Journal of Computer Virology and Hacking Techniques (2018) 14:291–311
https://doi.org/10.1007/s11416-018-0318-x

ORIGINAL PAPER

SEQUIN: a grammar inference framework for analyzing malicious system behavior

Robert Luh[1,2] · Gregor Schramm[1] · Markus Wagner[3] · Helge Janicke[2] · Sebastian Schrittwieser[1]

Received: 11 July 2017 / Accepted: 14 March 2018 / Published online: 26 March 2018

Abstract
Targeted attacks on IT systems are a rising threat to the confidentiality of sensitive data and the availability of critical systems. The emergence of Advanced Persistent Threats (APTs) made it paramount to fully understand the particulars of such attacks in order to improve or devise effective defense mechanisms. Grammar inference paired with visual analytics (VA) techniques offers a powerful foundation for the automated extraction of behavioral patterns from sequential event traces. To facilitate the interpretation and analysis of APTs, we present SEQUIN, a grammar inference system based on the Sequitur compression algorithm that constructs a context-free grammar (CFG) from string-based input data. In addition to recursive rule extraction, we expanded the procedure through automated assessment routines capable of dealing with multiple input sources and types. This automated assessment enables the accurate identification of interesting frequent or anomalous patterns in sequential corpora of arbitrary quantity and origin. On the formal side, we extended the CFG with attributes that help describe the extracted (malicious) actions. Discovery-focused pattern visualization of the output is provided by our dedicated KAMAS VA prototype.

Keywords Malware analysis · System behavior · Attribute grammar · Knowledge generation · Visual analytics

1 Introduction

IT systems are threatened by a growing number of cyber-attacks. With the emergence of Advanced Persistent Threats (APTs), the focus shifted from off-the-shelf malware to attacks that are tailored to one specific entity. These targeted threats are driven by varying motivations, such as espionage or high-profile sabotage, and often cause significantly more damage.

APTs are typically conducted by dedicated groups within organized crime, industry, or nation state intelligence and increasingly affect less prominent targets as well. In 2013 alone, "economic espionage and theft of trade secrets cost the American economy more than $19 billion" [35]. 60% of espionage attacks now target small and medium businesses whereas each reported data breach exposes over a million identities on average [51]. The retail, healthcare, and finance sectors find themselves in the crosshairs most often.

While APTs utilize malware like most other, more conventional attacks, their level of complexity and sophistication is usually much higher. This is problematic especially since defensive measures offered by security vendors typically employ the same signature-based detection approaches that have been used for years. The major drawback of these systems is that the binary patterns required for detection are unlikely to exist at the time of attack, since most APTs are tailored to one specific target and often utilize zero-day exploits [7,48]. In addition, meta- and polymorphic techniques, as well as packers and encryption routines may throw off signature-based systems while the multi-stage nature of

✉ Robert Luh
robert.luh@fhstp.ac.at

Gregor Schramm
gregor.schramm@fhstp.ac.at

Markus Wagner
markus.wagner@fhstp.ac.at

Helge Janicke
heljanic@dmu.ac.uk

Sebastian Schrittwieser
sebastian.schrittwieser@fhstp.ac.at

1 Josef Ressel Center TARGET, St. Pölten University of Applied Sciences, St. Pölten, Austria

2 De Montfort University, Leicester, UK

3 Institute for Creative Media Technologies, St. Pölten University of Applied Sciences, St. Pölten, Austria

APTs makes it generally difficult to interpret findings without additional context [18].

This increased complexity makes it necessary to explore novel techniques for threat intelligence and malicious activity detection on multiple layers. Behavior-based approaches [27] are a promising means to identify and understand illegal actions. Despite the stealth mechanisms employed, the attacker will sooner or later execute his or her action on target – be it data theft, sabotage, or other fraudulent activity. Behavior patters and anomalies signifying a deviation from a known baseline can then be used to detect such a threat.

However, both pattern and anomaly detection systems usually suffer from a lack of semantic interpretation; the so-called *semantic gap*, the hard-to-bridge difference in syntactic event information and actual attack semantics, remains an issue. Patterns of binary appearance or execution behavior are often manually assigned to represent analyst knowledge, while anomaly detection systems do not usually attempt to explain the identified deviations. This makes potential victims vulnerable to unknown attacks and does little to further the exploration of meaning and intent behind the actions of a malicious actor.

Successfully discovering potentially harmful system behavior boils down to three major problem domains: i) the automated generation of patterns that contribute to detecting and understanding complex multi-stage attacks, ii) attack semantics, and iii) the holistic view on targeted attacks and their many properties. Arguably, a powerful formal definition of malicious behavior is the foundation for addressing most of these aspects.

In this article, which is an extended and improved version of [29], we propose a novel IT system behavior inference and classification methodology based on the Sequitur algorithm [37], which we formalize through a context-free grammar (CFG) extended by semantic attributes (attribute grammar). The approach combines a condensed formal definition with the generation of knowledge linked to the information security and malware analysis domains. Instead of manually defining the many terminals and production rules that the description of a behavior trace would require, we automate the process through an extension of Sequitur that is fully capable of determining and evaluating significant rules. This, together with our output visualization and analytics system, eliminates the analysts' need to define fixed patterns describing harmful or benign behavior.

Specifically, we contribute by:

- Defining an attribute grammar capable of depicting sequential behavior while retaining information about the triggering process and its parameters,
- Developing a grammar inference framework based on the Sequitur algorithm capable of performing input data compression and anomaly detection on arbitrary system traces,
- Expanding this approach to a knowledge discovery system supporting automated evaluation and extraction of potentially interesting patterns through our novel KAMAS visualization tool.

The remainder of this paper is structured as follows: In Section 2, similar works in the area of security-related inference are reviewed. In Section 3, the specifics of grammar inference and available algorithms are discussed. We furthermore introduce our input event data as well as the developed attribute grammar for describing (malicious) system activity, and present our inference algorithm of choice. Applied grammar inference and data analysis procedures are detailed in Section 4. Our implementation, which includes a visual knowledge extraction prototype (Sections 5 and 7), as well as several evaluated applications of the approach (Section 6) conclude the article.

2 Related work

In light of the large number of operating systems and programming languages currently available, a universal means of abstraction and classification of malicious behavior into a more generic representation is paramount. [22] present a detection system based on attribute grammars, where syntactic rules describe possible combinations of operations constituting certain behavior, while semantic rules control the data flow between events and assign general meaning to a sequence. Jacob et al's system is intended as formal foundation for developing robust intrusion and malware detection automata. On the modeling side, Filiol et al. [18] propose a generalized model for malware recognition which considers both sequence-based and behavior-based detection. An evaluation methodology for behavioral engines of existing products is proposed. The major difference to SEQUIN is the system's reliance on manually defined behavioral rules. Jacob et al's approach focuses on specified duplication and propagation behavior of investigated PE and VBA samples. Close to 20 Windows native API calls are directly mapped to interaction classes (create, read, write, etc.) and object types such as file, registry, or network operations. SEQUIN's grammar inference automates this process, but relies on an effective naming schema (see 4.4) to retain the semantic link.

In general, the discovery of program behavior is key to understanding benign and malicious software. Zhao et al. [62] present a semi-automatic graph grammar approach to retrieving the hierarchical structure of an application's activity. This is achieved by mining recurring behavioral patterns from execution traces using VEGGIE with SubdueGL [4,5],

a Minimum Description Length (MDL)-based compression algorithm. The inferred graph grammar and a syntactic parse tree visually represent reused structures found. Unlike SEQUIN, Zhao et al.'s semi-automated approach uses a more computationally complex context-sensitive grammar to identify common call subgraphs, which are ultimately used in code verification scenarios.

Joo and Chellappa [24] introduce a method for representing and recognizing specific event anomalies in a video by using attribute grammars. This limited domain makes it possible to model most of the events expected to occur and to define anomalies that do not fit the model. Matches are represented by degree of certainty expressed as a probability. While Joo and Chellappa's and our system share the formal foundation, they differ significantly in inference capabilities and automation.

Thompson and Flynn [53] use a similar algorithm to detect and identify the polymorphic instances of a given malware. The approach represents program structure as a context-free grammar, and compares grammars by checking for homomorphism between them. The system makes it possible to identify variants of software by abstracting the control flow of the code. Non-structural elements are removed and the complexity of code quantified by counting the number of remaining elements within the function. For comparison, the resulting grammar is serialized (similar to SEQUIN's zero rule) and checked against another string. Inference mechanisms or CFG attributes for function arguments are not employed. Thompson and Flynn's purely theoretical approach does not specify concrete algorithms or evaluation scenarios.

On the more traditional anomaly detection side, Creech and Hu [12] introduce a host-based detection method that uses discontiguous system call patterns. The authors use a context-free grammar to describe (but not infer) benign and malicious call traces. Several decision engines were tested and compared in the paper, making it a good starting point for the selection of learning algorithms applicable to system call sequences.

In a patent submitted by Eiland et al. [17], the authors describe an intrusion masquerade detection system that includes a grammar inference engine based on MDL compression. The compression algorithm is applied to sets of input data to build user-specific grammars. The use of intrusion masquerade is ultimately based on the determined distance between template and observed algorithmic minimum sufficient statistic.

Visualization is a predominant theme in this field. With GrammarViz, Senin et al. [45] introduce a grammar mining and visualization tool based on CFG induction. While GrammarViz does not specifically consider attributes or malicious software scenarios in general, it describes a practical approach to manually analyzing time series data. In a more recent paper, Senin et al. [46] expand on the concept of algorithmic incompressibility for anomaly detection and present practical examples using spatial trajectory data. Senin et al.'s Sequitur-based system heavily relies on the ratio between rules and terminals for identifying anomalies. Unlike SEQUIN, the employed visualization tool does not fully support internal pattern extraction and classification while primarily depicting time series charts.

Specific grammar inference algorithms considered during the design stages of SEQUIN are introduced in Section 3.1 below.

3 Preliminaries

In this chapter, we introduce the grammar inference background of our solution and specify type of system event data used as the foundation of semantic pattern analysis. Sequitur, as our compression algorithm of choice, is used to determine patterns of interest. Furthermore, a formal definition of the information used is presented in the form of an attribute grammar.

3.1 Grammar inference

Grammar inference is the process of automatically learning a grammar by examining the sentences of an unknown language [50]. In the IT sector, grammar inference is primarily used for pattern recognition, computational biology, natural language processing, language design programming, data mining, and machine learning. The effectiveness of grammar inference is influenced by the language class and by the information available about the target language. To raise the effectiveness of grammar inference, a combination of learning modules and language classes are used. Grammar inference has been proven to be a feasible approach to anomaly detection, since "algorithmic incompressibility is a necessary and sufficient condition for randomness" [34]. We use grammar inference as key component in the process of compressing a sequential trace for extracting relevant behavioral patterns.

The main *issues* of grammar inference are over-specialization and over-generalization. Over-specialization (over-fitting) describes the problem when the inference process produces a grammar whose language is smaller than the unknown target language. This is typically countered by defining an appropriate validation set from the available data and by measuring the performance on this data after each training example has been processed [15]. Over-generalization occurs when the inference process produces a grammar whose language is larger than the unknown language. The use of negligible items results in an unnecessarily

Table 1 Grammar inference algorithms and applications by category

	supervised	semi-supervised	unsupervised
statistical	Co-training [49]	Self-training [32]	ADIOS [47]
evolutionary	GA-based [42]		LAgts [8]
heuristic	ALLiS [13]		Inductive CYK [36]
			ABL [54]
MDL			e-GRIDS [38]
			CDC [10]
			VEGGIE [4,5]
			Eiland et al. [17]
greedy search			ADIOS
			CDC
			Incremental parsing [3,44]
			Sequitur [37]
			GraphViz [45,46]
clustering	EMILE [1]		CDC

large grammar. To limit the impact of over-generalization, it is recommended to also use a set of negative examples.

