Threat Intelligence Computing

Xiaokui Shu
IBM Research
xiaokui.shu@ibm.com

Frederico Araujo
IBM Research
frederico.araujo@ibm.com

Douglas L. Schales
IBM Research
schales@us.ibm.com

Marc Ph. Stoecklin
IBM Research
mpstoeck@us.ibm.com

Jiyong Jang
IBM Research
jjang@us.ibm.com

Heqing Huang
IBM Research
hhuang@us.ibm.com

Josyula R. Rao
IBM Research
jrrao@us.ibm.com

ABSTRACT

Cyber threat hunting is the process of proactively and iteratively formulating and validating threat hypotheses based on security-relevant observations and domain knowledge. To facilitate threat hunting tasks, this paper introduces *threat intelligence computing* as a new methodology that models threat discovery as a graph computation problem. It enables efficient programming for solving threat discovery problems, equipping threat hunters with a suite of potent new tools for agile codifications of threat hypotheses, automated evidence mining, and interactive data inspection capabilities.

A concrete realization of a threat intelligence computing platform is presented through the design and implementation of a domain-specific graph language with interactive visualization support and a distributed graph database. The platform was evaluated in a two-week DARPA competition for threat detection on a test bed comprising a wide variety of systems monitored in real time. During this period, sub-billion records were produced, streamed, and analyzed, dozens of threat hunting tasks were dynamically planned and programmed, and attack campaigns with diverse malicious intent were discovered. The platform exhibited strong detection and analytics capabilities coupled with high efficiency, resulting in a leadership position in the competition. Additional evaluations on comprehensive policy reasoning are outlined to demonstrate the versatility of the platform and the expressiveness of the language.

CCS CONCEPTS

• **Security and privacy → Intrusion detection systems**; Formal security models; • **Computing methodologies**; • **Information systems →** *Query languages*;

KEYWORDS

Threat hunting; intrusion detection; computing methodology

ACM Reference Format:
Xiaokui Shu, Frederico Araujo, Douglas L. Schales, Marc Ph. Stoecklin, Jiyong Jang, Heqing Huang, and Josyula R. Rao. 2018. Threat Intelligence Computing. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3243734.3243829

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243829

1 INTRODUCTION

After decades of research and development on intrusion and anomaly detection, malware analysis, domain expert systems, and security information and event management systems, modern cybersecurity practice still relies heavily on human experts for information digestion and decision making in tasks such as context completion, false positive elimination, and end-to-end attack story construction. The problem can be traced in part to the involved threat campaign discovery process, which often demands the ability to establish causal inferences based on missing contextual information and domain knowledge not easily enumerable by current techniques.

Not surprisingly, one major gap between completely autonomous threat detection and today's automated systems is the inability to effectively model all required knowledge in pre-programmed systems. Conventional approaches either *i)* program specific human knowledge into their detection logic (such as rule-based detection, intrusion detection expert systems, and behavior-based detection), or *ii)* acquire detection knowledge from limited and scarce training domains (such as binary analysis, anomaly detection, and automatic feature engineering). The former category exhibits incomplete cognitive traits associated with the encoding of contextual domain knowledge applied to pre-programmed threat detection models, while the latter subsumes the difficult and tedious task of eliminating false positives to separate *anomalous* (e.g., evolution of user behaviors) from *malicious* (e.g., an actual exploit) behaviors.

Moreover, machine learning-based detection schemes learn from preset domains (e.g., feature domains for unsupervised clustering), and these learning domains are neither adaptive nor guaranteed to cover the ever-evolving attack techniques. For example, `*.*.*.255` is commonly set as a broadcast IP, which is not security-relevant until an analyst links it to a data exfiltration campaign that hides the destination of a cross-host information movement. Such context-sensitive knowledge is usually opaque and hard to generalize given the commonplace dependency on exogenous factors that elude fully-automated threat discovery approaches. To aggravate the problem, typical solutions introduce interpretation gaps that often lead to erroneous conclusions and increased burden on cyber combatants.

Therefore, human involvement remains indispensable for effectively uncovering cyber threats buried in the myriad of loosely-related events captured by sensors deployed across systems and networks. Analysts in security operation centers (SOCs) digest indicators of compromise (IOCs) fired by various automated detection modules, perform event triage, sift through false alarms, and search for correlations to discover potential threats or attack campaigns. The extracted *threat intelligence* (including context, actionable advice, etc.) can be shared and consumed by other analysts to support

agile detection strategies. Among SOC analysts, a special task force composed of *threat hunters* actively creates and validates new attack hypotheses to derive additional threat intelligence.

As a formal security practice, threat hunting can be strenuous to conduct due to its dynamic nature and uncertainty, requiring a fluid interplay between human deliberation and specialized tooling for inspection and reasoning. Existing toolkits include *i)* security information and event management (SIEM) systems such as HP ArcSight [31], *ii)* threat intelligence sharing platforms such as IBM X-Force [35], and *iii)* individual task scripting such as process-level back-tracking in Cb Response [9]. Unfortunately, these tools do not cope with dynamic reasoning [75] requirements, and are difficult to customize and interoperate. For example, a threat hunter who desires to quickly investigate the dynamic loading behavior used by a 0-day attack, needs to first backtrack data- and control-flows with unique constraints associated with the particular exploit, which is difficult to express and program using existing tools.

To overcome these disadvantages and facilitate human-machine agile detection strategy co-development, we introduce *threat intelligence computing* as a new methodology for rapidly programming threat hunting workflows, searching for threat evidence, and iteratively validating threat hypotheses to uncover attack campaigns with loosely coupled attack steps. By recasting threat hunting as a programming task, the new paradigm provides (1) a standard representation for traces, logs, alerts, and threat intelligence holding heterogeneous formats and interfaces, (2) programmability for much faster development iterations than traditional security software development, and (3) an interoperable, *metasploit*-like programming environment, in which threat hunters can easily and declaratively create and execute threat hunting workflows.

To ensure uniform data representation, we express computations on one or many computing devices as a temporal graph, defined as a *computation graph* (CG). Similar to process calculi [1], basic elements in a CG are entities (e.g., processes, files, sockets) and events (e.g., file read, process fork). A CG references the entire history of computation including any entities or events associated with attacks or threats. Security-relevant data such as alerts, IOCs, and intermediate threat analysis results are subgraphs, which can be denoted by labels on elements of a CG. As a result, threat detection becomes a *graph computation problem* whose solution is to iteratively deduce threat-inducing subgraphs in a CG.

To manage CGs and program graph computations atop them, we conceptualize, formalize, and evaluate *τ-calculus*, a graph computation platform for threat intelligence computing. It comprises *i)* a Turing-complete domain-specific language (DSL) with syntax tailored for programming on CGs, *ii)* a graph database designed and implemented to cope with efficient data storage and retrieval for live and forensic threat investigations, and *iii)* peripheral components for supporting interactive programming. In addition to basic features, such as variable reference and declarative programming, we conceptualize the language for superior code composability and reusability than existing general-purpose graph languages, e.g., Gremlin [73], Cypher [64]. We also back our graph database with a distributed key-value store for low-level CG operation optimization targeting unique CG properties, such as data locality and immutability. This architecture gives τ-calculus an edge over conventional graph databases, e.g., Neo4J [65], which cannot meet the performance requirements of typical threat hunting scenarios.

Our contributions can be summarized as follows:

- We formalize threat intelligence computing as a new security paradigm, define a CG as an abstract data model compatible with diverse monitoring granularities, and describe threat hunting as an application of threat intelligence computing.
- We design τ-calculus as a graph computation platform for threat intelligence computing. It consists of a domain-specific language with syntax tailored for CG computations, and a distributed graph database optimized for CG operations.
- We realize τ-calculus and its peripherals in Haskell and TypeScript including the language interpreter, the graph database, the interactive console, and the CG Browser (for interactive CG visualization and inspection).
- We evaluate the practicality, effectiveness, and performance of τ-calculus in a two-week DARPA threat detection competition on live-monitored systems (Windows, FreeBSD, Linux, and Android) and demonstrate its strong capabilities for threat hunting, automated detection, and policy reasoning.

2 THREAT INTELLIGENCE COMPUTING

Threat intelligence computing discovers threats, deduces threat intelligence, and supports efficient threat hunting through a standard data representation (Section 2.1), programmability (Section 2.3), and an interactive programming environment (Section 4).

2.1 Computation Graph

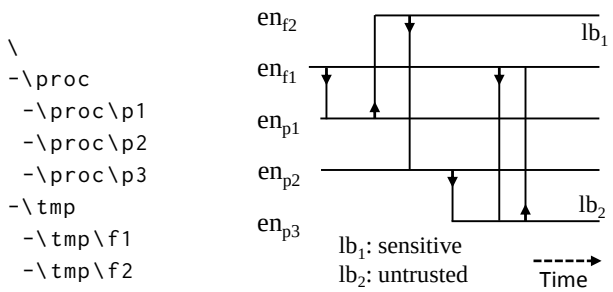
A computation graph (CG) is an abstract representation of computations inspired by process calculi [1]. At its core, a CG is a *labeled semi-directed temporal graph* which objectively records both intrusive and non-intrusive computations on computing devices plus security knowledge associated with the computations. Table 1 formally defines a CG as a 4-tuple $\langle \mathbb{T}, \mathbb{V}, \mathbb{L}, \Lambda \rangle$ where Ψ and Θ denote *time* and the monitoring *space*, and $\mathbb{T}$ and $\mathbb{V}$ denote monitored *entities* and traced *events* within $\Psi \times \Theta$. Labels $\mathbb{L}$ in a CG contain critical information for security reasoning. A label $\texttt{lb} \in \mathbb{L}$ associates a set of elements through Λ, where lb denotes one of three categories:

- *Element attribute* (objective information derived from computation recording): a label identifies a set of elements with a particular attribute, e.g., an event type `READ`.
- *Element relation* (objective information derived from computation recording): a label expresses some relation among a set of elements, e.g., a provenance linkage between `READ` and `WRITE` event of a process, which connects hundreds of `READ`/`WRITE` events. This label embeds finer-grained provenance information into an inter-process level CG.
- *Security knowledge* (subjective information regarding the security and privacy goals and reasoning procedures): a label marks a group of elements with some security knowledge. This label can be generated as either *i)* intermediate/final results of threat deduction, or *ii)* organization policies, IOCs, or anomaly scores imported from external detection systems, e.g., a set of confidential files, or IPs marked as C&C servers.

Table 1: Terms and Symbols in Computation Graph (Scope: Ψ, Θ; Composition: $\mathbb{T}$, $\mathbb{V}$, $\mathbb{L}$, Λ; Helper: $\mathbb{M}$)

Name	Symbol	Definition	Description	Visualization
time	$\Psi = \{\psi, \dots\}$	$\Psi = (\mathbb{Z}, +)$	time counted in machine cycles or aggregated units	x-axis
space	$\Theta = \{\theta, \dots\}$	$\Theta = (\mathbb{Z}, +)$	space of entities those can be monitored or traced	y-hyperplane
entities	$\mathbb{T} = \{\mathtt{en}, \dots\}$	$\mathtt{en} = \langle\theta, \psi, \psi'\rangle$	computation entities with their lifespans	line segments along x-axis
events	$\mathbb{V} = \{\mathtt{ev}, \dots\}$	$\mathtt{ev} = \langle\mathtt{en}, \mathtt{en}', \psi\rangle$	information flows as pairs of entities at specific times	line segments in y-hyperplane
labels	$\mathbb{L} = \{\mathtt{lb}, \dots\}$		an enumeratable set of labels	labels on line segments
mappings	Λ	$\mathbb{T} \times \mathbb{V} \leftrightarrow \mathbb{L}$	bi-directional mapping between elements and labels	
elements	$\mathbb{M} = \{\mathtt{el}, \dots\}$	$\mathtt{el} \in \mathbb{T} \cup \mathbb{V}$	an element is an alias referencing an entity or an event	

‡ Event directions are expressed in the order of entities. *Bi-directed* and *non-directed* events are also included in real-world uses regarding monitoring capabilities.



(a) host filesystem **(b) a computation graph**

Figure 1: CG example at host level (processes and files).

