A2
A	A1	A3
B	B1	B1
B	B1	B2
B	B1	B3

FIG. 25F

MID	login_time	sshd_FID_hash
A	t1	A1
B	t2	B1

FIG. 25C

parent_MID	parent_sshd_FID_hash	child_MID	origin_sshd_FID_hash
A	A1	B	B1

FIG. 25G

MID	sshd_FID_hash	login_time	login_username	src_ip_addr	src_port	dst_ip_addr	dst_port
A	A1	t1	X	1.1.1.10	10000	2.2.2.20	22
B	B1	t2	Y	2.2.2.20	10001	2.2.2.21	22

FIG. 25D

MID	sshd_FID_hash	parent_MID	parent_sshd_FID_hash	origin_MID	origin_sshd_FID_hash
A	A1	null	null	A	A1
B	B1	A	A1	A	A1

FIG. 25H

2600 →

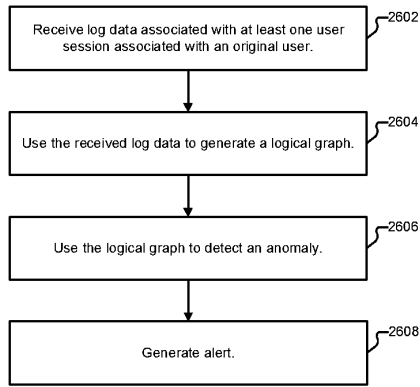


FIG. 26

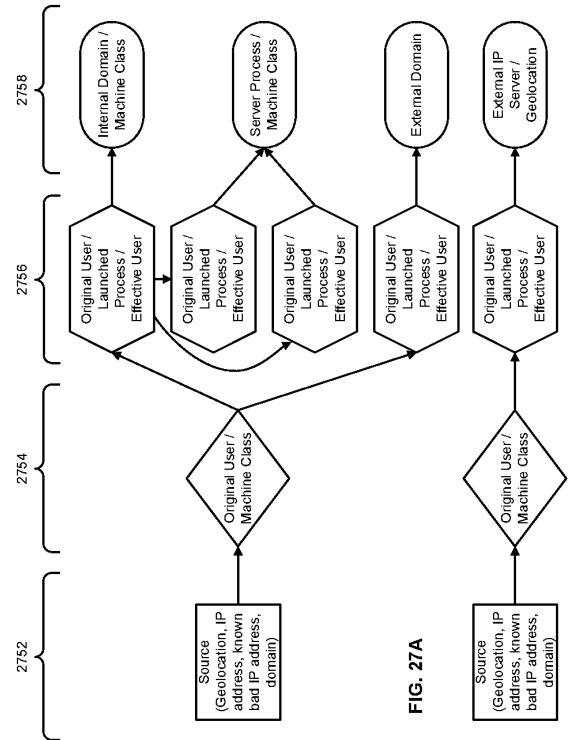


FIG. 27A

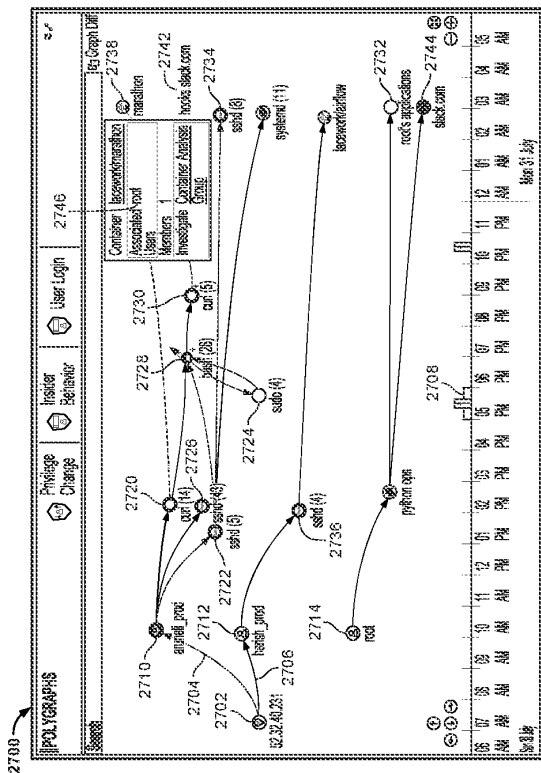


FIG. 27B

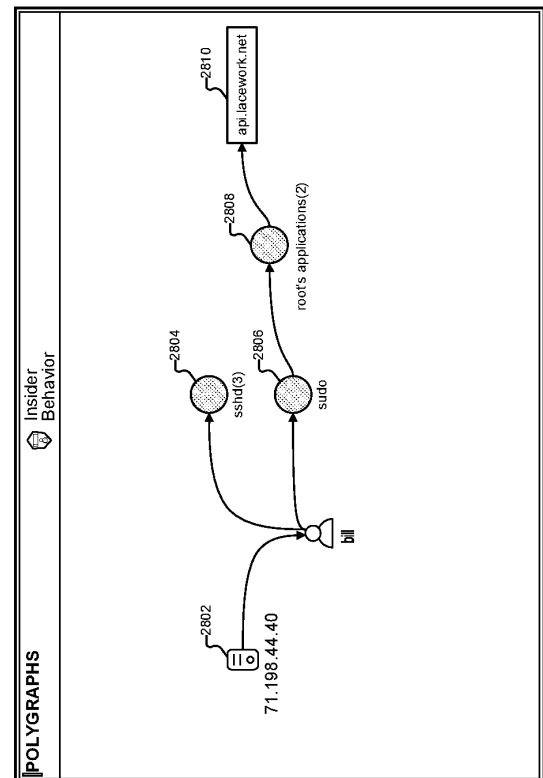
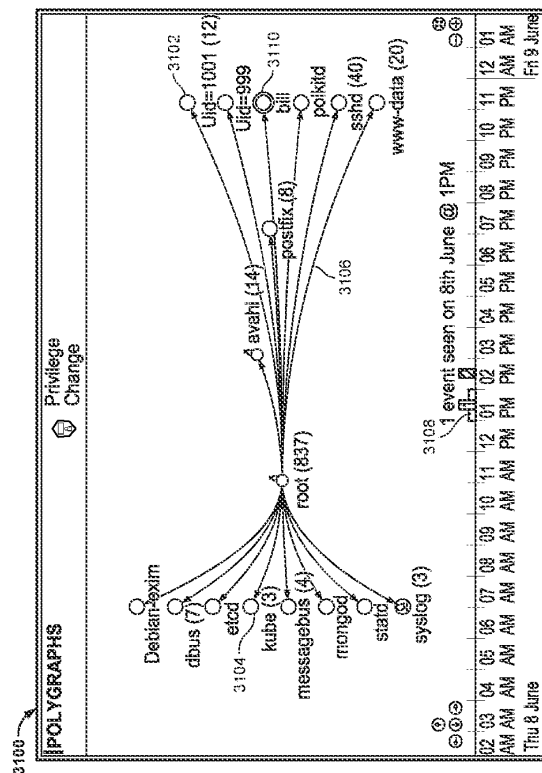
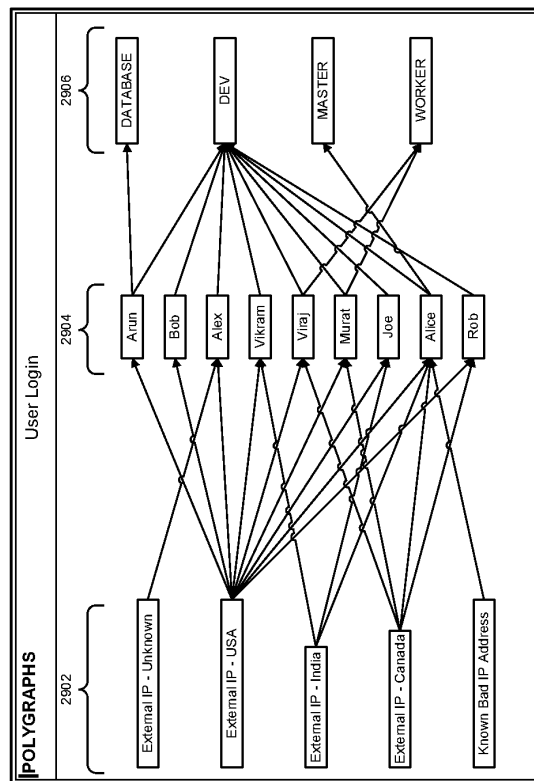
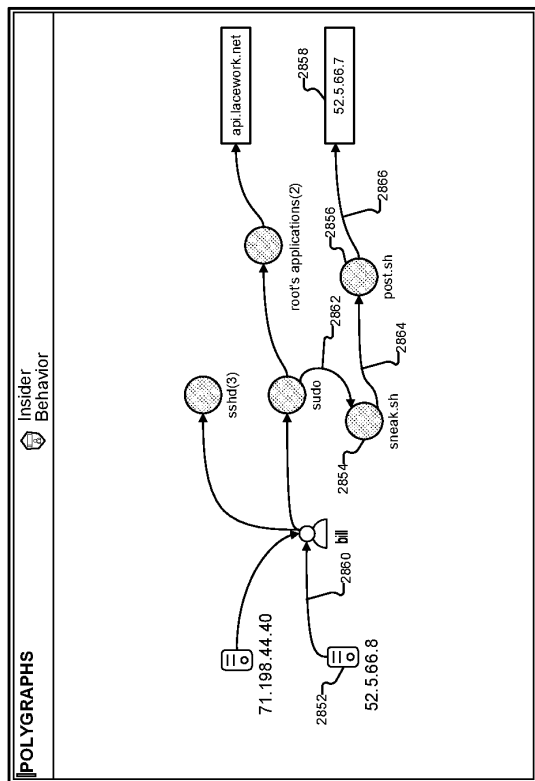
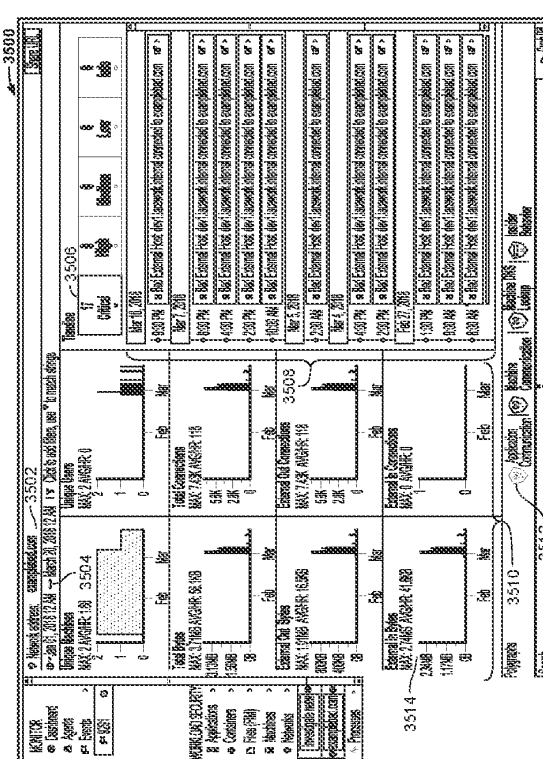
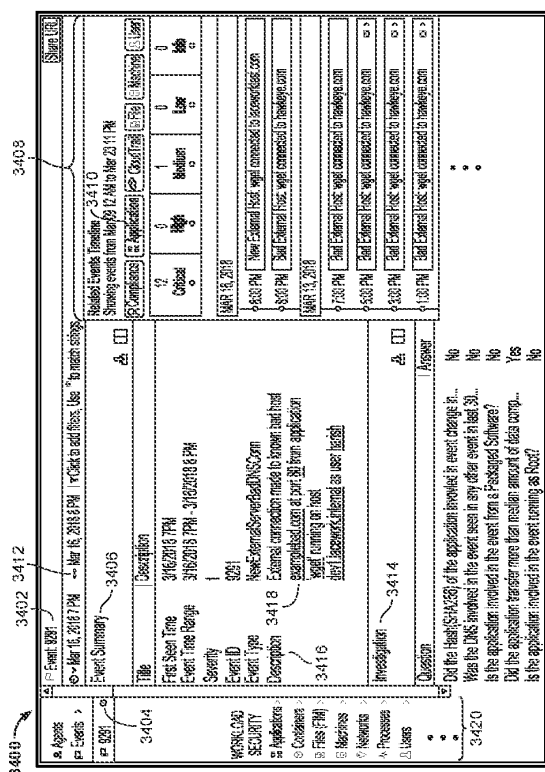
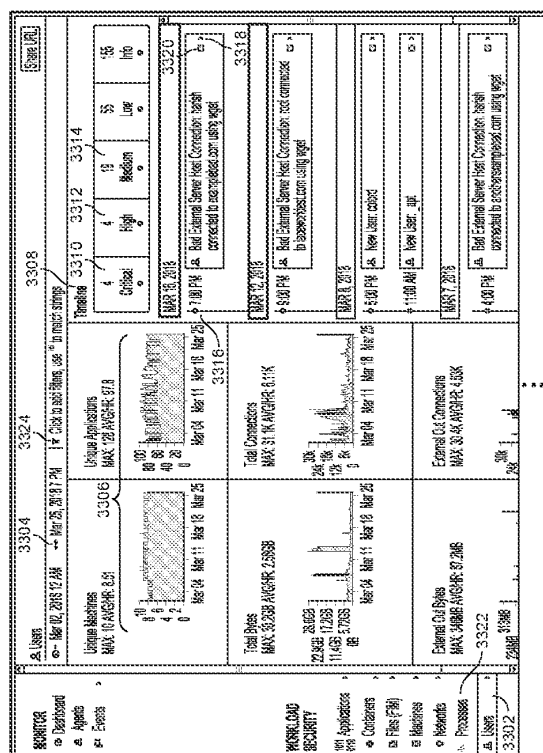
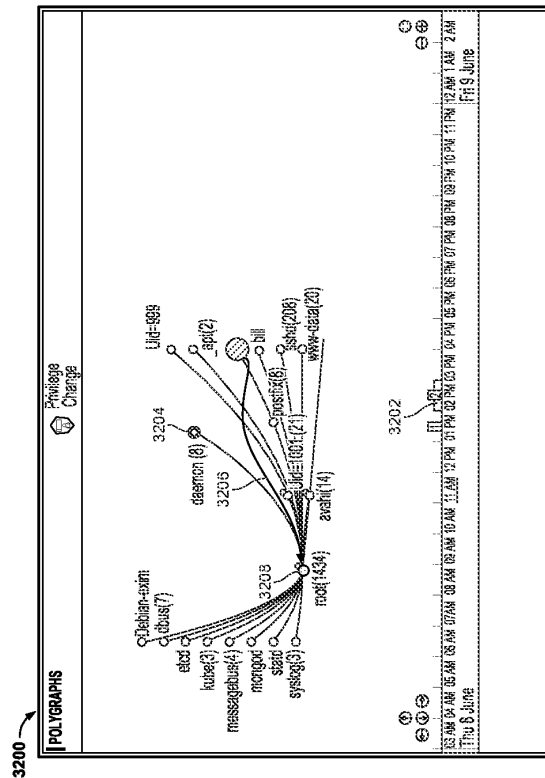