There are various *computational techniques* suitable for grammar inference. DUlizia et al. [15] surveyed various algorithms and categorized them into six groups:

- **Statistical methods** use probability distribution in a class of models derived from empirical data generally provided by a large body of text. Applications include self-training [32] and co-training [49]. ADIOS [47] also uses statistical information to derive regularities from sentences.
- **Evolutionary computing techniques**, often used in computational biology, regularly update (evolve) the initial model or grammar. Each new iteration is produced by removing less desired solutions. GA-based approaches and e.g. LAgts [8] both use genetic algorithms to eliminate unnecessary non-terminal symbols and production rules from the grammar.
- **Heuristic methods** generate training examples of sentences. In grammar inference, ALLiS [13] uses heuristics to reduce the number of similar rules as well as for selecting rules that have the most content. ABL [54] finds, with the help of heuristics, the longest common sequence shared between sentences.
- **Minimum description length** (MDL) [40] assumes that the simplest, most compact representation of data is its best and most probable depiction. The principle finds its primary application in data reduction, where "any regularity in a given set of data can be used to compress the data" [20]. Examples include CDC [10] and e-GRIDS [38].
- **Greedy search algorithms** make decisions based on their internal logic which may lead to the creation, removal or fusion of rules. For example, Sequitur [37] recursively replaces same-character sequences with new production rules. It produces a grammar that reflects repetitions and thereby infers the hierarchical structure of the grammar. ADIOS [47] also applies a greedy learning algorithm to its graph representation of sentences.
- **Clustering techniques** require a starting grammar that contains all possible sentences. They subsequently cluster syntactic units until the grammar has been constructed. For example, EMILE [1] clusters expressions that occur in the same context, while CDC [10] creates sets of sequences within a context before selecting clusters that satisfy the MDL principle (see above).

Grammar inference algorithms are further classified by their learning approach [15]. Refer to Table 1 for an overview of the above-mentioned systems, among others. In subsection 3.2 below, we further discuss the choice of utilizing a greedy, unsupervised inference approach for our prototype.

3.2 Algorithm selection

We have identified three prerequisites for the successful identification and extraction of interesting behavior from a trace:

- **Unsupervised learning** – The learning module used for generating knowledge from malicious system behavior must be unassisted. Human intervention in the decision of whether a grammar is valid or not would contradict the automation requirements set by most analysts.
- **Context-free grammar** (CFG) – A compromise between a regular and a context-sensitive grammar, CFGs offer a good balance between ease of parsing and computational efficiency. The language created by a CFG can be recognized in $O(n^3)$ time, which will prove helpful in future

parsing efforts. This selection prerequisite was complemented by the decision to use an attribute grammar for formal representation (see 3.3).

– **Loss-less operation** – It is vital that the algorithm employed does not change the order or immutability of events, since both is likely to have an impact on the semantics of a behavioral sequence.

After surveying numerous algorithms such as the ones listed in Section 3.1 and Table 1, we decided against using an approach that uses equivalence classes in its vocabulary (like ADIOS [47]). While this feature might be interesting for studying mimicry attacks or the use of equivalent API functions at a later point, the first prototype needed to be precise and deterministic in its inference process in order to enable a more expressive evaluation of the general approach. The choice fell on Sequitur.

3.2.1 Sequitur

Sequitur is a greedy compression algorithm that creates a hierarchical structure (CFG) from a sequence of discrete symbols by recursively replacing repeated phrases with a grammatical rule [37]. The output is a representation of the original sequence, which effectively results in the creation of a context-free grammar. The algorithm creates this representation through two essential properties, which are called *rule utility* and *bigram uniqueness*. Rule utility checks if a rule occurs at least twice in the grammar, while bigram uniqueness observes if a bigram occurs only once. A bigram in this context describes two adjacent symbols or terms. Assuming we have a string `abcdbcabcd`, the first bigram would be `ab`, followed by a second bigram `bc`, and so forth. See Table 2 for a complete example of the process.

Sequitur is linear in space and time. In terms of data compression, the algorithm can outperform other designs that achieve data reduction by factoring out repetition. It is almost as performant as designs that compress data based on probabilistic predictions [37].

See Section 6.1 for an evaluation of processing time and RAM requirements of our adapted implementation. Generally, the system introduced in this paper not only applies the Sequitur algorithm, but also statistically evaluates the inferred grammar. Furthermore, it automatically highlights rules describing potentially relevant behavior, enabling anomaly or similarity detection (see 6.2).

Table 2 Operation of Sequitur after Nevill-Manning and Witten [37]). Property application is *italicized*

Sym	String	Grammar	Remarks
1	a	S → a	
2	ab	S → ab	
3	abc	S → abc	
4	abcd	S → abcd	
5	abcdb	S → abcdb	
6	abcdbc	S → abcdbc	bc appears 2x
		S → aAdA	*bigram uniq.*
		A → bc	
7	abcdbca	S → aAdAa	
		A → bc	
8	abcdbcab	S → aAdAab	
		A → bc	
9	abcdbcabc	S → aAdAabc	bc reappears
		A → bc	
		S → aAdAaA	*bigram uniq.*
		A → bc	aA appears 2x
		S → BdAB	*bigram uniq.*
		A → bc	
		B → aA	
10	abcdbcabcd	S → BdABd	Bd appears 2x
		A → bc	
		B → aA	
		S → CAC	*bigram uniq.*
		A → bc	B used only 1x
		B → aA	
		C → Bd	
		S → CAC	*rule utility*
		A → bc	
		C → aAd	

3.3 Formal definition

To enable the conversion of any kind of trace into an applicable ruleset for behavioral classification, it is necessary to formally define relevant (malicious) actions through distinct patterns that can be integrated into a grammatical hierarchy. Unlike many other solutions (see 2), we do not manually map system activity to concrete events but use Sequitur-enabled inference to automatically determine likely rules.

For this purpose, our system uses a context-free grammar extended by attributes, known as attribute grammar [2]. This decision followed an in-depth review of several grammars and languages, including graph grammars [6], state transition graphs based on NLC [41], trace languages [21], and the aforementioned attribute grammars.

Graph grammars (also known as graph rewriting systems) were identified as the main competitor to our final specification. Following the notation introduced by Benteler [6], we define a graph grammar as $GG = (N, T \cup \Delta, P, S)$, where node labels $n \in N$ are non-terminals and $t \in T$ are terminals, while the respective edge labels e are part of alphabet Δ. $S \in N$ describes the start axiom whereas a production

$p \in P$ is part of the set of production rules $P = (L, R, C)$, which describe the one-node graph $L \in p$ that replaces graph $R \in p$ using specific embedding rules $C \in p$. The two main issues with graph grammars are the inflexibility of the edge labels, which, unlike the attributes in attribute grammars, are typically limited to one element $e \in \Delta$, as well as the inherent computational complexity of graph rewriting operations/embedding rules: most algorithms require cubic or greater time to complete. Since our system is intended to be used as compression tool for simplified graph data (see 6.1), the performance factor was relevant.

In summary, the reason for our choice was grounded in the fact that semantically interesting connections between system events are often expressed by several parameters; parameters, that can be aptly modeled by the attributes of a context-free grammar. Performance and the availability of parsing tools also factored into the decision.

The inferred patterns and, by extension, the full attribute grammar as per our specification, can be defined as follows:

Let $AG = (G, A, R, V)$ be an attribute grammar, where:

- $G = (N, T, P, S)$ is a context-free grammar
  - N... Set of non-terminal symbols (variables)
  - T... Set of terminal symbols (alphabet)
  - P... Production rules
  - S... Start symbol
- A is a finite set of attributes
- R is a finite set of attribution rules (semantic rules)
- V is a finite set of values assigned to an attribute

Every symbol $X \in (N \cup T)$ is assigned a finite set of attributes A_X. The attribute $a \in A_X$ is denoted $X.a$. Every attribute $a \in A_X$ also has a set of values $V(X.a)$. Typically, an attribute a of symbol $X \in (N \cup T)$ that is e.g. assigned the value "0" is denominated as $X.a = 0$.

Our methodology uses attributes to store parameters of system events, such as the names of particular files that are being accessed or IP addresses that are being contacted in the course of a network operation. Attributes are also used to retain the connection to the invoking process of an event. In our case, attributes are used to e.g. represent the name of a file being created and the name of the process triggering that particular operation (see Sections 3.4 and 5.1 for more examples).

Two frequently used attributes are $a_1 = X.trigger\_name$ and $a_2 = X.element\_name$. The value $v_i \in V(X.a_1)$ identifies the actual name of the observed process responsible for triggering the individual event $X \in (N \cup T)$. Value $v_j \in V(X.a_2)$ denotes the process or file system element the process interacted with.

Raw system events (see 3.4) captured by our monitoring agent are processed by Sequitur, which infers a full grammar in accordance to above definitions. Our system is able to depict an arbitrary number of input traces with several attributes $a \in A$. The resulting grammar enables further parsing and semantic analysis. See Section 4.2 for more information about the inference process.

3.4 Event data

SEQUIN is based on event traces defined as descriptions of operating system kernel behavior invoked by applications and, by extension, a legitimate or illegitimate user. These events are abstractions of raw system and API calls that yield information about the general behavior of a sample [56]. Raw calls may include wrapper functions (e.g. `CreateProcess`) that offer a simple interface to the application programmer, or native system calls (e.g. `NtCreateProcess`) that represent the underlying OS or kernel support functions. In the context of SEQUIN, event data is collected directly from the Windows kernel. We employ a driver-based monitoring agent [31] designed to collect and forward a number of events to a database server. This gives us unimpeded access to events depicting operations related to process and thread control, image loads, file management, registry modification, network socket interaction, and more. For example, a shell event that creates a new text file on a system may be simply denoted as a triple `explorer.exe,file-create,document.txt`.

An alternative notation allows us to also specify each event as a graph element $G = (U, V, E)$, where U and V are nodes and E is the respective edge. In one of our own applications (see Section 6.1), the edge label is used as basis for minimal cost calculation of same-size star structures. Here, the above event would consist of the triple `explorer.exe,0.75,file-document.txt`, whereby 0.75 is the edge label value currently associated with 'create' operations in a file context. Regardless of format, Sequitur still transforms the supplied input to sequences of strings during processing.

Additional information captured in the background includes various process and thread ID information required to uniquely identify an event within a system session. To maintain event chronology, each individual activity is time-stamped and can be linked to a specific process or thread through their respective ID. This allows us to construct both process and full system traces that consider process and thread context, and reduce the impact of multi-threading and mimicry attacks while making it possible to determine specific dependencies between processes or general events. Concatenated into a full system trace, the resulting sequence describing the monitored session is assembled without runaway entries (see Fig. 1) interrupting the process flow. These *smart traces* are the basis for all further processing.

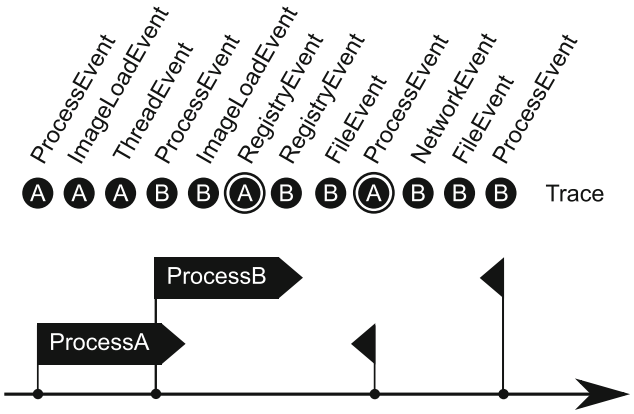

Fig. 1 Runaway events in a context-unaware trace. The 'smart trace' approach reorders events to maintain their process and thread context

Specifically, there we use two methods for assembling smart traces: The first approach reorders events to retain chronology within the context of each process and thread. New processes launched or threads spawned are appended after the current process or thread terminates. This trade-off sacrifices inter-contextual precision for a high accuracy within process/thread bounds (see Section 8 for further discussion). The second approach adds thread ID information to graph element U (or the first tuple in general), allowing for later filtering of undesired events.

