Enabling Efficient Cyber Threat Hunting With Cyber Threat Intelligence

Peng Gao*, Fei Shao†, Xiaoyuan Liu*, Xusheng Xiao†, Zheng Qin‡, Fengyuan Xu‡
Prateek Mittal§, Sanjeev R. Kulkarni§, Dawn Song*
*University of California, Berkeley †Case Western Reserve University
‡National Key Lab for Novel Software Technology, Nanjing University §Princeton University
*{penggao,xiaoyuanliu,dawnsong}@berkeley.edu †{fxs128,xusheng.xiao}@case.edu
‡{qinzheng,fengyuan.xu}@nju.edu.cn §{pmittal,kulkarni}@princeton.edu

2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00024

Abstract—**Log-based cyber threat hunting has emerged as an important solution to counter sophisticated attacks. However, existing approaches require non-trivial efforts of manual query construction and have overlooked the rich external threat knowledge provided by open-source Cyber Threat Intelligence (OSCTI). To bridge the gap, we propose THREATRAPTOR, a system that facilitates threat hunting in computer systems using OSCTI. Built upon system auditing frameworks, THREATRAPTOR provides (1) an unsupervised, light-weight, and accurate NLP pipeline that extracts structured threat behaviors from unstructured OSCTI text, (2) a concise and expressive domain-specific query language, TBQL, to hunt for malicious system activities, (3) a query synthesis mechanism that automatically synthesizes a TBQL query for hunting, and (4) an efficient query execution engine to search the big audit logging data. Evaluations on a broad set of attack cases demonstrate the accuracy and efficiency of THREATRAPTOR in practical threat hunting.**

I. Introduction

Recent cyber attacks have plagued many well-protected businesses [1], [2]. These attacks often exploit multiple types of vulnerabilities to infiltrate into target systems in multiple stages, posing challenges for effective countermeasures. To counter these attacks, *ubiquitous system auditing* has emerged as an important approach for monitoring system activities [3]–[6]. System auditing collects system-level auditing events about system calls from OS kernel as system audit logs. The collected system audit logging data further enables approaches to hunt for cyber threats via query processing [7]–[9].

Cyber threat hunting is the process of proactively and iteratively searching for malicious actors and indicators in various logs, which is critical to early-stage detection. Despite numerous research outcomes [7]–[9] and industry solutions [10], [11], existing approaches, however, require non-trivial efforts of manual query construction and have overlooked the rich external knowledge about threats provided by open-source Cyber Threat Intelligence (OSCTI). Hence, the current threat hunting process is labor-intensive and error-prone.

OSCTI [12] is a form of evidence-based knowledge and has received growing attention from the community, enabling companies and organizations to gain visibility into the fast-evolving threat landscape. Commonly, knowledge about threats is presented in a vast number of publicly available OSCTI sources. Structured OSCTI feeds [13]–[16] have primarily focused on Indicators of Compromise (IOCs) [17], which are forensic artifacts of an intrusion such as malicious file/process names, virus signatures, and IPs/domains of botnets. Though useful in capturing fragmented views of threats, these disconnected IOCs lack the capability to uncover the complete threat scenario as to how the threat unfolds into multiple steps. Consequently, defensive solutions that rely on these low-level, fragmented indicators [10], [11] can be easily evaded when the attacker re-purposes the tools and changes their signatures. In contrast, unstructured OSCTI reports [18]–[20] contain more comprehensive knowledge about threats. For example, descriptive relationships between IOCs contain knowledge about *multi-step threat behaviors* (e.g., “read” relationship between two IOCs “/bin/tar” and “/etc/passwd” in Figure 2), which is critical to uncovering the complete threat scenario. Besides, such connected threat behaviors are tied to the attacker’s goals and thus more difficult to change. Unfortunately, prior approaches do not provide an automated way to harvest such knowledge and use it for threat hunting.

Challenges. In this work, we seek to design automated techniques to (1) extract knowledge about threat behaviors (IOCs and their relationships) from unstructured OSCTI reports, and (2) use the extracted knowledge to facilitate threat hunting. We identify two major challenges. First, accurately extracting threat knowledge from natural-language OSCTI text is not trivial. This is due to the presence of massive nuances particular to the security context, such as special characters (e.g., dots, underscores) in IOCs. These nuances confuse most NLP modules (e.g., sentence segmentation, tokenization), making existing information extraction tools ineffective [21], [22]. Second, system auditing often produces a huge amount of daily logs (0.5 GB $\sim$ 1 GB for 1 enterprise host [23]), and hence threat hunting is a procedure of “finding a needle in a haystack”. Such big data poses challenges for solutions to store and query the data efficiently to hunt for malicious activities. To meet the requirement of timely threat hunting, knowledge extraction from OSCTI text also needs to be efficient.

Contribution. We propose THREATRAPTOR, a system that facilitates threat hunting in computer systems using OSCTI. We built THREATRAPTOR ($\sim$ 25K LOC) upon mature system auditing frameworks [24]–[26] for system audit logging data collection (Section III-A), and databases (PostgreSQL [27],

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00024

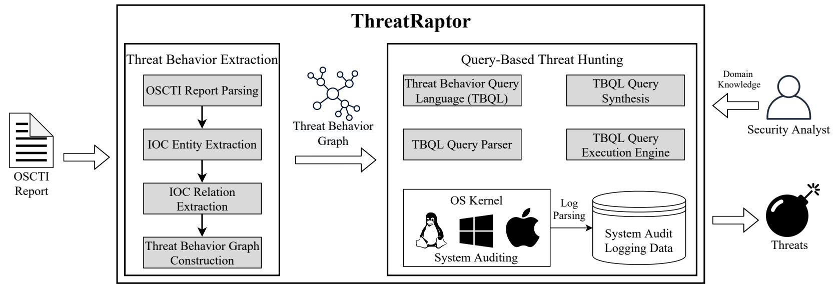


Fig. 1: The architecture of THREATRAPTOR

Neo4j [28]) for data storage (Section III-B). This enables our system to leverage the services provided by these mature infrastructures, such as data management, querying, and recovery. Besides, THREATRAPTOR has three novel designs:

(1) *Unsupervised, Light-Weight, and Accurate NLP Pipeline for Threat Behavior Extraction:* THREATRAPTOR employs a specialized NLP pipeline that targets the unique problem of IOC and IOC relation extraction from OSCTI text, which has not been studied in prior work. To handle nuances and meet the requirement of timely threat hunting, the pipeline adopts a collection of techniques (e.g., IOC protection, dependency parsing-based IOC relation extraction) to achieve accurate and efficient threat behavior extraction. The extracted threat behaviors are represented in a structured *threat behavior graph*, in which nodes represent IOCs and edges represent IOC relations. Compared to the unstructured OSCTI text, such structured threat behavior representation is more amenable to automated processing and integration (Section III-C).

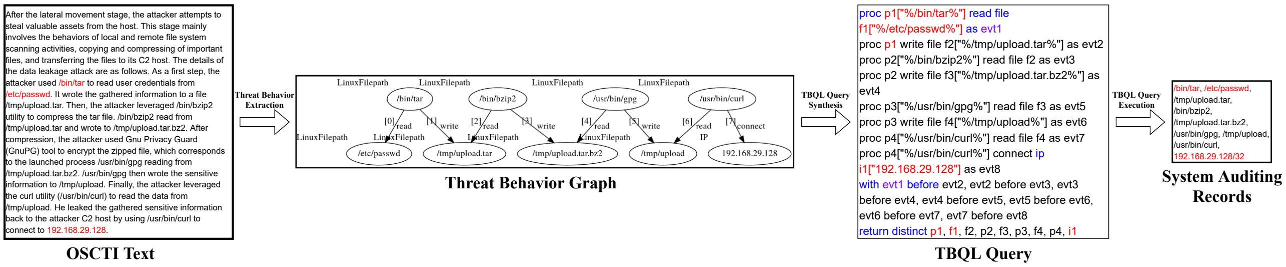
(2) *Domain-Specific Query Language & Query Synthesis*: To facilitate threat hunting over system audit logging data, THREATRAPTOR has an efficient query subsystem that employs a concise and expressive domain-specific query language, *Threat Behavior Query Language (TBQL)*, to query the log data stored in database backends. TBQL is a declarative language that integrates a collection of critical primitives for threat hunting in computer systems. For example, TBQL treats system entities (i.e., files, processes, network connections) and system events (i.e., file events, process events, network events) as first-class citizens, and provides explicit constructs for entity/event types, event operations, and event path patterns. With TBQL, complex multi-step system behaviors can be easily specified and searched (Section III-D).

To bridge the threat behavior graph with the query subsystem, THREATRAPTOR employs a *query synthesis mechanism* that automatically synthesizes a TBQL query from the constructed graph. This way, external knowledge about threat behaviors can be automatically integrated in threat hunting. No prior work has proposed a query language for threat hunting that supports the same set of features as supported in TBQL, and has considered the automation of the threat hunting procedure via query synthesis (Section III-E).

It is important to note that THREATRAPTOR also supports human-in-the-loop analysis via query editing: the security analyst can further revise the synthesized query to encode domain knowledge about the specific enterprise. In practice, threat hunting is an iterative process that involves multiple rounds of query editing and execution, and the conciseness and declarative nature of TBQL make this process efficient.

(3) *Efficient Query Execution:* To query the big data efficiently, THREATRAPTOR employs specialized optimizations for data storage and query execution engine. Specifically, THREATRAPTOR employs data reduction techniques to merge excessive system events while preserving adequate information. To execute a TBQL query, THREATRAPTOR decomposes it into parts and compiles each part into a semantically equivalent data query (i.e., a small SQL [29] or Cypher [30] query that will be executed in PostgreSQL or Neo4j databases). THREATRAPTOR then employs a *scheduling algorithm* to schedule the execution of these data queries, based on their estimated pruning power and semantic dependencies. Compared to the naive plan that compiles the TBQL query into a giant SQL or Cypher query to execute, our execution plan avoids the weaving of many joins and constraints together (which often leads to slow performance) and leverages the query semantics to speed up the execution. In addition to the exact search mode, THREATRAPTOR supports a *fuzzy search mode* based on inexact graph pattern matching, by extending [6]. This mode generalizes to cases where the searched graph pattern in a TBQL query deviates from the ground truth (could due to IOC changes or structural differences), which improves the generality of TBQL queries in threat hunting (Section III-F).

