CCDCOE
NATO COOPERATIVE
CYBER DEFENCE
CENTRE OF EXCELLENCE

IDS for logs: Towards implementing a streaming Sigma rule engine

Markus Kont
NATO CCDCOE Technology Branch Researcher
Mauno Pihelgas
NATO CCDCOE Technology Branch Researcher

Tallinn 2020

About the author

Markus Kont is a Researcher in the Technology branch of the NATO Cooperative Cyber Defence Centre of Excellence (CCDCOE). His area of expertise is packet capture and log processing, DevOps tools and techniques and data science. His current work involves researching stream processing techniques and he is responsible for teaching network security monitoring tools in the CCDCOE. In his previous life, he was a server administrator in a hosting company for over five years, focusing mostly on Linux systems and back-end infrastructure development. He holds a Master's degree in Cyber Security from Tallinn University of Technology.

Mauno Pihelgas has been a Researcher in the Technology branch of the CCDCOE since 2013. His area of expertise is monitoring, data mining and situational awareness. Prior experience includes five years as a monitoring administrator and developer for the largest telecommunications operator in Estonia. In addition to being a GIAC GMON Continuous Monitoring Certified Professional, he is also a Red Hat Certified System Administrator, Red Hat Certified Engineer and a Red Hat Certified Specialist in Ansible Automation. Mauno holds a Master's of Science degree in Cyber Security and is pursuing a PhD at the Tallinn University of Technology.

CCDCOE

The NATO CCDCOE is a NATO-accredited cyber defence hub focusing on research, training and exercises. It represents a community of 29 nations providing a 360-degree look at cyber defence, with expertise in technology, strategy, operations and law. The heart of the Centre is a diverse group of international experts from military, government, academia and industry backgrounds.

The CCDCOE is home to the *Tallinn Manual 2.0*, the most comprehensive guide on how international law applies to cyber operations. The Centre organises the world's largest and most complex international live-fire cyber defence exercise Locked Shields and hosts the International Conference on Cyber Conflict (CyCon), a unique annual event in Tallinn, joining key experts and decision-makers from the global cyber defence community. As the Department Head for Cyberspace Operations Training and Education, the CCDCOE is responsible for identifying and coordinating education and training solutions in the field of cyber defence operations for all NATO bodies across the Alliance.

The Centre is staffed and financed by its member nations: Austria, Belgium, Bulgaria, Croatia, the Czech Republic, Denmark, Estonia, Finland, France, Germany, Greece, Hungary, Italy, Latvia, Lithuania, Montenegro, the Netherlands, Norway, Poland, Portugal, Romania, Slovakia, Slovenia, Spain, Sweden, Switzerland, Turkey, the United Kingdom and the United States. NATO-accredited centres of excellence are not part of the NATO Command Structure.

www.ccdcoe.org
publications@ccdcoe.org

Table of Contents

1. Abstract

Signatures are the staple of threat detection. The Sigma rule format has emerged in recent years to fulfil this role for event logs and has become increasingly popular in security operations and threat hunting communities. The Public Sigma project provides rules and tooling for conversion into various SIEM provider alerting formats. It is not a match engine, so users are still expected to deploy a log management solution. In many cases, this is not ideal and real-time integration into existing log streams would be preferred. NATO CCDCOE has organised the Crossed Swords exercise for red-team training and the yellow team feedback system is one such use. We implemented an experimental rule engine in Golang and made the source code publicly available. Since then, we have rewritten that engine so it would serve as a better reference for anyone who needs to implement such a solution in their own environment. To support the goal, this paper provides a detailed technical outline of our implementation. We also conducted performance benchmarks to assess potential limitations of our approach and propose further developments.

2. Introduction

Pattern-based intrusion detection systems (IDS) form the cornerstone of network security monitoring (NSM) and threat detection. Network defenders must be able to learn from past mistakes to cut off the attacker's kill-chain as soon as possible. Defenders must also be able to easily share indicators of compromise (IoC) to collaboratively tackle threats and improve incident response efficiency. The network security community has used real-time IDS patterns for decades as a solution to this problem, with Suricata ("Suricata," 2020) and Snort ("Snort," 2020) having emerged as *de facto* open-source tools for the task. Likewise, YARA ("YARA," 2020) fulfills this role for files in malware analysis.

Pattern matching on event logs is usually relegated to data analytics platforms that function as SIEM systems and data warehouses. They usually ingest logs to an internal archive while maintaining indexes for fast data retrieval, commonly with custom query language and APIs. Elasticsearch ("Elastic stack," 2020) and Splunk ("Splunk," 2020) are well-known and heavily adopted products that employ this method. These tools commonly implement mechanisms for defining alert rules, but not in a way that is interoperable with competing products. Each organisation uses different software stacks and thus deals with different kinds of logs. With security data being sensitive by nature, this has left the logging field largely without a central repository of open and community-driven alert rules, something that packet capture communities have enjoyed for decades, at least until recent years.

Sigma (Roth, 2020a) was developed as a *generic signature format for SIEM systems*. It defines a flexible rule structure in yaml format, provides python tools for rule parsing and conversion to supported SIEM rule formats and maintains an up-to-date repository of community-provided rules for threat detection. Essentially, the goal of the project is to avoid vendor lock-in and to simplify event log IoC sharing, as alert rules written in sigma could be converted to any vendor format. Sigma does not do any pattern matching or alerting by itself; rather it acts as a translation layer and IoC sharing platform. Thus, users are still expected to maintain a fully-fledged SIEM system with alerting capabilities.

These systems are well known for their significant resource requirements not only in terms of hardware and licensing fees, but also in daily system administration. Someone, usually a team of people, needs to monitor those tools, keep them up to date, re-index data if needed, keep up with new features and depreciations, communicate support requests, verify supply chain integrity, etc. By comparison, implementing a custom tool is often frowned upon, especially in corporate and military environments, as it is often seen as the more difficult option. This paper has two goals. First, to demonstrate a counterpoint that a small custom-built streaming tool can easily handle a task usually relegated to much larger databases. Second, to present a technical specification for implementing a streaming Sigma rule engine, something that to the best of our knowledge did not exist during Crossed Swords 2020.

2.1 Real-time engine for Crossed Swords 2020

The NATO CCDCOE has organised Exercise Crossed Swords (XS) each year since 2015. It aims to train the red team, whereas the yellow team is tasked with providing real-time feedback to players. During those years, we have been developing *Frankenstack* (Kont et al., 2017), a pipeline of open-source security and analytics tools for collecting raw data from targets and game network capture, to filter, to correlate and finally to present digested information to players as feedback on their activities. While designing this stack, we are faced with numerous constraints.

Feedback should reach players as soon as possible, with minimal human intervention. If a red teamer executes an action within the gamenet, the arteafacts, if any, should trigger a timely notification on a central dashboard. A delayed data feed would most likely go unseen by players, as the exercise already subjects them to significant cognitive pressure. Data platforms that rely on bulk database

ingestion and indexing already have an inherent delay before data becomes available for querying, being near-real-time as opposed to true-real-time systems. A five- to fifteen-minute delay would make information on dashboards obsolete as playing teams might have moved on to their next objectives.

The core of the feedback system must be vendor-agnostic and minimalist in design. Building the exercise means a heavy reliance on volunteer contributions and supporting nations. If the system core relied on an external vendor or contributor and that contributor was unable to attend, the entire exercise could be compromised. The compressed time frame of an exercise also leaves little or no time for debugging, even for a very familiar tool. SIEM solutions are infamous for their monolithic design and therefore not a good fit for this role.