Refer to [31] for more information about the smart trace format. For the evaluation detailed in this paper, we primarily utilized method 1. The example in Section 5.1 has been simplified and retains its original event order.

4 Inference and analysis process

4.1 Preprocessing

Before Sequitur can be used on log files, behavioral traces or other, sequential reports describing the activity of potentially malicious programs, the traces need to be reduced to their core components. In this normalization stage we have the choice to either strip away all attributes, or to retain them in an abstracted fashion as part of the set of terminals. As we want to construct a full, semantics-aware attribute grammar, most information is typically kept. We only reduce volatile information such as (user) IDs, memory addresses, and registry paths to a more manageable set of terminals. Names of known system processes and libraries are not modified in any way while unknown binaries and modules (which are possibly randomly named) are represented by extension-aware placeholders (e.g. `1.txt` or `2.exe`).

In order to compare the impact of different levels of detail and granularity, we defined a total of three input formats. A full example input and output scenario is discussed in Section 5.1.

- **Verbose** – This trace format uses full, attribute-enabled events as individual words of the corpus. In verbose mode, the input data is transformed into the following format: `triggering-process,operation,element-name`, which translates to $v_i \in V(X.a_1)$,$t_x \in T$,$v_j \in V(X.a_2)$. For example, a specific file creation operation triggered by the known `explorer.exe` process would be preprocessed into the following textual input format: `explorer.exe,file-create,1.txt`.
- **Reduced** – In this preprocessing mode, we omit attribute a_2 to generate a quick view of the high-level activity exhibited by the processes under scrutiny. Here, v_j is not processed, resulting in a reduced format of `triggering-process,operation`, depicted as e.g. `explorer.exe,file-create`.
- **Granular** – The goal in granular mode is to investigate operations not as single word, but as elementary components. Each of the elements processed in verbose mode is treated by Sequitur as one terminal of the bigram. To maintain a level of separation between event triples, a forth item denoting the start of a new event is prepended before each v_i. This results in the following input (items delimited by semicolon): `<start>;triggering-process;operation;element-name`.

4.2 Rule extraction

Since Sequitur only takes a single input file per default, we added additional functionality to the algorithm in order to retain information of origin and to enable multi-file inference with various evaluation subroutines. This way, SEQUIN's grammar inference can be applied to several files at once without having to concatenate the input in advance. Specifically, we altered Sequitur to be capable of constructing rules across file boundaries denoted by a unique separator, which is ignored by the inference engine. This ultimately enables comparative analyses of larger, disconnected data sets that do not necessarily share repeating behavior within a single trace, which, under normal circumstances, is required for the inference process to trigger. The main stages of the rule extraction process are the following:

- **Lexical analysis** – In this initial step, each unique terminal $t \in T$ is assigned a corresponding symbol, called a token. This numerical representation is used to streamline the process by reducing the processing complexity of string-only comparisons. Each new terminal is addi-

tionally stored in a translation (symbol) table for later reference.
- **Grammatical inference** – After the lexical analysis process, the Sequitur algorithm is applied to generate an execution trace grammar consisting of tokenized terminal symbols. The first rule $p \in P$ of each grammar is the start rule, or zero rule, which depicts the full grammar of the compressed input data. In our implementation, every line thereafter contains the following extracted information:
  - Rule – The rule consists of a left-side rule name (variable), which is sequentially numbered, as well as right-side variables and terminals. The non-terminals are, again, references to finer-grained rules while the terminals represent the actual system events. In line with the definition of CFGs, there is only one single variable on the left side of a rule.
  - Resolved rule – In order to provide a detailed view on individual rules, we recursively resolve each sequence of non-terminals $n \in N$ to their base terminals $t \in T$.

4.3 Rule evaluation

As part of the evaluation process, the final grammar is parsed to determine how many times a specific derivation occurs in each of the investigated input files. Semantically interesting patterns include specific sequences that e.g. occur exactly once in each input trace, making them potential common denominators for a class of malicious behavior. Computed information includes:

- File rule (FR) count – This number shows how many times a rule occurs in the current input file.
- Grammar rule (GR) count – The overall count across all supplied input files is specified here. For a single trace, this number is identical to the FR count.
- Prevalence count – This value specifies the number of input files a particular derivation has been found in. The result is displayed as x/y (x in y), where x is the number of files the pattern is prevalent and y is the overall count of individual input files.
- Match flag – The extraction of interesting rules is facilitated by determining rules that are identical in occurrence and number across all of the processed input files, indicated by a Boolean flag.
- Rule length – this value defines the overall number of items seen in the entire derivation (i.e. the resolved rule). Multiples of the input file count y are likely to represent recursively compressed rules.
- Rule density – this support metric facilitates anomaly detection by calculating the ratio between inferred rules and single terminals that are present in the input as well as rule zero.

The various counts calculated always include references to the original input files, which help retain each pattern's connection to its semantic source. In Section 5.1, we show an exemplary scenario for a 'verbose' (see Section 4.1) input set.

4.4 Rule transformation

In order to transform the newly inferred rules into an attributed grammar as defined in Section 3.3, a set mechanism is required. This mechanism allows the analyst to easily define a naming schema for inferred rules, optimally resulting in a semantic description of the terminals and non-terminals contained within. In the initial version of our tool, we map each operation to an attribute-enhanced terminal while rule identifiers are transformed into descriptive variables: Specifically, each rule is dubbed in accordance to its semantic nature. For example, a rule describing a `process-create` operation followed by a `file-delete` operation is transformed into the descriptive variable CREATE-PROC_DELETE-FILE. A rule that describes the loading of two image files is dubbed LOAD2-IMG.

The full naming schema NS is currently defined as follows:

- $NS = (O, E, MO, ME, L)$, where
  - Operation O = {CREA, MOD, START, LOAD, KILL, DEL, CONN}
  - Event type E = {PROC, THR, IMG, FILE, REG, NET}
  - Operation mapping rules MO = {
    CREA → create,
    MOD → modify | change | edit,
    START → start | spawn,
    LOAD → load,
    KILL → kill | stop | terminate,
    DEL → delete,
    CONN → connect
    }
  - Event mapping rules ME = {PROC → process, THR → thread, IMG → image, FILE → file, REG → registry, NET → network}
- and labeling rules L, where
  - $(O_1 || \text{"-"} || E_1 || \text{"\_"}, ..., O_n || \text{"-"} || E_n)$
  - If $O_n == O_{n+1}$ then $O_n ||$ "2"

The triggering process and element name are then transformed into the attributes tp ($X.a_1$) and en (a_2). Recursive

variable descriptors are supported – above naming schema always considers the fully resolved rule. See Section 5.1 for several examples of automatically determined variables.

Future versions of the method will replace the current mapping with a true semantic descriptor that identifies specific attacker actions or objectives. While a manual assignment of such variables is already possible [14], it is not feasible in larger analysis scenarios. The automation of the process is an important research challenge to come.

5 Implementation

Our grammar inference and evaluation tool is based on the Sequitur application developed by Eibe Frank[1]. All core and extended functionality has been fully implemented in Java. The data used as basis for the analysis process is collected using a specifically created kernel driver agent deployed on 10 actively used and malware-free (fresh installations performed by security analysts) Windows 7 and Windows 10 machines within our company partner's environment. An additional virtual Windows instance is used for dynamically analyzing malicious software. All machines at least provide common user applications such as Microsoft Office, Adobe Reader, various browsers, as well as common OS extensions such as Java SE and the .NET framework. The collected events are stored and processed on a dedicated PostgreSQL database server that generates verbose or reduced traces of specific processes or even entire system sessions. These traces are ultimately used as input for the Sequitur approach: SEQUIN concatenates the respective files and keeps them apart by inserting a file delimiter that is ignored by the inference engine. This way we can handle an arbitrary number of traces without major changes to the underlying algorithm. Following the inference and analysis stage, our KAMAS prototype visualizes the grammar and helps discover and classify relevant elements. See Fig. 2 for a full process overview.

In practical scenarios, it might be prudent to use clustering algorithms to pre-classify traces that are likely to share common behavior. While these algorithms typically do not yield insight into event semantics, this intermediate step helps an analyst to select sequences that e.g. belong to a similar class of malware or describe a comparable attack stage. In such a scenario, our inference tool can be used to specifically extract behavioral patterns for a particular use case. In our initial tests, we used Malheur [39] for this purpose.

5.1 Example

With or without preselection, our grammar inference and evaluation tool will generate variables and production rules for a dynamically growing number of terminals and attributes. Below example demonstrates the use of our tool for two simplified 'verbose' input files generated from aforementioned kernel event traces. Thread context information and smart reordering has been omitted for better legibility. The character tokens ($a..m$) were added manually for better understanding.

```
Input file 1: Verbose mode (default sequence).
    Delimiter: newline.
explorer.exe,file-create,1.exe (a)
explorer.exe,process-start,1.exe (b)
1.exe,image-load,kernel32.dll (c)
1.exe,image-load,advapi32.dll (d)
1.exe,registry-modify,hklm/software/
  microsoft (e)
1.exe,registry-modify,hklm/software/
  microsoft (e)
1.exe,process-create,cmd.exe (f)
cmd.exe,process-create,net.exe (g)
1.exe,registry-create,machine/system (h)
1.exe,registry-modify,hklm/software/
  microsoft (e)
1.exe,registry-modify,hklm/software/
  microsoft (e)
cmd.exe,process-kill,net.exe (i)
1.exe,thread-terminate,thread (j)
explorer.exe,file-delete,1.exe (k)
```

The second input file has been determined by Malheur to be similar, however the commonalities are yet unclear. This is where our pattern evaluation extension comes in.

```
Input file 2: Verbose mode (default sequence).
    Delimiter: newline.
explorer.exe,file-create,1.exe (a)
explorer.exe,process-start,1.exe (b)
1.exe,thread-create,thread (l)
1.exe,image-load,kernel32.dll (c)
1.exe,image-load,advapi32.dll (d)
1.exe,image-load,ws2_32.dll (m)
1.exe,registry-modify,hklm/software/
  microsoft (e)
1.exe,registry-modify,hklm/software/
  microsoft (e)
1.exe,process-create,cmd.exe (f)
cmd.exe,process-create,net.exe (g)
cmd.exe,process-kill,net.exe (i)
1.exe,thread-terminate,thread (j)
explorer.exe,file-delete,1.exe (k)
```

Sequitur now infers the following rules and evaluates the frequency and similarity. Below output has been reformatted to contain the resolved events only for the remaining, recursively created rules (rule 3), as well as rule zero. Rule density is only calculated for the latter.