CG is an abstraction of computations. It is able to represent Turing-complete computations at different monitoring granularities, which supports threat reasoning and detection at different levels. Figure 1 describes one example of a CG at the host level and Appendix A describes two other CG examples at the network and process levels. In Figure 1, system activities are logged via syscall monitoring and program instrumentation. Entities in this CG consist of *subjects* (e.g., processes and threads) and *objects* (e.g., files, pipes, and network sockets). Security data is embedded in labels: $\mathtt{lb}_1$:`sensitive` indicates that $\mathtt{en}_{f2}$ contains sensitive information, and $\mathtt{lb}_2$:`untrusted` indicates that $\mathtt{en}_{p3}$ is not certified by the company. Data leakage occurs when $\mathtt{en}_{p3}$ can be traversed from $\mathtt{en}_{f2}$, as shown in Figure 1.

2.2 Security Model For Threat Hunting

Given a CG that records objective computation histories regarding both intrusive and non-intrusive data, threat discovery reduces to the graph query problem of iteratively computing the closure over the subset of security-relevant subgraphs in the CG, and finally yielding the subgraph that describes the threat or intrusion. Graph queries can be pre-programmed into IDSes or behavior anomaly detection systems, or they can be accomplished through on-demand agile reasoning development. Threat hunting composes sequences of graph queries to iteratively and interactively conceive, verify, revise, and confirm threat hypotheses.

The process of composing and executing graph queries in a CG is graph computation. During the computation, any variable referencing a subgraph is also a label to the set of entities and events of that subgraph, and the variable can be stored as a label on CG (*security knowledge* label category in section 2.1). Since the outcome of each iterative graph computation step is a subgraph or a label, each step can be implemented natively in a graph computation language or in an external module as a black-box, which outputs a set of events and entities as the subgraph.

Threat intelligence is therefore generated in the graph query when a threat is discovered. The query, especially the graph pattern, describes the threat and can be executed to search other CGs for the specific threat.

2.3 Programming Platform Requirements

Programming on CGs enables dynamic reasoning and agile reasoning development. Next, we detail the requirements to achieve programmability for threat intelligence from the *language design* and *platform realization* perspectives.

Comprehensive Graph Pattern Composition. Graph pattern matching is at the core of graph querying. A graph pattern, in essence, is a set of constraints describing the subgraph(s) to be matched, where a constraint over graph elements describes (1) a single graph element (e.g., a label/property of an entity), or (2) an element relation (e.g., an entity connects to an event). Pattern composition allows for embedding human domain knowledge into the deduction procedure. Simple pattern examples, which can be expressed by most graph languages, include:

- *behavior of typical DLL injections*[1]: two entities with `PROCESS` labels are connected by an event with label `CREATE_THREAD`.
- *behavior of untrusted executions*: an entity with `FILE` label but not `TRUSTED_EXE`[2] label connects to an event labeled `EXECUTE`, then to an entity labeled `PROCESS`.
- *behavior of data leak*: an entity labeled with `SENSITIVE` connects to an entity labeled `NETFLOW` within 10 hops.

What makes pattern matching powerful in threat intelligence computing is its ability to treat patterns as values and compose larger patterns based on others, thus enabling pattern reuse and abstraction. One common task in threat hunting is to back traverse from a pre-matched subgraph following some unique traversal guidance for the case. Such a traversal can be expressed as a pattern, which matches a subset of entities/events from a given subgraph (as

[1] DLL injections can be benign or malicious. This pattern does not distinguish them.
[2] An organization may only permit a whitelist of applications for execution.

Table 2: Platform Candidates for the Realization of Threat Intelligence Computing

	Titan (Gremlin)	Neo4J (Cypher)	τ-calculus
Data Model: Language Built-in Element/Algorithm Support			
graph element labeling (intermediate analysis result storing/retrieval)	●	●	●
temporal graph (native temporal operation/algorithm support)	○	○	●
graph integrity guarantee (immutability of computation history)	○	○	●
Programming: Abstraction, Code Reusability, and Computability			
declarative programming (minimizing low-level execution exposure)	●	●	●
comprehensive pattern composition (abstraction and code reusability)	○	○	●
function composition and application (modular program design)	●	○	●
Turing completeness (potential detecting/analytical algorithm design)	●	○	●
Performance: Real-time and Forensic Uses			
streaming ingestion (24/7 system events and security log feeding)	●	●	●
distributed storage and data retrieval (scalablity potential)	●	○	●
data awareness (fine-grained data locality and indexing tuning)	○	○	●
Peripherals: Interfaces, Extensions, and Helpers			
external language API (user-defined functions, etc.)	●	●	●
built-in console (interactive tasks, e.g., threat hunting)	●	●	●
interactive visualization tool (interface to opaque human reasoning)	●	●	●

its argument) based on element attributes or element relations. An abstract traversal pattern can be expressed as a pattern similar to the simple *data leak pattern* described above but takes two arguments: one refers to the subgraph as the source for traversal, and the other refers to the guidance pattern, which dynamically expands to the set of constraints on the traversed variable in the matching procedure.

To implement pattern matching as a constraint-solving problem for compiler reasoning and optimization, the construction of patterns should be limited to graph element constraints and other patterns, making them pure and solvable. Full-fledged functions should be defined as a separate first-class object in the language.

Declarative Language Design. A language supporting threat intelligence computing should be declarative, so that threat hunters need only to specify *what* to do, e.g., matching a pattern, instead of *how* to do it, e.g., steps of constraint solving for matching a pattern. Additional features of this language include *native function composition* and *external function interface*.

Functions and Computability. Though patterns are similar to functions as in a pure functional language, I/O and other operations are inevitable to make a language useful. The ability to compose functions with side effects in native graph languages is critical for code reusability and abstraction. Furthermore, recursive function referencing is desirable since lots of graph algorithms are recursive in nature (e.g., graph traversal). In addition, though a Turing-complete graph language can program any Turing-computable problem in theory, many algorithms have weak or inexistent graph language equivalents in reality. It is therefore important for graph languages to provide external function interfaces, or user-defined functions (UDF), to delegate dedicated tasks like anomaly score calculation to external libraries such as `scikit-learn` in Python.

Domain-Specific Language Syntax. A CG is a labeled semi-directed temporal graph. Its entities/events must be immutable, though new labels can be added. General graph languages do not reflect this property and therefore need to be extended or modified. In addition, a user-friendly programming language should enforce that all executable programs are well-typed according to a typing system in order to reduce programming errors.

Distributed and Data-Aware Storage. Supporting a distributed database architecture is critical for coping with a large number of devices or CGs, or/and long-term monitoring and historical data support. It is also important to optimize the database design regarding unique CG properties, e.g., temporal locality tuning for fast graph traversal. Moreover, an effective threat intelligence computing platform must be capable of ingesting monitored data as live streams concurrently from multiple sources to enable threat hunting on multiple systems. This becomes vital if one wants to capture early stages of APTs and take actions before later destructive APT stages.

Interactive Programming and Visualization. To effectively and efficiently support human-machine cooperation, the platform must be conducive to agile detection strategy development and rapid interpretation of execution outcomes. Toward this end, we envision a threat hunting ecosystem that enables analysts to obtain interactive feedback from threat hunting tasks coupled with visual representations that reduce the semantic gap between threat behavior understanding and the CG presentation.

Summary. Though many graph languages/databases are available today for general and domain-specific programming, the programmability requirements of threat intelligence computing are hardly met by existing solutions. To overcome this limitation, we introduce τ-calculus as a new graph computation platform, including a DSL, a graph database, and peripheral systems. Table 2 compares τ-calculus with the most popular graph programming platforms Neo4J/Cypher [65] and Titan/Gremlin [12] for threat intelligence computing. τ-calculus meets and exceeds the minimal requirements

Table 3: Semantics of τ-calculus Relational Operators

Op	Predicate	Semantics
has	el **has** lb	el has label lb
conn	en **conn** ev	en connects to ev via a join point
reach	$\texttt{el}_1$ **reach**$_n$ $\texttt{el}_2$	$\texttt{el}_1$ reaches $\texttt{el}_2$ in at most n hops
prec	$\texttt{ev}_1$ **prec** $\texttt{ev}_2$	$\texttt{ev}_1$ precedes $\texttt{ev}_2$
at	ev **at** ψ	ev occurs at timestamp ψ
bf	ev **bf** ψ	ev occurs before timestamp ψ
af	ev **af** ψ	ev occurs after timestamp ψ
in	el **in** gr	el is in graph gr

for realizing threat intelligence computing, providing a new powerful platform for threat hunters to validate threat hypotheses and uncover entire attack campaigns deeply buried in the sea of logging events and monitoring data.

3 τ-CALCULUS LANGUAGE

We design the τ-calculus language as a domain-specific language for programming computations on CG. The language is declarative and embraces *patterns* as a first-class language construct. It supports parameterized pattern definition and pattern application similar to functions. Well-typed terms (Table 3) in a pattern definition are sets of predicate expressions used for constraint solving purposes as explained in Section 2.3.

For explanatory precision, we formally define the τ-calculus language in terms of the simple, typed Turing-complete language shown in Figure 2. The simplified programming language abstracts irrelevant implementation details, capturing only essential features for formalizing threat intelligence computing.

3.1 Language Syntax

Programs $\mathcal{P}$ are lists of commands, denoted $\overline{c}$. Commands consist of variable assignments, graph label operations, identifier-dereferencing assignments (stores), graph visualizations, pattern/function application, pattern abstraction, pattern matching, function abstraction, and fixed-point operator. Expressions evaluate to values u and value-labeled tuples $gr \in \mathbb{T} \times \mathbb{V} \times \mathbb{L}$, where u ranges over primitive value representations, and gr denotes CGs.

Variable names range over identifiers and function names, and the type system supports primitive types, graph abstract data types, predicate expression types, pattern types, and function types (τ). We use $2^{\texttt{en}}$ to denote the type of the set of values of type en. Execution contexts are comprised of *i)* a store σ relating locations to CGs and graph identifiers to locations, and *ii)* an environment ρ mapping variables to values, graphs, and functions.

Pattern matching on graphs is the problem of finding a homomorphic or isomorphic image of a given graph, called the pattern, in another graph. In τ-calculus, patterns are defined as n-ary functions $(\tau_1, \ldots, \tau_n) \rightarrow \tau_{gr}$, specified as constraints $\overline{p}$ on computational graph elements (predicate expressions) and pattern applications (cf. Section 3.3). Patterns are pure and do not have side effects. Pattern matching expressions are evaluated by a constraint solver.

The language does not enforce any particular *label model* — a notation for labeling graphs together with *policies* governing those

programs	$\mathcal{P} ::= \overline{c}$
commands	$c ::= v := e \mid \texttt{label}\ e_1\ e_2 \mid \texttt{store}\ e_1\ e_2 \mid \texttt{show}\ e$
expressions	$e ::= u \mid gr \mid v \mid \texttt{load}\ e$
	$\mid \overline{p} \mid \texttt{pattern}\ \overline{v}\ .\ e \mid e\,\overline{e}$
	$\mid \texttt{match}\ e_{\overline{p}}\ \texttt{with}\ \overline{e}\ \texttt{in}\ e$
	$\mid \texttt{fn}\ v : \tau\ .\ e \mid e_1 e_2 \mid \mu\, v : \tau\ .\ e$
pred expressions	$p ::= \ \textbf{not}\, p \mid e_1 \diamond_p e_2$
relational ops	$\diamond_p ::= \ \textbf{has} \mid \textbf{conn} \mid \textbf{reach}_n \mid \textbf{prec} \mid \textbf{at} \mid \textbf{bf} \mid \textbf{af} \mid \textbf{in}$
variables	v
values	$u ::=$ values of the underlying language
element types	$\texttt{el} ::= \texttt{en} \mid \texttt{ev}$
primitive types	$\tau_p ::= id \mid bool \mid \psi \mid \texttt{lb} \mid \texttt{el}$
graph type	$\tau_{gr} ::= 2^{\texttt{en}} \times 2^{\texttt{ev}} \times 2^{\texttt{lb}}$
pattern type	$\tau_{\overline{p}} ::= (\tau_1, \ldots, \tau_n) \rightarrow \tau_{gr}$
types	$\tau ::= \tau_p \mid \tau_{gr} \mid \tau_{\overline{p}} \mid \tau \rightarrow \tau'$
CGs	$gr \in \mathbb{T} \times \mathbb{V} \times \mathbb{L}$
visualization	$O ::=$ output methods
locations	$\ell ::=$ memory addresses
identifiers	$id \in ID$ (computation graph identifiers)
join points	$jp ::= \langle\rangle \mid \langle \texttt{en}, \texttt{ev} \rangle \mid \langle \texttt{ev}, \texttt{en} \rangle$
environment	$\rho : v \rightharpoonup (u \cup gr \cup \overline{p} \cup$
	$((u \cup gr \cup \overline{p}) \rightarrow (u \cup gr \cup \overline{p})))$
stores	$\sigma : (\ell \rightharpoonup (gr \cup \overline{p})) \cup (id \rightharpoonup \ell)$

Figure 2: τ-calculus language syntax.

labels. This choice of design enables τ-calculus to support different label embeddings for a variety of threat reasoning tasks such as traversals for root cause analysis or policy reasoning.