The core needs a simple rule language that fits into an asynchronous multi-consumer pub-sub model. We do not exclude existing analytic platforms, SIEM systems or vendor products from our stack. Elasticsearch has been a part of our stack since the beginning and many security vendors have joined our team to test their products in a unique live-fire environment. Rather, we do not treat any one security product as an end-all solution, but as another input source that produces a processed feed back to the system. Many tools consume the same input independently and, in some cases, somewhat redundantly and provide us with their unique insight. We need to be able to tie these feeds together.

Previously, we used simple event correlator (SEC) for this task (Vaarandi, Blumbergs & Çalişkan, 2015). However, we soon concluded that SEC was not ideal for the role, being initially designed for complex event correlation on unstructured messages by using regular expressions to extract significant variables. All relevant event streams were already configured to output structured JSON events at source or normalised at the pre-processing phase. Rules therefore needed to select the correct JSON key and do a basic variable comparison or string matching on the value, but this meant writing rather unseemly Perl functions into most of our SEC rules. While calling Perl functions was indeed possible, accommodating this within the SEC rule syntax was cumbersome. Thus, our ruleset did not use more powerful correlation features, being limited mostly to *Simple*, *SimpleWithThreshold* and relatively few *EventGroup* rules. Put simply, we found the complexity unwarranted relative to our gains.

Processing complex nested data structures directly within a full-blown programming environment seemed more promising. In hindsight, it seems we tried to correlate events too soon in the processing pipeline; SEC is an event correlation tool, not a programming language or data processing tool. Unfortunately, in our ruleset we tried to accommodate many of the data processing and transformation tasks which should have been completed prior to pushing events into SEC. This hindered the rule-writing process and the lack of proper post-processing meant that even minor changes in the input event structure resulted in the need to rewrite a large portion of the rules.

Atomic patterns recognition must be separated from correlation logic. Atomic events are raw messages emitted by a target system or sensor. Domain-specific knowledge and prior experience is needed to decide if they indicate a relevant security event. Some people specialise in Windows systems, others in Linux or network security. Event processing and correlation is a form of data science and big data analytics. Finding a person with such extensive set of skills is difficult, yet many rule formats combine those two sides. Security practitioners should be able to easily express their knowledge to produce contextualised events to analytics experts.

We believe those requirements to also be applicable to good systems design in real-world production environments.

2.2 Implementation language

Sigma rule format fits these requirements well, but no open-source project existed for implementing it as a match engine and the existing Sigma toolset is written entirely in Python 3, an interpreted high-level programming language with *batteries included* mentality that is designed to be easily readable and

simple to use. That is an excellent choice when the project scope is limited to processing highly dynamic input, executing offline conversion operations and sharing the logic in an easily approachable and newcomer-friendly manner. However, dynamically typed programming languages are not ideal for implementing streaming systems with heightened stability expectations that are constantly exposed to unknown input. Python is infamous for being easy to get started with and highly productive in small applications, yet incredibly difficult to debug and prone to unforeseen errors as the complexity of code base increases over the application's lifetime.

For example, passing a list to a function that is designed to process integers would result in a compile error in a statically typed language, yet be perfectly fine in a dynamically typed one and result in a runtime exception only when that code branch is executed. In some cases, this might not happen until the application has been in production for an extended amount of time. Changes to existing code base are also highly likely to introduce new bugs. Thus, statically typed languages require additional work during initial implementation, but produce more reliable software overall as many classes of bugs are discovered during compilation.

Golang (Go programming language, 2020) was chosen for implementing the rule parser and match engine. Like Python, it is a simple and productive language, being developed by Google as a less verbose, faster to build and easier to manage replacement for C++ and Java. Unlike Python, however, it does not compromise on type safety, preferring to compile the entire code base along with dependencies to a single binary. As a result, binaries built with native Go can be shipped to any system targeted by the compile phase just like a python script, but without any need for dependency management on the client. It was also designed from inception to have a powerful concurrency model, an area that Python is severally lacking.

Our initial core implementation of streaming Sigma rule engine used during XS2020 stands at 2,500 lines of Go including tests and is publicly available in Github (Go Sigma Rule Engine, 2020). Subsequent sections explain the sigma rule format and technical details of our implementation in detail.

3. Sigma rule structure

Before implementing a parser, let us explore the Sigma rule format. The analyst might be interested in tracking client-side beacons emitting from compromised workstations. A common red team technique is to deliver the malware payload with Powershell, whereas the malicious code is obfuscated in base64 encoding. The full malicious script may be visible in event logs if Powershell or Sysmon (Windows Sysinternals - Sysmon, 2020) logging has been enabled.

```
$s=New-Object IO.MemoryStream(,[Convert]::FromBase64String(\"OMITTED BASE64 STRING\"));
```

A truncated ECS-formatted (Elastic Common Schema Overview, 2020) structured event message would contain the following structure. Note that interesting values are in the *channel* and *ScriptBlockText* fields.

```
{
  "event_id": 4104,
  "channel": "Microsoft-Windows-PowerShell/Operational",
  "task": "Execute a Remote Command",
  "opcode": "On create calls",
  "version": 1,
  "record_id": 1559,
    "winlog": {
        "event_data": {
            "MessageNumber": "1",
            "MessageTotal": "1",
            "ScriptBlockText": "$s=New-Object IO.MemoryStream(,[Convert]::FromBase64String(\"OMITT
ED BASE64 STRING\"));",
            "ScriptBlockId": "ecbb39e8-1896-41be-b1db-9a33ed76314b"
        }
    }
}
```

The analyst can then describe the event in Sigma format. Note how the rule specifies the interesting pattern along with where it might be observed and defines various meta data fields present in Sigma specification (Roth, 2020b).

```
author: Mauno Pihelgas
description: >
  Detects suspicious PowerShell
  invocation command parameters
detection:
  condition: selection
  selection:
    winlog.event_data.ScriptBlockText:
    - ' -FromBase64String'
falsepositives:
- Penetration tests
- Very special / sneaky PowerShell scripts
fields:
- winlog.event_data.ScriptBlockText
id: 697e4279-4b0d-4b14-b233-9596bc1cacda
level: high
logsource:
```

```
  product: windows
  service: powershell
status: experimental
tags:
- attack.execution
- attack.defense-evasion
- attack.t1064
title: Encoded ScriptBlock Command Invocation
```

We can define following Go *struct* to express Sigma fields and their respective types. For the uninitiated, a *struct* is a custom type which acts as a collection of variables. These variables can be of any built-in or custom type. However, those types must be known beforehand.

```
type Rule struct {
    Author        string   `yaml:"author" json:"author"`
    Description   string   `yaml:"description" json:"description"`
    Falsepositives []string `yaml:"falsepositives" json:"falsepositives"`
    Fields        []string `yaml:"fields" json:"fields"`
    ID            string   `yaml:"id" json:"id"`
    Level         string   `yaml:"level" json:"level"`
    Title         string   `yaml:"title" json:"title"`
    Status        string   `yaml:"status" json:"status"`
    References    []string `yaml:"references" json:"references"`

    Logsource `yaml:"logsource" json:"logsource"`
    Detection `yaml:"detection" json:"detection"`
    Tags      `yaml:"tags" json:"tags"`
}
```

Most sigma fields are either *string* text or *lists* of them. They do not really serve a purpose in rule engine implementation, except for optional event tagging or grouping. Only the *detection* field is really needed for creating a pattern-matching object and thus the only struct element used by rule parser. However, *id*, *logsource* and *tags* fields were also used, albeit only on the calling application side for pre- and post-processing. Firstly, let us consider the *logsource* field.

```
type Logsource struct {
    Product    string `yaml:"product" json:"product"`
    Category   string `yaml:"category" json:"category"`
    Service    string `yaml:"service" json:"service"`
    Definition string `yaml:"definition" json:"definition"`
}
```

As the name implies, the *logsource* field defines the message source. However, we found the overall usage of this field to be fairly arbitrary and inconsistent. Therefore, we opted to transform the *logsource* values to fit our data model before parsing the ruleset. Our streaming tool would then use it as a pre-filter to match incoming events to particular rulesets. For example, an event originating from *snoopy* audit daemon should never be exposed to a ruleset written explicitly for Windows events. Likewise, the *tags* field was only used post-match for decorating detected events with metadata.

```
type Tags []string

type Result struct {
    Tags
```

```
    ID, Title string
}

type Results []Result
```

Each atomic event could match multiple rules. Therefore, our ruleset would return a list *Result* objects, each containing a matching rule name, identifier and predefined tags. Many rules in public Sigma rulesets are tagged with tactics and techniques from the MITRE ATT&CK (2020) framework. Our post-processing tool would then parse those values into unique identifiers in an Enterprise ATT&CK matrix that could be used for event correlation and deduplication, enrich matching events with those identifiers and emit the event to the alert channel.