Evaluation. We deployed THREATRAPTOR on a physical testbed and performed a broad set of attacks to evaluate the system. The evaluation results demonstrate that: (1) THREATRAPTOR is able to accurately extract threat behaviors from OSCTI text (96.64% F1 for IOC extraction, 92.59% F1 for IOC relation extraction), performing much better than general information extraction approaches ($<$ 5% F1); (2) THREATRAPTOR is able to accurately find malicious system activities using OSCTI text (98.34% F1); (3) the entire pipeline of THREATRAPTOR is efficient. The threat behavior extraction and query synthesis parts take 0.52s on average. For query execution (in exact search mode), TBQL queries execute 22.7x faster than SQL queries for PostgreSQL backend, and 9.1x faster than Cypher queries for Neo4j backend; (4) TBQL queries are more concise than SQL queries ($>$ 2.8x) and Cypher queries ($>$ 2.2x). To the best of our knowledge, THREATRAPTOR is *the first system that bridges OSCTI with system auditing to facilitate cyber threat hunting in computer systems*. A system demo video is available at [31].



After the lateral movement stage, the attacker attempts to steal valuable assets from the host. This stage mainly involves the behaviors of local and remote file system scanning activities, copying and compressing of important files, and transferring the files to its C2 host. The details of the data leakage attack are as follows. As a first step, the attacker used /bin/tar to read user credentials from /etc/passwd. It wrote the gathered information to a file /tmp/upload.tar. Then, the attacker leveraged /bin/bzip2 utility to compress the tar file. /bin/bzip2 read from /tmp/upload.tar and wrote to /tmp/upload.tar.bz2. After compression, the attacker used Gnu Privacy Guard (GnuPG) tool to encrypt the zipped file, which corresponds to the launched process /usr/bin/gpg reading from /tmp/upload.tar.bz2. /usr/bin/gpg then wrote the sensitive information to /tmp/upload. Finally, the attacker leveraged the curl utility (/usr/bin/curl) to read the data from /tmp/upload. He leaked the gathered sensitive information back to the attacker C2 host by using /usr/bin/curl to connect to 192.168.29.128.

```
proc p1["%/bin/tar%"] read file
f1["%/etc/passwd%"] as evt1
proc p1 write file f2["%/tmp/upload.tar%"] as evt2
proc p2["%/bin/bzip2%"] read file f2 as evt3
proc p2 write file f3["%/tmp/upload.tar.bz2%"] as
evt4
proc p3["%/usr/bin/gpg%"] read file f3 as evt5
proc p3 write file f4["%/tmp/upload%"] as evt6
proc p4["%/usr/bin/curl%"] read file f4 as evt7
proc p4["%/usr/bin/curl%"] connect ip
i1["192.168.29.128"] as evt8
with evt1 before evt2, evt2 before evt3, evt3
before evt4, evt4 before evt5, evt5 before evt6,
evt6 before evt7, evt7 before evt8
return distinct p1, f1, f2, p2, f3, p3, f4, p4, i1
```

```
/bin/tar, /etc/passwd,
/tmp/upload.tar,
/bin/bzip2,
/tmp/upload.tar.bz2,
/usr/bin/gpg, /tmp/upload,
/usr/bin/curl,
192.168.29.128/32
```

Fig. 2: An example data leakage attack case demonstrating the whole processing pipeline of THREATRAPTOR

II. SYSTEM OVERVIEW

Figure 1 shows the architecture of THREATRAPTOR, which consists of two subsystems: (1) a threat behavior extraction pipeline for automated threat knowledge extraction, and (2) a query subsystem built upon system auditing, which provides a domain-specific query language, TBQL, to hunt for threats in computer systems. In the query subsystem, monitoring agents built upon mature frameworks [24]–[26] are deployed across hosts to collect system audit logging data. The collected data is sent to the central database for storage. Given an input OSCTI report, THREATRAPTOR first extracts IOCs (e.g., file names/paths, IPs) and their relations, and constructs a threat behavior graph. THREATRAPTOR then synthesizes a TBQL query from the constructed graph, and executes the query to find the matched system auditing records. The security analyst can optionally revise the synthesized query to encode domain knowledge. In the situation where the OSCTI report is not available, THREATRAPTOR can be used as a proactive threat hunting tool with TBQL queries manually constructed.

Demo Example. Figure 2 shows an example data leakage attack case demonstrating the whole pipeline. The case was constructed based on the Cyber Kill Chain framework [32] and CVE [33], and used in our evaluation (i.e., Case *ra_2*). As we can see, the threat behavior graph clearly shows how the threat unfolds into multiple connected steps, where each step is represented by an IOC node-edge triplet. Furthermore, each edge is associated with a sequence number indicating the step order. Such sequential information is essential to uncovering the correct threat scenario and has not been considered in prior work [21], [22]. The synthesized TBQL query further encodes the threat knowledge into formal query constructs, which is more amenable to human-in-the-loop analysis and iterative exploration. Nodes and edges in the threat behavior graph are synthesized into system entities and system event patterns in the TBQL query, and sequence numbers of edges are used to construct a `with` clause that specifies the temporal order constraints of system event patterns. By default, the synthesized TBQL query specifies the default attributes of all system entities (i.e., "name" for files, "exename" for processes, and "dstip" for network connections) in the `return` clause.

Threat Model. Our threat model follows the prior work on system auditing [3]–[8]. We assume an attacker that attacks the computer system from outside: the attacker either utilizes the vulnerabilities in the system or convinces the user to download files with malicious payload. We also assume that OS kernels and kernel-layer auditing frameworks are part of our trusted computing base, and the system audit logging data collected from kernel space is not tampered. We also do not consider attacks that do not go through kernel-layer auditing (e.g., side channel attacks, memory-based attacks) and thus cannot be captured by system auditing frameworks.

Event Category	Relevant System Call
ProcessToFile	read, readv, write, writev, execve, rename
ProcessToProcess	execve, fork, clone
ProcessToNetwork	read, readv, recvfrom, recvmsg, sendto, write, writev

TABLE I: Representative system calls processed

Entity	Attributes
File	Name, Path, User, Group
Process	PID, Executable Name, User, Group, CMD
Network Connection	SRC/DST IP, SRC/DST Port, Protocol

TABLE II: Representative attributes of system entities

Operation Type	Read, Write, Execute, Start, End, Rename
Time	Start Time, End Time, Duration
Misc.	Subject ID, Object ID, Data Amount, Failure Code

TABLE III: Representative attributes of system events

III. DESIGN OF THREATRAPTOR

A. System Auditing

THREATRAPTOR leverages mature system auditing frameworks [24]–[26] to collect system-level audit logs about system calls from the OS kernel. The collected kernel audit logs consist of system events that describe the interactions among system entities, which are crucial for security analysis. As shown in previous studies [3]–[8], on mainstream operating systems, system entities in most cases are files, processes, and network connections, and the monitored system calls are mapped to three major types of system events: file access, processes creation and destruction, and network access. Hence, in THREATRAPTOR, we consider *system entities* as *files*, *processes*, and *network connections*. We consider a *system event* as the interaction between two system entities represented as ⟨subject_entity, operation, object_entity⟩, which consists of the initiator of the interaction, the type of the interaction, and the target of the interaction. Subjects are processes originating from software applications (e.g., Chrome), and objects can be files, processes, and network connections. We categorize system events into three types according to the types of their object entities: *file events*, *process events*, and *network events*.

THREATRAPTOR parses the collected audit logs of system calls (Table I) into a sequence of system events among system entities, and extracts a set of attributes that are crucial for security analysis (Tables II and III). To uniquely identify system entities, for a process entity, THREATRAPTOR uses the process executable name and PID as its unique identifier. For a file entity, ThreatRaptor uses the absolute path as its unique identifier. For a network connection entity, ThreatRaptor uses the 5-tuple ⟨srcip, srcport, dstip, dstport, protocol⟩ as its unique identifier [5]. Failing to distinguish different entities will cause problems in relating events to entities.

B. Data Storage

ThreatRaptor stores the parsed system entities and system events in databases, so that the system audit logging data can be persisted. Prior work has modeled system audit logging data as either relational tables [7] or provenance graphs [5]. Inspired by such designs, ThreatRaptor adopts two types of database models for its storage component: relational model and graph model. Relational databases come with mature indexing mechanisms and are scalable to massive data, which are suitable for queries that involve many joins and constraints. Graph databases represent data as nodes and edges, which are suitable for queries that involve graph pattern search.

Currently, ThreatRaptor adopts PostgreSQL [27] for its relational storage and Neo4j [28] for its graph storage. For PostgreSQL, ThreatRaptor stores system entities and system events in separate tables. For Neo4j, ThreatRaptor stores system entities as nodes and system events as edges. Data is replicated across the two databases, which supports the execution of different types of queries (Section III-F) and improves data availability. Indexes are created on key attributes (e.g., file name, process executable name, source/destination IP) for both databases to speed up the search.