3.2 Denotational Semantics

Figure 3 presents a formal denotational semantics for τ-calculus language that unambiguously identifies what a τ-calculus program means. Semantic domains $\mathcal{E}$, $\mathcal{P}$, and C denote functions that associate precise meanings with expressions, predicates, and commands, respectively. Meaning function $\mathcal{E}[\![\,.\,]\!]$ defines the denotational semantics of expressions by structural induction, mapping each expression in the language to a domain associated with that expression. For example, $\mathcal{E}[\![\,v\,]\!]\rho$ denotes the value of v given some ρ. Similarly, $\mathcal{P}[\![\,.\,]\!]$ and $C[\![\,.\,]\!]$ define the meanings of predicates and commands in the language. We assume that programs are well-typed according to the formal typing rules described in Section 3.3.

Table 3 provides an informal description of the valuation semantics for the set of relational operators that form the basis for constructing patterns in τ-calculus. To describe the semantics of predicate expression *et* **conn** *ev*, we introduce *join points* as the set of tuples describing adjacent entity-event pairs in a CG, encoding incoming events as $\langle \texttt{ev}, \texttt{en} \rangle$ and outgoing events as $\langle \texttt{en}, \texttt{ev} \rangle$. For instance, $\mathcal{P}[\![\,et\ \textbf{conn}\ ev\,]\!]\,jp\rho$ is interpreted as valid (T) if $\langle et, ev \rangle \in jp$.

Notation $\rho[v \mapsto u]$ denotes function ρ with v remapped to u. For example, command $C[\![\,v := u\,]\!]\rho$ denotes $\rho[v \mapsto u]$. Likewise, $C[\![\,\texttt{store}\ id\ gr\,]\!]\sigma\rho$ rebinds id to gr in stores σ, and $C[\![\,\texttt{load}\ id\,]\!]\sigma\rho$

$\mathcal{P} : p \rightarrow (jp \rightarrow \rho \rightharpoonup \{T, F\})$ predicate denotations

$\mathcal{E} : e \rightarrow (\rho \rightharpoonup (u \cup gr \cup \overline{p}))$ expression denotations

$C : c \rightarrow (\rho \rightarrow \rho)$ command denotations

$\mathcal{P}[\![\textbf{not } p]\!]\, jp\rho = \neg \mathcal{P}[\![p]\!]\, jp\rho$

$\mathcal{P}[\![e_1 \textbf{ has } e_2]\!]\, jp\rho = \pi_3(\mathcal{E}[\![e_1]\!]\rho) \cap \mathcal{E}[\![e_2]\!]\rho \neq \varnothing$

$\mathcal{P}[\![e_1 \textbf{ conn } e_2]\!]\, jp\rho = \langle \mathcal{E}[\![e_1]\!]\rho, \mathcal{E}[\![e_2]\!]\rho \rangle \in jp$

$\mathcal{P}[\![e_1 \textbf{ reach}_n \ e_2]\!]\, jp\rho = p \neq \varnothing,$
where $p = path(\mathcal{E}[\![e_1]\!]\rho, \mathcal{E}[\![e_2]\!]\rho)\rho,\ n \in \mathbb{N}$
and $length(p) \leq n$

$\mathcal{P}[\![e_1 \textbf{ prec } e_2]\!]\, jp\rho = time(\mathcal{E}[\![e_1]\!]\rho) < time(\mathcal{E}[\![e_2]\!]\rho)$

$\mathcal{P}[\![e_1 \textbf{ at } e_2]\!]\, jp\rho = time(\mathcal{E}[\![e_1]\!]\rho) = \mathcal{E}[\![e_2]\!]\rho$

$\mathcal{P}[\![e_1 \textbf{ bf } e_2]\!]\, jp\rho = time(\mathcal{E}[\![e_1]\!]\rho) < \mathcal{E}[\![e_2]\!]\rho$

$\mathcal{P}[\![e_1 \textbf{ af } e_2]\!]\, jp\rho = time(\mathcal{E}[\![e_1]\!]\rho) > \mathcal{E}[\![e_2]\!]\rho$

$\mathcal{P}[\![e_1 \textbf{ in } e_2]\!]\, jp\rho = \mathcal{E}[\![e_1]\!]\rho \cap \mathcal{E}[\![e_2]\!]\rho = \mathcal{E}[\![e_1]\!]\rho$

$\mathcal{E}[\![u]\!]\rho = u$

$\mathcal{E}[\![gr]\!]\rho = gr$

$\mathcal{E}[\![\overline{p}]\!]\rho = \overline{p}$

$\mathcal{E}[\![v]\!]\rho = \rho(v)$

$\mathcal{E}[\![\texttt{load } e]\!]\sigma\rho = \sigma(\mathcal{E}[\![e]\!]\rho)$

$\mathcal{E}[\![\texttt{pattern } \overline{v} \,.\, e]\!]\rho = \texttt{pattern } \overline{u} \,.\, \mathcal{E}[\![e]\!]\rho[\overline{v} \mapsto \overline{u}]$

$\mathcal{E}[\![e\, \overline{e}]\!]\rho = \mathcal{E}[\![e]\!]\rho\ (\mathcal{E}[\![e_1]\!]\rho, \ldots, \mathcal{E}[\![e_n]\!]\rho)$

$\mathcal{E}[\![\texttt{match } e_{\overline{p}} \texttt{ with } \overline{e_1 \ldots e_n} \texttt{ in } e]\!]\rho = gr,$
where $gr \subseteq \mathcal{E}[\![e]\!]\rho\ , \mathcal{E}[\![e_{\overline{p}}]\!]\rho = \overline{p_1 \ldots p_k}\ ,$
and $\bigwedge_{i=1}^{k} \mathcal{P}[\![p_i[\overline{\mathcal{E}[\![e_1]\!]\rho \ldots \mathcal{E}[\![e_n]\!]\rho}, \mathcal{E}[\![e]\!]\rho]]\!]\, jp\rho$

$\mathcal{E}[\![\texttt{fn } v : \tau \,.\, e]\!]\rho = \texttt{fn } u \,.\, \mathcal{E}[\![e]\!]\rho[v \mapsto u]$

$\mathcal{E}[\![e_1 e_2]\!]\rho = \mathcal{E}[\![e_1]\!]\rho\ \mathcal{E}[\![e_2]\!]\rho$

$\mathcal{E}[\![\mu\, v : \tau \,.\, e]\!]\rho = \mu\,(\mu\, v : \tau \,.\, e) \,.\, \mathcal{E}[\![e]\!]\rho[v \mapsto \mu\, v : \tau \,.\, e]$

$C[\![v := e]\!]\rho = \rho[v \mapsto u]$, where $u = \mathcal{E}[\![e]\!]\rho$

$C[\![\texttt{label } e_1\ e_2]\!]\rho =$
$(\pi_1(\mathcal{E}[\![e_1]\!]\rho), \pi_2(\mathcal{E}[\![e_1]\!]\rho), \pi_3(\mathcal{E}[\![e_1]\!]\rho) \sqcup \pi_3(\mathcal{E}[\![e_2]\!]\rho))$

$C[\![\texttt{store } e_1\ e_2]\!]\sigma\rho = \sigma[\mathcal{E}[\![e_1]\!]\rho \mapsto \mathcal{E}[\![e_2]\!]\rho]$

$C[\![\texttt{show } e]\!]\rho = \mathcal{O}(\mathcal{E}[\![e]\!]\rho)$

Figure 3: Denotational semantics for τ-calculus language.

denotes graph $\sigma(id)$. A graph labeling $C[\![\texttt{label } gr\ lb]\!]\rho$ denotes $\pi_3(gr) \sqcup lb$, the join of gr's original label with new label lb, where π_3 denotes a projection on the third element of gr.

3.3 Typing Rules

Figure 4 presents the language's static semantics. The typing rules determine which terms are well-formed τ-calculus programs. They are a set of rules that allow the derivation of type judgments of form $\Gamma \vdash e : \tau$, where $\Gamma : v \rightharpoonup \tau$ is the type environment — a partial map from variables to types used to determine the types of the free variables in e. The environment $\Gamma[v \mapsto \tau]$ is obtained by rebinding v to τ (or creating the binding anew if $v \notin dom(\Gamma)$).

Every well-typed τ-calculus term has a proof tree consisting of applications of the typing rules to derive a type for the term. Such proof trees form the basis for type checking terms in the language. For example, consider a computation graph depicting the scenario where a browser process writes to a user file. In this context, the predicate expression *process* **conn** *sys_write* evaluates to T, which is a *bool*. The expression is thus well-typed, which can be verified by constructing its typing derivation:

$$\frac{\Gamma \vdash process : \mathsf{en} \quad \Gamma \vdash sys\_write : \mathsf{ev}}{\Gamma \vdash process \textbf{ conn } sys\_write : bool}\ \textsc{Fwd}_{\mathsf{en}}$$

4 ARCHITECTURE AND REALIZATION

Figure 5 shows an overview of τ-calculus' architecture. The full-stack graph computation platform comprises a language interpreter, a graph database, and user-interface components, which include an interactive console (τ-REPL) and a CG visualization tool (CG Browser). The graph database employs a distributed key-value store, FCCE [74], for *i)* long-term monitoring data storage with data locality optimization, and *ii)* concurrent multi-source streaming data ingestion. All components of τ-calculus are implemented in Haskell except CG Browser, which is implemented in TypeScript. τ-REPL[3] and CG Browser together provide the interactive programming and data inspection environment required for threat reasoning (cf. Section 2.3). Next, we detail core platform subsystems.

4.1 Typing System

τ-calculus' type checker provides informative user feedback to help reducing programming errors. τ-calculus interpreter binds types to variables through variable *declaration* and *inference*. Local variables in a predicate (e.g., x in x **conn** y) must be declared with types before use (e.g., $x \in \mathbb{T}$). For simplicity, this paper uses the symbols defined in Table 1 to denote variable types (e.g., x_{en} **conn** y_{ev}). Type inference applies to function and pattern parameters (e.g., x **in** y indicates that the variable y is inhabited by type τ_{gr}). Type inference also applies to the abstract type `el`, which can be either an entity or an event (cf. Figure 4).

4.2 Constraint Solving

Pattern matching evaluation involves solving the set of constraints, or predicate expressions, defined by the pattern. This task is computationally non-trivial because *i)* a pattern can refer to other patterns via parameters, and *ii)* CG is stored on distributed external storage, making it expensive to check data associated with each predicate.

To cope with the parametric expressiveness of patterns, we developed a module to perform *pattern application* similar to *function application*. The constraint solving process efficiently decides when and how many times the pattern application needs to be performed for a single pattern reference. For instance, if a pattern reference associates with a variable that relates to a **reach** predicate, the referred pattern may be applied repeatedly in the traversal procedure to minimize on-disk data requests in subsequent traversal steps.