Finally, we defined our detection field simply as a wrapper of a hashmap with unknown elements.

```
type Detection map[string]interface{}
```

We encountered quite large variation of fields in this map, as the values can contain *strings*, *maps* of *strings* or numbers, lists of strings or numbers, etc. In Go, the only choice in this scenario is to use an empty *interface* which means that the value is unknown *a priori*. Each value in that map needs to be type cast in runtime, to handle all possible scenarios. Those cases will be explored in the next subsection.

3.1 Detection field

Having looked at an example rule and explored its structure in Go, let us investigate the structure of the *detection* element. Consider our rule example from the previous section:

```
detection:
  condition: selection
  selection:
    winlog.event_data.ScriptBlockText:
    - ' -FromBase64String'
```

The condition field defines the search expression whereas other fields are known as identifiers, acting as pattern containers. Note that the condition field could easily be omitted if the detection map only contains one identifier, but should be mandatory when two or more identifiers are tied together with a logical conjunction or disjunction.

```
detection:
  condition: selection1 AND selection2
  selection1:
    winlog.event_data.ScriptBlockText:
    - ' -FromBase64String'
  selection2:
    task: "Execute a Remote Command"
```

Keys inside identifiers represent the values that are to be extracted from structured event messages, whereas dot notation signifies nested JSON structures. Values can be strings, numbers, Booleans or lists of those respective data types. Lists represent all possible patterns which would indicate malicious activity and thus are joined by logical disjunction. Selections are joined by logical conjunction, as they should help the rule writer to specify a more exact pattern. Thus, the previous example could be rewritten thus:

```
detection:
  condition: selection1
```

```
selection1:
  winlog.event_data.ScriptBlockText:
  - ' -FromBase64String'
  task: "Execute a Remote Command"
```

Identifiers can broadly be classified into two categories – *selection* and *keywords*. The former represents key and value pairs in applicable event messages, as presented in prior examples. The latter is simply a list of possible patterns in an unstructured message or primary body section.

```
condition: keywords
keywords:
  - 'wget * - http* | perl'
  - 'wget * - http* | sh'
  - 'wget * - http* | bash'
  - 'python -m SimpleHTTPServer'
```

Note the lack of message keys. When faced with this rule while parsing structured messages, the match engine implementor is left with two options: either to exhaustively try all fields, or to make an educated assumption and select the most appropriate one. For example, the syslog message structure is well known for delivering most useful information in the unstructured *MSG* field. Likewise, Windows event logs have a multi-line *message* section which often delivers many useful information pieces which are not present in other fields. Audit logs such as snoopy often present full commands executed by the user. The choice would be obvious in these cases and the *keyword* identifier could therefore be used as shorthand notation for accessing these fields. Naturally, the rule writer can mix *keyword* and *selection* identifiers as needed. Users can also rename those identifiers to contextualise their purpose. For example, negated identifiers often have *filter* or *falsepositives* in their naming pattern.

```
condition: selection and not filter
```

The only way to differentiate between *selection* and *keyword* identifier type is to check for *keyword* prefix in the identifier name and verify that the corresponding item is a list or raw value. Otherwise, we assume *selection* and verify that item is a map or a list of maps. We found that this approach worked relatively well in practice while parsing the public rules, regardless of being a best-effort approach. While the rule language could be enhanced with standardised notation to make this distinction more clear, we do not believe extra complexity to be warranted as identifier data types need to be validated by the rule parser, thus making extra standardisation redundant in practice while possibly making grouped and nested expressions needlessly complicated.

```
condition: ( selection_1 and selection_2 ) or ( selection_3 and selection_4 )
```

These groupings are common in public Sigma repositories, though deep nesting is rarely used. Regardless, there is no theoretical limit to expression depth and any properly implemented parser should be able to recursively handle such a scenario. Expression groups can become quite lengthy. Many sigma rules use wildcards as a convenience feature.

```
condition: all of selection* and not 1 of filter* | count() > 10
```

Note that this expression also contains an aggregation which is separated from the main query expression via the *pipe* symbol. These constructs, along with the *them* identifier, were not implemented in the initial match engine due to time limitations before the exercise began. We did not deem the number of existing rules using these constructs sufficient to justify the extra implementation effort, particularly aggregations as their existence would expand our initial project scope from writing a streaming match engine to implementing a fully-fledged event correlator. Our stream platform makes extensive use of the Go fan-out concurrency model, whereby the processor module spawns a user-defined number of workers that consume messages from a common thread-safe channel. Each worker executes processing logic per message and produces processed messages to the output channel. Implementing

a *count* operator would mean either ensuring that all messages subject to a particular rule would be sent to the same worker, or maintaining a shared thread-safe counter using locking mechanisms or separate concurrent worker. *Near* keywords and streaming thresholds also require a full sliding window implementation, so a conscious decision was made to simply fall back on other correlation engines or batch queries if needed. This feature did present itself during the exercise and we will consider implementing these mechanisms in future work.

Likewise, we simply decided to rewrite any wildcard and *them* keyword rules to a simpler format in the initial engine prototype. To clarify, the *all of* expression is essentially a shorthand for logical conjunction, whereas *1 of* is for disjunction. A match expression presented above could therefore be rewritten thus:

```
condition: (selection1 AND selection2 AND selectionN) AND NOT (filter1 OR filter2 OR filterN)
```

The second engine version now supports this expansion, allowing the rule writer to use shorthand notation. Another construct missing in the initial version that is now supported is piped specifiers in the selection keys.

```
selection:
  CommandLine|endswith: '.exe -S'
  ParentImage|endswith: '\services.exe'
```

These expressions were relatively rare while building the stream engine but have since grown in popularity. The initial version simply splits those keys using the pipe symbol as delimiter, ignoring the specifier. However, second version uses *HasPrefix()*, *HasSuffix()* and *Contains()* methods from *strings* package in the Go standard library, depending on specifier *enum* value as extracted from the key.

Finally, many existing rule files contain multiple YAML documents in the same file separated by a YAML document start operator (---). We noted that these rules simply incorporate multiple *logsource* and *detection* sets that signify similar or related events (e.g., detecting various types of credential dumping) with just a few keys that are changing.

```
title: ...
id: ...
description: ...
---
logsource: ...
detection: ...
---
logsource: ...
detection: ...
```

While this shorthand notation makes writing the rules easier, these rules needed to be expanded with the otherwise identical base metadata before loading them into the match engine. As an interesting aside, most of the existing rule files omit the start operator at the beginning of the ´yaml´ file, which is fine in the case of the PyYAML parser, but some other parsers (e.g., Java's Jackson) might not process these files.