Data Reduction. ThreatRaptor further reduces the data size before storing it in databases, so that the search can be done more efficiently while the critical information about malicious behaviors is still preserved. System audit logging data often has many excessive events between the same entity pair [23]. The reason is that OS typically finishes a read/write task (e.g., file read/write) by distributing the data proportionally to multiple system calls. Inspired by a log reduction work [23], ThreatRaptor merges the events between two entities using the following criteria: (1) two events $e_1(u_1, v_1)$ and $e_2(u_2, v_2)$ (u_1, u_2 are subjects and v_1, v_2 are objects; suppose e_1 occurs before e_2) will be merged if: $u_1 = u_2$ && $v_1 = v_2$ && $e_1.operationType = e_2.operationType$ && $0 \leq e_2.startTime - e_1.endTime \leq threshold$; (2) the attributes of the merged event e_m are updated as: $e_m.startTime = e_1.startTime$, $e_m.endTime = e_2.endTime$, $e_m.dataAmount = e_1.dataAmount + e_2.dataAmount$. We experimented with different threshold and chose 1 second, as it has reasonable reduction performance in merging system events for file manipulations, file transfers, and network communications, with no false events generated.

C. Threat Behavior Extraction

As mentioned in Section I, massive nuances exist in OSCTI text (e.g., dots, underscores in IOCs), which limit the performance of most NLP modules and existing information extraction tools [21], [22]. To address the unique challenge, ThreatRaptor employs a specialized NLP pipeline to handle nuances and accurately extract IOCs and their relations to construct a threat behavior graph. Furthermore, our pipeline is unsupervised and light-weight. Algorithm 1 gives the pipeline:

Algorithm 1: Threat Behavior Extraction Pipeline

```
   Input  : OSCTI Text: document
   Output: Threat Behavior Graph: graph
 1 Initialize all_block_trees;
 2 Initialize all_ioc_relations;
 3 for block in SegmentBlock(document) do
 4     Initialize trees;
 5     block, replacementRecord ← ProtectIoc(block);
 6     for sentence in SegmentSentence(block) do
 7         tree ← ParseDependency(sentence);
 8         Align replacementRecord with tree;
 9         tree ← RemoveIocProtection(tree, replacementRecord);
10         tree ← AnnotateTree(tree);
11         tree ← SimplifyTree(tree);
12         Add tree to trees;
13     for tree in trees do
14         tree ← ResolveCoref(tree, trees);
15     Add all tree in trees to all_block_trees;
16 all_iocs ← ScanMergeIoc(all_block_trees);
17 for tree in trees do
18     ioc_relations ← ExtractIocRelation(tree, trees, all_iocs);
19     Add ioc_relations to all_ioc_relations;
20 graph ← ConstructGraph(all_iocs, all_ioc_relations);
```

Step 1: Block Segmentation (Line 3): We segment an article into blocks, and extract IOCs and their relations from each block. Later on, when we construct the threat behavior graph, we will link the same IOCs that appear across multiple blocks.

Step 2: IOC Recognition and IOC Protection (Line 5): We construct a set of regex rules by extending an open-source IOC parser [34] (we made improvements to improve its coverage, e.g., distinguishing Linux/Windows file paths) to recognize different types of IOCs (e.g., file name, file path, IP). Furthermore, we protect the security context by replacing the IOCs with a dummy word (we use the word "something") and leave a replacement record. This makes the NLP modules designed for processing general text work well for OSCTI text.

Step 3: Sentence Segmentation (Line 6): We segment a block into sentences using a sentence segmentation component [35].

Step 4: Dependency Parsing (Line 7): We construct a dependency tree for each sentence using a dependency parsing component [35] pretrained on a large general corpus. We then use the replacement record of IOCs to restore the security context by replacing the dummy word with the original IOCs.

Step 5: Tree Annotation (Line 10): Among all nodes in the dependency trees, there are some nodes whose associated tokens are particularly useful for coreference resolution and *relation extraction* (e.g., IOCs, candidate IOC relation verbs, pronouns). We annotate these nodes of interest in the trees. We curated a list of keywords that correspond to candidate IOC relation verbs (e.g., "read", "write", "download", "open").

Step 6: Tree Simplification (Line 11): We simplify the annotated trees by removing irrelevant nodes and paths (i.e., removing the trees without any candidate IOC relation verbs or the paths without any IOC nodes). This step does not influence the extraction outcome, but helps speed up the performance.

Step 7: Coreference Resolution (Line 14): Across all trees of all sentences within a block, we resolve the coreferenced nodes for the same IOC by checking their POS tags and dependencies, and create connections between the nodes in the trees. After this step, we have a set of final annotated, simplified dependency trees for the OSCTI text.

Step 8: IOC Scan and Merge (Line 16): As the same IOC may appear across different blocks in different phrases, we scan all IOCs in the dependency trees of all blocks, and merge similar ones based on both the character-level overlap and the semantic similarity of word vectors (we used word vectors in spaCy [35]). This is different from Step 7, which performs coreference resolution within a block. After this step, we have a set of final IOCs served as nodes in the threat behavior graph.

Step 9: IOC Relation Extraction (Line 18): We present the details of our *dependency parsing-based IOC relation extraction algorithm*: (1) For each dependency tree, we enumerate all pairs of IOCs nodes; (2) Then, for each pair of IOC nodes, we check whether they satisfy the subject-object relation by considering their dependency types in the tree. In particular, we consider three parts of their dependency path: one common path from the root to the LCA (Lowest Common Ancestor); two individual paths from the LCA to each of the nodes, and construct a set of dependency type rules to do the checking; (3) Next, for the pair that passes the checking, we extract its relation verb by first scanning all the annotated candidate verbs (annotations are done in Step 5 using our curated list) in the aforementioned three parts of dependency path, and then selecting the one that is closest to the object IOC node; (4) The candidate IOC node pair and the selected verb (after lemmatization) form the final IOC entity-relation triplet. Note that for a token to be output as the final relation verb, it needs to be both covered by our keyword list and form the correct subject-verb-object relation with the IOC node pair tokens.

Step 10: Threat Behavior Graph Construction (Line 20): We iterate over all IOC entity-relation triplets sorted by the occurrence offset of the relation verb in OSCTI text, and construct a threat behavior graph. Each edge in the graph is associated with a sequence number, indicating the step order.

D. Threat Behavior Query Language (TBQL)

ThreatRaptor provides a domain-specific language, TBQL (Grammar 1), to facilitate threat hunting over system audit logging data. Compared to low-level and verbose general query languages (SQL [29], Cypher [30]), TBQL integrates a collection of critical primitives, making it easy to specify complex multi-step system behaviors for hunting.

(1) Event Pattern Syntax: The basic syntax of TBQL follows our prior work [7], which specifies one or more system event patterns in the format of ⟨subject_entity, operation, object_entity⟩, with optional filters on temporal and attribute relationships between event patterns. System entities have explicit types and identifiers, with optional filters on attributes. Essentially, the specified event patterns form a subgraph of system events to be searched. Figure 2 shows an example.

Specifically, in Grammar 1, the ⟨*patt*⟩ rule specifies an event pattern, including the subject/object entity (⟨*entity*⟩), the operation (⟨*op_exp*⟩), the pattern ID (⟨*patt_id*⟩), and the optional time window (⟨*wind*⟩). The ⟨*entity*⟩ rule specifies the entity type, the entity ID, and the optional attribute filter expression (⟨*attr_exp*⟩). Operators (e.g., logical, comparison) are supported in ⟨*op_exp*⟩ and ⟨*attr_exp*⟩ to form complex expressions (e.g., `proc p[pid = 1 && exename = "%chrome.exe%"] read || write file f`, where `%` matches any character sequence). The ⟨*wind*⟩ rule specifies a time window that narrows down the search. The ⟨*global_filter*⟩ rule specifies the global filters for all event patterns. The ⟨*rel*⟩ rule specifies the relationship between event patterns. TBQL supports two types of relationships: temporal relationship (e.g., `evt1 before[0-5 min] evt2` specifies a temporal order of events), and attribute relationship (e.g., `p1.pid = p2.pid` specifies that the two processes have the same PID). The ⟨*return*⟩ rule specifies the attributes of the matched events for return.

```
⟨tbql⟩          ::=  (⟨global_filter⟩)* (⟨patt⟩)+ ⟨rel⟩? ⟨return⟩
⟨global_filter⟩ ::=  ⟨attr_exp⟩ | ⟨wind⟩
⟨patt⟩          ::=  ⟨entity⟩ (⟨op_exp⟩ | ⟨op_path⟩) ⟨entity⟩ ⟨patt_id⟩? ⟨wind⟩?
⟨entity⟩        ::=  ⟨entity_type⟩ ⟨id⟩ (‘[’ ⟨attr_exp⟩‘]’)?
⟨entity_type⟩   ::=  ‘file’ | ‘proc’ | ‘ip’
⟨op_exp⟩        ::=  ⟨op⟩
                 |   ‘!’⟨op_exp⟩
                 |   ⟨op_exp⟩ (‘&&’ | ‘||’) ⟨op_exp⟩
                 |   ‘(’ ⟨op_exp⟩ ‘)’
⟨op⟩            ::=  ‘read’ | ‘write’ | ‘start’ | ‘execute’|...
⟨op_path⟩       ::=  (‘~>’ | ‘->’) (‘(’⟨int⟩? ‘~’? ⟨int⟩? ‘)’)? (‘[’ ⟨op_exp⟩ ‘]’)?
⟨patt_id⟩       ::=  ‘as’ ⟨id⟩ (‘[’ ⟨attr_exp⟩‘]’)?
⟨attr_exp⟩      ::=  ⟨attr⟩ ⟨bop⟩ ⟨val⟩
                 |   ‘!’? ⟨val⟩
                 |   ⟨attr⟩ ‘not’? ‘in’ ⟨val_set⟩
                 |   ⟨attr_exp⟩ (‘&&’ | ‘||’) ⟨attr_exp⟩
                 |   ‘(’ ⟨attr_exp⟩ ‘)’
⟨attr⟩          ::=  ⟨id⟩ (‘.’ ⟨id⟩)?
⟨wind⟩          ::=  ‘from’ ⟨datetime⟩ ‘to’ ⟨datetime⟩
                 |   (‘at’ | ‘before’ | ‘after’) ⟨datatime⟩
                 |   ‘last’ ⟨num⟩ ⟨time_unit⟩
⟨rel⟩           ::=  ‘with’ ⟨id⟩ (‘before’ | ‘after’ | ‘within’) (‘[’ ⟨num⟩
                     ‘-’ ⟨num⟩ ⟨time_unit⟩‘]’)? ⟨id⟩
                 |   ‘with’ ⟨attr⟩ ⟨bop⟩ ⟨attr⟩
⟨return⟩        ::=  ‘return’ ‘distinct’? ⟨attr⟩ (‘,’ ⟨attr⟩)*
```

Grammar 1: Representative BNF grammar of TBQL

In addition, TBQL provides different types of syntactic sugars to facilitate the query construction:

- *Default attributes for system entities:* default attribute names will be inferred if the user only specifies attribute values in an event pattern, or entity IDs in the `return` clause. We select the most commonly used attributes in security analysis as default attributes: "name" for files, "exename" for processes, and "dstip" for network connections.
- *Entity ID reuse:* reusing an entity ID in multiple event patterns implicitly indicates that the entities are the same.