Rule 1 is found twice across the grammar (GR count) of the two appended input files. The two events `explorer.exe,file-create,1.exe` (abbreviated: a) and `explorer.exe,process-start,1.exe` (b) form the rule $1 \rightarrow$ a b. For a more formal breakdown summary of example input file 1, please consult below grammar depiction AG_1.

```
Rule: 1
explorer.exe,file-create,1.exe (a)
explorer.exe,process-start,1.exe (b)

Evaluation:
FR count (file 1, 2): 1, 1
GR count: 2
Prevalence: 2/2
Match: true
Rule length: 2
```

[1] https://github.com/craignm/sequitur/tree/master/java

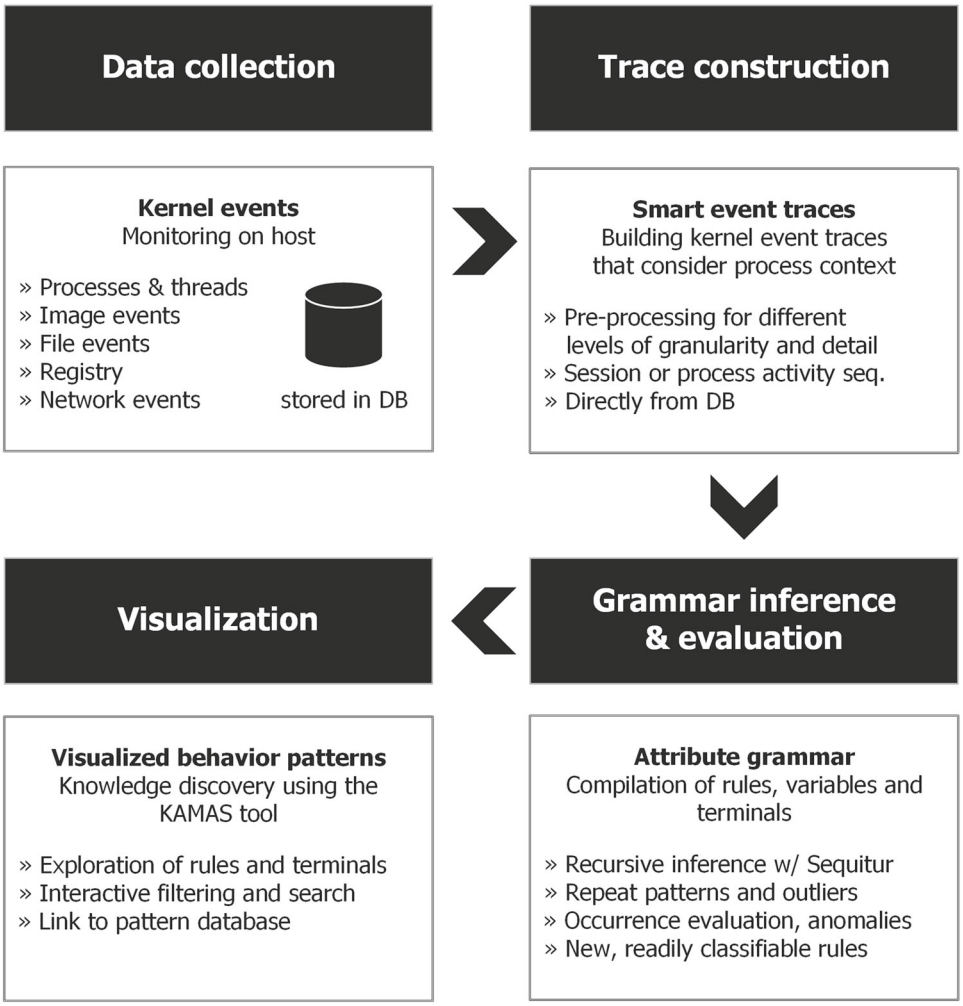

Fig. 2 Overview of SEQUIN

Rule 2 is inferred by events c and d: 2 → c d. It is part of both files and is thereby prevalent in the input.

```
Rule: 2
1.exe,image-load,kernel32.dll (c)
1.exe,image-load,advapi32.dll (d)

Evaluation:
FR count (file 1, 2): 1, 1
GR count: 2
Prevalence: 2/2
Match: true
Rule length: 2
```

Below rule is part of both input files, but does not perfectly match in terms of frequency: The second trace contains two identical occurrences instead of just one. It resolves to 3 → e e.

```
Rule: 3
1.exe,registry-modify,hklm/software/
  microsoft (e)
1.exe,registry-modify,hklm/software/
  microsoft (e)

Evaluation:
FR count (file 1, 2): 2, 1
GR count: 3
Prevalence: 2/2
Match: false
Rule length: 2
```

Prevalent rule 4 is the only persisting recursively inferred sequence of the example: It translates to 4 → 3 f g. However, Sequitur does not immediately build that rule: Initially, the system infers 4TEMP1 → 3 f, followed by 4TEMP2 → 4TEMP1 g. Because of the rule utility property, both TEMP rules are ultimately dissolved, resulting in the final rule 4 → 3 f g.

```
Rule: 4
1.exe,rule,rule-3
1.exe,process-create,cmd.exe (f)
cmd.exe,process-create,net.exe (g)

Evaluation:
FR count (file 1, 2): 1, 1
GR count: 2
Prevalence: 2/2
Match: true
Rule length: 3

Resolved:
1.exe,registry-modify,hklm/software/
  microsoft (e)
1.exe,registry-modify,hklm/software/
  microsoft (e)
1.exe,process-create,cmd.exe (f)
cmd.exe,process-create,net.exe (g)
```

The final rule summarizes the triple that concludes both traces: 5 → i j k. Like rule 4, this process has an intermediate step: Before settling on the final derivation, Sequitur builds

the rule 5TEMP1 → i j, followed by 5TEMP2 → 5TEMP1 k.

```
Rule: 5
cmd.exe,process-kill,net.exe (i)
1.exe,thread-terminate,thread (j)
explorer.exe,file-delete,1.exe (k)

Evaluation:
FR count (file 1, 2): 1, 1
GR count: 2
Prevalence: 2/2
Match: true
Rule length: 3
```

From these 5 final rules, a compressed zero rule can be inferred. The resolved rule yields the concatenated original input of both files separated by a file delimiter: 0 → (file 1) 1 2 800 4 h 3 5 (file 2) 1 l 2 m 4 5.

```
Rule: 0
explorer.exe,rule,rule-1
1.exe,rule,rule-2
1.exe,rule,rule-4
1.exe,registry-create,machine/system (h)
1.exe,rule,rule-3
cmd.exe,rule,rule-5
-
explorer.exe,rule,rule-1
1.exe,thread-create,thread (l)
1.exe,rule,rule-2
1.exe,image-load,ws2_32.dll (m)
1.exe,rule,rule-4
cmd.exe,rule,rule-5

Evaluation:
Rule density (input: 3 out of 27): 88.9
Rule density (rule 0: 3 out of 9): 66.6

Resolved:
(see concatenated input)
```

In our example, the tool has successfully extracted rules that describe behavior observed in both input files. With the uncompressed events `1.exe,registry-create, machine/system`, `1.exe,thread-create,thread`, and `1.exe,image-load,ws2_32.dll` highlighted, we can immediately spot the deviations from the otherwise recurring behavior.

In conclusion, the output is transformed into an attribute grammar as described in Section 3.3. Since semantics is a major factor of rule construction, we assign variables based on the nature of the inferred event. Specifically, above example (here: only for file 1 of rule 0) can be formalized into a grammar as follows:

Let $AG_1 = (G_1, A, R, V)$ be an inferred CFG extended by attributes, where:

- $G_1 = (N, T, P, S)$, and where:
  - N = {CREA-FILE_START-PROC; LOAD2-IMG; MOD2-REG_CREA2-PROC; MOD2-REG; KILL-PROC_KILL-THR_DEL-FILE}
  - T = {
    file-create.tp, en = explorer.exe, 1.exe;
    file-delete.tp, en = explorer.exe, 1.exe;
    process-create.tp, en = explorer.exe, 1.exe;
    process-create.tp, en = 1.exe, cmd.exe;
    process-create.tp, en = 1.exe, net.exe;
    process-kill.tp, en = cmd.exe, net.exe;
    image-load.tp, en = 1.exe, kernel32.dll;
    image-load.tp, en = 1.exe, advapi32.dll;
    registry-create.tp, en = 1.exe, machine/system;
    registry-modify.tp, en = 1.exe, hklm/software/microsoft;
    thread-terminate.tp, en = 1.exe, thread;
    }
  - P = {
    ZERO-RULE → CREA-FILE_START-PROC LOAD2-IMG MOD2-REG_CREA2-PROC registry-create.tp, en = 1.exe,machine/system MOD2-REG KILL-PROC_KILL-THR_DEL-FILE;
    CREA-FILE_START-PROC → file-create.tp, en = explorer.exe,1.exe process-create.tp, en = explorer.exe,1.exe;
    LOAD2-IMG → image-load.tp, en = 1.exe,kernel32.dll image-load.tp, en = 1.exe,advapi32.dll;
    MOD2-REG → registry-modify.tp, en = 1.exe,hklm/software/microsoft registry-modify.tp, en = 1.exe, hklm/software/microsoft;
    MOD2-REG_CREA2-PROC MOD2-REG → process-create.tp, en = 1.exe,cmd.exe process-create.tp, en = cmd.exe,net.exe;
    KILL-PROC_KILL-THR_DEL-FILE → process-kill.tp, en = cmd.exe, net.exe thread-terminate.tp, en = 1.exe,thread file-delete.tp, en = explorer.exe, 1.exe
    }
  - S = {ZERO-RULE}
- A = {tp; en}
- R is described as part of the preprocessing stage and defines which portion of the data translates into triggering process tp (v_i), operation (t_x), and element en (v_j).
- V = {explorer.exe; 1.exe; kernel32.dll; advapi32.dll; cmd.exe; net.exe; machine/software/microsoft; machine/system; thread}

Above attribute grammar for part 1 of the zero rule has been generated automatically and can now be used as the foundation for further (attribute-based) parsing efforts. The inferred variables, if stored, can be used as new behavioral templates for comparable input data sets. Above definition can easily be extended to encompass specific files, or all of them at once (entire zero rule).

The next Section discusses practical applications of this approach and evaluates them using real-world data.

6 Evaluation

The introduced system has a wide variety of applications. Ranging from preliminary knowledge extraction in malware

Table 3 Scenarios covered by experiments, in order of appearance by subsection

Scenario	Eval.	Significance
Compression	6.1	Reduction of input data size
		Reduction of processing complexity (third-party)
Anomaly detection	6.2	Detection and extraction of deviating behavior
Baselining	6.2	Identification of common patterns in traces
Visualization	7	Visual presentation of inferred rules
Discovery	7	Interactive filtering and extraction of terminals/rules
		Rule labeling, storing
		Highlighting of known rules

analysis scenarios to understanding more complex attacks, the adapted inference methodology is versatile in both terms of input data as well as practical benefit. Below, we introduce and evaluate some of its applications and discuss future and ongoing work. Table 3 provides an overview of the conducted experiments.

6.1 Preparatory data reduction

6.1.1 Concept

In many malware and APT attack stage analysis scenarios, analysts are often forced to deal with huge amounts of data. Be it kernel events, raw system calls or even assembler-level CPU instruction information, the abstraction and reduction of input data is essential to decrease the complexity of many an analysis task. Our solution provides the means through its easily adaptable preprocessing mechanism (see Section 4.1) and the grammar inference system itself. By using the Sequitur approach, it is possible to reduce the input corpus to only relevant n-grams ($n \geq 2$), instead of working with the full, unfiltered set of event or code snippet unigrams. The grammar transformation mechanism (see Section 4.4) also enables us to work with an automatically generated placeholder variable $n \in N$ instead of several compound terminals.

A second, closely related application of Sequitur compression is the extraction of recurring patterns. Dubbed 'baselining', this process takes the result of the inference process and regards sequences of terminals that have been turned into rules. The discovery process is best supported by our KAMAS VA prototype introduced in Section 7.

6.1.2 Evaluation

Current efforts include the pre-abstraction of behavioral data in graph notation (also see Section 3.4) subsequently used for edit distance calculations [30]. Minimizing the amount of data to be processed drastically reduces computation requirements of expensive (up to exponential complexity, depending on the application) graph transformation operations. Specifically, we evaluated several days' worth of benign system events monitored by our kernel driver (see 5 for event capture details), collecting 100k, 200k, and 400k sequential events with different size alphabets and rule density – all associated with instances of the Windows `svchost.exe` process. Under normal circumstances, this data would have to be assessed in its uncompressed entirety, as it is used for creating baseline graph templates utilized in behavior deviation analysis using a combination of Hungarian distance computation [26] and Malheur heuristic clustering [39]. Thanks to our Sequitur-enabled data reduction, we can focus on event sequences (rules) that are representative for specific processes – or on the remaining terminals which constitute a potential anomaly. Both significantly speed up all involved, polynomial complexity star graph matching operations (Fig. 4) used by the tested system [30]. At the same time, we increase the accuracy of the template creation process by drastically reducing the number of empty feature vectors that are normally produced in the clustering stage.