Therefore, we decided not to handle these raw rules within the match engine, but rather to pre-process the rules using a custom ruleset transformation script written in Python. This transformation was needed anyway, as the *logsource* and generic field names needed to be customised for data streams in our environment.

4. Ruleset transformation

Ruleset transformation was required for several reasons. First, we split rule files that contain one or more related rules into separate files. Second, we opted to employ our own *logsource* values and use them as a pre-filter to optimise the match engine and to scan only applicable rulesets (e.g., Windows event log messages should only be matched against rules written for Windows logs). Third, we needed to convert the generic field names used in the rule language into specific keys that are produced by tools in our data pipeline (e.g., Winlogbeat and Suricata).

The following steps are combined into a pre-processing Python script that loads all the rule files, makes the necessary transformations and writes the new file using the original directory structure (e.g., ´linux´, ´windows´, ´web´, etc.) of the original rules.

4.1 Split combined rules

To handle multiple rules in the same ´yaml´ file, the script first scans the file to detect if it contains multiple rules. If so, the script splits the file into separate YAML documents by document start operator (---). We then load the first YAML section in the file which typically contains all the required elements of a rule. The script processes and stores this first document as a separate rule, then proceeds to load the following documents, while retaining the first document as the template which contains the identical base metadata.

4.2 Log source

The sigma rules provide a flexible way to describe rules for any system, product or tool. In many cases the *product* field is fairly generic, such as *linux* or *windows*. However, sometimes the rule writer has been more specific and provided a rule that matches the output of a specific tool (e.g., the Qualys vulnerability scanner). Since we did not have these tools in our exercise environment, we simply left the *logsource* field unchanged in such cases. Other rules only provide a *category* instead of a specific system or product that would produce this log. For example, the categories *dns* and *firewall* were mapped to *suricata* as the most likely data source for network-specific rules. Overall, we specified four *product* values that matched our primary data sources: *Windows*, *Linux*, *Suricata* and *Snoopy*.

4.3 Field names

The Sigma rules often contain generic or standard field names for various types of systems and log sources. This is intentional, since rule writers cannot foresee all keys produced by different monitoring tools. For example, when describing the source IP of a network event, Sigma rules often use the *ClientIP* keyword. However, the Suricata EVE format uses the *src_ip* name and Zeek uses the *id.orig_h* field.

The same concept was applied for converting Windows Event Log field names into keys produced by the Winlogbeat host agent in our environment. For example, the field *ProcessCommandLine* is transformed into *winlog.event_data.ProcessCommandLine* by Winlogbeat.

Rules should be transformed according to the specific tools and log sources applicable in your environment. Sigma conversion tools already provided some mappings, but we needed to add many more to accommodate tools and data pipelines in our environment. For example, we wrote our custom

sigma rule that matched against the *ScriptBlockText* field in the Windows Event Log, so the corresponding *winlog.event_data.ScriptBlockText* translation had to be added.

5. Data acquisition

The first order of business is to access the data. The number of possible log data sources and event formats is vast and managing them is a well-known problem. Many standardisation efforts exist, for example Elastic Common Schema (ECS). However, each simply adds another format that needs to be parsed by the streaming engine. While the Sigma toolset does provide utilities for converting rules between popular event formats, our engine still has to access nested keys while validating key existence. Attempting to access a missing map key will cause runtime panic in most programming languages. The user might also be working with static *structs* instead of dynamic dictionaries, which is good for performance and type safety. Another use-case would be matching unstructured *strings* or *byte arrays* with simple *keyword* rules.

This section explores how these use-cases could be handled to be compatible with Sigma rule format.

5.1 Field access

Our general approach is quite simple. We define *interface* methods that users should implement for their own data types. Our code relies only on those methods and trusts the user to properly present proper fields that are defined in a rule or indicate field absence. To achieve this, we define an interface per rule type and embed it in our *Event* definition that will be used internally by the rule engine.

```
// Keyworder implements keywords sigma rule type on arbitrary event
// Should return list of fields that are relevant for rule matching
type Keyworder interface {
    // Keywords implements Keyworder
    Keywords() ([]string, bool)
}

// Selector implements selection sigma rule type
type Selector interface {
    // Select implements Selector
    Select(string) (interface{}, bool)
}

// Event implements sigma rule types by embedding Keyworder and Selector
// Used by rules to extract relevant fields
type Event interface {
    Keyworder
    Selector
}
```

This is a form of polymorphism that allows us to define deterministic input and output while abstracting implementation details. Unlike many other statically typed languages, Go version 1 does not support true compile-time generics, but can evaluate virtual method tables, or *vtables*, in runtime, thus allowing us to pass an *interface* to our internal match functions, as opposed to any concrete type.

Note that the *keyword* rule should only be applicable to textual events. Some structured events could contain multiple fields that should be considered. Thus, we define *Keyworder* as the method that returns a list of *string* data types and a Boolean indicating if a particular rule applies for event type. If *false*, the slice value should default to *nil*. Deriving the interface name from the implemented method name is idiomatic for Go.

A *selection* rule does key lookups and value comparisons for structured events. For simplicity, we assume the message to conform to JSON standard and always be textual. Thus, our input value to the *Selector* method is a *string*. However, the value could be any data type defined in JSON and YAML standards and thus the return value is an empty *interface* that is implemented by every Go type and the user does not need to implement separate methods for textual, numeric or Boolean data types. Lacking true generics, our Go engine needs to assert concrete data types in runtime to maintain type safety.

Consider some use-cases that our streaming engine would need to handle.

5.1.1 Dynamic JSON input

Firstly, consider a truncated Suricata IDS alert for a fairly insignificant, yet verbose, event for web application scanning. These events usually indicate potential reconnaissance.

```
{
  "timestamp": "2020-01-22T12:56:17.403749+0000",
  "event_type": "alert",
  "src_ip": "1.2.3.4",
  "src_port": 53646,
  "dest_ip": "2.3.4.5",
  "dest_port": 1521,
  "proto": "006",
  "alert": {
    "action": "allowed",
    "gid": 1,
    "signature_id": 2010936,
    "rev": 3,
    "signature": "ET SCAN Suspicious inbound to Oracle SQL port 1521",
    "category": "Potentially Bad Traffic",
    "severity": 2
  },
  "payload_printable": "TRUNCATED"
}
```

Extensible Event Format (EVE) (Eve JSON Output, 2020) is a highly dynamic nested JSON document, with more than 1,000 possible keys. Value presence and data types depend on Suricata daemon configuration. For example, a DNS event logger could be configured to log only some resource record types, like A and AAAA, while omitting others, like MX and NS. These records could be presented individually as text records or logged in bulk as lists of dictionaries.

While we opted to generate static structs during exercises to handle these variations, this approach could easily lose fields if any were overlooked in our definition and a lot of work is needed to verify each field type, to handle fields with multiple types and to keep this definition up to date for new Suricata releases. Alternatively, we could simply create a dynamic *map* data type that implements our *EventChecker* interface.

```
// DynamicMap is a reference type for implementing sigma Matcher
type DynamicMap map[string]interface{}

// Keywords implements Keyworder
func (s DynamicMap) Keywords() ([]string, bool) {
    return nil, false
}
```

```
// Select implements Selector
func (s DynamicMap) Select(key string) (interface{}, bool) {
    if val, ok := d[key]; ok {
        return val, true
    }
    return nil, false
}
```

The interface implementor simply needs to return the correct value when asked, or indicate a missing value when the key is absent. Note that the *keyword* rule type can simply be indicated as not applicable to this event by returning a *false* value from the respective *Keywords* method, thus avoiding needless memory allocations if a Suricata event is evaluated against this rule type.