FIG. 28A





3600

```

"Card109" : {
  "props" : {
    "question": "Hostname, Binary, and Username",
    "scope" : "external"
  },
  "parameters": {
    ("name" : "StartTimeRange", "type" : "TIMESTAMP"),
    ("name" : "EndTimeRange", "type" : "TIMESTAMP"),
  },
  "sources": {
    ("name" : "VIEW INTERNAL.ENTITY VIEW T", "alias" : "P"),
    ("name" : "ProcessClusterFilters", "alias" : "Act")
  },
  "returns": {
    ("field" : "P.EXE_PATH", "alias" : "EXE_PATH"),
    ("field" : "P.USERNAME", "props" : {"type" : "USERNAME"}),
    ("field" : "P.PROPS_PROCESS:SSH_SESSION:props:username", "alias" : "ORIGIN_USER"),
    ("field" : "P.HOSTNAME", "props" : {"type" : "HOSTNAME"}),
    ("expr" : "COUNT(DISTINCT #P.MID), #P.PID_HASH)", "type" : "NUMBER", "alias" : "NUM_PROCESSES")
  },
  "keys": {
    ("type" : "PID_KEY", "fieldRefs" : ["P.MID", "P.PID_HASH"])
  },
  "joins": {
    ("type" : "Implicit", "left": "PID_KEY", "right": "Act.P_PID_KEY")
  },
  "groups": {
    ("field" : "1"), ("field" : "2"), ("field" : "3"), ("field" : "4")
  },
  "filters": {
    ("type" : "eq", "field" : "START TIME", "value" : "#(StartTimeRange)"),
    ("type" : "le", "field" : "END TIME", "value" : "#(EndTimeRange)"),
    ("type" : "eq", "field" : "ENTITY TYPE", "value" : "Process"),
    ("type" : "ne", "field" : "PID_HASH", "value" : "0")
  },
  "filterMap": {
    "P.PROPS_PROCESS:SSH_SESSION:props:username" : ["SSH_ORIGIN_USERNAME"]
  }
}