For example, in the TBQL query in Figure 2, `proc p1["%/bin/tar%"]`, `file f1["%/etc/passwd%"]`, `ip i1["192.168.29.128"]`, and `return p1` will be inferred as `proc p1[exename = "%/bin/tar%"]`, `file f1[name = "%/etc/passwd%"]`, `ip i1[dstip = "192.168.29.128"]`, and `return p1.exename`. Besides, the entity ID `p1` is used in both `evt1` and `evt2`, indicating the same system entity.

(2) Variable-Length Event Path Pattern Syntax: In addition to the basic event pattern syntax, ThreatRaptor uniquely provides an advanced syntax that specifies various types of variable-length paths of system event patterns. The ⟨*op_path*⟩

rule gives the core syntax, which provides several alternatives:

- `proc p ~>[read] file f`: a path of arbitrary length from a process entity `p` to a file entity `f`. The operation type of the final hop (i.e., system event where `f` is an object) is `read`.
- `proc p ~>(2~4)[read] file f`: the path has a minimum length of 2 and a maximum length of 4.
- `proc p ~>(2~)[read] file f`: the path has a minimum length of 2. The maximum length is not restricted.
- `proc p ~>(~4)[read] file f`: the path has a maximum length of 4. The minimum length is 1.
- `proc p ->[read] file f`: the path has a length of 1. This is semantically equivalent to the basic event pattern syntax, e.g., `proc p read file f`. The difference lies in the execution: this length-1 event path pattern will be compiled into a Cypher data query executed on the Neo4j database, while the basic event pattern will be compiled into a SQL data query executed on the PostgreSQL database.
- `proc p ~> file f`: the operation type of the final hop is omitted, indicating that the search allows any operation type.

This syntax is particularly useful when doing query synthesis: in certain cases, an edge in the threat behavior graph (hence a threat step between two IOCs in OSCTI text) may correspond to a path of system events in system audit logs. This happens often when intermediate processes are created to chain system events, but are omitted in the OSCTI text by the human writer. With this syntax, the information flow between two system entities can be easily specified and the semantic gap between the OSCTI text and the system audit logs can be bridged. We use "TBQL pattern" to refer to both the event pattern and the variable-length event path pattern.

E. TBQL Query Synthesis

To facilitate threat hunting with OSCTI, ThreatRaptor provides a query synthesis mechanism that automatically synthesizes a TBQL query from the threat behavior graph.

Step 1: Pre-Synthesis Screening & IOC Relation Mapping: One challenge in query synthesis is the semantic gap between the types of IOCs and IOC relations, and the types of system entities and their operations. To bridge the gap, ThreatRaptor first performs a pre-synthesis screening to filter out nodes (and connected edges) in the threat behavior graph whose associated IOC types are not currently captured by the system auditing component (e.g., registry entries). Then, for each remaining edge, ThreatRaptor maps its associated IOC relation to the TBQL operation type (e.g., ⟨*op*⟩ rule in Grammar 1). We constructed a set of rules for IOC relation mapping, which consider both the semantic meaning of the IOC relation and the types of the connected IOC nodes. For example, the "download" relation between two "Filepath" IOCs will be mapped to the "write" operation in TBQL, indicating a process writes data to a file. In contrast, the "download" relation from a "Filepath" IOC to an "IP" IOC will be mapped to the "read" operation in TBQL, indicating a process reads data from a network connection. ThreatRaptor further filters out edges whose associated IOC relations do not match any rules.

Step 2: TBQL Pattern Synthesis: For each node in the threat behavior graph, ThreatRaptor synthesizes a TBQL system entity (i.e., rule ⟨*entity*⟩) and assigns a unique entity ID: (1) for a source node, ThreatRaptor synthesizes a process entity; (2) for a sink node, ThreatRaptor synthesizes a network connection entity if its associated IOC type is an IP. Otherwise, ThreatRaptor synthesizes a file entity or a process entity depending on the associated IOC relation of the edge. ThreatRaptor then synthesizes the attribute of the entity using the associated IOC content. Wildcard operators `%` are added around the attribute string by default.

ThreatRaptor synthesize a TBQL pattern (i.e., rule ⟨*patt*⟩) by connecting the synthesized TBQL subject & object entities and the mapped TBQL operation. By default, an event pattern is synthesized. System administrator can configure the system to synthesize a variable-length event path pattern.

Step 3: TBQL Pattern Relationship Synthesis: For TBQL event patterns, ThreatRaptor synthesizes their temporal relationships by following an ascending order of the sequence numbers of corresponding edges in the threat behavior graph. For variable-length event path patterns, this step is omitted since event paths in TBQL do not have temporal relationships.

Step 4: TBQL Return Synthesis: To synthesize the TBQL return clause, ThreatRaptor by default appends all entity IDs to the "return" string. Default attribute names will be inferred when the query is executed and the corresponding attribute values will be returned (i.e., TBQL syntactic sugars).

Figure 2 shows an example TBQL query synthesized using the default synthesis plan. In addition, ThreatRaptor supports user-defined synthesis plans to overwrite the default plan and synthesize attributes that are supported but not captured in the threat behavior graph (e.g., hostname, time window).

F. TBQL Query Execution

To efficiently execute a TBQL query with many TBQL patterns (could be a mix of event patterns and variable-length event path patterns), ThreatRaptor extends our prior work [7] by (1) compiling each TBQL pattern into a semantically equivalent SQL or Cypher data query, and (2) scheduling the execution of these data queries in different database backends (i.e., PostgreSQL and Neo4j) based on their estimated pruning power and semantic dependencies ([7] does not involve variable-length event path patterns and Neo4j backend). Specifically, for an event pattern, ThreatRaptor compiles it into a SQL data query, so that the mature indexing mechanism and the efficient support for joins in relational databases can be leveraged. The compiled SQL query joins two system entity tables with one system event table, and applies the filters in the WHERE clause. For a variable-length event path pattern, since it is difficult to perform graph pattern search using SQL, ThreatRaptor compiles it into a Cypher data query by leveraging Cypher's path pattern syntax [30].

We now present our *data query scheduling algorithm*: For each TBQL pattern, ThreatRaptor computes a pruning score by counting the number of constraints declared; a TBQL pattern with more constraints has a higher score. For a variable-length event path pattern, we additionally consider the length of the path when computing the score; a pattern with a smaller maximum path length has a higher score. Then, when scheduling the execution of the data queries, THREATRAPTOR considers both the pruning scores and the pattern dependencies: if two TBQL patterns have dependencies (e.g., connected by the same system entity), THREATRAPTOR will first execute the data query whose associated pattern has a higher pruning score, and then use the execution results to constrain the execution of the other data query (by adding filters). This way, complex TBQL queries with various TBQL patterns can be efficiently executed in different database backends seamlessly.

In addition to the search mode based on exact matching, THREATRAPTOR supports a *fuzzy search mode* based on inexact graph pattern matching [6]. The user can use this mode as an alternative when the exact search mode fails to retrieve meaningful results, allowing the generality of searching while at the cost of a longer execution time. Intuitively, a TBQL query specifies a subgraph of system events to be searched, and inexact graph pattern matching can be naturally leveraged in query execution to enable fuzzy search. In the current design, our fuzzy search mode leverages Poirot [6] to search for both node-level alignment and graph-level alignment: (1) For node-level alignment, we use Levenshtein distance [36] to perform similarity matching of IOC strings specified in the TBQL query and attributes of system entities stored in the database, so that typos or small changes in IOCs can still retrieve the correct system entities; (2) For graph-level alignment, we match the subgraph pattern specified in the TBQL query with the provenance graph of system events. We borrow Poirot's idea that measures the potential attacker influence by the number of compromised ancestor processes. By calculating Poirot's graph alignment scores for all candidate alignments, THREATRAPTOR's query execution engine produces an exhaustive searching result for aligned subgraphs of system events, and returns the entity attributes specified in the `return` clause as final results. Sections IV-B4 and VI include evaluation results and comparison with Poirot.

IV. EVALUATION

We built THREATRAPTOR ($\sim$ 25K LOC) upon several tools: Sysdig [26] for system auditing, PostgreSQL [27] and Neo4j [28] for system audit logging data storage, Python and spaCy [35] for threat behavior extraction, ANTLR 4 [37] for TBQL language parser, and Java for the whole system.

We deployed THREATRAPTOR on a physical testbed to collect real system audit logs and hunt for malicious activities. We evaluated THREATRAPTOR on a broad set of attack cases. In total, the audit logs used in our evaluations contain $47,688,033$ system entities and $55,840,381$ system events. We aim to answer the following research questions:

- **RQ1:** How accurate is THREATRAPTOR in extracting threat behaviors from OSCTI text compared to general information extraction approaches?
- **RQ2:** How accurate is THREATRAPTOR in finding malicious system activities using OSCTI text?
- **RQ3:** How efficient is THREATRAPTOR in extracting threat behaviors from OSCTI text, constructing a threat behavior graph, and synthesizing a TBQL query?
- **RQ4:** How efficient is THREATRAPTOR in executing TBQL queries over the big system audit logging data?
- **RQ5:** How concise is TBQL in specifying malicious system behaviors compared to general-purpose query languages?