This example lacks an important feature. It does not support nested JSON structures and the analyst would be unable to access any field in *alert* or any protocol substructures. A common notation for accessing these values is by concatenating nested keys with a *dot* symbol. For example, *alert.category* would return the value *Potentially Bad Traffic*. This feature can be implemented as a recursive function.

```
// GetField is a helper for retrieving nested JSON keys with dot notation
func GetField(key string, data map[string]interface{}) (interface{}, bool) {
    if data == nil {
        return nil, false
    }
    bits := strings.SplitN(key, ".", 2)
    if len(bits) == 0 {
        return nil, false
    }
    if val, ok := data[bits[0]]; ok {
        switch res := val.(type) {
        case map[string]interface{}:
            return GetField(bits[1], res)
        default:
            return val, ok
        }
    }
    return nil, false
}
```

Runtime type casting is unavoidable when dealing with dynamic input. Our approach splits the key into two , delimited by the first dot symbol, and attempts to extract a map element using the first value. A *switch* statement then attempts to cast the result into a *map* datatype. If successful, the function will recursively call itself with the second part and extracted container as arguments. Otherwise, the value is returned as a positive match.

We can then implement our *Event* interface methods as wrappers around this function and satisfy the Sigma engine requirements with relative ease.

```
// Keywords implements Keyworder
func (s DynamicMap) Keywords() ([]string, bool) {
    if val, ok := s.Select("alert.signature"); ok {
        if str, ok := val.(string); ok {
            return []string{str}, true
        }
    }
```

```
    return nil, false
}
// Select implements Selector
func (s DynamicMap) Select(key string) (interface{}, bool) {
    return GetField(key, s)
}
```

Note that multiple fields could be returned to satisfy the *keyword* lookup, but each must be properly cast into a *string* data type and appended to the resulting list.

5.1.2 Static input

On the opposite side of the spectrum, the user might be working with simple unstructured text messages.

```
pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=2a08:1172:142:713::201
```

Normalising these messages is outside the scope of the rule engine and many existing tools are already solving this problem. Nevertheless, we can demonstrate how even they can be used in our rule engine by wrapping a built-in *string* into a custom type.

```
type Message string
```

Lacking keys, the *selection* rule can be implemented as stub interface and only the *keyword* conversion truly matters.

```
func (m Message) Keywords() ([]string, bool) {
    return []string{string(m)}, true
}
func (m Message) Select(key string) (interface{}, bool) {
    return nil, false
}
```

This idea can be taken further when considering well-known and standardised event formats. For example, consider the RFC3164 (Lonvick, 2001) formatted syslog message as presented with the following Go *struct*.

```
type Syslog struct {
    Timestamp time.Time `json:"@timestamp"`
    Host      string    `json:"host"`
    Program   string    `json:"program"`
    Pid       int       `json:"pid"`
    Severity  int       `json:"severity"`
    Facility  int       `json:"facility"`
    Sender    net.IP    `json:"ip"`

    Message `json:"message"`
}
```

Note that the unstructured textual *message* field embeds our *Message* type, thus giving access to all associated methods. The following example illustrates a *Select* method that statically implements a key lookup. This approach provides better performance and type safety than the dynamic JSON example but requires more work, especially when handling complex messages. Code generation can be a perfectly acceptable solution when dealing with standardised formats that are unlikely to change, such as BSD syslog events.

```
func (m Syslog) Select(key string) (interface{}, bool) {
    switch key {
    case "timestamp", "@timestamp":
        return m.Timestamp, true
    case "host":
        return m.Host, true
    case "program":
        return m.Program, true
    case "pid":
        return m.Pid, true
    case "severity":
        return m.Severity, true
    case "facility":
        return m.Facility, true
    case "sender":
        if m.Sender == nil {
            return nil, false
        }
        return m.Sender.String(), true
    case "message", "msg":
        return m.Keywords(), true
    default:
        return nil, false
    }
}
func (m Syslog) Keywords() ([]string, bool) {
    return m.Message.Keywords()
}
```

Custom logic can be applied where needed. For example, IP address objects can be converted to *string* representation to be compatible with textual patterns in Sigma rules. As the *net.IP* type is actually a set of methods around *slice* datatype and therefore a pointer, its existence must always be checked or the tool might crash in runtime. Also, remember that a *keyword* identifier is simply shorthand for message *selection* and thus *message* key retrieval can be implemented by wrapping methods. This makes the overall code more explicit and consistent, as changes to how the *Message* object is handled would propagate to other types that embed it.

6. Rule components

Having defined data extraction objects, we can now define one for a Sigma rule as well. A rule can be conceptualised as a function that produces a deterministic verdict for a particular input value. As before, we use polymorphism and can thus define a rule as an *interface* with an *Event* as input and *True* or *False* values as output. Subsequently, we embed this interface into *Branch* to make our interface naming more explicit. *Branch* can also be used as a platform for adding optimisation methods without overcomplicating *Matcher*. A rule of thumb is to keep the overall number of interface methods minimal to avoid excessive work during implementation or refactoring and to maintain a consistent naming scheme between interfaces and the methods that they implement.

```
// Matcher is used for implementing Abstract Syntax Tree for Sigma engine
type Matcher interface {
    // Match implements Matcher
    Match(Event) bool
}
// Branch implements Matcher with additional methods for walking and debugging the tree
type Branch interface {
    Matcher
  // Other interface for walking the tree can be added later
}
// Tree represents the full AST for a sigma rule
type Tree struct {
    Root Branch
}
```

These core structures are enough as a basic rule framework. Subsequent subsections will explore individual nodes that will comprise the tree.

6.1 Identifier leaf nodes

As the reader might already guess, the final leaf nodes, the ones that will be doing actual pattern matching, will consist of *Keyword* and *Selection* objects. The atomic data structure for both is a list of patterns that are joined by logical disjunction. Each pattern is simply an object that holds a token and returns a verdict when presented with value extracted from an event. While we could re-use the *Matcher* interface, it might not be ideal for runtime efficiency and code readability. Instead, we opted to create another interface for matching on concrete types.

```
// StringMatcher is an atomic pattern that could implement glob, literal or regex matchers
type StringMatcher interface {
    // StringMatch implements StringMatcher
    StringMatch(string) bool
}
```

We use *interface* instead of a concrete type because Sigma rules can express multiple string matching techniques, like literal text patterns, prefix and suffix checks, wildcard *glob* patterns and regular expressions. For example, we implemented a regular expression pattern as a custom *struct* that wraps a compiled regular expression .

```
// RegexPattern is for matching messages with regular expressions
type RegexPattern struct {
```

```
    Re *regexp.Regexp
}
// StringMatch implements StringMatcher
func (r RegexPattern) StringMatch(msg string) bool {
    return r.Re.MatchString(msg)
}
```

Alternatively, the *StringMatch* method could wrap around matching methods in *strings* or *bytes* packages in a standard library, or *glob* methods from third party libraries. Our implementation uses all these methods. Consider the following Sigma pattern list as illustration of why this approach is useful:

```
- "/ssh\s+-R/"
- "cat /etc/*"
- "python -m SimpleHTTPServer"
- "python2 -m SimpleHTTPServer"
- "python -m http.server"
- "python3 -m http.server"
- "/python\d? -m (http.server|SimpleHTTPServer)/"
```

The first pattern is a regular expression signified by being enclosed with *slash* symbols. It can be powerful and precise, but is also the slowest by orders of magnitude. An attacker could enter any number of whitespace symbols between the *ssh* command and flag signifying a reverse connection, yet the pattern would still match. Second is a wildcard, or *glob*, that will also match on any arbitrary text proceeding the presented pattern. As yet, rule writers cannot specify whether proceeding symbols are alphanumeric, special, etc., nor can they specify advanced constructs like alterations or list of possible options. However, performance is much better than a regular expression engine and the user can write the pattern intuitively without having to be proficient in the regular expression language. Subsequent patterns are literal, requiring an exact match, so variations need to be expressed manually. For example, python versions 2 and 3 have different syntax for starting a web server and different operating systems default to different versions. Literal matches are also the most sensitive to insignificant variations like extra whitespace symbols, and thus are easiest to bypass. However, they are by far the fastest, as standard library string and byte operations are highly optimised. Handling different cases with multiple patterns could be faster than invoking complex regular expression state machines.