In our first, medium-sized exemplary dataset of 100k Windows kernel events (alphabet of 665 words), we reduced the number events to a list of 1,715 terminals and 6,415 first-level rules (contained in the zero rule), which effectively compressed the data by 90.7%. This cut the processing time for graph template generation and graph transformation calculations by 77.64%, down to a total of 8.75 + 1.9 minutes, instead of 48.2 minutes sans compression. Performance evaluation showed a maximum memory utilization of around 1.3 GiB, with a total processing time of 1.9 minutes (9.1 minutes with full rule resolving) on a dual core virtual machine equipped with 64 GiB of RAM.

The second and third datasets encompassed an alphabet of 1,310 and 1,569 words, respectively. We achieved a compression rate of 95.95% and 98.26%, which reduced the graph processing time by 64.91% and 77.26%. RAM usage increased to 2.35 GiB for the 200k dataset, and to 4 GiB for the 400k trace. Figure 3 and 4 show basic regression analyses of grammar inference and graph processing times as well as RAM utilization during compression. See [30] for more information on star graph template generation and matching.

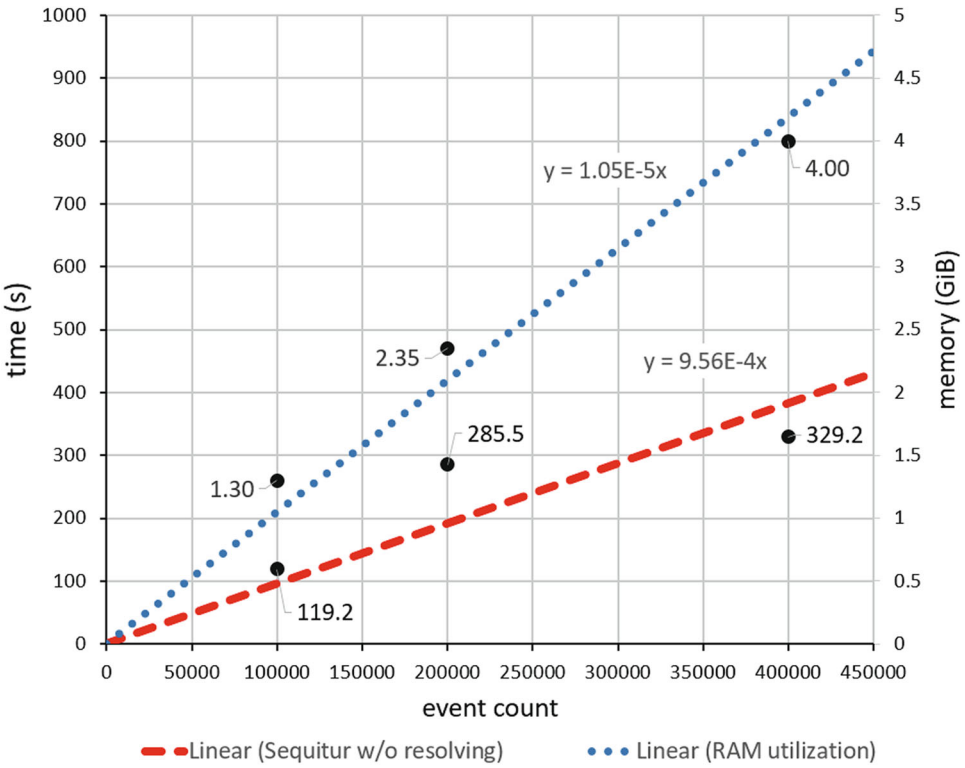

Fig. 3 Performance analysis of Sequitur inference. Both processing time and RAM utilization grow at a linear rate. The alphabet size (not pictured) was determined to have a greater impact on the time required than on physical memory use

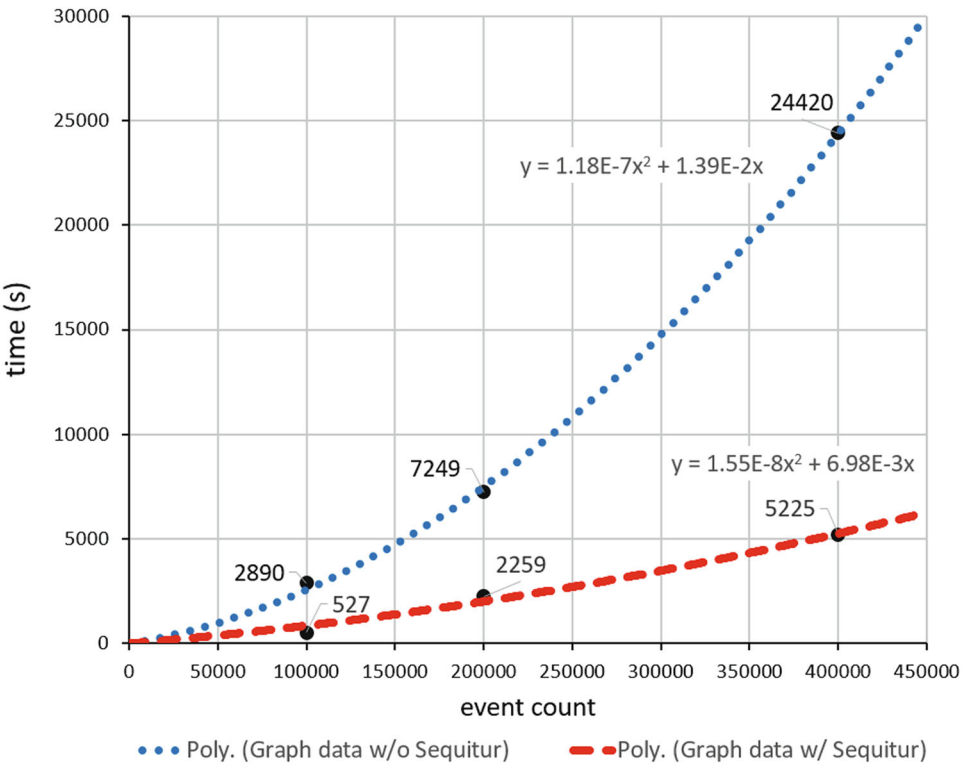

Fig. 4 Performance impact on graph matching operations. The chart compares the time required for creating a template using uncompressed data versus the non-terminals inferred by Sequitur. Input filtered to rules with a prevalence count $PC > (n/m)$, where $m < 0.2 * n$ (here: $m = 1000$) and $n =$ number of unique processes per dataset. Results show an average speed-up of around 73% for kernel event data

6.1.3 Discussion

The overall process was determined as scaling at linear time $O(n)$, putting it in line with e.g. basic search algorithms and confirming the results disseminated by [37]. RAM consumption scaled linearly as well – in case of our machine, we expect to hit the memory ceiling of 64 GiB at around 6 million events, provided the size of the alphabet grows at a similarly steady rate. SEQUIN without rule resolving performed best overall, saving up to 90% of its processing time for the largest (400k) dataset. Because of this significant overhead, rule resolving will in the future be performed independently from compression – directly in our growing database of known productions. This will offer a convenient way to compute and look up patterns without negatively impacting performance during the actual inference process.

In terms of semantic accuracy, over-zealous recursive compression needs to be considered when applying the approach to data with large repeating sequences of terminals, as it may combine too many individual events of significance into one long rule, which in turn consists of other lengthy rules describing similar behavior. This could not only skew the result, but also eliminate some of the performance benefits evaluated above. In our experiments, we have found it prudent to impose a limit on maximum resolved length for terminal-only rules. Another approach is to set a minimum limit on prevalence count (see Section 4.3) to remove non-terminals that occur in only a few input files. The thresholds for these operations largely depend on the nature of the data used and will have to be determined by experimentation on a case-to-case basis.

SEQUIN has proven to be well suited to the task of preprocessing/reducing input data needed by other, more expensive algorithms. We achieved an average speed-up in star-graph data processing of 73% when employing our system to the same dataset. Data sizes were reduced by up to 98%.

6.2 Anomaly detection

6.2.1 Concept

In our above preprocessing example, we use grammar inference to determine interesting repeating patterns that are representative of the corpus under investigation. However, the reverse is also a viable scenario: By focusing attention on patterns that do not excessively reoccur, our approach can be used to identify anomalies in a sequence or set of sequences. Parts of the trace that are not replaced by variables during rule construction (i.e. the remaining terminals in between) represent unique events that, in such a scenario, are of particular interest as they represent deviating (abnormal) behavior. Rule density (see 4.3) is also important in scenarios where stable behavior is expected: the higher the share of terminals, the higher the overall entropy, and, by extension, the likelihood of anomalous behavior. All anomaly detection efforts can be aided by visualization tools such as GrammarViz [45] as well as our own VA research introduced in Section 7 below.

6.2.2 Evaluation

The Sequitur tool is not limited to system events but can be used with a wide range of sequential input data formats. In the following, we specifically evaluated an APT anomaly detection scenario on a set of temperature, speed, and photoelectric sensor data generated by a Siemens Simatic industrial control system (ICS) within a testbed environment. We assessed 13 full production runs in total, whereas two of the runs were maliciously altered by illegally interfering with the rotation. This resulted in some atypical sensor readings that are nigh impossible to spot manually.

The full evaluated grammar for a total of 34,000 observed events was constructed within 5 seconds. Sequitur inferred a total of 2,155 rules (sans zero rules), resulting in a 93.7% data compression rate. In stage one, anomaly detection was conducted by assessing rules with a low rule density value. By that metric alone, it was already possible to identify anomalous traces. With a terminal-to-rule ratio (TRR) of over 62.7% (rule density of 37.3%), the malicious samples contained less uniform behavior patterns than the remainder of 11 traces with a mean ratio of 59.3%. Only two benign traces came close to that number, exceeding a TRR of 60%. The comparatively small margin is due to the fact that, in this scenario, anomalous data did not cause sensor spikes but rather triggered a slow, continuous change in behavior.

Further analysis of the possibly deviating behavior was (and is typically) required to solidify the initial verdict. To this end, we used our evaluation system to filter rules that are present in only a minority of files and that have a prevalence count of 1 out of 12. Armed with the pre-selection based on rule density, we particularly focused on traces with a TRR of $>60\%$. Specifically, we normalized the scores for TRR (x_T), mean rule length (x_L), and the count of low-prevalence rules (x_P) and computed a weighted total X_{WT}:

$$X_{WT} = 0.4\frac{x_T - min(x_T)}{max(x_T) - min(x_T)} + 0.1\frac{x_L - min(x_L)}{max(x_L) - min(x_L)} + 0.5\frac{x_P - min(x_P)}{max(x_P) - min(x_P)} \quad (1)$$

Table 4 lists the computed values for each trace. As a result, the Youden index BC_a bootstrap confidence interval [16] was determined as 0.55..1, resulting in an associated criterion score threshold of 0.66 for distinguishing true from false matches. For ICS data, a manual adjustment to a higher threshold (e.g. 0.75) can further increase result confidence.