Case ID	Case Name
tc_clearscope_1	20180406 1500 ClearScope – Phishing E-mail Link
tc_clearscope_2	20180411 1400 ClearScope – Firefox Backdoor w/ Drakon In-Memory
tc_clearscope_3	20180413 ClearScope
tc_fivedirections_1	20180409 1500 FiveDirections – Phishing E-mail w/ Excel Macro
tc_fivedirections_2	20180411 1000 FiveDirections – Firefox Backdoor w/ Drakon In-Memory
tc_fivedirections_3	20180412 1100 FiveDirections – Browser Extension w/ Drakon Dropper
tc_theia_1	20180410 1400 THEIA – Firefox Backdoor w/ Drakon In-Memory
tc_theia_2	20180410 1300 THEIA - Phishing Email w/ Link
tc_theia_3	20180412 THEIA - Browser Extension w/ Drakon Dropper
tc_theia_4	20180413 1400 THEIA - Phishing E-mail w/ Executable Attachment
tc_trace_1	20180410 1000 TRACE – Firefox Backdoor w/ Drakon In-Memory
tc_trace_2	20180410 1200 TRACE – Phishing E-mail Link
tc_trace_3	20180412 1300 TRACE – Browser Extension w/ Drakon Dropper
tc_trace_4	20180413 1200 TRACE – Pine Backdoor w/ Drakon Dropper
tc_trace_5	20180413 1400 TRACE – Phishing E-mail w/ Executable Attachment
password_crack	Password Cracking After Shellshock Penetration
data_leak	Data Leakage After Shellshock Penetration
vpnfilter	VPNFilter

TABLE IV: 18 attack cases in our evaluation benchmark

RQ1 aims evaluate the accuracy of THREATRAPTOR in threat behavior extraction. RQ2 aims to evaluate the end-to-end accuracy of THREATRAPTOR in threat hunting using OSCTI. RQ3 aims to evaluate the efficiency of THREATRAPTOR in threat behavior extraction, threat behavior graph construction, and TBQL query synthesis. RQ4 aims to evaluate the efficiency of THREATRAPTOR in TBQL query execution, and measure the performance speedup achieved by the TBQL query scheduler. RQ5 aims to evaluate the conciseness of TBQL in expressing complex system behaviors.

A. Evaluation Setup

The deployed server has an Intel(R) Xeon(R) CPU E5-2637 v4 (3.50GHz), 256GB RAM running 64bit Ubuntu 18.04.1. The server is frequently used by > 15 active users to perform various daily tasks, including file manipulation, text editing, and software development. To evaluate THREATRAPTOR, we constructed an evaluation benchmark of 18 attack cases from two sources: 15 cases released in the DARPA TC dataset [38], and 3 multi-step intrusive attacks that we performed ourselves on the testbed based on the Cyber Kill Chain framework [32] and CVE [33]. When we perform the attacks and conduct the evaluations, the sever continues to serve other users. This setup ensures that enough noise of benign background traffic is collected in together with malicious activities, representing the real-world deployment. Furthermore, benign activities significantly outnumber attack activities (55 million vs. thousands), demonstrating the challenge in threat hunting. Table IV shows the list of cases. The total monitoring length is 41 days for DARPA TC cases and 16 hours for our three attacks.

1) DARPA TC Attack Cases: We selected 15 cases from the DARPA TC Engagement 3 data release [38], which cover various combinations of OSs (e.g., Linux, Windows, Android), vulnerabilities (e.g., Nginx backdoor, Firefox backdoor, browser extension), and exploits (e.g., Drakon APT, micro APT, phishing email with malicious Excel attachment).

Specifically, the dataset consists of the captured audit logs of six performer systems (ClearScope, FiveDirections, THEIA, TRACE, CADETS, and TA5.2) under the penetration of the red team using different attack strategies, which include both benign and malicious system activities. The dataset also includes a ground-truth report that has attack descriptions for the cases. After examining the descriptions and the logs, we found that the logs for TA5.2 are missing in the released dataset and the logs for CADETS lack key attributes (e.g., file name). This makes us unable to confirm the attack ground truth to conduct faithful evaluations. Thus, we do not consider these cases in our evaluations. Nevertheless, similar attacks were already performed for other performer systems and their descriptions and logs are covered in our evaluations. For the other four performer systems, we selected all the 15 attack cases in our evaluation benchmark. For each case, we parsed the provided audit logs and loaded the data in THREATRAPTOR's databases. We then extracted the attack description text from the ground-truth report and use it as input to THREATRAPTOR.

2) Multi-Step Intrusive Attack Cases: To increase the coverage of our benchmark, we constructed 3 multi-step intrusive attack cases, based on CVE [33] and capture the important traits of attacks depicted in the Cyber Kill Chain framework [32] (e.g., including the stages of initial penetration, data exfiltration). We performed these attacks on the testbed and collected system audit logs. The attack description texts were constructed according to the way the attacks were performed.

Attack 1: Password Cracking After Shellshock Penetration. The attacker penetrates into the victim host (i.e., the testbed) by exploiting the Shellshock vulnerability [39]. After penetration, the attacker first connects to cloud services (Dropbox) and downloads an image where C2 (Command & Control) server's IP address is encoded in the EXIF metadata. This behavior is a common practice shared by APT attacks [40] to evade DNS blacklisting based detection systems. Using the IP, the attacker downloads a password cracker from the C2 server to the victim host, and then runs the password cracker against password shadow files to extract clear text.

Attack 2: Data Leakage After Shellshock Penetration. After the reconnaissance, the attacker attempts to steal all the valuable assets from the victim host. This stage mainly involves the behaviors of local and remote file system scanning activities, copying and compressing of important files, and transferring the files to the C2 server. The attacker scans the file system, scrapes files into a single compressed file, and transfers it back to the C2 server.

Attack 3: VPNFilter. The attacker seeks to maintain a direct connection to the victim host from the C2 server. The attacker utilizes the notorious VPNFilter malware [41] which infected millions of IoT devices by exploiting a number of known or zero-day vulnerabilities. After the initial penetration on the victim host, the attacker downloads the VPNFilter stage 1 malware from the C2 server, which accesses a public image repository to get an image. In the EXIF metadata of the image, the IP address for the stage 2 server is encoded. The stage 1 malware then downloads the VPNFilter stage 2 malware from the stage 2 server, and executes it to launch the VPNFilter attack, which establishes a direct connection to the C2 server.

B. Evaluation Results

1) RQ1: Accuracy of Threat Behavior Extraction: To evaluate the accuracy of THREATRAPTOR in extracting threat behaviors from OSCTI text, we labeled the OSCTI texts based on the ground truth and measure the precision, recall, and F1 of the extracted IOC entities and IOC relations. We compare THREATRAPTOR with two state-of-the-art open information extraction approaches for entity and relation extraction from general text: Stanford Open IE [21] and Open IE 5 [22]. Furthermore, we are interested in studying the effect of IOC Protection on the accuracy of IOC entity and relation extraction. Thus, we also compare THREATRAPTOR with the version of THREATRAPTOR without IOC Protection, Stanford Open IE with IOC Protection, and Open IE 5 with IOC Protection.

Table V shows the precision, recall, and F1 score aggregated over all evaluation cases. We have the following observations: (1) THREATRAPTOR achieves the highest precision, recall, and F1 score for both IOC entity extraction and IOC relation extraction. In particular, THREATRAPTOR has 96.64% F1 for IOC entity extraction and 92.59% F1 for IOC relation extraction. In contrast, the scores of Stanford Open IE and Open IE 5 are very low: $< 5\%$ F1 for IOC entitiy extraction and 0% F1 for IOC relation extraction. These results demonstrate the effectiveness of THREATRAPTOR's specialized threat behavior extraction NLP pipeline in processing OSCTI text; (2) When removing IOC Protection, the scores of THREATRAPTOR drop significantly (59.26% F1 for IOC entity extraction and 16.39% F1 for IOC relation extraction). The reason for the accuracy drop is that if the OSCTI text is processed directly by an NLP component without first applying IOC Protection, the sentence segmentation component and the tokenizer will break the IOC entities (e.g., file paths, process executable names, IPs) into pieces, making it impossible to annotate the pieces and analyzing the correct grammatical structure of sentences. This demonstrates the effectiveness of IOC Protection in protecting the security context in OSCTI text; (3) When adding IOC Protection, the scores of Stanford Open IE and Open IE 5 increase a bit, but not much. This again demonstrates the effectiveness of IOC Protection in improving the accuracy of other NLP components. Though, as these approaches target general information extraction instead of threat behavior extraction from OSCTI text, their performance is limited.

2) RQ2: Accuracy of Threat Hunting: To measure the end-to-end accuracy of THREATRAPTOR in threat hunting, for each attack case, we compare the system events found by the event patterns in the synthesized TBQL query, and the ground-truth system events that are related to the attack. Table VI shows the precision and recall. We have the following observations: (1) THREATRAPTOR is able to accurately find malicious system events using OSCTI texts, achieving 100% precision, 96.74% recall, and 98.34% F1. This is largely due to the high accuracy achieved by THREATRAPTOR's threat behavior extraction pipeline; (2) Though some excessive event patterns may be

Approaches	Entity Precision	Entity Recall	Entity F1	Relation Precision	Relation Recall	Relation F1
THREATRAPTOR	96.00%	97.30%	**96.64%**	96.15%	89.29%	**92.59%**
THREATRAPTOR- IOC Protection	94.12%	43.24%	59.26%	100.00%	8.93%	16.39%
Stanford Open IE	1.82%	14.86%	3.24%	0.00%	0.00%	0.00%
Stanford Open IE + IOC Protection	4.39%	36.49%	7.84%	0.88%	8.93%	1.59%
Open IE 5	0.25%	1.35%	0.43%	0.00%	0.00%	0.00%
Open IE 5 + IOC Protection	3.49%	20.27%	5.95%	0.00%	0.00%	0.00%

TABLE V: Precision, recall, and F1 of IOC entity extraction and IOC relation extraction of THREATRAPTOR and baseline approaches. The results are aggregated overall all 18 cases.