On the engine side, this pattern list can be implemented as a *slice* of *StringMatcher* interfaces. Thus, the message can be evaluated in a loop. Since the patterns are joined by logical disjunction, the first match is enough and subsequent patterns no longer need to be evaluated. Our rule engine optimises this property at parse time by sorting the list according to concrete types and ensures that fast literal patterns are evaluated first and slow regular expressions last. This pattern list can also implement the *StringMatcher* interface, so it could be used interchangeably with atomic patterns.

```
type StringMatchers []StringMatcher

// StringMatch implements StringMatcher
func (s StringMatchers) StringMatch(msg string) bool {
    for _, m := range s {
        if m.StringMatch(msg) {
            return true
        }
    }
    return false
}
```

6.1.1 Keyword

Implementing a *keyword* identifier type is quite simple as it is essentially just a wrapper around the *StringMatcher* interface introduced in the previous section. Naturally, the object can also hold meta data like rule statistics, etc.

```
type Keyword struct {
    S StringMatcher
    Stats
}
```

Compatibility with the *Matcher* interface can be achieved by creating a method that extracts all relevant keywords from the event and checks them against the embedded matcher. As before, the first positive match is sufficient for an early return.

```
// Match implements Matcher
func (k Keyword) Match(msg Event) bool {
    msgs, ok := msg.Keywords()
    if !ok {
        return false
    }
    for _, m := range msgs {
        if k.S.StringMatch(m) {
            return true
        }
    }
    return false
}
```

Note that the rule will skip all matching if *Keyworder* is implemented as a stub.

6.1.2 Selection

While the *selection* type requires more complex logic, we can nevertheless reuse existing code. The atomic matcher is a value or list of values that needs to be evaluated against an event, allowing us to implement it with existing objects. Unlike *keyword*, however, *selection* rules contain multiple objects, each associated with a particular structured event key. *Selection* rules also define numeric and Boolean data types. Since items need to be checked sequentially, we wrap them into an object that also stores the key.

```
type SelectionStringItem struct {
    Key     string
    Pattern StringMatcher
}
```

Note that we could use a *map* type as pattern container, but map is optimised for random access and its elements are not stored as a continuous memory block, thus making it sub-optimal for our implementing a pattern list.

```
type Selection struct {
    N []SelectionNumItem
    S []SelectionStringItem
    Stats
}
```

Selection follows the opposite logic to *keyword*. While individual atomic values per element are still joined with logical disjunction, the elements themselves are joined with logical conjunction. Thus, *selection* is a map where every key must have a positive match while values are pattern lists where only the first match is sufficient. This means traversing a list of elements, first ensuring that the event has a key defined in the identifier and then verifying that the extracted value matches a pattern.

```
// Match implements Matcher
func (s Selection) Match(msg Event) bool {
    for _, v := range s.S {
        val, ok := msg.Select(v.Key)
        if !ok {
            return false
        }
    // Numeric matching omitted
        switch vt := val.(type) {
        case string:
            if !v.Pattern.StringMatch(vt) {
                return false
            }
        default:
            s.incrementMismatchCount()
            return false
        }
    }
    return true
}
```

The example omits numeric matching for brevity, but users should consider all possible numeric types when implementing a matcher. All JSON numbers are by definition *float64,* but decoding structured events into custom types could yield signed or unsigned integers of any length, thus causing a false negative due to type mismatch.

Some *selection* rules are defined as lists of maps where each element follows a *selection* convention, but only one element need match. We simply parsed each element into a distinct *selection* and merged them into a disjunction object, rather than treating it as special variation of the rule format.

6.1.3 Logic operations

Having defined our leaf nodes, the next step is to construct their logical connections. The simplest method for doing that is by constructing a *slice* of branches and evaluating each element sequentially when implementing the *Matcher* interface.

As each logic gate also implements the *Matcher* interface, thus qualifying as *Branch*, it fully supports nested expressions. An element could be a leaf identifier as described in prior sections, or another logic gate with an arbitrary number of recursive child nodes. The logic gate simply functions as a container that modifies the Boolean results from the child elements, but does not know how those results are achieved. While not built as an actual binary search tree, the resulting construct functions very similarly to one.

```
// NodeSimpleAnd is a list of matchers connected with logical conjunction
type NodeSimpleAnd []Branch

// Match implements Matcher
func (n NodeSimpleAnd) Match(e Event) bool {
```

```
    for _, b := range n {
        if !b.Match(e) {
            return false
        }
    }
    return true
}
```

A logical conjunction, or *AND* gate, can only return a positive result if all the child elements have evaluated as *true*. Thus, the first negative result is sufficient for early return and subsequent elements are not evaluated. The opposite logic applies to the logical disjunction, or *OR* gate. The first positive result is sufficient for early return, otherwise all elements would be evaluated for a definitive *false*. As the reader might already assume, a *NOT* gate is simply an object that negates its only child node, whereas that child could be any leaf or branch in a tree.

```
// NodeNot negates a branch
type NodeNot struct {
    B Branch
}

// Match implements Matcher
func (n NodeNot) Match(e Event) bool {
    return !n.B.Match(e)
}
```

Note that these logic gates could easily be implemented as nodes in a binary tree or any other efficient data structure such as a linked list. Our prototype converts two element *slice* types into binary tree nodes with left and right branches which are evaluated with static logic expressions against direct pointers rather than dynamic loops although this design might not yield a significant performance gain in practice as sequential looping over a continuous block of memory could be better for CPU cache efficiency. However, our code does completely needless runtime loops when the logic expression only has one element. In that case, the first and only element is simply extracted and returned as-is. As mentioned before, the parent object that ends up wrapping the result is agnostic to *Branch* type. Likewise, lists with only two elements are converted into a simple binary tree node.

```
func (n NodeSimpleAnd) Reduce() Branch {
    if len(n) == 1 {
        return n[0]
    }
    if len(n) == 2 {
        return &NodeAnd{L: n[0], R: n[1]}
    }
    return n
}
```

Our approach simply uses the list method in other cases. However, these lists could be converted into a right-leaning binary tree with a recursive builder function.

```
func newConjunction(s NodeSimpleAnd) Branch {
    if l := len(s); l == 1 || l == 2 {
        return s.Reduce()
    }
    return &NodeAnd{
        L: s[0],
```

```
        R: newConjunction(s[1:]),
    }
}
```

7. Abstract syntax tree

Our goal was to convert Sigma rule expression with associated pattern containers into an *abstract syntax tree* (AST), which is a hierarchical machine-readable representation of source code. This section outlines the overall process for achieving that. For example, consider the following expression:

```
condition: (selection_1 and selection_2) and not ( filter1 or filter2 )
```

The resulting rule object would have following hierarchy. Note that circles denote leaf identifiers which do the actual pattern matching, and diamonds denote logic functions that simply pass the event while modifying the results from leaf nodes.