Table 4 Scores for TRR, mean rule length (Length), and rules with minimum (= 1) prevalence count (Preval). Columns marked with an asterisk (*) mark values normalized to 0..1 as per Equation (1). Normalized scores ≥ 0.8 are printed in bold

Sample trace	TRR	TRR*	Length	Length*	Preval.	Preval.*	**Overall**
Ben-1	58.60	0.33	8.87	0.44	2	0.20	0.274
Ben-2	**64.00**	**1.00**	**9.28**	**0.95**	**9**	**0.90**	**0.945**
Ben-3	59.30	0.41	8.9	0.48	2	0.20	0.313
Ben-4	56.00	0.00	8.51	0.00	1	0.10	0.050
Ben-5	56.80	0.10	8.52	0.01	1	0.10	0.091
Ben-6	58.30	0.29	8.79	0.35	2	0.20	0.250
Ben-7	59.20	0.40	8.69	0.22	0	0.00	0.182
Ben-8	60.00	0.50	8.72	0.26	4	0.40	0.426
Ben-9	**63.80**	**0.98**	**9.32**	**1.00**	0	0.00	0.490
Ben-10	57.30	0.16	8.6	0.11	3	0.30	0.226
Ben-11	58.50	0.31	8.79	0.35	**10**	**1.00**	0.660
Mal-1	**62.80**	**0.85**	9.01	0.62	**9**	**0.90**	**0.852**
Mal-2	**62.74**	**0.84**	8.99	0.59	**9**	**0.90**	**0.846**

Table 5 Extracted and evaluated rules for ICS sensor data traces with low rule density (≤ 40%) and prevalence count (= 1). Each rule describes an anomaly not typically seen in other input data. FR...file rule, GR...grammar rule

File	Rule	FR #	GR #	Prevalence	Length
Ben-2	3 → 139 139	2	2	1/12	16
Ben-2	4 → 140 140	3	3	1/12	4
Ben-2	12 → 1-0-1-(...)-0-40 1-0-1-(...)-0-40	2	2	1/12	2
Ben-2	23 → 1-0-1-(...)-1-56 1-0-1-(...)-1-56	2	2	1/12	2
Ben-2	69 → 1-0-1-(...)-5-59 1-0-1-(...)-5-59	2	2	1/12	2
Ben-2	98 → 1-0-1-(...)-8-60 1-0-1-(...)-8-60	2	2	1/12	2
Ben-2	102 → 0-0-1-(...)-8-54 0-0-1-(...)-8-54	2	2	1/12	2
Ben-2	139 → 4 4	2	2	1/12	8
Ben-2	140 → 0-0-0-(...)-0-20 0-0-0-(...)-0-20	2	2	1/12	2
Mal-1	57 → 1-0-1-(...)-4-60 1-0-1-(...)-4-60	2	2	1/12	2
Mal-1	72 → 1-0-1-(...)-6-52 1-0-1-(...)-6-52	2	2	1/12	2
Mal-1	81 → 82 82	2	2	1/12	64
Mal-1	82 → 83 83	3	3	1/12	32
Mal-1	83 → 157 157	3	3	1/12	16
Mal-1	84 → 85 85	3	3	1/12	4
Mal-1	85 → 1-0-1-(...)-7-60 1-0-1-(...)-7-60	3	3	1/12	2
Mal-1	86 → 1-0-1-(...)-7-59 1-0-1-(...)-7-59	2	2	1/12	2
Mal-1	157 → 84 84	2	2	1/12	8

Now, we can now analyze the outcome of the inference process in detail: Benign trace 2 ("Ben-2" in Table 5) contained 9 rules that were not seen in the the remaining corpus. Likewise, both malicious traces (pictured here: "Mal-1") contained 9 unique rules. A direct comparison of the remaining anomalous candidates highlights one particularly interesting, recursively compressed block per trace, which resolved into 16 (benign) and 64 (malicious) terminals, respectively. It stands to note that patterns with a higher average length are particularly interesting as they identify larger, uninterrupted sequences unique for the dataset under scrutiny.

See Table 5 for a direct comparison of two of the most deviating behavior traces. Rule 3 of "Ben-2", which resolves into 8 terminal pairs as inferred by rule 140 (a rare, but valid sensor state), contributes most to the trace's analysis verdict. For "Mal-1", the same applies to rule 81 (32 iterations of rule 85), effectively identifying the anomalous sensor state. The accuracy of this evaluation scenario is detailed in Fig. 5 and boasts a high false positive and false negative rate, especially when removing faulty runs. The extracted, semantically relevant rules can now be formalized and stored for future parsing efforts.

For a simple baselining experiment, we used our SEQUIN tool on two lengthy traces depicting the Windows 7 update process performed on two machines with the same OS patch level. With 10,769 events in update 1 and 11,057 events

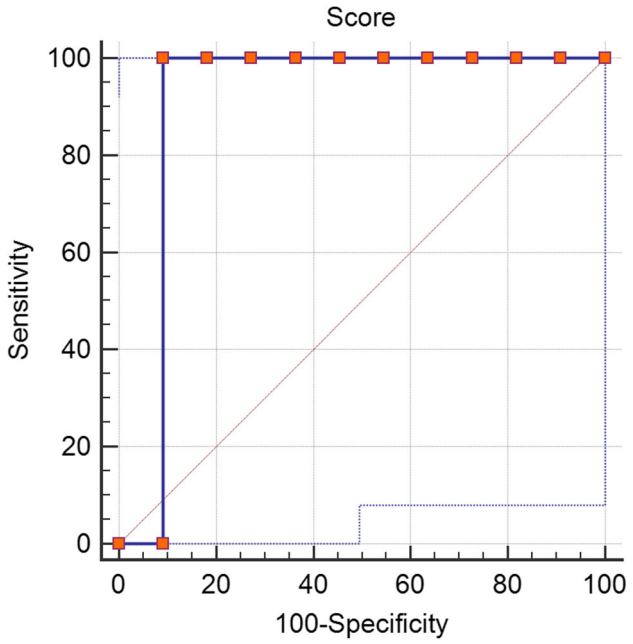

Fig. 5 ROC curve of the anomaly detection run. 12 out of 13 sets of sensor data traces were classified correctly with a sensitivity (true positive rate) of 100% and a specificity (true negative rate) of 100%. ROC area (AUC) was determined as 0.909. Since the deviating benign run is actually faulty despite not being a deliberate attack, its removal would boost the overall accuracy to 100% in our particular test case

in the second update, the combined compression generated 250 rules in total, 218 of which were contained in both input files. The randomly generated 32-bit strings used as names for the respective download directories within the `windows/softwaredistribution/download` folder structure were replaced during normalization as part of the preprocessing stage (Section 4.1). The remaining unique rules as well as terminals were deviations from the computed baseline caused by random temporary file names and minor changes in update chronology. In practice, this variant of the inference process can be used to create whitelists, templates for benign process behavior, or, again, for extracting anomalies from seemingly benign sequences of application behavior.

6.2.3 Discussion

In terms of accuracy, anomaly detection or baselining efforts are less likely to require fine-tuning than the compression routine (see 'Discussion' in Section 6.1), as the number of recursive rule-building iterations does not negatively impact the result. Instead of limiting the length of resolved rules, the extraction of relevant data is based on choosing the correct rule density and prevalence for the dataset under investigation. Here, analysts need to keep in mind that choosing the latter is tied to the expected number of malicious input traces that might share the same characteristics (identical inferred rules): The more often harmful sequences are assumed to repeat in the corpus, the higher the prevalence threshold needs to be and the less likely it becomes to spot the behavior using anomaly detection.

Initial tests determined an accuracy of well above 92% for the detection of deliberate attacks. Relevant anomalies were found in all of the inspected cases. For more complex scenarios it is recommended to apply visual analytics techniques that enable interactive data exploration: After the extraction of possible anomalies or baselines, a domain expert can investigate further to determine the individual events t that truly describe an illegal action. While this can be done textually using only our inference system, a visuals-assisted solution such as KAMAS promises (see 7 below) even better results.

7 Visualization &knowledge discovery

7.1 Visual analytics

One of the major applications of our proposed solution is undoubtedly the extraction of new domain knowledge. Inferred patterns can be compiled into a permanent grammar used to detect similar behavior in unknown traces. This process is supported by interactive visualization to drastically improve usability. This area of research, typically referred to as visual analytics (VA), forms the basis for KAMAS, our novel system used to visualize SEQUIN's output for pattern discovery and semantic annotation. In the following, we briefly explain the concept of VA, KAMAS' design considerations, and the prototype implementation itself.

VA is "the science of analytical reasoning facilitated by interactive visual interfaces" [52]. A major tenet of VA is that analytical reasoning is not a routine activity that can be automated completely [60]. Instead it depends heavily on the analysts initiative and domain experience, which is exercised through interactive visual interfaces. Such interfaces, especially information visualizations, are high bandwidth gateways for the depiction of structures, patterns, and connections hidden in the data. Furthermore, visual analytics often involves automated analysis methods that perform various computations on potentially large volumes of data.

When analysts solve real world problems they typically have vast amounts of complex and heterogeneous data at their disposal, as is evidenced by above application scenarios (see Sections 6.1 and 6.2). Externalization and storing of implicit knowledge will make it available as *explicit domain knowledge*, which is defined as knowledge that "represents the results of a computer-simulated cognitive process, such as perception, learning, association, and reasoning (...)" [9].

Through visualization, explicit knowledge can be used to graphically summarize and abstract a dataset. Put sim-

ply, it enables quicker and more precise analyses of complex input data such as the set of traces used in our ICS example.

Using VA for security applications is a widely accepted practice. In Wagner et al. [55], the authors surveyed tools for behavior-based malware analysis in addition to visual representations best suited to various domain challenges. Through a data–users–tasks analysis [33], they ascertained that the parse tree of a grammar such as the one generated by the Sequitur algorithm, can be abstracted to a directed acyclic graph, where each node represents part of a sequence.

7.2 Visualization considerations

The nature of the event data and the inference algorithm employed builds the foundation for our knowledge-assisted malware analysis tool, dubbed KAMAS [57], which is intended to support analysts in their task of identifying relevant behavioral patterns. In preliminary research [55], problem characterization and abstraction [43] was performed to elaborate the analysts' needs when using visual analytics for behavior-based malware analysis. This way, a common terminology describing i) the data to be visualized; ii) the users of the system; and iii) the tasks to be fulfilled, was established. Based on the outcome of this problem characterization, the initial design decisions for the visualization of data generated by the Sequitur algorithm can be established:

- **Representation of explicit knowledge**. To support the analysts in their task of behavior-based malware analysis, explicit expert knowledge should be made available in the system. Additionally, the actual generation of explicit knowledge needs to be facilitated. For the visualization of that knowledge, a basic hierarchy of events is required. We utilized the malicious behavior schema by Dornhackl et al. [14]: Using the provided semantic categorization, it is possible for analysts to explore currently stored knowledge and to add newly inferred rules to the system. The visualization of the malicious behavior schema employs a tree structure, where the nodes are the different types of malicious behavior and the leaves are the rules for its representation (see Fig. 6:1).
- **Representation of events**. For the representation of the events included in an analysis file, two important aspects have to be covered. On the one hand, the name of the event (see Section 3.4 for more information on input data) is essential for the analyst who is trying to ascertain its purpose. On the other hand, it is very important to learn how often a single event is included in the analysis file. We use a table structure for the visualization of this data, whereby the event name is represented as string and its occurrence is represented as a bar chart including the total number as an overlay. Employing this visualization technique, the analyst gains the ability to quickly find events of interest (e.g., by visually analyzing the size of the bar charts) (see Fig. 6:3).
- **Representation of rules**. Since a rule is a sequential structure containing several events, it is prudent to use a similar representation as for individual items. In contrast to the representation of all events included in a rule (resolved rule, see 4.2), a more abstract visualization can be applied here. The transformation of events based on their unique ID into a graphical representation (which is called 'Graphical Summary' [57]) helps to more effectively locate unknown patterns in the data. Additionally, all the other related information can be visualized as bar charts in combination with a label representing the total number (e.g., the rule's prevalence and length as introduced in Section 4.3). The original order of events within a rule is highlighted (see Fig. 6:2).