```

FIG. 36

```

"UserTrackingMemberOfIPAddress_MV" : {
  "type": "Function",
  "sources": [
    ("name": "ClusterMemberOfIPAddress_MV")
  ],
  "parameters": [
    ("name": "ClusterType", "value": "UserTracking")
  ]
},
"UserTrackingMemberOfUser_MV" : {
  "type": "Function",
  "sources": [
    ("name": "ClusterMemberOfUsername_MV")
  ],
  "parameters": [
    ("name": "ClusterType", "value": "UserTracking")
  ]
},
"UserTrackingMemberOfWSSep_MV" : {
  "type": "Function",
  "sources": [
    ("name": "ClusterMemberOfWSSep_MV")
  ],
  "parameters": [
    ("name": "ClusterType", "value": "UserTracking")
  ]
}

```

FIG. 37

```

curl "localhost:8080/card/describe/
Card262DCGUID=DEVTEST_CDB_DEVTEST_00_B10C8DBE1CFA99C1F274B"
{
  "ok": true,
  "schema": [
    {
      "name": "MID",
      "type": "Number",
      "props": {
        "utype": "MID",
        "title": "Machine ID"
      }
    },
    {
      "name": "MACHINE_NAME",
      "type": "String",
      "props": {
        "utype": "MACHINE_NAME",
        "title": "Machine Name"
      }
    },
    ...
  ],
  "parameters": [
    {
      "name": "JoinOption",
      "type": null,
      "default": "OR",
      "props": null
    },
    {
      "name": "StartTimeRange",
      "type": null,
      "default": null,
      "props": null
    },
    {
      "name": "EndTimeRange",
      "type": null,
      "default": null,
      "props": null
    }
  ]
}

```

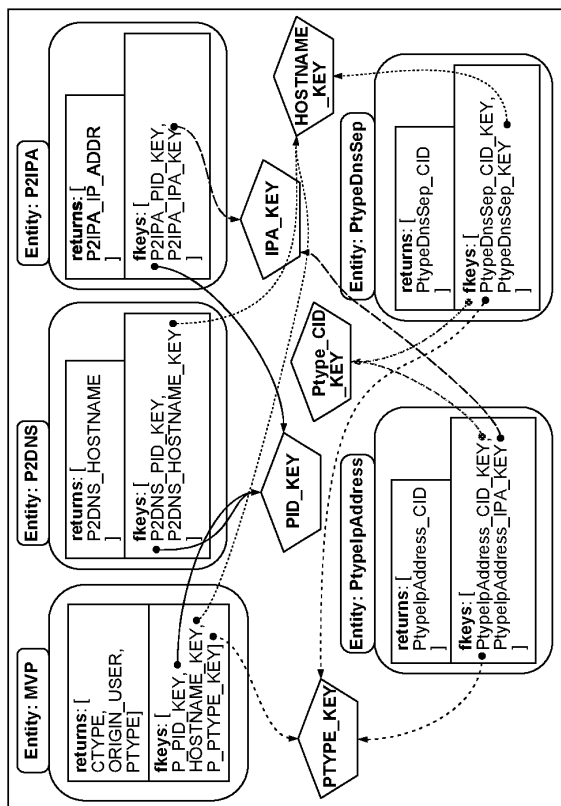
FIG. 38

```

curl "localhost:8080/card/describe/Card217DCGUID=DEVTEST_CDB_DEVTEST_00_B10C8DBE1CFA99C1F274B"
{
  "ok": true,
  "schema": {
    "nodes": [
      ("name": "ID", "type": "Number"),
      ("name": "TYPE", "type": "String"),
      ("name": "KEY", "type": "JSON"),
      ("name": "PROPS", "type": "JSON")
    ],
    "edges": [
      ("name": "ID", "type": "Number"),
      ("name": "TYPE", "type": "String"),
      ("name": "SRC_TYPE", "type": "String"),
      ("name": "DST_TYPE", "type": "String"),
      ("name": "SRC_KEY", "type": "JSON"),
      ("name": "DST_KEY", "type": "JSON"),
      ("name": "PROPS", "type": "JSON")
    ]
  },
  "graph": {
    "SOURCE": {
      "type": "Ptype",
      "key": {
        ("name": "cid", "filterName": "KEY_CID", "type": "Number")
      },
      "props": [
        ("name": "client", "filterName": "PROPS_CLIENT", "type": "Number"),
        ("name": "members", "filterName": "PROPS_MEMBERS", "type": "Number"),
        ("name": "tier", "filterName": "PROPS_TIER", "type": "Number"),
        ("name": "server", "filterName": "PROPS_SERVER", "type": "Number"),
        ("name": "ctype", "filterName": "PROPS_ctype", "type": "String")
      ]
    },
    "TARGET": {
      "type": "Ptype",
      "key": {
        ("name": "cid", "filterName": "KEY_CID", "type": "Number")
      },
      "props": [
        ("name": "client", "filterName": "PROPS_CLIENT", "type": "Number"),
        ("name": "members", "filterName": "PROPS_MEMBERS", "type": "Number"),
        ("name": "tier", "filterName": "PROPS_TIER", "type": "Number"),
        ("name": "server", "filterName": "PROPS_SERVER", "type": "Number"),
        ("name": "ctype", "filterName": "PROPS_ctype", "type": "String")
      ]
    },
    "EDGE": {
      "type": "ConnectedTo",
      "props": []
    }
  }
}

```

FIG. 39



```
curl -s -H "Content-Type: application/json" -X POST -d '{
  "filters": {
    "type": "like",
    "field": "ProcessClusterFilters.F2INS_HOSTNAME",
    "value": "google"
  },
  "paramInfo": {
    "name": "StartTimeRange",
    "value": "6/17/2017"
  },
  {
    "name": "EndTimeRange",
    "value": "6/18/2017"
  }
}

' "localhost:8080/card/query/
Card1897DCGUD=STAGING_CDB_STAGING_00_DEEP95AE3C794A7FF1D08"
"data": {
  "EXE_PATH": "/usr/lib/jvm/java-8-oracle/jre/bin/java",
  "USERNAME": "root",
  "PASSWORD": "root",
  "HOSTNAME": "staging_worker5.lacework.internal",
  "NUM_PROCESSES": "1"
},
"ok": true
}
```