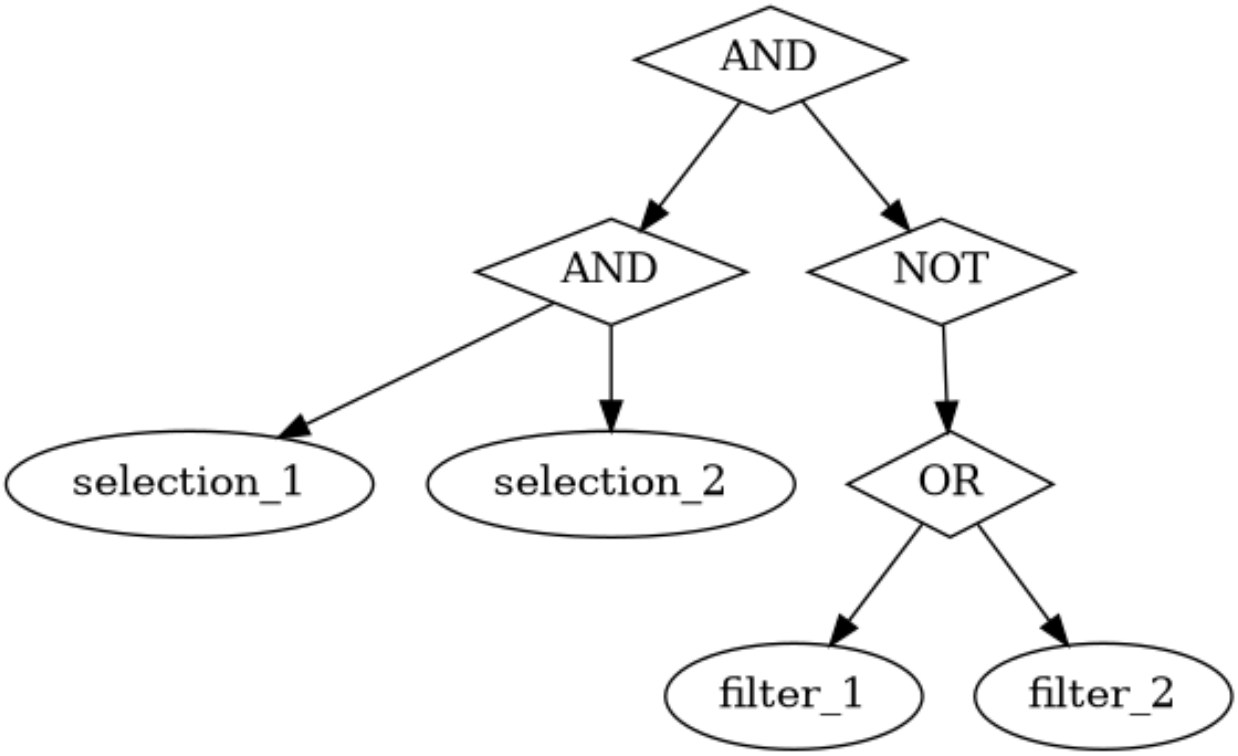

FIGURE 1: AST

To construct this hierarchical object, we need two components: a *lexer* for tokenising the raw input and a *parser* for constructing a rule from the tokens. An in-depth review of those modules will be covered in subsequent sections.

7.1 Lexer

The term *lexer* stands for *lexical analyser*, but in practice no real analysis happens at this stage. Rather, it simply converts a raw human-written expression into a stream of machine-readable tokens that can be used by a parser. A *token* is a variable corresponding to an enumerated value, or *enum*. While Golang lacks a dedicated *enum* type, we are able to achieve the same result by creating a custom *int* type. The example below shows constants with automatically assigned sequential numeric values.

```
// Token is a lexical token extracted from condition field
type Token int

const (
```

```
    TokBegin Token = iota
    TokIdentifier
    TokSepLpar
    TokSepRpar
    TokKeywordAnd
    TokKeywordOr
  // omitted
)
```

Each subsequent value is automatically incremented by one. A token is then attached to a *struct* called *Item*, along with raw value extracted from *condition*. That value is used later for extracting pattern containers from the *detection* map.

```
type Item struct {
    T   Token
    Val string
}

func (i Item) String() string { return i.Val }
```

Generally, the *lexer* can be implemented in three ways. The first is to use a tool to generating one. Many exist that are well known and heavily adopted, such as Lexx, Rager, YACC and Bison. However, they would introduce undesirable dependencies to our otherwise lightweight and self-contained project and are known for having quite complex syntax. The resulting tokens would still need to be integrated into the match engine, so we would conserve effort by using one. The second option would be to rely on regular expressions. While a perfectly acceptable method, the Sigma *condition* format is not complex enough to warrant their use. Token extraction is relatively straightforward compared to rule tree parsing, as we simply need to scan non-whitespace characters and compare them against predefined values. In addition, Lexer does not need to function as a full validator, as the parser is responsible for contextualising the extracted tokens and ensuring their proper sequence.

Instead, we chose the third option, which is implementing a custom lexical scanner. While seemingly the most difficult of the three, the actual process is quite straightforward and often used in the Go community, in no small part thanks to excellent presentation on lexical scanning by Rob Pike, one of the authors of Golang (Pike, 2011) and the large quantity of available resources inspired by that talk. A *lexer* type is a *struct* that contains the input expression, along with offsets for current cursor *position* and last extracted token. The *start* offset will be moved to the value of *position* whenever a new token is extracted. All tokens will be used to instantiate an *Item* which is sent to the *items* channel. The parser will collect those items, validate the sequence and construct an AST for a Sigma rule.

```
type lexer struct {
    input    string    // we'll store the string being parsed
    start    int       // the position we started scanning
    position int       // the current position of our scan
    width    int       // we'll be using runes which can be double byte
    items    chan Item // the channel we'll use to communicate between the lexer and the parser
}

func (l *lexer) scan() {
    for fn := lexCondition; fn != nil; {
        fn = fn(l)
    }
    close(l.items)
}
```

A lexer is a state machine. It will loop over state functions depending on current input until the end of input is signified with a *nil* return type. Syntactically, this means a custom function type that takes a *lexer* as the input and returns another state function as a result. For example, if the scanner encounters the *EOF* symbol, it would return *lexEOF* function that collects everything from the last offset to the end of input and returns an empty value that breaks the function loop.

```
type stateFn func(*lexer) stateFn

func lexEOF(l *lexer) stateFn {
   if l.position > l.start {
      l.emit(checkKeyWord(l.collected()))
   }
   l.emit(TokLitEof)
   return nil
}
```

Our condition scanner would consume characters one by one until a separator or whitespace is encountered. When whitespace is seen, it will call another state function that moves position offset back by one character and emits the collected token after identifying it.

```
func lexAccumulateBeforeWhitespace(l *lexer) stateFn {
   l.backup()
   // emit any text we've accumulated.
   if l.position > l.start {
      l.emit(checkKeyWord(l.input[l.start:l.position])
   }
   return lexWhitespace
}
```

The *emit* method is simply a helper that instantiates a new *Item* and sends the token with collected raw value to the parser. The function will then move our cursor to latest offset.

```
func (l *lexer) emit(k Token) {
   i := Item{T: k, Val: l.input[l.start:l.position]}
   l.items <- i
   l.start = l.position
}
```

After emitting the token, another state function is called that skips all whitespace characters until a non-whitespace is found or input ends. Note that the latest offset will back up by one character once content is found.

```
func lexWhitespace(l *lexer) stateFn {
   for {
      switch r := l.next(); {
      case r == eof:
         return lexEOF
      case !unicode.IsSpace(r):
         l.backup()
         return lexCondition
      default:
         l.ignore()
      }
   }
}
```