Case	Precision TP/(TP+FP)	Recall TP/(TP+FN)
tc_clearscope_1	6/6	6/6
tc_clearscope_2	3/3	3/3
tc_clearscope_3	1/1	1/1
tc_fivedirections_1	51/51	51/51
tc_fivedirections_2	3/3	3/3
tc_fivedirections_3	0/0	0/3
tc_theia_1	3/3	3/3
tc_theia_2	115/115	115/115
tc_theia_3	3/3	3/3
tc_theia_4	421/421	421/421
tc_trace_1	39/39	39/76
tc_trace_2	7/7	7/7
tc_trace_3	0/0	0/2
tc_trace_4	1/1	1/3
tc_trace_5	578/578	578/578
password_crack	10/10	10/12
data_leak	6/6	6/8
vpnfilter	178/178	178/178
Total	1425/1425 = 100.00%	1425/1473 = 96.74%

TABLE VI: Precision and recall of THREATRAPTOR in finding malicious system events

occasionally synthesized (e.g., in *password_crack*, one excessive event pattern is synthesized: `proc p3["%/tmp/libfoo.so%"] write file f2["%/tmp/john.zip%"] as evt5`), the design of THREATRAPTOR ensures that these excessive event patterns will rarely retrieve benign activities. The reason is because these excessive patterns have IOCs as subject/object constraints, which are extracted by a set of highly-precise regex rules in THREATRAPTOR. As a result, very few benign activities are falsely retrieved (e.g., 0 false positive rate in our evaluation benchmark); (3) For queries that have false negatives, the primary reason is due to the semantic ambiguity in query synthesis for certain IOC relations. For example, in *tc_trace_1*, there is an edge pointing from the "Filepath" IOC "/home/admin/cache" to itself with the "run" relation. Both the IOC and the relation are corrected extracted from OSCTI text. However, when performing query synthesis, there is no way to differentiate whether it represents a file event `proc p1["%/home/admin/cache%"] execute file f1["%/home/admin/cache%"]` or a process event `proc p1["%/home/admin/cache%"] start proc p2["%/home/admin/cache%"]`, as both events are related to process creation. The default synthesis plan in THREATRAPTOR synthesizes the first pattern, while for this case, the second pattern has matched ground-truth system events. As a result, 37 system events are missed. One way to mitigate this is to let the security analyst revise the query to improve the coverage, and the synthesized event patterns serve as a base for exploration.

It is worth mentioning that the three cases for ClearScope were conducted on Android OS and the ground-truth system events have Android package names as process executable names (e.g., `proc p1["%com.android.defcontainer%"] open file f1["%MsgApp-instr.apk%"]`), which are different from other cases in which process executables are normal Linux/Windows files. Nevertheless, THREATRAPTOR is able to accurately extract such information and use the information to find the malicious system events, thanks to the coverage of a wide range of IOC types and IOC relations in THREATRAPTOR's threat behavior extraction pipeline.

3) RQ3: Efficiency of Threat Behavior Extraction: Table VII shows the execution time of different stages of THREATRAPTOR: threat behavior extraction, threat behavior graph construction, and query synthesis. For threat behavior extraction, we also compare with other baseline approaches. We have the following observations: (1) THREATRAPTOR is efficient in processing the input OSCTI texts, constructing threat behavior graphs, and synthesizing TBQL queries. The average time for the three stages is 0.52s; (2) Stanford Open IE and Open IE 5 are more expensive in extracting threat behaviors compared to THREATRAPTOR (0.76s and 13.46s vs. 0.42s), since these general information extraction approaches spend a long time analyzing texts that are unrelated to threat behaviors; (3) IOC Protection adds trivial overhead.

4) RQ4: Efficiency of TBQL Query Execution: We measure the runtime performance of THREATRAPTOR in executing TBQL queries, particularly the performance speedup provided by the TBQL query scheduler in different database backends. To prepare for evaluation, for each case, we construct four types of semantically equivalent queries according to the corresponding synthesized TBQL query by THREATRAPTOR:

(a) TBQL query using the event pattern syntax (e.g., `proc p open file f as evt`).
(b) SQL query that encodes all event patterns and filters in the FROM and WHERE clauses.
(c) TBQL query using the length-1 event path pattern syntax (e.g., `proc p ->[open] file f as evt`).
(d) Cypher query that encodes all length-1 event path patterns and filters in the MATCH and WHERE clauses.

All these four types queries search for the same system behaviors and return the same results. The difference lies in the query scheduler and the database: Queries (a) and (b) are executed in PostgreSQL, and Queries (c) and (d) are executed in Neo4j. Queries (a) and (c) benefit from optimizations in TBQL query scheduler, and Queries (b) and (d) do not.

Table VIII shows the execution time of the queries aggregated over 20 rounds. We have the following observations: (1) TBQL query scheduler provided by THREATRAPTOR is generally more efficient than the query schedulers provided by PostgreSQL and Neo4j. Specifically, for PostgreSQL backend, THREATRAPTOR is $3810.17/168.18 = 22.7$x faster; for Neo4j backend, THREATRAPTOR is $3104.68/343.04 = 9.1$x faster; (2) There also exist a few cases in which TBQL queries run slightly slower than SQL queries and Cypher queries. Particularly, when the TBQL query only contains 1 pattern (i.e., *tc_clearscope_3*, *tc_trace_3*), TBQL query runs slower

Case	ThreatRaptor			ThreatRaptor − IOC Protection	Stanford Open IE	Stanford Open IE + IOC Protection	Open IE 5	Open IE 5 + IOC Protection
	Text -> E. & R.	E. & R. -> Graph	Graph -> TBQL	Text -> E. & R.	Text -> E. & R.	Text -> E. & R.	Text -> E. & R.	Text -> E. & R.
tc_clearscope_1	0.43	0.08	0.00	0.42	0.74	0.74	7.69	7.69
tc_clearscope_2	0.42	0.08	0.00	0.39	0.46	0.46	1.53	1.53
tc_clearscope_3	0.37	0.02	0.00	0.24	0.76	0.76	15.29	15.29
tc_fivedirections_1	0.33	0.08	0.01	0.35	0.61	0.61	7.84	7.84
tc_fivedirections_2	0.39	0.08	0.01	0.38	0.56	0.56	0.67	0.67
tc_fivedirections_3	0.36	0.08	0.00	0.35	0.63	0.63	0.50	0.50
tc_theia_1	0.52	0.09	0.01	0.46	0.59	0.59	16.53	16.53
tc_theia_2	0.52	0.08	0.01	0.47	0.31	0.31	48.85	48.85
tc_theia_3	0.56	0.09	0.01	0.51	0.72	0.72	7.79	7.79
tc_theia_4	0.26	0.09	0.01	0.27	0.73	0.73	0.19	0.19
tc_trace_1	0.44	0.11	0.01	0.48	1.06	1.06	4.01	4.01
tc_trace_2	0.45	0.16	0.01	0.44	0.63	0.63	47.13	47.13
tc_trace_3	0.31	0.09	0.00	0.31	1.10	1.10	1.00	1.00
tc_trace_4	0.46	0.09	0.00	0.46	0.96	0.96	1.76	1.76
tc_trace_5	0.42	0.08	0.00	0.43	0.91	0.91	41.92	41.92
password_crack	0.43	0.09	0.01	0.44	0.87	0.87	25.54	25.54
data_leak	0.47	0.09	0.01	0.51	0.84	0.84	10.06	10.07
vpnfilter	0.35	0.08	0.01	0.30	1.15	1.16	4.05	4.05
Total	7.50	1.55	0.11	7.21	13.63	13.64	242.35	242.36
Average	0.42	0.09	0.01	0.40	0.76	0.76	13.46	13.46

TABLE VII: Execution time (second) of different stages of ThreatRaptor: threat behavior extraction (text -> E. & R.), threat behavior graph construction (E. & R. -> graph), and TBQL query synthesis (graph -> TBQL)

Case	TBQL		SQL		TBQL (length-1 path)		Cypher	
	20-r mean	20-r std	20-r mean	20-r std	20-r mean	20-r std	20-r mean	20-r std
tc_clearscope_1	1.07	0.14	1.41	0.54	3.86	0.21	3.91	0.19
tc_clearscope_2	1.39	0.16	1.34	0.12	4.14	0.33	3.93	0.20
tc_clearscope_3	0.92	0.15	0.90	0.11	3.51	0.24	3.47	0.17
tc_fivedirections_1	2.48	0.04	32.24	0.48	5.38	0.38	44.79	1.01
tc_fivedirections_2	1.79	0.16	1.94	0.12	4.52	0.31	4.50	0.36
tc_fivedirections_3	1.46	0.14	1.87	0.17	3.89	0.28	5.40	0.36
tc_theia_1	3.86	0.08	43.15	0.55	10.41	0.38	234.31	5.31
tc_theia_2	1.91	0.17	1.88	0.17	5.17	0.41	77.66	1.11
tc_theia_3	4.43	0.40	12.07	0.34	9.84	0.35	32.66	0.59
tc_theia_4	4.37	0.15	5.28	0.29	8.54	0.32	8.09	0.43
tc_trace_1	44.21	0.57	85.63	1.91	82.16	0.81	366.11	18.74
tc_trace_2	50.66	0.77	52.94	0.42	85.90	0.66	348.23	1.94
tc_trace_3	1.97	0.05	1.95	0.06	2.25	0.01	2.24	0.01
tc_trace_4	37.66	0.37	38.88	0.26	89.39	0.85	91.33	0.53
tc_trace_5	5.43	0.10	13.79	0.37	6.46	0.12	6.19	0.11
password_crack	1.52	0.20	40.32	0.45	6.06	0.34	57.70	1.02
data_leak	1.45	0.43	3,456.12	67.43	6.28	0.67	1,803.14	34.92
vpnfilter	1.60	0.29	18.44	0.28	5.28	0.42	11.04	0.61
Total	168.18		3,810.17		343.04		3,104.68	
Average	9.34		211.68		19.06		172.48	

TABLE VIII: Execution time (second) of queries in TBQL, SQL, TBQL in length-1 event path pattern syntax, and Cypher. Each query was executed for 20 rounds.