FIG. 41

FIG. 42

[illegible]

```

WITH GRAPH_EDGE_VIEW_0 AS (
    WITH GRAPH_EDGE_VIEW_0 AS (
        SELECT ID, SRC_TYPE, DST_KEY, DST_TYPE, DST_KEY_VAL, TYPE, PROPS, START_TIME, END_TIME
        FROM GRAPH_INTERNAL_EDGE_0
        WHERE START_TIME >= TO_TIMESTAMP('6/17/2017')
          AND END_TIME <= TO_TIMESTAMP('6/18/2017')
    )
    SELECT ID, SRC_TYPE, DST_KEY, DST_TYPE, DST_KEY_VAL, TYPE, PROPS, START_TIME, END_TIME
    FROM GRAPH_EDGE_VIEW_0
)
Card189 AS (
    WITH P AS (
        SELECT START_TIME, END_TIME, ENTITY_TYPE, MED, FID_HASH, USERNAME, EXE_PATH, HOSTNAME, PROPS_PROCESS
        FROM NVTM_INTERNAL_ENTITY_VIEW_1
    ),
    Act_MVP AS (
        WITH ENTITY_VIEW_T_42 AS (
            SELECT START_TIME, END_TIME, ENTITY_TYPE, MED, FID_HASH, USERNAME, EXE_PATH, HOSTNAME, CONTAINER, NET_STATS,
            DISK_USAGE, TASKS, CPU_USAGE, FID_HASH_MAP, COLLECTED_WAY, PROPS_MACHINE, PROPS_PROCESS, PROPS_USER, PROPS_EXE_PATH, PROPS_CONTAINER
            FROM NVTM_INTERNAL_ENTITY_VIEW_1
        )
        SELECT ENTITY_VIEW_T_42.MED.MED, ENTITY_VIEW_T_42.FID_HASH.FID_HASH, ENTITY_VIEW_T_42.HOSTNAME.HOSTNAME,
        ENTITY_VIEW_T_42.USERNAME.USERNAME, ENTITY_VIEW_T_42.EXE_PATH.EXE_PATH, ENTITY_VIEW_T_42.CONTAINER.CONTAINER,
        ENTITY_VIEW_T_42.COLLECTED_WAY."COLLECTED_WAY", ENTITY_VIEW_T_42.TASKS.TASKS,
        ENTITY_VIEW_T_42.PROPS_PROCESS."PROPS_PROCESS"
        FROM ENTITY_VIEW_T_42
        WHERE ENTITY_VIEW_T_42.ENTITY_TYPE = 'Process'
        AND ENTITY_VIEW_T_42.START_TIME >= TO_TIMESTAMP('6/17/2017')
        AND ENTITY_VIEW_T_42.END_TIME <= TO_TIMESTAMP('6/18/2017')
    ),
    Act_P2INS AS (
        WITH ProcessToExternalInsView_6 AS (
            SELECT DISTINCT SRC_KeyvalId,SRCKey,MED,
            COLLECTED_INS_ID,IPV4_IPv6,(?)REGINT FID_HASH, DST_KEY\hostname::VARCHAR(255)
            FROM GRAPH_End_View
            WHERE TYPE = "Connected"
            AND SRC_Type IN ('Process','Mechanic')
            AND DST_Key\hostname <> ''
        )
        SELECT MED, FID_HASH, HOSTNAME
        FROM ProcessToExternalInsView_6
    )
    SELECT P.EXE_PATH.EXE_PATH, P.USERNAME.USERNAME, P.PROPS_PROCESS.SSH_SESSIONID::username ORIGIN_USER,
    P.HOSTNAME.HOSTNAME, CARD(DISTINCT P.MED, P.FID_HASH) NM Processes
    FROM P
    WHERE P.START_TIME >= TO_TIMESTAMP('6/17/2017')
      AND P.END_TIME <= TO_TIMESTAMP('6/18/2017')
      AND P.ENTITY_TYPE = 'Process'
      AND P.FID_HASH < 0
      AND IF(MED, P.FID_HASH) IN (
        SELECT Act_MVP.MED, Act_MVP.FID_HASH FROM Act_MVP
        WHERE Act_MVP.MED, Act_MVP.FID_HASH IN (
          SELECT Act_P2INS.MED, Act_P2INS.FID_HASH FROM
          Act_P2INS
          WHERE Act_P2INS.HOSTNAME LIKE ?||? ESCAPE '\\'
        )
      )
    GROUP BY 1, 2, 3, 4
)
SELECT Card189.EXE_PATH.EXE_PATH, Card189.USERNAME.USERNAME, Card189.ORIGIN_USER ORIGIN_USER, Card189.HOSTNAME
HOSTNAME,
Card189.NM_Processes NM_Processes FROM Card189 LIMIT 5000