Since on-disk data queries only support solving one constraint at a time, we developed a constraint-solving algorithm to solve constraints iteratively and propagate the latest solved constraint to all variables associated with previously satisfied constraints. When a pattern is parsed and the abstract syntax tree (AST) is

[3] τ-REPL is to threat hunters as `msfconsole` (Metasploit console) is to pen testers [69].

$$\frac{}{\Gamma \vdash u : \tau_p}\ \textsc{Val} \qquad \frac{}{\Gamma \vdash gr : \tau_{gr}}\ \textsc{Graph} \qquad \frac{}{\Gamma \vdash p : bool}\ \textsc{Pred} \qquad \frac{}{\Gamma \vdash v : \Gamma(v)}\ \textsc{Var} \qquad \frac{\Gamma \vdash e : id}{\Gamma \vdash \texttt{load}\ e : \tau_{gr}}\ \textsc{Load}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash v := e : \Gamma[v \mapsto \tau]}\ \textsc{Assign} \qquad \frac{\Gamma \vdash e_1 : \tau_{gr} \quad \Gamma \vdash e_2 : \texttt{lb}}{\Gamma \vdash \texttt{label}\ e_1\ e_2 : \Gamma}\ \textsc{Label} \qquad \frac{\Gamma \vdash e_1 : id \quad \Gamma \vdash e_2 : \tau_{gr}}{\Gamma \vdash \texttt{store}\ e_1\ e_2 : \Gamma}\ \textsc{Store}$$

$$\frac{\Gamma[v \mapsto \tau] \vdash e : \tau'}{\Gamma \vdash (\texttt{fn}\ v : \tau\ .\ e) : \tau \to \tau'}\ \textsc{Abs} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}\ \textsc{App} \qquad \frac{\Gamma[v \mapsto \tau] \vdash e : \tau}{\Gamma \vdash (\mu\, v : \tau\ .\ e) : \tau}\ \textsc{Fix} \qquad \frac{\Gamma \vdash e : \tau_{gr}}{\Gamma \vdash \texttt{show}\ e : \Gamma}\ \textsc{Show}$$

$$\frac{n > 0 \quad \Gamma[\overline{v_1 \cdots v_n} \mapsto \overline{\tau_1 \cdots \tau_n}] \vdash e : \tau_{gr}}{\Gamma \vdash (\texttt{pattern}\ \overline{v}\ .\ e) : (\tau_1, \ldots, \tau_n) \to \tau_{gr}}\ \textsc{Pat} \qquad \frac{n > 0 \quad \Gamma \vdash e : (\tau_1, \ldots, \tau_n) \to \tau_{gr} \quad \Gamma \vdash e_i : \tau_i \text{ for } i = 1, \ldots, n}{\Gamma \vdash e\,\overline{e} : \tau_{gr}}\ \textsc{NApp}$$

$$\frac{\Gamma \vdash e : \tau_{gr} \quad \Gamma \vdash e_i : \tau_i \text{ for } i = 1, \ldots, n \quad \tau_1 \cdots \tau_n . \Gamma \vdash e_{\overline{p}} : (\tau_1, \ldots \tau_n) \to \tau'_{gr}}{\Gamma \vdash \texttt{match}\ e_{\overline{p}}\ \texttt{with}\ \overline{e}\ \texttt{in}\ e : \tau'_{gr}}\ \textsc{Match}$$

$$\frac{\Gamma \vdash e : bool}{\Gamma \vdash \textbf{not}\ e : bool}\ \textsc{Not} \qquad \frac{\Gamma \vdash e_1 : \texttt{el} \quad \Gamma \vdash e_2 : \texttt{lb}}{\Gamma \vdash e_1\ \textbf{has}\ e_2 : bool}\ \textsc{Has} \qquad \frac{\Gamma \vdash e_1 : \texttt{en} \quad \Gamma \vdash e_2 : \texttt{ev}}{\Gamma \vdash e_1\ \textbf{conn}\ e_2 : bool}\ \textsc{Fwd}_{\texttt{en}} \qquad \frac{\Gamma \vdash e_1 : \texttt{ev} \quad \Gamma \vdash e_2 : \texttt{en}}{\Gamma \vdash e_1\ \textbf{conn}\ e_2 : bool}\ \textsc{Fwd}_{\texttt{ev}} \qquad \frac{\Gamma \vdash e_1 : \texttt{el} \quad \Gamma \vdash e_2 : \texttt{el}}{\Gamma \vdash e_1\ \textbf{reach}_n\ e_2 : bool}\ \textsc{Imp}$$

$$\frac{\Gamma \vdash e_1 : \texttt{ev} \quad \Gamma \vdash e_2 : \psi}{\Gamma \vdash e_1\ \textbf{at}\ e_2 : bool}\ \textsc{At} \qquad \frac{\Gamma \vdash e_1 : \texttt{ev} \quad \Gamma \vdash e_2 : \psi}{\Gamma \vdash e_1\ \textbf{bf}\ e_2 : bool}\ \textsc{Bef} \qquad \frac{\Gamma \vdash e_1 : \texttt{ev} \quad \Gamma \vdash e_2 : \psi}{\Gamma \vdash e_1\ \textbf{af}\ e_2 : bool}\ \textsc{Aft} \qquad \frac{\Gamma \vdash e_1 : \texttt{ev} \quad \Gamma \vdash e_2 : \texttt{ev}}{\Gamma \vdash e_1\ \textbf{prec}\ e_2 : bool}\ \textsc{Prec} \qquad \frac{\Gamma \vdash e_1 : \texttt{el} \quad \Gamma \vdash e_2 : \tau_{gr}}{\Gamma \vdash e_1\ \textbf{in}\ e_2 : bool}\ \textsc{In}$$

Figure 4: Typing rules for τ-calculus language.

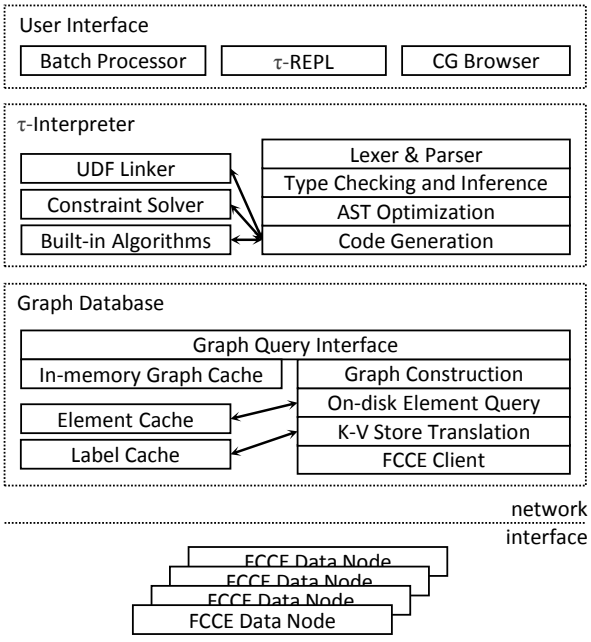


Figure 5: τ-calculus platform architecture.

created, the interpreter determines how constraints are connected and stores the constraint relations as a *graph of constraints* (GoC) into a supplementary data structure in the AST. To evaluate a pattern, the constraint solver orders constraints by heuristics and user guidance and iteratively satisfies all constraints, including single-element constraints, (e.g., x **has** $\langle type : READ \rangle$) and multi-element constraints (e.g., x **conn** y). After each iterative constraint-solving step, the variables associated with the pattern may undergo a state change, which is propagated to all previously solved variables through a graph traversal on GoC, from the changed variables to all previously solved variables.

4.3 Built-in Traversal Support

Backward and forward traversals are common tasks in threat intelligence for root cause discovery and impact analysis [42]. While one can implement traversal as a native τ-calculus function with recursive function support, it can be useful to encode the traversal semantics as a built-in primitive pattern predicate. To this end, the built-in relation **reach** (cf. Section 3) provides four functionalities:

- *Forward traversal* (touched x, untouched y): x **reach** y
- *Backward traversal* (untouched x, touched y): x **reach** y
- *Reachability Filter* (touched x, y): x **reach** y
- *Pathfinder* (touched x, z, untouched y): x **reach** y, y **reach** z

The traversal computes the graph closure over all subgraphs reachable from a provided subgraph or set of entities/events. A touched/untouched variable refers to whether any constraint associated with that variable has been solved in previous iterating constraint-solving steps (Section 4.2). The last pattern expression (*Pathfinder*) is useful for searching connections (a subgraph y) between two sets of elements or subgraphs x and z.

To solve constraints expressed as traversal predicates, the system takes into account (1) *event direction*, if present (information-/control-flow direction), (2) *temporal requirement* (e.g., events in a backward step can only occur earlier than events in the current step), and (3) *variable constraints*, if any (from other predicates or patterns in arguments). The two most important optimizations applied to the traversal procedure are:

(1) *dynamic programming*: bookkeeping results of all traversal sub-problems solved in previous iterations. A traversal sub-problem is defined by its domain (a connected entity and the query time range) and its codomain (a set of events).

(2) *proactive constraint solving*: if a variable in a traversal predicate has other constraints (either as direct predicates or referenced patterns), the additional constraints are proactively and repeatedly solved in each iterating step of the traversal to minimize on-disk data queries, especially for hub entities.

Table 4: DARPA TC Monitored Systems: CG Statistics†

OS	#(entities)	#(events)	#(labels)⋆	Case Study
Windows	0.9M	19.1M	20.2M	Section 5.1
FreeBSD	0.5M	8.4M	43.7M	Section 5.2
Android	0.1M	77.4M	149.2M	Section 5.3
Linux	11.6M	26.0M	84.3M	Section 5.4

† Each row represents one monitored host selected for demonstration purpose.
⋆ Labels are stored as dictionary items described in Section 4.4.

4.4 Graph Database

The graph database stores both in-memory and on-disk CG portions, and provides graph query APIs to the interpreter. The two main functionalities of the graph database are to *i)* bridge the semantics of CG and low-level data storage, and *ii)* optimize graph retrieval throughput using multi-layer caches and data arrangement based on CG properties such as temporal locality of events.

We utilize FCCE [74] as the low-level key-value data store in our graph database realization. FCCE is designed for security data storage and processing; it supports concurrent multi-source asynchronous ingestion, distributed data storage, and data locality management. To optimize graph queries based on special CG properties, we compose FCCE schema to represent CG in key-value pairs and replicate critical values in multiple schemas for data locality preservation and fast retrieval from different perspectives. For instance, one replica of events deals with temporal locality: *i)* events are indexed by time, and *ii)* events occurring within a time window are managed on one memory page and stored at consecutive filesystem blocks. Other event replicas deal with labels and shared entities.

To process a graph query, the graph database first checks whether any portion of the data is already loaded into memory through previous queries. If not, it will split the graph query into one or more on-disk element queries, each of which is to be translated into key-value queries that FCCE can process. Labels are expressed as dictionary items to express complex element attributes. A simple element query searching for file entities whose `path` contains a substring `firefox` translates into two FCCE queries: the first searches for all satisfied labels, and the second searches for raw data to construct elements associated with these labels.

When raw data is retrieved from disk, buckets of key-value pairs are first cached in the FCCE client where data within a bucket has tight data locality and high probability to be queried in the same high-level graph query or following queries. Then, different components of an element are constructed and some are cached for frequent referencing, e.g., the *principal* label for processes contains multiple pieces of information including the username, uid, group, etc., and it is cached as a reference. Next, elements are constructed and cached. Lastly, the requested graph is assembled and returned.

5 WAR ROOM CASE STUDY

The security industry is evolving with ubiquitous monitoring facilities and enduring data generation mechanisms. The DARPA Transparent Computing (TC) program aims to push this trend to an extreme and investigate how security analysts can benefit from a complete recording of computations across a small network. Several research teams from academia and industry provided real-time monitoring of computations on diverse systems (Table 4). All monitoring systems emitted traces that can construct host-level CGs.

DARPA held a two-week threat detection competition in 2017 where a red team planned and conducted various attack campaigns covering all monitored devices with in-house planning and tool development. Traditional detection mechanisms such as anti-virus were nullified due to the unknown tools/malware and the inability to discover campaign stories. Anomaly detection mechanisms were limited due to the lack of training data and the variance of user behaviors between training and detection periods[4]. Since the attacks were unknown to the detection teams, it was also impossible to perform machine learning-based detection on attack data.