Case	ThreatRaptor-Fuzzy				Poirot			
	Loading	Preprocessing	Searching	Total	Loading	Preprocessing	Searching	Total
tc_clearscope_1	7.71	6.03	577.01	590.75	7.94	6.08	542.59	556.61
tc_clearscope_2	7.69	6.32	>3600	>3600	7.74	6.20	431.22	445.17
tc_clearscope_3	7.69	6.02	21.30	35.01	7.75	5.86	22.21	35.81
tc_fivedirections_1	18.11	12.99	58.33	89.42	18.15	12.80	57.83	88.78
tc_fivedirections_2	18.05	12.52	705.13	735.69	18.15	12.36	663.13	693.64
tc_fivedirections_3	18.09	12.37	33.28	63.74	18.08	12.08	32.64	62.80
tc_theia_1	37.25	34.10	>3600	>3600	37.35	33.25	>3600	>3600
tc_theia_2	37.36	33.06	>3600	>3600	37.36	31.68	>3600	>3600
tc_theia_3	36.98	35.15	>3600	>3600	37.34	34.23	>3600	>3600
tc_theia_4	37.02	33.10	>3600	>3600	37.31	32.65	>3600	>3600
tc_trace_1	297.35	510.99	249.71	1,058.05	296.65	489.12	109.20	894.97
tc_trace_2	295.01	504.08	>3600	>3600	297.02	483.31	>3600	>3600
tc_trace_3	295.30	340.92	111.55	747.77	297.22	328.44	114.32	739.97
tc_trace_4	294.23	412.82	301.30	1,008.34	297.43	401.04	304.09	1,002.56
tc_trace_5	294.53	401.28	>3600	>3600	297.24	389.22	>3600	>3600
password_crack	1.36	1.12	37.97	40.45	1.37	1.54	22.02	24.93
data_leak	2.37	1.75	19.01	23.13	2.36	1.72	18.46	22.55
vpnfilter	2.07	1.83	>3600	>3600	2.07	1.83	2,069.69	2,073.59

TABLE IX: Execution time (second) of ThreatRaptor's fuzzy search mode and Poirot [6].

than SQL query and Cypher query as additional time is taken to parse the TBQL query and compile into SQL or Cypher data queries. When the number of patterns becomes large, SQL queries and Cypher queries become much slower (e.g., *data_leak*), as these giant queries have many joins and constraints mixed together, which may suffer from indeterministic optimizations and take long to finish execution; (3) PostgreSQL is generally faster than Neo4j, as relational databases have better support for joins; (4) The standard deviation is small compared to the mean. This indicates that the 20-round mean values are representative. These results demonstrates the superiority of TBQL query scheduler in speeding up the execution of TBQL queries in different database backends.

Performance of ThreatRaptor*'s Fuzzy Search Mode:* We further study the performance of ThreatRaptor's fuzzy search mode and Poirot [6] (the difference is that Poirot does not search for all aligned system provenance subgraphs). Table IX shows the execution time (second). The execution consists of three parts: loading all system entities and system events from database into memory (loading time), constructing the provenance graph from system entities and system events (preprocessing time), and searching for alignments in the provenance graph (searching time).

We have the following observations: (1) ThreatRaptor's fuzzy search mode (ThreatRaptor-Fuzzy) based on Poirot improves generality at the cost of efficiency. In particular, ThreatRaptor's exact search mode is a lot faster than ThreatRaptor-Fuzzy and Poirot. Besides, given that ThreatRaptor-Fuzzy additionally performs an exhaustive search, it is within our expectation that ThreatRaptor-Fuzzy in general runs longer than Poirot; (2) There are several cases that both Poirot and ThreatRaptor-Fuzzy cannot finish within 1 hour (e.g., tc_theia cases). After profiling the execution, we identify two major bottlenecks in Poirot's searching iterations that affect both approaches: (a) a large number of candidate node alignments often result in a longer running time; (b) graph traversals required by candidate selection on dense graphs are time-consuming. For example, although the provenance graph of tc_theia_1 (1.5M nodes, 8.7M edges) is a lot smaller than tc_trace_1 (44.7M nodes, 39.8M edges), tc_theia_1 (78K alignments) has more candidate node alignments than tc_trace_1 (131 alignments). Furthermore, the average degree of tc_theia is 5.9 while the average degree of tc_trace is only 0.9. As a result, searching for aligned subgraphs for tc_theia cases is more time-consuming; (3) For cases that ThreatRaptor-Fuzzy finishes within 1 hour, all ground-truth attack activities are found.

Based on the evaluation results, we recommend the user to use ThreatRaptor's exact search mode if possible, which is much more efficient. The fuzzy search mode can be used as an alternative when the exact search mode fails to retrieve meaningful results. How to enable more efficient inexact graph pattern matching (and exhaustive graph alignment search) in dense, high-alignment system provenance graphs is an open question, which we leave for future work.

5) RQ5: Conciseness of TBQL: For the four types of queries mentioned in RQ4, we further compare their conciseness by measuring the number of characters (excluding spaces and comments) and words. Table X shows the results. We observe that: (1) TBQL is more concise than SQL and Cypher for all cases. Specifically, for # characters, TBQL is $15007/4460 = 3.4$x more concise than SQL and $13601/4772 = 2.9$x more concise than Cypher; for # words, TBQL is $2670/945 = 2.8$x more concise than SQL

Case	# Patterns	TBQL		SQL		TBQL (length-1 path)		Cypher	
		# Chars	# Words	# Chars	# Words	# Chars	# Words	# Chars	# Words
tc_clearscope_1	2	131	30	584	102	145	32	530	84
tc_clearscope_2	2	213	36	588	102	224	37	556	83
tc_clearscope_3	1	93	16	218	40	97	16	205	30
tc_fivedirections_1	3	219	47	737	134	234	48	674	105
tc_fivedirections_2	3	164	39	750	133	182	41	665	105
tc_fivedirections_3	2	111	25	376	70	125	26	330	51
tc_theia_1	3	238	51	711	130	253	52	648	101
tc_theia_2	3	235	51	888	152	253	53	803	124
tc_theia_3	5	329	77	1385	252	361	81	1261	206
tc_theia_4	2	195	39	562	102	206	40	530	83
tc_trace_1	4	249	55	927	169	268	56	833	130
tc_trace_2	3	264	57	933	168	282	59	882	140
tc_trace_3	1	109	19	240	42	116	20	222	33
tc_trace_4	3	187	40	656	120	202	41	599	91
tc_trace_5	2	211	40	585	103	222	41	553	84
password_crack	8	551	122	1866	311	586	123	1639	252
data_leak	8	561	121	1841	328	596	122	1614	249
vpnfilter	5	400	80	1160	212	420	80	1057	162
Total	60	4460	945	15007	2670	4772	968	13601	2113
Average	3.33	247.78	52.50	833.72	148.33	265.11	53.78	755.61	117.39

TABLE X: Conciseness of queries in TBQL, SQL, TBQL in length-1 event path pattern syntax, and Cypher

and $2113/968 = 2.2$x more concise than Cypher. This is because TBQL directly models the high-level, domain-specific concepts like system entities and system events, instead of the low-level concepts like tables or nodes/relationships; (2) The conciseness saving of TBQL compared to SQL and Cypher increases when more patterns are declared (e.g., *password_crack*, *data_leak*); (3) Cypher queries are generally more concise than SQL queries. This is within our expectation as Cypher has a concise syntax to specify linked nodes and relationships, while SQL models everything as tables and has to explicitly specify table joins to represent system events.

V. Discussion

Limitations. As mentioned in Section II, attacks on OS kernels, system auditing frameworks, and databases, and attacks that are not captured by system auditing are not considered by ThreatRaptor. Besides, ThreatRaptor's threat behavior extraction pipeline is not applicable if the OSCTI text for the attack is not available or contains little useful information (e.g., no IOCs, no sentence structures that contain IOC relations). When there are deviations between the OSCTI text and the ground truth (e.g., typos or changes in IOCs), ThreatRaptor's exact search mode may miss attack activities. In such cases, ThreatRaptor's fuzzy search mode can be used as an alternative to increase the generality of searching. Once some attack activities are found, the user can switch back to the exact search mode and revise the query (e.g., adding event patterns) to expand the search for connected activities.

CTI Collection. In general, CTI reports can be collected from various public sources (a.k.a., OSCTI), such as security websites [18], [19] and blogs [20]. Enterprises may also have access to proprietary sources such as internal reports provided by domain experts, which might better reflect the particular enterprise environment. This work does not target CTI collection. Instead, ThreatRaptor targets automated extraction of threat knowledge from an input report and use of the extracted knowledge for threat hunting.

Design Alternatives. ThreatRaptor currently leverages regex rules to extract IOCs and dependency parsing to extract IOC relations. Besides IOCs, other types of entities may also exist in OSCTI text that constitute threat behaviors, such as threat actors (e.g., CozyDuke [42]) and security tools (e.g., Mimikatz [43]), which are hard to extract using fixed regex rules. To extend the support for these entities and their relations, one approach is to adopt learning-based approaches to perform Named Entity Recognition (NER) [44] and Relation Extraction (RE) [45]. Different from ThreatRaptor's current unsupervised NLP pipeline, these approaches are typically supervised, which require large annotated corpora for model training. Such large annotated corpora is very costly to obtain manually. In future work, we plan to explore techniques to programmatically synthesize annotations for OSCTI texts (e.g., via data programming [46]) and leverage learning-based approaches to expand our threat behavior extraction scope.