```

FIG. 43

FIG. 44

FIG. 46

FIG. 45

FIG. 47A

FIG. 47B

FIG. 47C

FIG. 47D

4800

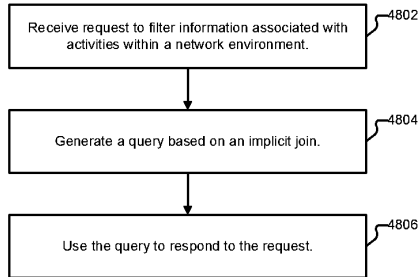


FIG. 48

```

public String sqlGen(QueryContext context) {
    presqlGen();
    this.pullUpCachable(context);
    StringBuilder sb = new StringBuilder();
    String with = sqlWith(context);
    String from = sqlFrom(context);
    String where = sqlWhere(context);
    String projections = sqlProjection(context);
    String gb = sqlGroupBy(context);
    String ob = sqlOrderBy(context);
    // stitch everything together
    if (!with.isEmpty()) {
        sb.append("WITH ");
        if (this.parentRef.isRoot()) {
            int cnt = context.sqlGenCachable(sb);
            if (cnt > 0)
                sb.append(", ");
        }
        sb.append(with).append(" ");
    }
    sb.append(projections).append(" ").append(from);
    if (!where.isEmpty())
        sb.append(" ").append(where);
    if (!gb.isEmpty())
        sb.append(" ").append(gb);
    if (!ob.isEmpty())
        sb.append(" ").append(ob);
    sb.append(sqlLimitOffset(context));
    return sb.toString();
}
  
```

FIG. 49A

```

"SQL1" : {
  "type" : "SQL",
  "sql" : [
    "SELECT MID, PID_HASH FROM PROCESS_DETAILS_T"
  ],
  "returns" : [
    { "field" : "MID", "type" : "NUMBER" },
    { "field" : "PID_HASH", "type" : "NUMBER" }
  ]
},
"SQL2" : {
  "type" : "SQL",
  "sql" : [
    "WITH S1 AS ({SQL1}) ",
    "SELECT MID, COUNT(DISTINCT PID_HASH) CNT FROM S1 ",
    "GROUP BY MID"
  ],
  "returns" : [
    { "field" : "MID", "type" : "NUMBER" },
    { "field" : "CNT", "type" : "NUMBER" }
  ]
},
"Entity1" : {
  "sources" : [
    { "name" : "SQL2", "alias" : "S2" }
  ],
  "returns" : [
    { "field" : "CNT" }
  ],
  "filters" : [
    { "type" : "eq", "field" : "S2.MID", "value" : "1" }
  ]
}
  
```

FIG. 49B

```

"ProcessClusterFilters" : {
  "props" : { "scope" : "external" },
  "type" : "Filters",
  "parameters" : [
    { "name" : "JoinOption", "default" : "AND" }
  ],
  "sources" : [
    { "name" : "DummyClusterMember MVP", "alias" : "Proxy" },
    { "name" : "ActiveProcessDetailsNoDistinct", "alias" : "MVP" },
    { "name" : "InterfaceFilterView", "alias" : "I1" },
    { "name" : "ProcessToExternalINSView", "alias" : "P2DNS" },
    { "name" : "MachineToExternalINSView", "alias" : "M2DNS" },
  ],
  "returns" : [
    {
      "field" : "MVP.MID",
      "alias" : "P_MID",
      "props" : {
        "title" : "Process Machine ID", "utype" : "MID",
        "filterGroup" : "Machine",
        "format" : "singleValue",
        "functions" : [
          { "timeSeries" : "MachineLTimeSeries" }
        ]
      }
    },
    {
      "field" : "MVP.MID",
      "alias" : "MID",
      "props" : {
        "title" : "Process Machine ID", "utype" : "MID",
        "filterGroup" : "Machine",
        "format" : "singleValue",
        "functions" : [
          { "timeSeries" : "MachineLTimeSeries" }
        ]
      }
    },
    {
      "field" : "MVP.PID_HASH",
      "alias" : "PID_HASH",
      "props" : {
        "title" : "Process Hash",
        "filterGroup" : "Process",
        "format" : "singleValue",
        "utype" : "PID_HASH"
      }
    }
  ],
  "filters" : [
    { "type" : "eq", "field" : "S2.MID", "value" : "1" }
  ]
}
  
```

FIG. 50A

```
curl -H "Content-Type: application/json" -X POST -d '{
  "returns": {
    "field": "PID",
    "field": "HOSTNAME"
  },
  "filters": {
    "type": "like", "field": "ProcessName",
    "type": "like", "field": "ProcessName",
    "limit": 5,
    "orders": {
      "field": "HOSTNAME", "order": "desc",
      "field": "PID"
    }
  }
}
```

FIG. 51

```
curl -H "Content-Type: application/json" -X POST -d '{"filters": [{"type": "in", "field": "MID", "values": ["153"]}], "returns": [{"field": "PID HASH"], "orders": [{"field": "1"], "limit": "3"}]}' "localhost:8080/card/query/ActiveProcessByConnection?DCGUID=DEVTEST_CDB_DEVTEST_00_B10C8DB61CFA89C1F274B&StartTimeRange=3/30/2016&2010:00:00&EndTimeRange=3/30/2016&2011:00:00"
```

FIG. 52A

```

WITH ActiveProcessByConnection AS (
  WITH NODE T_1 AS (
    SELECT TYPE, KEY, START_TIME, END_TIME
    FROM GRAPH_INTERNAL.NODE_T
  )
  SELECT DISTINCT
    KEY:mld MID, KEY:pid_hash PID_HASH
    FROM NODE_T_1
  WHERE NODE_T_1.TYPE = 'Process'
  AND KEY:mld > 0
  AND (KEY:pid_hash < 0 OR KEY:pid_hash > 65535)
  AND NODE_T_1.START_TIME >= TO_TIMESTAMP('3/30/2016 10:00:00')
  AND NODE_T_1.END_TIME <= TO_TIMESTAMP('3/30/2016 11:00:00')
)
SELECT ActiveProcessByConnection.PID_HASH PID_HASH
FROM ActiveProcessByConnection
WHERE ActiveProcessByConnection.MID IN (153)

ORDER BY 1
LIMIT 3

```

FIG. 52C

```
curl -H "Content-Type: application/json" -X POST -d '{}' 'localhost:8080/
card/query/
CardID:CGUID=DEVTEST_CDB_DEVTEST_00_B10C8DBE1CF89C1F274B&startTimeRange=3
/30/2016&2010:00:00&endTimeRange=3/30/2016&2011:00:00"
{"data":[{"COUNT":3}], "ok":true}
```