To determine analysts' requirements for behavior-based malware analysis with regards to usable visualization metaphors [55], a basic set of well-known visualization techniques was evaluated. Most of the participants indicated a combination of Multiple View [19], Arc Diagrams [59] and Wordtree [58] as being the most helpful, followed by OutFlow [61] and Pixel-Oriented Visualization [25]. In contrast, the Parallel Tag Cloud [11], has been described as the least complementing solution in respect to behavioral trace analysis. In the following, we detail the visualization concepts leading up to the development of research prototypes for both data processing and visualization.

7.3 Implementation

Aforementioned visualization considerations were used as a guideline to define the design rationale of KAMAS [57]. We decided to use an interface concept akin to well-known programming IDE interfaces (e.g. Eclipse) as the conceptual foundation for our prototype. This helps create a familiar environment based on multiple, vertically separated views (see Fig. 6).

In respect to these interface structures, we placed the tree view of the 'Knowledge Database (KDB) on the left side of the window (see Fig. 6:1). The KDB element contains all the explicit knowledge used for automated analysis; newly extracted patterns can be stored in a database for later use, whereas existing ones serve as real-time filter that automatically highlights known patterns. The key 'Rule Explorer' element is positioned in the center of the screen, where most user actions are performed during the analysis of a trace (see Fig. 6:2). Here, the analyst can see the different rules generated by Sequitur, including all information pertaining to its file and grammar count as well as its length. Select-

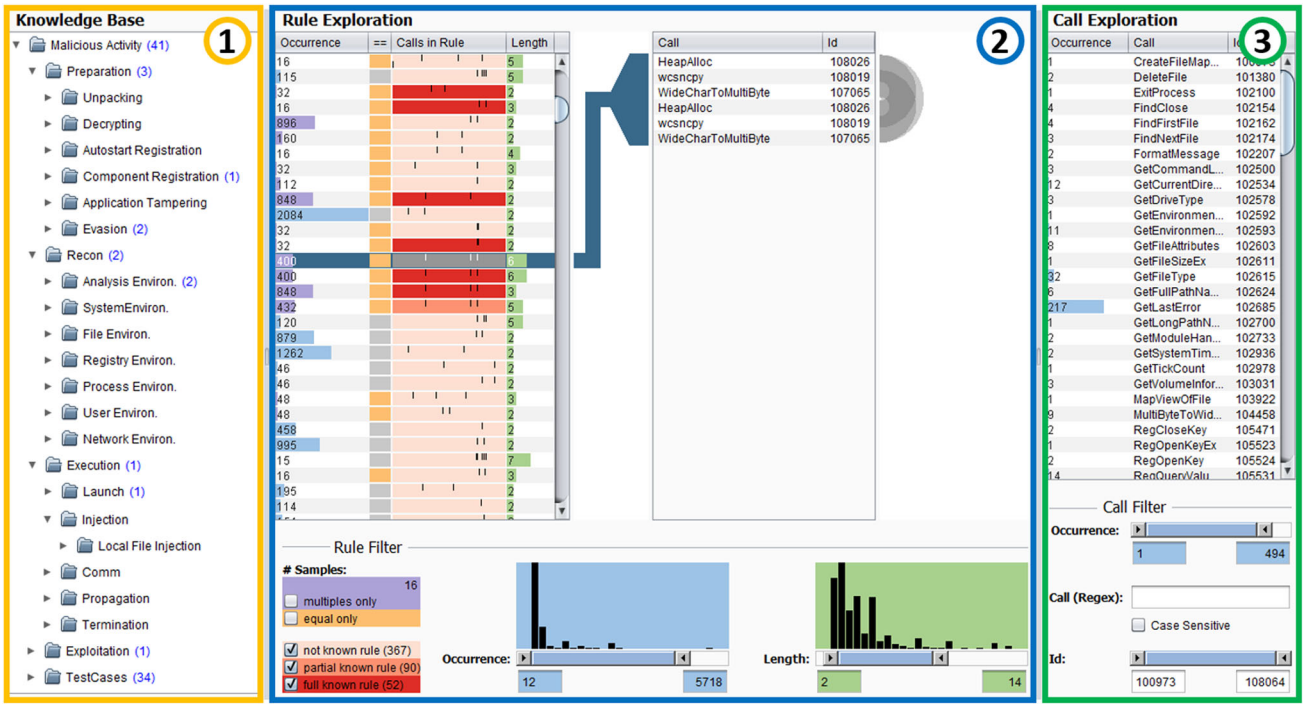

Fig. 6 Illustration of the Knowledge-assisted Malware Analysis System (KAMAS) designed to support malware analysts in their work. The interface encompasses a (1) 'Knowledge Base' for automated analysis and knowledge sharing between the analysts, a (2) 'Rule Exploration' area, and a (3) 'Call Exploration' area used to investigate individual events. Various filters at the bottom help to remove redundant data

ing a specific element expands the fully resolved rule for further study. Additionally, an arc diagram will be shown to highlight events that constitute a known sequence. The 'Call Exploration' area is positioned on the right side of the interface (Fig. 6:3), similar to the functions overview area in commonly used programming IDEs. Here, the analyst is provided with the ability to explore all of the events included in the rules that are presented/highlighted in the center element. The 'Call Exploration' view lists all events contained in the loaded file(s). To cope with the potentially huge amount of data, the analyst has the ability to use different, regular-expression-enabled filters to locate data of interest. Newly discovered patterns can be added into the KDB via a simple drag and drop action.

In addition to the IDE-like design decision, we used the Gestalt principles of proximity and similarity [23] to improve interface clarity. Each exploration area (Rule Explorer and Call Explorer) is expanded with its own filtering elements located directly below the visualized data. Based on all aforediscussed findings and concepts, we finally created the KAMAS VA prototype for visualizing and assessing the patterns generated by SEQUIN. Figure 6 depicts a screenshot of the main interface. Analysts have found the tool a useful addition to their workflow of exploring potentially harmful traces of events, be it system calls or OS kernel operations. A full evaluation of KAMAS, including a comprehensive usability study, is disseminated in [57]. The prototype has been released online under Creative Commons Attribution license[2].

8 Limitations and future work

While SEQUIN is a versatile solution that offers support to IT security experts and (malware) analysts alike, it is constrained by a number of limitations. As suggested in the discussion of Sections 6.1 and 6.2, the system does not currently provide an automated way of determining the optimal length of rules representative for an anomaly. The recursive nature of operation of the Sequitur algorithm is likely to produce rules within rules that, semantically speaking, amount to the same result. Right now, it is up to the analyst to impose length, TRR, and prevalence thresholds to counter this limitation.

On the data input side, we are currently required to internally concatenate the individual files and separate them with a file delimiter, which is skipped by Sequitur's rule building engine. This puts greater emphasis on the selection process of input traces, since SEQUIN will only start to infer rules that occur at least twice in the overall input. Too diverse an input will result in a lower compression ratio and anomaly detection accuracy. At the same time, the use of smart traces

[2] https://phaidra.fhstp.ac.at

sacrifices some of the trace's chronology by reordering events according to their process and thread context. While this usually improves the results, long-running processes of lengthy sessions will skew the timeline more than processes/threads that execute, run, and terminate within a shorter window. Here, a mechanism of intelligently segmenting traces will have to be developed to improve accuracy.

One of the other main areas of future improvement is undoubtedly the automated semantic interpretation of inferred variables, which, in the tool's current iteration, are assigned based on the operations (terminals) that constitute the respective rule. In the future, this representation will be updated to include actual attacker goals and (malicious) actions that go beyond the purpose-neutral label currently assigned by the naming schema. Ultimately, we plan to link the process to the team's previous work, which focuses on the development of a targeted attack ontology [28]. Automatically extracted events will be mapped to said ontology, thereby providing a complete view on likely attack scenarios induced by the events in question.

Another area of future research is the improved inclusion of the temporal domain. Currently, the order of events is maintained only within process and thread context (see 3.4). This allows for an investigation of sequences but does not consider the delay between two behavioral instances. As part of further abstraction efforts it is planned to prepend a temporal identifier to each event, which will transport information about the relative time and duration of execution.

In terms of validation, future work will encompass further proof of soundness for the attribute grammar specification used in the paper. Furthermore, we will formally test our behavioral engine against evaluation systems such as the one introduced by Filiol et al. [18]. Specific applications like the anomaly detection functionality discussed in Section 6.2 will also be evaluated using an even larger and more diverse set of behavioral traces to determine the best suited application scenarios. Furthermore, semantic synergies to the graph-based anomaly detection system [30] mentioned in Section 6.1 will be explored in depth. Optimizations for both systems are currently in the works.

On the knowledge discovery side, it is planned to continue development of the KAMAS visualization tool presented in Section 6. Specific functionality enabling further statistical assessment will be included to facilitate (malware) forensics, automated sample classification, and various intrusion detection scenarios coupled with a database of explicit domain knowledge.

In general, the automated cross-integration of visual analytics and knowledge discovery methods will be an integral part of our future research into the practical applications of the Sequitur approach.

9 Conclusion

This paper presented a grammar inference system based on an adapted version of Nevill-Mannings' Sequitur algorithm. Thanks to its versatile nature, the tool offers various benefits to the information security community, ranging from knowledge discovery in sequential system activity or malware execution traces to applied anomaly detection, baseline template generation, and grammar-enabled behavior discovery and interpretation.

We have successfully tested the induction and analysis system with several classes of input data. When used to streamline input traces for other, computationally expensive processes, we have achieved a significant reduction in complexity by extracting representative variables that describe relevant patterns. In our tests, the mean processing time for polynomial graph operations was reduced by more than 70%. Whilst comparative performance evaluations have not been performed, the results presented in Section 6 demonstrate the feasibility of using our grammar inference approach over systems that rely on the manual definition of rules. In addition, anomaly detection based on the rule density metric showed promising results by identifying deviating traces and their behavioral sequences in close to real time. Attack detection accuracy of an evaluated ICS sensor data scenario was well above 92% with a specificity of close to 91% (100% each when removing faulty benign runs), while each and every deviating behavioral pattern (benign and malicious both) was successfully extracted for further investigation. With KAMAS, we additionally introduced a visual analytics platform that uses the generated data to assist analysts in locating, extracting, and classifying relevant rules.

All in all, the introduced grammar inference tool can be used to quickly and accurately discover and highlight recurring patterns in sequential sets of arbitrary host and network event traces, thereby aiding in bridging the semantic gap between captured data and attacker behavior. The wide range of tested applications makes SEQUIN a unique tool in the repertoire of malware analysts and researchers alike.

Acknowledgements Open access funding provided by FH St. Pölten - University of Applied Sciences. The financial support by the Austrian Federal Ministry of Science, Research and Economy and the National Foundation for Research, Technology and Development is gratefully acknowledged. Furthermore, this work was supported by the Austrian Science Fund (FWF) via the KAVA-Time project (no. P25489).