Given the fact that automatic security knowledge acquisition is largely unavailable in the setup and cannot cover unforeseen learning domains, embedding human domain knowledge into preprogrammed automated detection algorithms, e.g., SLEUTH [30], and dynamically composed analytical programs (based on fresh observations and related knowledge) achieved significant outcomes.

We deployed τ-calculus in the DARPA's threat hunting *war room* and trained two research scientists on τ-calculus to perform threat hunting tasks. During the evaluation, data from live-monitored devices was streamed through Apache Kafka, translated into CG format, and ingested into four FCCE data nodes. τ-calculus was run on a CentOS VM (16 vcores and 140GB mem).

τ-calculus established the threat hunting environment for reasoning over ongoing threats atop tens of millions of records per day from multiple systems. First-line alarms were raised by either τ-calculus detectors, external detectors, or pure manual discoveries. Threat hunters inspected alarms to eliminate false positives, connect attack steps, and explore undetected portions of attack campaigns. Overcoming fundamental issues in conventional threat hunting environments, τ-calculus helped us lead the competition with 67.5% attack campaign plots detected[5].

This section presents four concrete threat hunting tasks dynamically planned and completed during the competition on all monitored OSes to demonstrate the capability of τ-calculus: IOC detection (Section 5.1), interactive reasoning (Section 5.2), provenance tracking (Section 5.3), and threat evaluation (Section 5.4).

5.1 IOC Detection on Windows

Threat hunting usually starts from IOCs or observables provided by automatic detectors. The ability to promptly implement new IOC detectors regarding the on-site observations and newly acquired knowledge is critical to the discovery of newly developed attacks. τ-calculus was used for rapid development of a Windows IOC detector against *reflective DLL injection* (RDI) [19] during the competition. The automatic detector with a few lines of τ-calculus code captured the operation of DoublePulsar [15], which was used in conjunction with EternalBlue [21] by the red team in their attacks.

[4] It is common in the security industry that the availability and the quality of training data impoverish well-designed anomaly detection mechanisms.
[5] The result is calculated by the red team based on multiple hit-points for each campaign story. The result is not simply an evaluation of our detection, but an evaluation of the combined monitoring and detection capabilities with multiple monitoring teams.

```
function detectRDI () {
  pattern patRDI (whitelist) {
    ev_createT has ⟨type : create_thread⟩
    en_injector conn ev_createT
    en_injector has ⟨type : subject⟩
    en_injector = not whitelist ()
    ev_createT conn en_injectee
    en_injectee has ⟨type : subject⟩
  }
  pattern knownBenign () {
    en_monitorProc has ⟨cmdline : winlogbeat.exe⟩
    en_searchProc has ⟨cmdline : SearchIndexer.exe⟩
    ...
  }
  cg = load "windows-live-graph"
  rdi = match patRDI (knownBenign) in cg
  label rdi ⟨alert : RDI⟩
}
```

Figure 6: τ-calculus IOC detector against RDI.

```
pattern patPWDDB () {
  en_pwddb has ⟨path : /etc/pwd.db⟩
  en_pwddb conn ev_readdb
  ev_readdb has ⟨type : read⟩
  ev_readdb conn en_reader
}
```

Figure 7: τ-calculus pattern for tapping `pwd.db`.

The IOC detector (Figure 6) essentially comprised two graph patterns: the first described the general DLL injection behavior that *a process creates a remote thread into another process*, and the second whitelisted known benign DLL injection instances. In the first pattern, CG labels provided a level of abstraction from the raw data so that threat hunters did not need to enumerate *subjects* as *process*, *thread*, etc., and *events* with the `create_thread` type[6], `CreateRemoteThread` (Window API), `RtlCreateUserThread` (undocumented Windows API), or even direct syscalls.

The `detectDRI()` function executed in seconds even on the entire two-week dataset. It captured 17 injection instances without whitelisting (denoted by an empty `knownBenign()` pattern). The `knownBenign()` pattern was created and revised throughout the days to exclude anomalous (in terms of not seen in previous data) but not malicious processes (verified with external knowledge via Google). After detection, post-IOC analyses were performed for false positive elimination, of which Section 5.4 gives an example. Lastly, three alerted RDIs were credited by the red team, which belonged to the operation of DoublePulsar — injections into `lsass.exe` and `winlogon.exe` from `rundll32.exe`.

5.2 Interactive Reasoning on FreeBSD

The main task of threat hunting is to interactively program human reasoning procedures, inspect the outcomes, and iteratively revise the threat hypotheses and the reasoning procedures based on observations and related knowledge. We performed such threat hunting procedure using τ-REPL and CG Browser during the competition. On the FreeBSD system, accesses to known sensitive files were checked periodically, e.g., `read` and `write` of `/etc/passwd`, `/etc/pwd.db`, and `/etc/spwd.db`, using simple τ-calculus patterns.

[6]This abstract label was provided by the monitoring team.

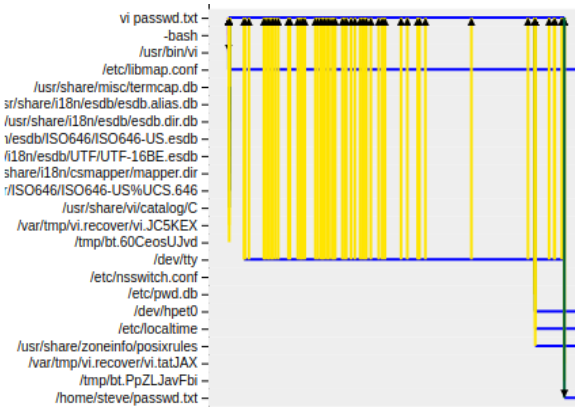


Figure 8: `vi` data movement shown in a CG Browser (visualization interpretation: Section 2.1 and Table 1).

```
pattern patFORW (startSubG, n) {
  en_start in startSubG
  en_start reach_n el_fwd
}
```

Figure 9: τ-calculus pattern for forward discovery.

The first suspicious access pinned by a threat hunter was matched by pattern `patPWDDB()` in Figure 7 — process `vi` read the FreeBSD user list database `/etc/pwd.db`, which is a *binary* file. By writing a new pattern in τ-REPL to match forward and backward events/entities connected to the process, the hunter noticed the `vi` process wrote into `/home/steve/passwd.txt` (looking like a *text* file) before it exited (Figure 8). Though nothing happened to `passwd.txt` for several hours, it was added to the tracked file list. On the next day, when inspecting the tracked files, the threat hunter applied a forward information tracking pattern in Figure 9 on `passwd.txt`. A chain of information flows was returned with process `fcgiwrap` and a `UNIX_SOCKET` feeding into each other at the end (Figure 10).

At this point, a data exfiltration alert was raised for this single attack step. We then wrote a set of τ-calculus functions to explore the bigger picture or the entire attack campaign around the data exfiltration path and provide answers to the following questions:

(1) *Why did not our automatic data leak detector fire?*
(2) *Where did the user, who used* `vi`*, come from?*
(3) *Did the user perform any other harmful actions?*
(4) *How did the* `fcgiwrap` *process started and is it benign?*
(5) *Are there any other files exfiltrated via* `fcgiwrap`*?*
(6) *Did the attacker use* `fcgiwrap` *for other type of attacks?*

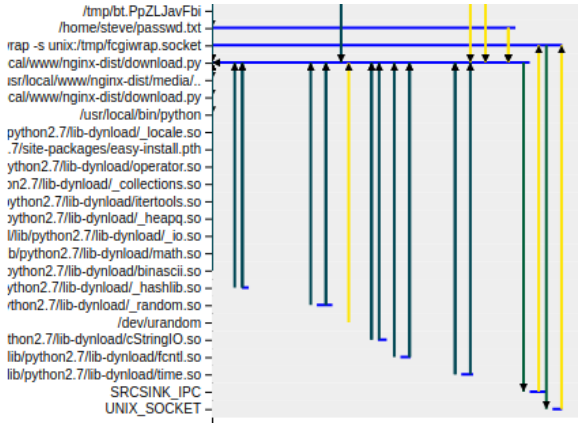


Figure 10: `passwd.txt` exfiltration shown in a CG Browser (visualization interpretation: Section 2.1 and Table 1).

Answers. (1) there is some missing information among `fcgiwrap`, `nginx`, and `UNIX_SOCKET`. The automatic data leak detector was fooled by the missed data, while the threat hunter obtained enough evidence from the name of `fcgiwrap` to conclude the leak; (2)/(3) the user logged in via ssh and did not have any other suspicious activities in that ssh session. (4) `fcgiwrap` was a forked worker from its parent process, which was launched via a `sudo` command in an ssh session where no suspicious activities such as privilege escalation actions were found. (5)/(6) several other file exfiltration activities were found from the sibling processes of the `fcgiwrap` worker process. While it was difficult for the automatic data leak detector to fire alerts (we had no knowledge of which files were sensitive on the target system), the opaque human knowledge that recognized sensitive keywords from the file names worked.

Lastly, the red team credited us for finding the data leaks and gave us the ground truth that one attacker knew about *i)* a vulnerability of the `download.py` FastCGI script for downloading arbitrary files via an online vulnerability database lookup and *ii)* sensitive file paths on the target machine via social engineering. `download.py` was normally installed and invoked by `fcgiwrap`.

5.3 Provenance Tracking on Android

It is straightforward to detect information leakage in τ-calculus due to its built-in traversal operator **reach**, employed as graph pattern construct. This case study illustrates the flexibility of τ-calculus when composing information leakage detection programs. A threat hunter should be able to dynamically allocate/modify the source, destination, and traversal constraints during a hunt regarding his growing knowledge about the underlying dataset.

Android owns its unique inter-process communication (IPC) mechanism, a.k.a., *Binder framework*. All system services, such as WiFi and notifications, are accessible to Apps through APIs on top of Binder. Instead of syscalls, the Android monitoring team provided us traces at the Binder level. For instance, a `read` event from a `Binder` entity had an attribute (encoded as a CG label) recording the name of the API call passed through Binder.

We composed several patterns in τ-calculus including the main pattern (Figure 11) to discover links between two subgraphs. `srcs` in

```
pattern patLeakDet (srcs, sinks, n, pathGuide) {
  ev_src  in  srcs
  en_sink  in  sinks
  ev_src reach_n el_path
  el_path reach_n en_sink
  el_path  =  pathGuide  (el_path)
}
```

Figure 11: Information leak detection pattern in τ-calculus.

`patLeakDet()` referred to a subgraph — selected sensitive sources described in another pattern, e.g., an event $\texttt{ev}_{wifi}$ that satisfies $\texttt{ev}_{wifi}$ **has** $\langle api : .*getConnectionInfo.* \rangle$ and comes from a `Binder` entity. `sinks` was matched with network entities. `n` was the maximum information-flow hops to check, and `pathGuide` was another pattern to guide the traversal with heuristics. This pattern combines two **reach** operators to function as a pathfinder (Section 4.3), and paths from the source to the sink subgraphs are stored in $\texttt{el}_{path}$.

One alert confirmed with `patLeakDet()` was a collusive leak: App `LobiwApp` requested WiFi information from a `Binder` entity via an event with label `getConnectionInfo`. `LobiwApp` wrote the information into `/storage/emulated/0/gather.txt`, which was then read by `SetexApp`. `SetexApp` leaked the information via an event `SEND_TO` into a `NETFLOW` entity with IP address `255.255.255.255`, i.e., broadcasting to hidden receiver(s).

5.4 Threat Evaluation on Linux

One common threat hunting task is to evaluate whether a given security alert is a false positive before further attack story discovery.