FIG. 53B

```
"Card1": {
  "props": { "scope": "external",
  "parameters": {
    { "name": "StartTimeRange", "required": true},
    { "name": "EndTimeRange", "required": true}
  },
  "sources": [
    { "name": "ActiveProcessByConnection", "alias": "S"},
    { "type": "Table", "name": "PUBLIC.PROCESS_DETAILS_T", "alias": "P"},
    { "name": "ProcessActivity", "alias": "Act"},
    { "name": "NetworkActivity", "alias": "Net"}
  ],
  "returns": [
    {
      "agg": "COUNT DISTINCT",
      "field": "P.USERNAME",
      "alias": "COUNT",
      "props": { "title": "# Of Distinct User Names" }
    }
  ],
  "fkeys": [
    { "type": "PID_XKEY", "fieldRefs": { "P.MID", "P.PID_HASH" } }
  ],
  "joins": [
    { "type": "in", "left": "P.MID", "right": "S.MID"},
    { "type": "in", "left": "P.PID_HASH", "right": "S.PID_HASH"},
    { "type": "Application", "left": "P.ID_XKEY", "right": "Net.P_Act_XKEY"},
    { "type": "Application", "left": "P.ID_XKEY", "right": "Net.SPC_XKEY"},
    { "type": "Application", "left": "P.ID_XKEY", "right": "Net.OSX_XKEY"}
  ],
  "filters": [
    { "type": "le", "field": "P.CREATED_TIME", "value": "${EndTimeRange}" }
  ]
},
}
```

FIG. 53A

```
WITH Card1 AS (
  WITH P AS (
    SELECT MID, CREATED_TIME, PID_HASH, USERNAME
    FROM PUBLIC.PROCESS_DETAILS_T
  ),
  S AS (
    WITH NODE_T_1 AS (
      SELECT TYPE, KEY, START_TIME, END_TIME FROM GRAPH_INTERNAL.NODE_T
    )
    SELECT DISTINCT KEY:mid MID, KEY:pid_hash PID_HASH
    FROM NODE_T_1
    WHERE NODE_T_1.TYPE = 'Process'
    AND KEY:mid > 0
    AND (KEY:pid_hash < 0 OR KEY:pid_hash > 65535)
    AND NODE_T_1.START_TIME >= TO_TIMESTAMP('3/30/2016 10:00:00')
    AND NODE_T_1.END_TIME <= TO_TIMESTAMP('3/30/2016 11:00:00')
  )
  SELECT COUNT(DISTINCT P.USERNAME) COUNT
  FROM P
  WHERE P.CREATED_TIME <= TO_TIMESTAMP('3/30/2016 11:00:00')
  AND (P.MID, P.PID_HASH) IN (SELECT MID, PID_HASH FROM S)
)
SELECT Card1.COUNT COUNT
FROM Card1 LIMIT 5000
```

FIG. 53C

```
curl -H "Content-Type: application/json" -X POST -d '{"filters":
[{"type":"eq","field":"ProcessActivity.USERNAME","value":"root"}]}' 'localhost:8080/
card/query/CardID:CGUID=DEVTEST_CDB_DEVTEST_00_B10C8DBE1CF89C1F274B&startTimeRange=3/
30/2016&2010:00:00&endTimeRange=3/30/2016&2011:00:00"
```

FIG. 53D

```
WITH Card1 AS (
  WITH P AS (
    SELECT MID, CREATED_TIME, PID_HASH, USERNAME
    FROM PUBLIC.PROCESS_DETAILS_T
  ),
  S AS (
    WITH NODE_T_1 AS (
      SELECT TYPE, KEY, START_TIME, END_TIME
      FROM GRAPH_INTERNAL.NODE_T
    )
    SELECT DISTINCT
    KEY:mid MID,
    KEY:pid_hash PID_HASH
    FROM NODE_T_1
    WHERE NODE_T_1.TYPE = 'Process'
    AND KEY:mid > 0
    AND (KEY:pid_hash < 0 OR KEY:pid_hash > 65535)
    AND NODE_T_1.START_TIME >= TO_TIMESTAMP('3/30/2016 10:00:00')
    AND NODE_T_1.END_TIME <= TO_TIMESTAMP('3/30/2016 11:00:00')
  ),
  Act_P AS (
    WITH ProcessDetailsTable_7 AS (
      SELECT LSN, MID, CREATED_TIME, START_TIME,
      PID, PID_HASH, PPID, PPID_HASH, PGID,
      PGID_HASH, SID, SID_HASH, TTY,
      UID, USERNAME, EUID, CMDLINE_HASH, EXE_PATH
      FROM PUBLIC.PROCESS_DETAILS_T
      WHERE CREATED_TIME <= TO_TIMESTAMP('3/30/2016 11:00:00')
    )
    UNION
    SELECT 0 LSN, MID, CREATED_TIME, LAST_BOOT_TIME,
    NULL PID, 0 PID_HASH, NULL PPID, NULL PPID_HASH, NULL PGID,
    NULL PGID_HASH, NULL SID, NULL SID_HASH,
    NULL TTY, NULL UID, NULL USERNAME, NULL EUID,
    NULL CMDLINE_HASH, NULL EXE_PATH
    FROM (
      SELECT MID, CREATED_TIME, LAST_BOOT_TIME,
      ROW() OVER (PARTITION BY MID ORDER BY CREATED_TIME) RNK
      FROM MACHINE_DETAILS_T
      WHERE CREATED_TIME <= TO_TIMESTAMP('3/30/2016 11:00:00')
    )
    WHERE RNK = 1
  )
  SELECT MID, PID, PID_HASH, PPID, PPID_HASH, PGID, PGID_HASH, TTY, UID,
  USERNAME, EUID, CMDLINE_HASH, EXE_PATH
  FROM ProcessDetailsTable_7
)
SELECT COUNT(DISTINCT P.USERNAME) COUNT
FROM P
WHERE P.CREATED_TIME <= TO_TIMESTAMP('3/30/2016 11:00:00')
AND (P.MID, P.PID_HASH) IN (SELECT MID, PID_HASH FROM S)
AND (P.MID, P.PID_HASH) IN (
  SELECT Act_P.MID, Act_P.PID_HASH FROM Act_P WHERE Act_P.USERNAME = 'root'
)
}
SELECT Card1.COUNT COUNT
FROM Card1 LIMIT 5000
```