In query synthesis, ThreatRaptor has a pre-synthesis screening step to filter out nodes in the threat behavior graph whose associated IOC types are not captured by the system auditing component. In future work, we plan to expand our monitoring scope by including more types of entities and events (e.g., Windows registry entries and Linux pipes).

VI. Related Work

Forensic Analysis via System Audit Logs. Research has proposed to leverage system audit logs for forensic analysis. Causality analysis plays a critical role in identifying root causes and ramifications of attacks [3]. Efforts have been made to mitigate the dependency explosion problem by performing fine-grained causality analysis [4], prioritizing dependencies [5], and reducing data size [23]. Besides, research has proposed to query system audit logs for attack investigation and anomaly detection [7]–[9], [47], [48]. The scope of ThreatRaptor is different from these works, as none of these works proposed to facilitate threat hunting via automated extraction of threat knowledge from OSCTI text and automated synthesis of threat hunting queries from the extracted knowledge. Besides, the TBQL query language provided in ThreatRaptor has a set of features particularly designed for threat hunting (e.g., variable-length event path pattern syntax) that are not supported in prior query tools.

Poirot [6] is an approach for threat hunting that finds the aligned system provenance subgraph of an input query graph. Its core contribution is an inexact graph pattern matching algorithm for finding the alignment. It is important to note that Poirot's scope is significantly different from ThreatRaptor's scope: Poirot does not search for all aligned subgraphs. Instead, Poirot stops its searching iteration after finding the first acceptable alignment that surpasses a threshold. This is different from the goal of ThreatRaptor's query subsystem. Besides, unlike ThreatRaptor's automated threat behavior graph construction, Poirot's query graph requires non-trivial efforts of cyber analysts to manually construct it. Furthermore, unlike ThreatRaptor, Poirot does not involve database storage for storing the massive log data, a query language for proactive threat hunting, and a query synthesis mechanism for automating the process. Nevertheless, Poirot's inexact graph pattern matching algorithm can be leveraged to improve the generality of searching: ThreatRaptor's current fuzzy search mode extends it to support exhaustive search.

OSCTI Analysis and Management. Research progress has been made for automated OSCTI analysis, including extracting IOCs [17], extracting threat action terms from semi-structured Symantec reports [49], and measuring information inconsistency [50]. There also exist platforms and standards for OSCTI management and exchange [14]–[16], [51], [52]. ThreatRaptor distinguishes from these works in the sense that it seeks to extract both IOCs and IOC relations from OSCTI text, and use the extracted knowledge for threat hunting.

Open Information Extraction. Information extraction (IE) extracts structured information from unstructured natural language text. Open information extraction (Open IE) is a new paradigm of IE that is not limited to a restricted set of target relations known in advance, but rather extracts all types of relations found in the text. Research has proposed to leverage rule-based approaches or learning-based approaches for more accurate Open IE [21], [22]. ThreatRaptor distinguishes from these works in the sense that it focuses on threat behavior extraction from OSCTI text, which requires special designs to handle massive nuances particular to the security domain.

VII. Conclusion

We have proposed ThreatRaptor, a system that facilitates cyber threat hunting in computer systems using OSCTI.

Acknowledgement. Peng Gao, Xiaoyuan Liu, and Dawn Song are supported in part by DARPA N66001-15-C-4066 and the Center for Long-Term Cybersecurity. Fei Shao and Xusheng Xiao are supported in part by NSF CNS-2028748. Zheng Qin and Fengyuan Xu are supported in part by NSFC-61872180, Jiangsu "Shuang-Chuang" Program, and Jiangsu "Six-Talent-Peaks" Program. Prateek Mittal is supported in part by NSF CNS-1553437 and CNS-1704105, the ARL's Army Artificial Intelligence Innovation Institute (A2I2), the Office of Naval Research Young Investigator Award, and the Army Research Office Young Investigator Prize. Sanjeev R. Kulkarni is supported in part by the Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370.

References

[1] "Target Data Breach Incident," http://www.nytimes.com/2014/02/27/business/target-reports-on-fourth-quarter-earnings.html?_r=1.

[2] "The Equifax Data Breach," https://www.ftc.gov/equifax-data-breach.

[3] S. T. King and P. M. Chen, "Backtracking intrusions," in *SOSP*, 2003.

[4] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition." in *NDSS*, 2013.

[5] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, "Towards a timely causality analysis for enterprise security," in *NDSS*, 2018.

[6] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting," in *CCS*, 2019.

[7] P. Gao, X. Xiao, Z. Li, F. Xu, S. R. Kulkarni, and P. Mittal, "AIQL: Enabling efficient attack investigation from system monitoring data," in *USENIX ATC*, 2018.

[8] P. Gao, X. Xiao, D. Li, Z. Li, K. Jee, Z. Wu, C. H. Kim, S. R. Kulkarni, and P. Mittal, "SAQL: A stream-based query system for real-time abnormal system behavior detection," in *USENIX Security*, 2018.

[9] T. Pasquier, X. Han, T. Moyer, A. Bates, O. Hermant, D. Eyers, J. Bacon, and M. Seltzer, "Runtime Analysis of Whole-system Provenance," in *CCS*, 2018.

[10] "Splunk Search Processing Language," https://www.splunk.com/en_us/resources/search-processing-language.html.

[11] "Elastic SIEM," https://www.elastic.co/siem.

[12] "Open Source Threat Intelligence Feeds," https://www.senki.org/operators-security-toolkit/open-source-threat-intelligence-feeds/.

[13] "PhishTank," https://www.phishtank.com/.

[14] "Structured Threat Information eXpression," http://stixproject.github.io/.

[15] "MISP - Open Source Threat Intelligence Platform & Open Standards For Threat Information Sharing," https://www.misp-project.org/.

[16] "The History of OpenIOC," https://www.fireeye.com/blog/threat-research/2013/09/history-openioc.html.

[17] X. Liao, K. Yuan, X. Wang, Z. Li, L. Xing, and R. Beyah, "Acing the ioc game: Toward automatic discovery and analysis of open-source cyber threat intelligence," in *CCS*, 2016.

[18] "AlienVault," https://www.alienvault.com/blogs/labs-research/.

[19] "SecureList," https://securelist.com/.

[20] "KrebsonSecurity," https://krebsonsecurity.com/.

[21] G. Angeli, M. J. J. Premkumar, and C. D. Manning, "Leveraging linguistic structure for open domain information extraction," in *ACL*, 2015.

[22] "Open IE 5," https://github.com/dair-iitd/OpenIE-standalone.

[23] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High fidelity data reduction for big data security dependency analyses," in *CCS*, 2016.

[24] "The Linux Audit Framework," https://github.com/linux-audit/.

[25] "Event Tracing for Windows," https://docs.microsoft.com/en-us/windows/win32/etw/event-tracing-portal.

[26] "Sysdig," http://www.sysdig.org/.

[27] "PostgreSQL," http://www.postgresql.org/.

[28] "Neo4j," http://neo4j.com/.

[29] "SQL: Structured Query Language," http://www.iso.org/iso/catalogue_detail.htm?csnumber=45498.

[30] "Cypher Query Language," http://neo4j.com/developer/cypher/.

[31] "ThreatRaptor Demo Video," https://youtu.be/SrcTDQwRF_M.

[32] "Cyber Kill Chain," https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html.

[33] "Common Vulnerabilities and Exposures," https://cve.mitre.org/.

[34] "ioc-parser," https://github.com/armbues/ioc_parser.

[35] "spaCy," https://spacy.io/usage/linguistic-features.

[36] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, 1966.

[37] "ANTLR," http://www.antlr.org/.

[38] "Transparent computing engagement 3 data release," https://github.com/darpa-i2o/Transparent-Computing/blob/master/README-E3.md.

[39] "CVE-2014-6271," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271.

[40] "VPNFilter: New Router Malware with Destructive Capabilities," https://symantec-enterprise-blogs.security.com/blogs/threat-intelligence/vpnfilter-iot-malware.

[41] "Router Vulnerability and the VPNFilter Botnet," https://www.schneier.com/blog/archives/2018/06/router_vulnerab.html.

[42] "APT29," https://attack.mitre.org/groups/G0016/.

[43] "Mimikatz," https://attack.mitre.org/software/S0002/.

[44] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer, "Neural architectures for named entity recognition," in *NAACL-HLT*, 2016.

[45] Y. Lin, S. Shen, Z. Liu, H. Luan, and M. Sun, "Neural relation extraction with selective attention over instances," in *ACL*, 2016.

[46] A. J. Ratner, C. M. De Sa, S. Wu, D. Selsam, and C. Ré, "Data programming: Creating large training sets, quickly," in *NeurIPS*, 2016.

[47] P. Gao, X. Xiao, Z. Li, K. Jee, F. Xu, S. R. Kulkarni, and P. Mittal, "A query system for efficiently investigating complex attack behaviors for enterprise security," in *VLDB*, 2019.

[48] P. Gao, X. Xiao, D. Li, K. Jee, H. Chen, S. R. Kulkarni, and P. Mittal, "Querying streaming system monitoring data for enterprise system anomaly detection," in *ICDE*, 2020.

[49] G. Husari, E. Al-Shaer, M. Ahmed, B. Chu, and X. Niu, "Ttpdrill: Automatic and accurate extraction of threat actions from unstructured text of cti sources," in *ACSAC*, 2017.

[50] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *USENIX Security*, 2019.

[51] P. Gao, X. Liu, E. Choi, B. Soman, C. Mishra, K. Farris, and D. Song, "A system for automated threat intelligence gathering and management," in *SIGMOD*, 2021.

[52] "ThreatMiner," https://www.threatminer.org/.