We used τ-calculus to manually evaluate a *download-and-execute* alert on a Linux workstation flagged by an external behavior detector: executable `tedit` was downloaded to disk through `Firefox` and then executed. The key question to answer was *whether this is a benign download-and-execute activity or a malicious one.* An effective way to answer it, given the CG at inter-process level[7], is to explore related activities to infer and compare user intentions with process behaviors. Using τ-REPL and CG Browser, we iteratively followed the steps below for evaluating the reported alert:

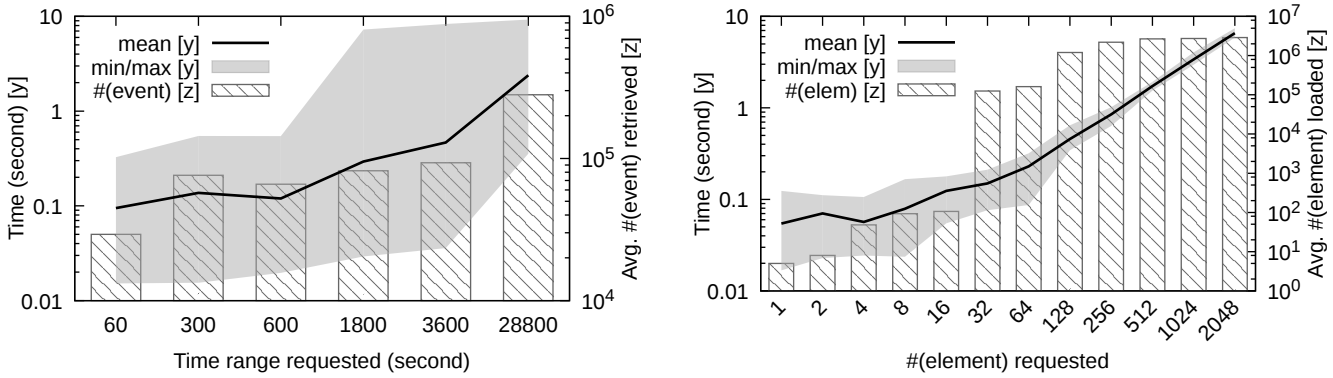
(1) Root cause analysis with backward discovery patterns
(2) Impact analysis with forward discovery patterns
(3) Backward/forward inspection of connected processes/files

We confirmed this alert to be a real threat within half an hour via τ-calculus threat hunting: the executed program connected and sent information to an external host `128.55.12.167:443`, and forked `dash` that executed `hostname`, `whoami`, `ifconfig`, and `netstat`. The behavior was suspicious considering the expected functionality (an editor) indicated by its name `tedit`.

6 POLICY REASONING WITH τ-CALCULUS

This section outlines a use case in which τ-calculus was used to provide reasoning for enforcing comprehensive policies on live systems. It was evaluated as a DARPA demonstration where several monitoring teams streamed live system traces to our platform. One

[7] Other means, e.g., drive-by download behavior identification [98], may require more fine-grained CGs at the process level.



(a) Event retrieval by time range.

(b) Element (entity/event) retrieval by UUID.

Figure 12: τ-calculus graph retrieval performance.

team who intercepted specific network requests on a monitored server queried our checking system with policies to check whether these requests should be blocked. Our policy checking system then reasoned over data from monitored clients which talked to the server and provided answers to the queries.

Our policy reasoning system employed τ-calculus for processing policy checking queries, which were translated into τ-calculus programs and evaluated. For instance, a query with policy *"whether the request was originated from a specific user"* generated a τ-calculus program with four graph patterns for (1) searching the request socket, (2) executing parameterized back traversal dealing with user change(s) via `sudo`, `setuid`, etc., (3) guiding the traversal along the call chain, and (4) halting the traversal when needed[8] to test for user match. Some policies were flow-sensitive (e.g., linking a download action and an upload action with more than 10 steps of operations, such as `tar`, `mv`, `gzip`, `unzip`, and `pipe`). Others were not about information flow (e.g., a C program triggered a network connection with `system("curl ...")` that created a chain of entities $\texttt{p} \xrightarrow{e_1} \texttt{sh} \xrightarrow{e_2} \texttt{curl} \xrightarrow{e_3} \texttt{socket}$; the policy required testing the existence of the process tree given network information about its leaves).

In summary, τ-calculus provided a comprehensive pattern composition mechanism that translated policy reasoning into graph searching problems similar to threat discovery. In the demo, our system answered all 26 queries[9] (on the same set of DARPA monitored systems as mentioned in Section 5) with 100% correctness, some of which are pre-programmed queries, and some are impromptu.

7 PERFORMANCE EVALUATION

This section reports on the performance throughput and scalability of τ-calculus, which supports both historical and real-time threat detection. For this evaluation, 161GB of two-week CG data from 6 monitored hosts (including the four listed in Table 4) were stored on four FCCE data nodes (CentOS VM with four virtual cores and 16GB memory) on four physical blades.

[8]Everything is started by `init` in Linux/FreeBSD with user `root`.
[9]Excluding queries with data issues.

We evaluated the fastest and the slowest graph queries to the τ-calculus database for throughput and scalability. With data locality tuning built into the low-level key-value store schema, the fastest/slowest graph queries are the ones associated with tight/loose data localities. They share similar concepts of sequential/random data access but in the context of distributed graph retrieval.

7.1 Graph Query by Event Time Range

Temporal locality on stores is one of the key features realized through the τ-calculus database layer (Section 4.4). Events occurring close in time are stored in one or nearby time slices, and they can be promptly retrieved for fast temporal reasoning.

We randomly selected time ranges of 60, 300, ..., 28800 seconds, and issued a simple τ-calculus query to retrieve a graph composed of all events within the time range (100 runs per each range with caches cleared between runs). During the DARPA competition, a monitored host emitted 100k events per hour on average. Figure 12a shows that it took on average 0.47s to retrieve all these events from storage across the network, construct, and return the graph.

7.2 Graph Query by Element UUIDs

Data locality does not always exist. For instance, it is difficult to define locality for a vital CG label—UUID. RFC 4122 defines UUID as a 128-bit number [48], and it is random and unique for each element (entity/event) in a CG. Element retrieval by UUID is to event retrieval by time range what random data access is to sequential data access. To process a graph query with multiple UUIDs, τ-calculus database retrieves multiple FCCE buckets that contain the requested UUIDs. Events and entities are constructed and additional information may be requested to complete the graph before return. On average, it took 0.46s to retrieve 128 elements shown in Figure 12b, and 1.2 million unrelated elements were also fetched by the FCCE client due to the lack of data locality. In this (graph queried by UUID) case with no data locality, τ-calculus graph database still displayed strong scalability in Figure 12b across distributed storage.

8 DISCUSSION

Beyond demonstrating the capabilities of threat intelligence computing with τ-calculus, this paper also aims to outline promising research opportunities enabled by the new security paradigm.

Incomplete Computation Graph. Incomplete data is the hurdle for any threat discovery procedure, including the ones realized with threat intelligence computing. It is non-trivial to implement a monitor that guarantees a complete view of a system at select monitoring levels. A new line of research pushes the state-of-the-art monitoring techniques to a new level [2, 26, 38, 49, 57, 72, 96], yet transmission and long-term storage are still open problems.

Given the realistic assumption that monitoring may not be complete due to implementation limitations, it raises another research challenge—how to deal with incomplete CGs in threat intelligence computing. Though analysts may fix missing information in an ad-hoc manner, e.g., Section 5.2, it is crucial to design a systematic solution to deal with missing data and strike a balance between data collection costs, e.g., computing and storage resources, and data quality, e.g., level of details and completeness.

Multi-Level Computation Graph. Threat intelligence often deals with data at various granularities and switches between them to fulfill different tasks. Threat intelligence computing offers a new opportunity for analyses that involve granularity switching—folding or unfolding CGs to easily zoom from one CG level to another.

Graph Pattern Constraint Solving. The current τ-calculus graph pattern constraint solver implementation employs heuristics to reorder constraints before solving. It is beneficial to research constraint ordering algorithms for achieving optimal constraint-solving procedure. This is challenging due to two main observations:

- *Data dependence of the problem*: the optimal constraint order on one CG may not be optimal on another CG.
- *Complex pattern composition*: pattern references and applications may invalidate inlining of constraints.

Machine Learning with Graph Languages. Many useful detection algorithms, especially the ones based on machine learning [11, 22], do not have straightforward graph language equivalents (even with Turing-complete graph languages like Gremlin). While the shortcut is to provide external function interfaces as explained in Section 2.3, it is beneficial to develop native data mining and machine learning algorithms in graph computation languages.

Higher-Order Graph Computation. Graph computation performed on top of a graph, e.g., CG, can be described as another graph [73]. One can collect large numbers of threat intelligence computing processes and apply graph computation on the high-order graphs for knowledge extraction and mining threat intelligence.

9 RELATED WORK

Threat discovery is the procedure of acquiring security knowledge from potential sources and applying it in the monitored environment to discover the computation steps resulting from attack campaigns. The security knowledge can be manually summarized and programmed into detection systems such as intrusion detection expert systems [14]. Some knowledge can also be mined by algorithms such as normal behavior models [16, 20, 50, 92], permitted behavior models [17], vulnerabilities [77], and specific attack models [98].

An orthogonal view to manual/automatic knowledge acquisition is static/dynamic threat model development. Threat detection is a never-ending game where new classes of threats are rapidly developed as well as variations of threats in different setups. While designing a detection system targeting a fixed set of threat models is an effective approach to threat detection, another proven practice is to dynamically adjust threat models and promptly create and test new threat hypotheses, a.k.a., threat hunting.

The major enabler from static to dynamic threat discovery is *agility* of threat hypotheses creation and verification. The dynamic paradigm heavily involves human for opaque knowledge and reasoning, which does not prohibit threat hunters from specifying new feature domains, programming automatic learning algorithms, and obtaining insight from the agilely created automatic detection modules. Note that no existing approach is fully autonomous: human knowledge for detection is *i)* embedded in the machine learning algorithm designs, and *ii)* employed to define learning domains — classification/clustering features or feature domains.

9.1 Static Threat Model Approaches

A large body of effective approaches have been developed with fixed threat models, which can be incorporated into threat intelligence computing as either security knowledge labels (e.g., anomaly scores of processes and sensitivity scores of files) or knowledge in reasoning algorithms (e.g., UDFs discussed in Section 2.2 and 2.3):

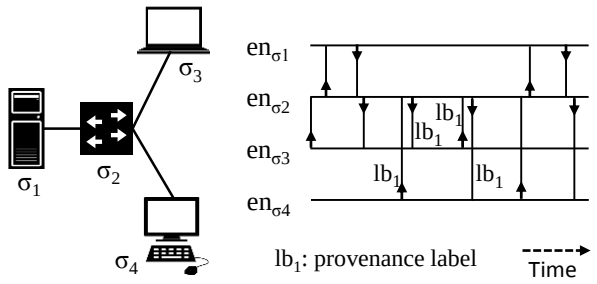
- Application of human-defined knowledge [3, 7, 30, 55, 87]
- Modeling trojan/ransomware behaviors [6, 41, 46, 89]
- Modeling botnet behaviors [5, 28, 37, 62, 97]
- Modeling malicious download behaviors [36, 45, 46, 84]
- Modeling malicious browser extension behaviors [40]
- Modeling malware behaviors [4, 18, 44, 51, 60, 86]
- Modeling malicious graph communities [39, 70, 91, 97]
- Modeling permitted behaviors [17, 24, 25, 29, 82, 88]
- Knowledge discovery on graphs [71, 81, 94]
- Attack causality tracking and inference [42, 47, 54, 85]
- Anomaly detection [16, 20, 23, 50, 58, 59, 61, 78, 80, 92]

Anomaly detection is a general threat model for identifying deviations from the training environment [10, 79]. It is useful but limited by training data quality (Section 5), manually specified feature domains, and the misalignment of *anomalous* and *malicious* alerts; and it does not replace other threat models. Though not existing yet, new anomaly detection algorithms are awaited to be developed in native Turing-complete graph languages (Section 8).

9.2 Dynamic Threat Model Approaches

Approaches with dynamic threat models have been introduced to deal with rapid threat variation/evolution and the creation/evaluation of new threat models in a prompt manner.

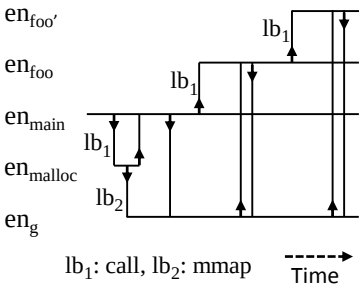
Existing threat hunting practices fulfill this need with a mash-up solution — importing security and non-security data of all kinds into a SIEM [31, 34] and employing SOC analysts for connecting the



(a) network topology (b) a computation graph

Figure 13: Example: CG at network level (link layer).

```
main () {
  g = malloc()
  *g = 1
  foo (g)
}
foo (x) {
  *x ++
  foo (x)
}
```



(a) sample code (b) a computation graph

Figure 14: Example: CG at process level (stack and heap).

dots with the human languages/concepts as the universal interface. The procedure is aided by static threat model approaches for well-modeled tasks, such as call chain traversal [9, 54] or knowledge standardization for retrieval and sharing [35, 52, 68, 83, 95].