FIG. 53E

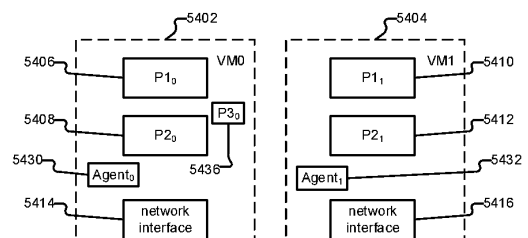


FIG. 54A

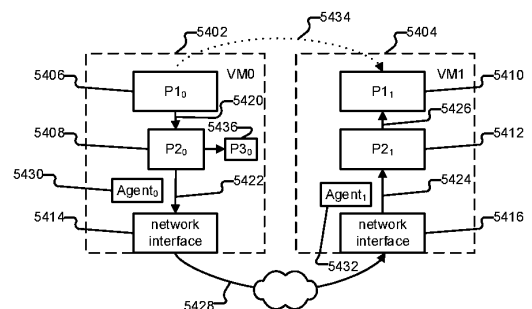


FIG. 54B

Electronic Patent Application Fee Transmittal				
Application Number:				
Filing Date:				
Title of Invention:		EDGE BASED DATA PLATFORM		
First Named Inventor/Applicant Name:		VIKRAM KAPOOR		
Filer:		Edward Joseph Lenart/Kate Hill		
Attorney Docket Number:		LACEP040		
Filed as Large Entity				
Filing Fees for Provisional				
Description	Fee Code	Quantity	Amount	Sub-Total in USD(\$)
Basic Filing:				
PROVISIONAL APPLICATION FILING	1005	1	300	300
Pages:				
Claims:				
Miscellaneous-Filing:				
Petition:				
Patent-Appeals-and-Interference:				
Post-Allowance-and-Post-Issuance:				

Description	Fee Code	Quantity	Amount	Sub-Total in USD(\$)
Extension-of-Time:				
Miscellaneous:				
Total in USD (\$)				300

Electronic Acknowledgement Receipt	
EFS ID:	43687745
Application Number:	63240818
International Application Number:	
Confirmation Number:	1130
Title of Invention:	EDGE BASED DATA PLATFORM
First Named Inventor/Applicant Name:	VIKRAM KAPOOR
Customer Number:	182115
Filer:	Edward Joseph Lenart/Kate Hill
Filer Authorized By:	Edward Joseph Lenart
Attorney Docket Number:	LACEP040
Receipt Date:	03-SEP-2021
Filing Date:	
Time Stamp:	19:44:44
Application Type:	Provisional

Payment information:

Submitted with Payment	yes
Payment Type	CARD
Payment was successfully received in RAM	\$ 300
RAM confirmation Number	E202193J45084477
Deposit Account	
Authorized User	
The Director of the USPTO is hereby authorized to charge indicated fees and credit any overpayment as follows:	

File Listing:					
Document Number	Document Description	File Name	File Size(Bytes)/ Message Digest	Multi Part /.zip	Pages (if appl.)
1	Application Data Sheet	2021-09-03_LACEP040_ADS.pdf	1873977	no	8
			9f3c3a543fa17e3735ec499d722a85187a9eb0d		
Warnings:					
Information:					
2		2021-09-03_LACEP040_ProvisionalApplication.pdf	533820	yes	31
			75ad9b84434405a3c52f02b1c5834096710eb311		
	Multipart Description/PDF files in .zip description				
	Document Description		Start	End	
	Specification		1	30	
	Claims		31	31	
Warnings:					
Information:					
3	Drawings-only black and white line drawings	2021-09-03_LACEP040_ProvisionalDrawings.pdf	2312713	no	16
			f4a05f5212a32a77717051283ed1c1f645096990		
Warnings:					
Information:					
4	Fee Worksheet (SB06)	fee-info.pdf	37205	no	2
			8952251b3305c789f69787ec8ffadba729ae9405		
Warnings:					
Information:					
Total Files Size (in bytes):			4757715		

This Acknowledgement Receipt evidences receipt on the noted date by the USPTO of the indicated documents, characterized by the applicant, and including page counts, where applicable. It serves as evidence of receipt similar to a Post Card, as described in MPEP 503.

New Applications Under 35 U.S.C. 111

If a new application is being filed and the application includes the necessary components for a filing date (see 37 CFR 1.53(b)-(d) and MPEP 506), a Filing Receipt (37 CFR 1.54) will be issued in due course and the date shown on this Acknowledgement Receipt will establish the filing date of the application.

National Stage of an International Application under 35 U.S.C. 371

If a timely submission to enter the national stage of an international application is compliant with the conditions of 35 U.S.C. 371 and other applicable requirements a Form PCT/DO/EO/903 indicating acceptance of the application as a national stage submission under 35 U.S.C. 371 will be issued in addition to the Filing Receipt, in due course.

New International Application Filed with the USPTO as a Receiving Office

If a new international application is being filed and the international application includes the necessary components for an international filing date (see PCT Article 11 and MPEP 1810), a Notification of the International Application Number and of the International Filing Date (Form PCT/RO/105) will be issued in due course, subject to prescriptions concerning national security, and the date shown on this Acknowledgement Receipt will establish the international filing date of the application.