References

1. Adriaans, P.W., et al.: Learning Shallow Context-Free Languages Under Simple Distributions. Institute for Logic, Language and Computation (ILLC), University of Amsterdam, Amsterdam (1999)
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles, Techniques. Addison Wesley, Boston (1986)
3. Angelov, K.: Incremental parsing with parallel multiple context-free grammars. In: Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics, pp. 69–76. Association for Computational Linguistics (2009)
4. Ates, K., Zhang K.: Constructing VEGGIE: machine learning for context-sensitive graph grammars. In: 19th IEEE International Conference on Tools with Artificial Intelligence, 2007. ICTAI 2007, vol. 2, pp. 456–463. IEEE (2007)
5. Ates, K., Kukluk, J., Holder, L., Cook, D., Zhang, K.: Graph grammar induction on structural data for visual programming. In: 18th IEEE International Conference on Tools with Artificial Intelligence, 2006. ICTAI'06, pp. 232–242. IEEE (2006)
6. Benteler, F.: Layout-graphgrammatiken für die darstellung von hierarchisch strukturierten graphen am beispiel von wellendigitalstrukturen. (2002)
7. Bilge, L., Dumitras, T.: Before we knew it: an empirical study of zero-day attacks in the real world. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 833–844. ACM (2012)
8. Briscoe, E.J.: Language as a complex adaptive system: co-evolution of language and of the language acquisition device. In: Proceedings of Eighth Computational Linguistics in the Netherlands Conference (1998)
9. Chen, M., Ebert, D., Hagen, H., Laramee, R.S., Van Liere, R., Ma, K.-L., Ribarsky, W., Scheuermann, G., Silver, D.: Data, information, and knowledge in visualization. Comput. Graph. Appl. **29**(1), 12–19 (2009). https://doi.org/10.1109/MCG.2009.6. ISSN 0272-1716
10. Clark, A.: Unsupervised induction of stochastic context-free grammars using distributional clustering. In: Proceedings of the 2001 Workshop on Computational Natural Language Learning, Vol. 7, p. 13. Association for Computational Linguistics (2001)
11. Collins, C., Viegas, F.B., Wattenberg, M.: Parallel tag clouds to explore and analyze faceted text corpora. In: Sympossium on Visual Analytics Science and Technology, pp. 91–98 (2009). https://doi.org/10.1109/VAST.2009.5333443
12. Creech, G., Hu, J.: A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns. IEEE Trans. Comput. **63**(4), 807–819 (2014)
13. Déjean, H.: ALLiS: a symbolic learning system for natural language learning. In: Proceedings of the 2nd Workshop on Learning Language in Logic and the 4th Conference on Computational Natural Language Learning, vol. 7, pp. 95–98. Association for Computational Linguistics (2000)
14. Dornhackl, H., Kadletz, K., Luh, R., Tavolato, P.: Defining malicious behavior. In: Ninth International Conference on Availability Reliability and Security (ARES), pp. 273–278. IEEE (2014)
15. DUlizia, A., Ferri, F., Grifoni, P.: A survey of grammatical inference methods for natural language learning. Artif. Intell. Rev. **36**(1), 1–27 (2011)
16. Efron, B., Tibshirani, R.J.: An Introduction to the Bootstrap. CRC Press, Boca Raton (1994)
17. Eiland, E.E., Evans, S.C., Markham, T.S., Impson, J.D.: MDL compress system and method for signature inference and masquerade intrusion detection, December 4 (2012) https://www.google.com/patents/US8327443. US Patent 8,327,443
18. Filiol, E., Jacob, G., Le Liard, M.: Evaluation methodology and theoretical model for antiviral behavioural detection strategies. J. Comput. Virol. **3**(1), 23–37 (2007)
19. Gotz, D., Stavropoulos, H., Sun, J., Wang, F.: ICDA: a platform for intelligent care delivery analytics. In: AMIA Annual Sympossium Proceedings, vol. 2012, pp. 264–273, (2012). ISSN 1942-597X http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3540495/
20. Grünwald, P.D.: A Tutorial Introduction to the Minimum Description Length Principle. In: Advances in Minimum Description Length: Theory and Applications. MIT Press (2005)
21. Hoffman, D., Richard, S.: Trace specifications: methodology and models. IEEE Trans. Softw. Eng. **14**(9), 1243–1252 (1988)
22. Jacob, G., Debar, H., Filiol, E.: Malware behavioral detection by attribute-automata using abstraction from platform and language. In: International Workshop on Recent Advances in Intrusion Detection, pp. 81–100. Springer (2009)
23. Johnson, Jeff.: Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Guidelines. Morgan Kaufmann, Amsterdam ; Boston, 2 edition, (2014). ISBN 978-0-12-407914-4
24. Joo, S.W., Chellappa, R.: Attribute grammar-based event recognition and anomaly detection. In: Conference on Computer Vision and Pattern Recognition Workshop, 2006. CVPRW'06, pp. 107–107. IEEE (2006)
25. Keim, D.A.: Designing pixel-oriented visualization techniques: theory and applications. IEEE Trans. Vis. Comput. Graph. **6**(1), 59–78 (2000). https://doi.org/10.1109/2945.841121. ISSN 1077-2626
26. Kuhn, H.W.: The Hungarian method for the assignment problem. Naval Res. Logist. Q. **2**(1–2), 83–97 (1955)
27. Luh, R., Marschalek, S., Kaiser, M., Janicke, H., Schrittwieser, S.: Semantics-aware detection of targeted attacks: a survey. J. Comput. Virol. Hack. Tech. **13**, 1–39 (2016)
28. Luh, R., Schrittwieser, S., Marschalek, S.: TAON: An ontology-based approach to mitigating targeted attacks. In: Proceedings of the 18th International Conference on Information Integration and Web-Based Applications & Services. ACM (2016)
29. Luh, R., Schramm, G., Wagner, M., Schrittwieser, S.: Sequitur-based inference and analysis framework for malicious system behavior. In: Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017), pp. 632–643, (2017). ISBN 978-989-758-209-7. https://doi.org/10.5220/0006250206320643
30. Luh, R., Schrittwieser, S., Marschalek, S., Janicke, H.: Design of an anomaly-based threat detection & explication system. In: Proceedings of the 3rd International Conference on Information Systems Security & Privacy. SCITEPRESS (2017)
31. Marschalek, S., Luh, R., Kaiser, M., Schrittwieser, S.: Classifying malicious system behavior using event propagation trees. In: Proceedings of the 17th International Conference on Information Integration and Web-Based Applications & Services. Association for Computational Linguistics (2015)
32. McClosky, D., Charniak, E., Johnson, M.: Effective self-training for parsing. In: Proceedings of the Main Conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, pp. 152–159. Association for Computational Linguistics (2006)
33. Miksch, S., Aigner, W.: A matter of time: applying a data-users-tasks design triangle to visual analytics of time-oriented data. Comput. Graph. **38**, 286–290 (2014). https://doi.org/10.1016/j.cag.2013.11.002
34. Ming, L., Vitányi, P.: An Introduction to Kolmogorov Complexity and Its Applications. Springer, Heidelberg (1997)
35. Munsey, C.: Economic Espionage: Competing For Trade By Stealing Industrial Secrets. (2013) https://leb.fbi.gov/2013/

october-november/economic-espionage-competing-for-trade-by-stealing-industrial-secrets. Accessed 15 Sept 2015
36. Nakamura, K., Ishiwata, T.: Synthesizing context free grammars from sample strings based on inductive CYK algorithm. In: International Colloquium on Grammatical Inference, pp. 186–195. Springer (2000)
37. Nevill-Manning, C.G., Witten, I.H.: Identifying hierarchical structure in sequences: a linear-time algorithm. J. Artif. Intell. Res. (JAIR) **7**, 67–82 (1997)
38. Petasis, G., Paliouras, G., Karkaletsis, V., Halatsis, C., Spyropoulos, C.D.: e-GRIDS: computationally efficient gramatical inference from positive examples. Grammars **7**, 69–110 (2004)
39. Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic analysis of malware behavior using machine learning. J. Comput. Secur. **19**(4), 639–668 (2011). ISSN 0926-227X
40. Rissanen, J.: Modeling by shortest data description. Automatica **14**(5), 465–471 (1978)
41. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1. World Scientific, Singapore (1997)
42. Sakakibara, Y., Kondo, M.: Ga-based learning of context-free grammars using tabular representations. In: ICML, vol. 99, pp. 354–360 (1999)
43. Sedlmair, M., Meyer, M., Munzner, T.: Design study methodology: reflections from the trenches and the stacks **18**(12), 2431–2440 (2012). ISSN 1077-2626. https://doi.org/10.1109/TVCG.2012.213
44. Seginer, Y.: Fast unsupervised incremental parsing. In: Annual Meeting-association for Computational Linguistics, vol. 45, p. 384 (2007)
45. Senin, P., Lin, J., Wang, X., Oates, T., Gandhi, S., Boedihardjo, A.P., Chen, C., Frankenstein, S., Lerner, M.: Grammarviz 2.0: a tool for grammar-based pattern discovery in time series. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pp. 468–472. Springer (2014)
46. Senin, P., Lin, J., Wang, X., Oates, T., Gandhi, S., Boedihardjo, A.P., Chen, C., Frankenstein, S.: Time series anomaly discovery with grammar-based compression. In: EDBT, pp. 481–492 (2015)
47. Solan, Z., Horn, D., Ruppin, E., Edelman, S.: Unsupervised learning of natural languages. Proc. Nat. Acad. Sci. U.S.A. **102**(33), 11629–11634 (2005)
48. Sood, A.K., Enbody, R.J.: Targeted cyberattacks: a superset of advanced persistent threats. IEEE Secur. Priv. **11**(1), 54–61 (2013)
49. Steedman, M., Osborne, M., Sarkar, A., Clark, S., Hwa, R., Hockenmaier, J., Ruhlen, P., Baker, S., Crim, J.: Bootstrapping statistical parsers from small datasets. In: Proceedings of the Tenth Conference on European Chapter of the Association for Computational Linguistics, vol. 1, pp. 331–338. Association for Computational Linguistics (2003)
50. Stevenson, A., Cordy, J.R.: A survey of grammatical inference in software engineering. Sci. Comput. Program. **96**, 444–459 (2014)
51. Symantec. Symantec Internet Security Threat Report, vol. 20. Whitepaper, (2015)
52. Thomas, J.J., Cook, K.A. (eds.): Illuminating the Path: The Research and Development Agenda for Visual Analytics. IEEE (2005). ISBN 0769523234
53. Thompson, G.R., Flynn, L.A.: Polymorphic malware detection and identification via context-free grammar homomorphism. Bell Labs Tech. J. **12**(3), 139–147 (2007)
54. Van Zaanen, M.: ABJ: alignment-based learning. In: Proceedings of the 18th Conference on Computational Linguistics, vol. 2, pp. 961–967. Association for Computational Linguistics (2000)
55. Wagner, M., Aigner, W., Rind, A., Dornhackl, H., Kadletz, K., Luh, R., Tavolato, P..: Problem characterization and abstraction for visual analytics in behavior-based malware pattern analysis. In: Whitley, K., Engle, S., Harrison, L. Fischer, F., Prigent, N. (Eds.) Proceedings of 11th Workshop on Visualization for Cyber Security, VizSec, pp. 9–16. ACM (2014) https://doi.org/10.1145/2671491.2671498
56. Wagner, M., Fischer, F., Luh, R., Haberson, A., Rind, A., Keim, D., Aigner, W., Borgo, R., Ganovelli, F., Viola, I.: A survey of visualization systems for malware analysis. In: Eurographics Conference on Visualization, pp. 105–125. EuroGraphics, (2015)
57. Wagner, M., Rind, A., Thür, N., Aigner, W.: A knowledge-assisted visual malware analysis system: design, validation, and reflection of kamas. Comput. Secur. **67**, 1–15 (2017). https://doi.org/10.1016/j.cose.2017.02.003. ISSN 0167-4048
58. Wattenberg, M., Viegas, F.B.: The word tree, an interactive visual concordance. IEEE Trans. Vis. Comput. Graph. **14**(6), 1221–1228 (2008). https://doi.org/10.1109/TVCG.2008.172. ISSN 1077-2626
59. Wattenberg, M.: Arc diagrams: visualizing structure in strings. In: Proceeding of IEEE Sympossium Information Visualization (InfoVis), pp. 110–116. (2002) https://doi.org/10.1109/INFVIS.2002.1173155
60. Wegner, P.: Why interaction is more powerful than algorithms. Commun. ACM **40**(5), 80–91 (1997). ISSN 0001-0782
61. Wongsuphasawat, K., Gotz, D.: Outflow: visualizing patient flow by symptoms and outcome. In: IEEE VisWeek Workshop on Visual Analytics in Healthcare, Providence, Rhode Island, USA (2011)
62. Zhao, C., Kong, J., Zhang, K.: Program behavior discovery and verification: a graph grammar approach. IEEE Trans. Softw. Eng. **36**(3), 431–448 (2010)