Performing threat hunting through graph computations establishes new programmability requirements beyond existing graph programming platforms [12, 65] (discussed in Section 2). Automation based on existing threat hunting practices [8, 13, 27, 32, 33, 43, 66, 67], graph-based forensics [53, 63, 90], and temporal graph retrieval development [56, 93, 99] inspired the design of τ-calculus.

10 CONCLUSION

This paper introduces threat intelligence computing as a methodology for agile threat hypotheses composition and validation regarding dynamic threat models. By reshaping threat discovery into a graph computation problem, it eliminates heterogeneous data representation in different modules and provides an interactive programming environment for rapid automated task development and opaque human knowledge codification. We demonstrate the utility, practicality, and potential of the methodology by presenting the design, implementation, and evaluation of τ-calculus — a domain-specific graph computation platform designed for threat intelligence computing. Lastly, the paper sheds light on new challenges and opens further opportunities for future research and development in the realm of threat intelligence computing.

A COMPUTATION GRAPH AT DIFFERENT GRANULARITIES

Enterprises and organizations inspect computations at multiple levels for threat discovery. CG describes computations at a selected monitoring level, such as network, host, or process level. Given a monitoring level, e.g., network, the activities within an entity, e.g., process communications within a host, are usually out of the monitoring scope and not expressed in CG. Finer-grained computation information is either expressed in a lower-level CG, e.g., CG at the host level, or embedded into the CG as labels, e.g., provenance labels (Section 2.1). We describe CG examples at network and process levels in addition to the host-level CG (Figure 1):

(1) *CG at network level* (Figure 13): the metadata of link layer communications of a small network is logged for threat intelligence computing. $\texttt{lb}_1$ is a provenance label linking four events among $\texttt{en}_{\sigma 2}$, $\texttt{en}_{\sigma 3}$, $\texttt{en}_{\sigma 4}$. $\texttt{lb}_1$ helps identify the causal chain between $\texttt{en}_{\sigma 3}$ and $\texttt{en}_{\sigma 4}$ avoiding impossible paths. Attack steps such as port scans and cross-host lateral movements can be identified and reasoned on this CG.

(2) *CG at process level* (Figure 14): activities within a process are monitored via dynamic program analysis. Entities are memory addresses of code and data; events are instructions (e.g., `call`) or syscalls (e.g., `mmap`). The infinity of Θ supports the representation of recursive calls, e.g., instances of `foo()` are described as $\texttt{en}_{foo}$, $\texttt{en}'_{foo}$, $\cdots$. Software exploit activities such as return-to-libc and return-oriented programming (ROP) [76] can be captured and inspected on this CG.

ACKNOWLEDGMENTS

The authors would like to thank their DARPA teammates Dr. R. Sekar, Dr. Venkat Venkatakrishnan, and Dr. Yan Chen for collaborative detection and analytics. The authors thank all monitoring teams in the DARPA program for providing provenance data on a wide variety of systems. The authors thank the red team in the DARPA program for their effort in developing stealthy attacks and the infrastructure team for their responsive support. The authors would also like to thank the anonymous reviewers for their valuable comments and helpful suggestions.

This project was sponsored by the Air Force Research Laboratory (AFRL) and the Defense Advanced Research Agency (DARPA) under the award number FA8650-15-C-7561. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

[1] Luca Aceto and Andy Gordon. 2005. *Algebraic Process Calculi: The First Twenty Five Years and Beyond.* BRICS publications, Bertinoro, Forli, Italy.

[2] Adam Bates, Dave Tian, Kevin R B Butler, and Thomas Moyer. 2015. Trustworthy Whole-System Provenance for the Linux Kernel. In *Proceedings of USENIX Security Symposium.* ACM, Washington, DC, USA, 319–334.

[3] Mick Bauer. 2006. Paranoid Penguin: An Introduction to Novell AppArmor. *Linux Journal* 2006, 148 (Aug 2006), 13.

[4] Konstantin Berlin, David Slater, and Joshua Saxe. 2015. Malicious Behavior Detection Using Windows Audit Logs. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security (AISec '15).* ACM, Denver, Colorado, USA, 35–44.

[5] Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, and Christopher Kruegel. 2012. Disclosure: Detecting Botnet Command and Control Servers

Through Large-scale NetFlow Analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC) (ACSAC '12)*. ACM, Orlando, Florida, USA, 129–138.

[6] Kevin Borders and Atul Prakash. 2004. Web Tap: Detecting Covert Web Traffic. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*. ACM, Washington, DC, USA, 110–120.

[7] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. 2012. Towards Taming Privilege-Escalation Attacks on Android. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, California, USA.

[8] Ahmet Salih Buyukkayhan, Alina Oprea, Zhou Li, and William Robertson. 2017. Lens on the endpoint: Hunting for malicious software through endpoint data analysis. In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Springer International Publishing, Atlanta, USA, 73–97.

[9] Carbon Black. 2018. Cb Response | Incident Response & Threat Hunting | Carbon Black. Retrieved August 10, 2018 from https://www.carbonblack.com/products/cb-response/

[10] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly Detection: A Survey. *ACM Comput. Surv.* 41, 3 (July 2009), 15:1–15:58.

[11] Chen Chen, Cindy X. Lin, Matt Fredrikson, Mihai Christodorescu, Xifeng Yan, and Jiawei Han. 2009. Mining Graph Patterns Efficiently via Randomized Summaries. *Proc. VLDB Endow.* 2, 1 (Aug 2009), 742–753.

[12] DataStax. 2018. Titan: Distributed Graph Database. Retrieved August 10, 2018 from http://titan.thinkaurelius.com/

[13] Hervé Debar and Andreas Wespi. 2001. Aggregation and correlation of intrusion-detection alerts. In *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer International Publishing, Davis, CA, USA, 85–103.

[14] Dorothy E. Denning. 1987. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering* 13, 2 (February 1987), 222–232.

[15] Sean Dillon. 2017. DoublePulsar Initial SMB Backdoor Ring 0 Shellcode Analysis. Retrieved August 10, 2018 from https://zerosum0x0.blogspot.com/2017/04/doublepulsar-initial-smb-backdoor-ring.html

[16] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In *Proceedings of the 2017 ACM Conference on Computer and communications security (CCS)*. ACM, Dallas, Texas, USA, 1285–1298.

[17] H.H. Feng, J.T. Giffin, Yong Huang, S. Jha, Wenke Lee, and B.P. Miller. 2004. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*. IEEE Press, Oakland, California, USA, 194–208.

[18] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. 2017. Automated Synthesis of Semantic Malware Signatures using Maximum Satisfiability. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, California, USA.

[19] Stephen Fewer. 2013. Reflective DLL injection library. Retrieved August 10, 2018 from https://github.com/stephenfewer/ReflectiveDLLInjection

[20] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. 1996. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*. IEEE Press, Oakland, California, USA, 120–128.

[21] Thomas Fox-Brewster. 2017. An NSA Cyber Weapon Might Be Behind A Massive Global Ransomware Outbreak. Retrieved August 10, 2018 from https://www.forbes.com/sites/thomasbrewster/2017/05/12/nsa-exploit-used-by-wannacry-ransomware-in-global-explosion/

[22] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. 2010. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. In *2010 IEEE Symposium on Security and Privacy*. IEEE Press, Oakland, California, USA, 45–60.

[23] Hamid Reza Ghaeini, Daniele Antonioli, Ferdinand Brasser, Ahmad-Reza Sadeghi, and Nils Ole Tippenhauer. 2018. State-Aware Anomaly Detection for Industrial Control Systems. In *Proceedings of Security Track at the ACM Symposium on Applied Computing (SAC)*. ACM, Pau, France, 1620–1628.

[24] Jonathon T. Giffin, David Dagon, Somesh Jha, Wenke Lee, and Barton P. Miller. 2006. Environment-sensitive Intrusion Detection. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer International Publishing, Hamburg, Germany, 185–206.

[25] Rajeev Gopalakrishna, Eugene H. Spafford, and Jan Vitek. 2005. Efficient intrusion detection using automaton inlining. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*. IEEE Press, Oakland, California, USA, 18–31.

[26] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the 22th Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, California, USA.

[27] Guofei Gu, Alvaro A. Cárdenas, and Wenke Lee. 2008. Principled Reasoning and Practical Applications of Alert Fusion in Intrusion Detection Systems. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security (ASIACCS '08)*. ACM, Tokyo, Japan, 136–147.

[28] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. 2008. BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In *Proceedings of the 17th USENIX Security Symposium (Security'08)*. ACM, San Jose, CA, USA, 139–154.

[29] Zhongshu Gu, Kexin Pei, Qifan Wang, Luo Si, Xiangyu Zhang, and Dongyan Xu. 2015. LEAPS: Detecting Camouflaged Attacks with Statistical Learning Guided by Program Analysis. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE Press, Rio de Janeiro, Brazil, 57–68.

[30] Md Nahid Hossain, Sadegh M. Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R. Sekar, Scott Stoller, and V.N. Venkatakrishnan. 2017. SLEUTH: Real-time Attack Scenario Reconstruction from COTS Audit Data. In *26th USENIX Security Symposium (USENIX Security 17)*. ACM, Vancouver, BC, 487–504.

[31] HP. 2018. ArcSight - HP. Retrieved August 10, 2018 from http://www.hp.com/go/ArcSight

[32] C. Huber, P. McDaniel, S. E. Brown, and L. Marvel. 2016. Cyber Fighter Associate: A Decision Support System for cyber agility. In *2016 Annual Conference on Information Science and Systems (CISS)*. IEEE Press, Princeton, NJ, USA, 198–203.

[33] Ghaith Husari, Ehab Al-Shaer, Mohiuddin Ahmed, Bill Chu, and Xi Niu. 2017. TTPDrill: Automatic and Accurate Extraction of Threat Actions from Unstructured Text of CTI Sources. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC 2017)*. ACM, Orlando, FL, USA, 103–115.

[34] IBM. 2018. IBM QRadar Security Intelligence Platform. Retrieved August 10, 2018 from http://www.ibm.com/software/products/en/qradar

[35] IBM. 2018. IBM X-Force. Retrieved August 10, 2018 from https://www.ibm.com/security/xforce/

[36] Luca Invernizzi, Stanislav Miskovic, Ruben Torres, Sabyasachi Saha, Sung-Ju Lee, Marco Mellia, Christopher Kruegel, and Giovanni Vigna. 2014. Nazca: Detecting Malware Distribution in Large-Scale Networks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, California, USA.

[37] Gregoire Jacob, Ralf Hund, Christopher Kruegel, and Thorsten Holz. 2011. JACKSTRAWS: Picking Command and Control Connections from Bot Traffic. In *USENIX Security Symposium*. ACM, San Francisco, CA, USA, 29–29.

[38] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2017. RAIN: Refinable Attack Investigation with On-demand Inter-Process Information Flow Tracking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, Dallas, Texas, USA, 377–390.

[39] Xuxian Jiang, A. Walters, Dongyan Xu, E. H. Spafford, F. Buchholz, and Yi-Min Wang. 2006. Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*. IEEE Press, Lisboa, Portugal, 38–38.

[40] Alexandros Kapravelos, Chris Grier, Neha Chachra, Chris Kruegel, Giovanni Vigna, and Vern Paxson. 2014. Hulk: Eliciting Malicious Behavior in Browser Extensions. In *Proceedings of USENIX Security Symposium*. ACM, San Diego, CA, USA, 641–654.

[41] Amin Kharraz, Sajjad Arshad, Collin Mulliner, William K Robertson, and Engin Kirda. 2016. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware. In *Proceedings of USENIX Security Symposium*. ACM, Vancouver, BC, Canada, 757–772.

[42] Samuel T. King and Peter M. Chen. 2003. Backtracking Intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, Bolton Landing, New York, 223–236.

[43] Samuel T King, Zhuoqing Morley Mao, Dominic G Lucchetti, and Peter M Chen. 2005. Enriching Intrusion Alerts Through Multi-Host Causality. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, California, USA.

[44] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. 2009. Effective and Efficient Malware Detection at the End Host. In *Proceedings of the 18th Conference on USENIX Security Symposium (SSYM'09)*. ACM, Montreal Canada, 351–366.

[45] Bum Jun Kwon, Jayanta Mondal, Jiyong Jang, Leyla Bilge, and Tudor Dumitras. 2015. The Dropper Effect: Insights into Malware Distribution with Downloader Graph Analytics. In *Proceedings of the 2015 ACM Conference on Computer and Communications Security (CCS)*. ACM, Denver, Colorado, US, 1118–1129.

[46] Bum Jun Kwon, Virinchi Srinivas, Amol Deshpande, and Tudor Dumitras. 2017. Catching Worms, Trojan Horses and PUPs: Unsupervised Detection of Silent Delivery Campaigns. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, California, USA.

[47] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela Ciocarlie, et al. 2018. MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, California, USA.

[48] P. Leach, M. Mealling, and R. Salz. 2005. RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace. Retrieved August 10, 2018 from https://tools.ietf.org/html/rfc4122

[49] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*. The Internet Society,

San Diego, California, USA.
[50] Wenke Lee and Salvatore J. Stolfo. 1998. Data Mining Approaches for Intrusion Detection. In *Proceedings of USENIX Security Symposium*, Vol. 7. ACM, San Antonio, Texas, 6–6.
[51] Charles Lever, Manos Antonakakis, Bradley Reaves, Patrick Traynor, and Wenke Lee. 2013. The Core of the Matter: Analyzing Malicious Traffic in Cellular Carriers. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, California, USA.
[52] Xiaojing Liao, Kan Yuan, XiaoFeng Wang, Zhou Li, Luyi Xing, and Raheem Beyah. 2016. Acing the IOC Game: Toward Automatic Discovery and Analysis of Open-Source Cyber Threat Intelligence. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS) (CCS '16)*. ACM, Vienna, Austria, 755–766.
[53] Richard Lippmann, Kyle Ingols, Chris Scott, Keith Piwowarski, Kendra Kratkiewicz, Mike Artz, and Robert Cunningham. 2006. Validating and restoring defense in depth using attack graphs. In *Military Communications Conference, 2006. MILCOM 2006. IEEE*. IEEE Press, Washington, DC, USA, 1–10.
[54] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. 2018. Towards a Timely Causality Analysis for Enterprise Security. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, California, USA.
[55] Teresa F Lunt, Ann Tamaru, and F Gillham. 1992. *A real-time intrusion-detection expert system (IDES)*. SRI International Computer Science Laboratory, Menlo Park, CA, USA.
[56] Shuai Ma, Renjun Hu, Luoshu Wang, Xuelian Lin, and Jinpeng Huai. 2017. Fast Computation of Dense Temporal Subgraphs. In *Proceedings of the 2017 IEEE International Conference on Data Engineering*. IEEE Press, San Diego, California, USA, 361–372.
[57] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. 2016. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *Proceedings of the 23th Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, California, USA.
[58] Emaad Manzoor, Sadegh M Milajerdi, and Leman Akoglu. 2016. Fast Memory-efficient Anomaly Detection in Streaming Heterogeneous Graphs. In *ACM SIGKDD International Conference on Knowledge Discovery and Data mining*. ACM, San Francisco, CA, US, 1035–1044.
[59] J. Mao, J. Bian, G. Bai, R. Wang, Y. Chen, Y. Xiao, and Z. Liang. 2018. Detecting Malicious Behaviors in JavaScript Applications. *IEEE Access* 6 (Jan 2018), 12284–12294.
[60] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, et al. 2017. Deep Android Malware Detection. In *Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy*. ACM, Scottsdale, Arizona, USA, 301–308.
[61] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. 2006. Anomalous System Call Detection. *ACM Transactions on Information and System Security* 9, 1 (February 2006), 61–93.
[62] Shishir Nagaraja, Prateek Mittal, Chi-Yao Hong, Matthew Caesar, and Nikita Borisov. 2010. BotGrep: Finding P2P Bots with Structured Graph Analysis. In *USENIX Security Symposium*. ACM, Washington, DC, USA, 95–110.
[63] Milad Nasr, Amir Houmansadr, and Arya Mazumdar. 2017. Compressive Traffic Analysis: A New Paradigm for Scalable Traffic Analysis. In *Proceedings of the 2017 ACM Conference on Computer and Communications Security (CCS)*. ACM, Dallas, Texas, USA, 2053–2069.
[64] Neo Technology. 2018. Cypher Query Language. Retrieved August 10, 2018 from https://neo4j.com/developer/cypher/
[65] Neo Technology. 2018. Neo4j Graph Platform. Retrieved August 10, 2018 from https://neo4j.com/
[66] Peng Ning and Dingbang Xu. 2003. Learning Attack Strategies from Intrusion Alerts. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*. ACM, Washington, DC, USA, 200–209.
[67] S. Noel, E. Robertson, and S. Jajodia. 2004. Correlating intrusion events and building attack scenarios through attack graph distances. In *Proceedings of the 20th Annual Computer Security Applications Conference*. ACM, Tucson, Arizona, USA, 350–359.
[68] OASIS. 2018. The OASIS Cyber Threat Intelligence. Retrieved August 10, 2018 from https://oasis-open.github.io/cti-documentation/
[69] Jim O'Gorman, Devon Kearns, and Mati Aharoni. 2011. *Metasploit: The penetration tester's guide*. No Starch Press, San Francisco, CA, USA.
[70] A. Oprea, Z. Li, T. F. Yen, S. H. Chin, and S. Alrwais. 2015. Detection of Early-Stage Enterprise Infection by Mining Large-Scale Log Data. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE Press, Toulouse, France, 45–56.
[71] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. 2016. HERCULE: Attack Story Reconstruction via Community Discovery on Correlated Log Graph. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC) (ACSAC '16)*. ACM, Los Angeles, California, USA, 583–595.
[72] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. 2012. Hi-Fi: collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*. ACM, Orlando, Florida, USA, 259–268.
[73] Marko A. Rodriguez. 2015. The Gremlin Graph Traversal Machine and Language (Invited Talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. ACM, Pittsburgh, PA, USA, 1–10.
[74] D. L. Schales, X. Hu, J. Jang, R. Sailer, M. P. Stoecklin, and T. Wang. 2015. FCCE: Highly scalable distributed Feature Collection and Correlation Engine for low latency big data analytics. In *Proceedings of the 31st IEEE International Conference on Data Engineering*. IEEE Press, Seoul, Korea, 1316–1327.
[75] Daniel G. Schwartz. 2015. Dynamic Reasoning Systems. *ACM Trans. Comput. Logic* 16, 4 (Nov 2015), 32:1–32:42.
[76] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*. ACM, Alexandria, VA, USA, 552–561.
[77] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. 2017. Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance. In *Proceedings of the 2017 ACM Conference on Computer and communications security (CCS)*. ACM, Dallas, Texas, USA, 347–362.
[78] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. 2015. Unearthing Stealthy Program Attacks Buried in Extremely Long Execution Paths. In *Proceedings of the 2015 ACM Conference on Computer and communications security (CCS)*. ACM, Denver, Colorado, US, 401–413.
[79] Xiaokui Shu, Danfeng Yao, and Barbara G. Ryder. 2015. A Formal Framework for Program Anomaly Detection. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Springer, Hamburg, Germany, 270–292.
[80] Xiaokui Shu, Danfeng (Daphne) Yao, Naren Ramakrishnan, and Trent Jaeger. 2017. Long-Span Program Behavior Modeling and Attack Detection. *ACM Trans. Priv. Secur.* 20, 4 (September 2017), 12:1–12:28.
[81] Milivoj Simeonovski, Giancarlo Pellegrino, Christian Rossow, and Michael Backes. 2017. Who controls the internet? Analyzing global threats using property graph traversals. In *Proceedings of the 26th International Conference on World Wide Web*. ACM, Perth, Western Australia, 647–656.
[82] M. Smith, F. Schwarzer, M. Harbach, T. Noll, and B. Freisleben. 2009. A Streaming Intrusion Detection System for Grid Computing Environments. In *Proceedings of the 11th IEEE International Conference on High Performance Computing and Communications*. IEEE Press, Seoul, Korea, 44–51.
[83] spiderfoot. 2018. SpiderFoot. Retrieved August 10, 2018 from http://www.spiderfoot.net/
[84] Gianluca Stringhini, Yun Shen, Yufei Han, and Xiangliang Zhang. 2017. Marmite: Spreading Malicious File Reputation Through Download Graphs. In *Proceedings of the 2017 Annual Conference on Computer Security Applications (ACSAC)*. ACM, San Juan, Puerto Rico, 91–102.
[85] X. Sun, J. Dai, P. Liu, A. Singhal, and J. Yen. 2016. Towards probabilistic identification of zero-day attack paths. In *2016 IEEE Conference on Communications and Network Security (CNS)*. IEEE Press, Philadelphia, PA, USA, 64–72.
[86] Kimberly Tam, Salahuddin J. Khan, Aristide Fattoriy, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the 23th Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, California, USA.
[87] Lawrence Teo and Gail-Joon Ahn. 2013. Extensible Policy Framework for Heterogeneous Network Environments. *Int. J. Inf. Comput. Secur.* 5, 4 (Dec 2013), 251–274.
[88] D. Wagner and D. Dean. 2001. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*. IEEE Press, Oakland, California, USA, 156–168.
[89] Hao Wang, Somesh Jha, and Vinod Ganapathy. 2006. NetSpy: Automatic Generation of Spyware Signatures for NIDS. In *22nd Annual Computer Security Applications Conference (ACSAC)*. ACM, Los Angeles, California, USA, 99–108.
[90] Wei Wang and Thomas E. Daniels. 2008. A Graph Based Approach Toward Network Forensics Analysis. *ACM Trans. Inf. Syst. Secur.* 12, 1 (October 2008), 4:1–4:33.
[91] Yong Wang, Zhaoyan Xu, Jialong Zhang, Lei Xu, Haopei Wang, and Guofei Gu. 2014. SRID: State Relation based Intrusion Detection for False Data Injection Attacks in SCADA. In *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS'14)*. Springer International Publishing, Wroclaw, Poland, 401–418.
[92] C. Warrender, S. Forrest, and B. Pearlmutter. 1999. Detecting intrusions using system calls: alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*. IEEE Press, Berkeley, California, USA, 133–145.
[93] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. 2016. High Fidelity Data Reduction for Big Data Security Dependency Analyses. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*.

ACM, Vienna, Austria, 504–516.

[94] Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William Robertson, Ari Juels, and Engin Kirda. 2013. Beehive: Large-scale Log Analysis for Detecting Suspicious Activity in Enterprise Networks. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC) (ACSAC '13)*. ACM, New Orleans, Louisiana, USA, 199–208.

[95] yeti community. 2018. YETI: Your Everyday Threat Intelligence. Retrieved August 10, 2018 from https://yeti-platform.github.io/

[96] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, Washington, DC, USA, 116–127.

[97] J. Zhang, S. Saha, G. Gu, S. J. Lee, and M. Mellia. 2015. Systematic Mining of Associated Server Herds for Malware Campaign Discovery. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE Press, Columbus, OH, USA, 630–641.

[98] Junjie Zhang, Christian Seifert, Jack W Stokes, and Wenke Lee. 2011. Arrow: Generating signatures to detect drive-by downloads. In *Proceedings of the 20th international conference on World wide web*. ACM, Lyon, France, 187–196.

[99] Bo Zong, Xusheng Xiao, Zhichun Li, Zhenyu Wu, Zhiyun Qian, Xifeng Yan, Ambuj K. Singh, and Guofei Jiang. 2015. Behavior Query Discovery in System-generated Temporal Graphs. *VLDB Endow.* 9, 4 (Dec 2015), 240–251.