Overall, we expect to encounter the following token sequence in a typical Sigma rule:

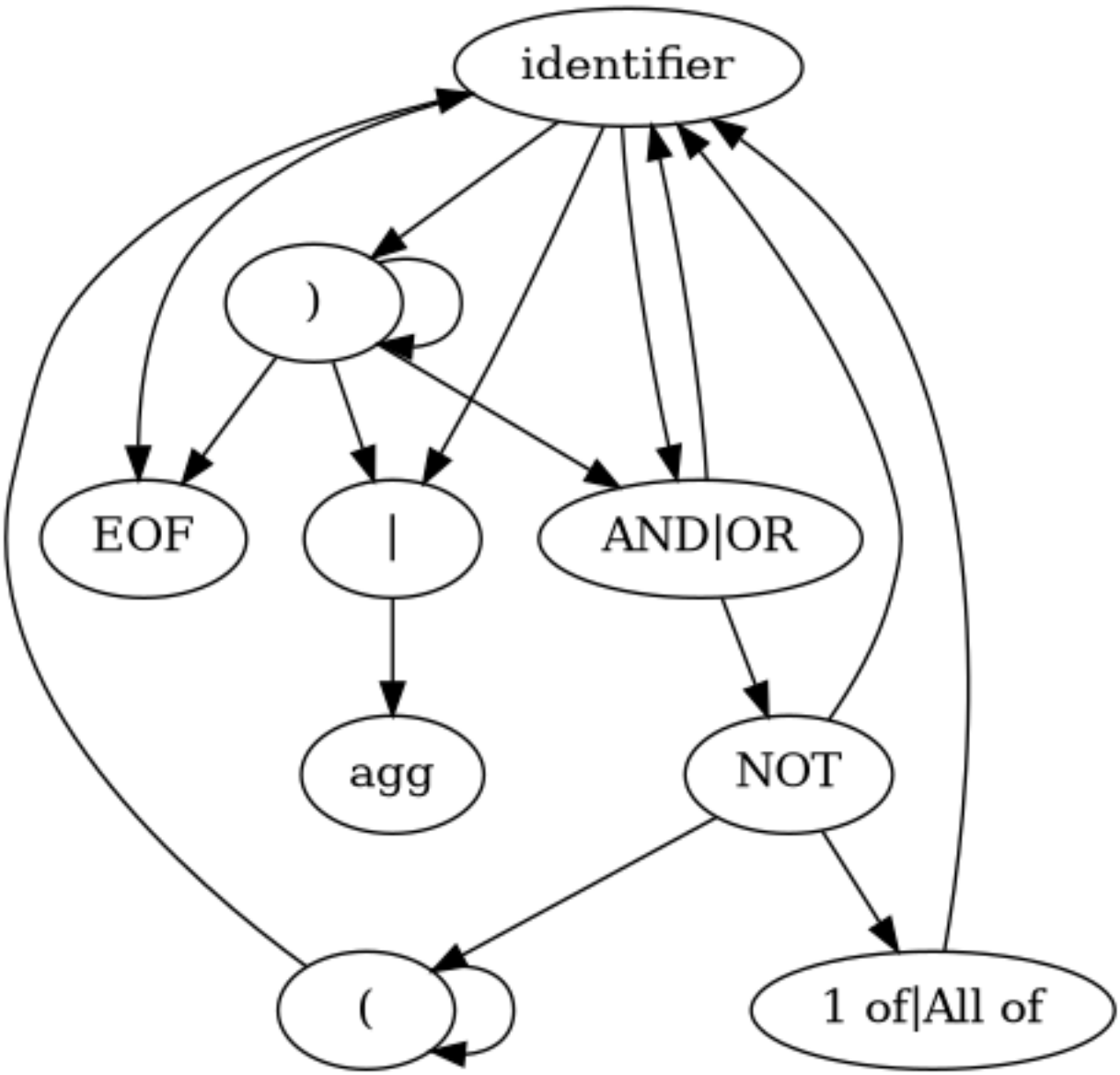

FIGURE 2: TOKEN SEQUENCES

7.2 Parser

The second stage in AST construction involves the actual parsing of *tokens* into a *Tree* object. Like most prior elements, the tree also implements *Matcher* and could be used interchangeably with its *root* branch. However, the user could also use the raw *struct* version with additional methods for statistics, updating, pre-filtering, etc.

```
// Tree represents the full AST for a sigma rule
type Tree struct {
    Root Branch
}
```

The custom *parser* type would wrap around the *lexer*, but also holds collected tokens and stores the last seen item to verify the correct token sequence during collection. It also extracts individual pattern

containers from the *detection* map and parses them into leaf nodes if the *identifier* is encountered in the token stream.

```
type parser struct {
    lex       *lexer
    tokens    []Item
    previous  Item
    sigma     Detection
    result    Branch
}
```

The resulting *Branch* is stored in *result* element if no error occurs during the parse step or in prior token collection. Branch construction is done by a recursive function to handle nested conditions.

```
func (p *parser) parse() error {
    res, err := newBranch(p.sigma, p.tokens, 0)
    if err != nil {
        return err
    }
    p.result = res
    return nil
}
```

Sigma rule expression is very small from a lexical standpoint and only needs to be done once during rule load and would need to do multiple passes over the token stream to extract nested expressions. Thus, the parser operates on a collected *slice* rather than a stream of items from the lexer channel. A *channel* is a language primitive in Golang to enable safe communication between workers, with semantics similar to *pipes* and *sockets* in UNIX-like operating systems, albeit in intra-process synchronisation rather than for inter-process communication. An arbitrary number of producers can send objects into a channel and an arbitrary number of consumers can read those items. An unbuffered channel blocks each send until something consumes the message from the other side, similar to the *yield* keyword in other programming languages.

We could rely on the items channel in the lexer type, but that would make recursion difficult when dealing with nested expressions. The channel follows a *first in first out* (FIFO) logic and thus we are unable to back up or scan ahead when encountering a left paragraph token. Another approach would be simply to iterate over a slice while maintaining offset indices. Nested elements could then be extracted by their position within the array, while moving the offset forward or backward as needed. Alas, this approach makes the code unintuitive to read and prone to human error, as offsets need to be maintained manually by the programmer. Common problems are extracting too much or moving out of the array bounds. The former would be difficult to debug when working with recursive code, and the latter would result in runtime panic or crash as the compiler provides no protection against this bug, thus requiring more testing for the code.

Our method combines these methods to create a third option. To do that, we created a helper which turns the token slice into a generator:

```
func genItems(t []Item) <-chan Item {
    tx := make(chan Item, 0)
    go func(ctx context.Context) {
        defer close(tx)
        for _, item := range t {
            tx <- item
        }
    }(context.TODO())
```

```
    return tx
}
```

This function would create a new unbuffered channel and return after spawning a concurrent *goroutine*. An iterator sends items to the channel one by one and closes the channel once the asynchronous task returns, thus making the channel safe to loop over. The channel will not deadlock if all items are pulled by the consumer, though *context* could be used if timeout is needed.

Our rule builder can then iterate over this channel with no need for offset management. We simply define modifier variables that can be set when special tokens are encountered. For example, encountering a *NOT* keyword would set *negated* Boolean and encountering an identifier would instantiate a new leaf matcher wrapped in *NodeNot* and reset the modifier.

```
func newBranch(d Detection, t []Item, depth int) (Branch, error) {
    rx := genItems(t)
    and := make(NodeSimpleAnd, 0)
    or := make(NodeSimpleOr, 0)
    var negated bool
    var wildcard Token
    for item := range rx {
        switch item.T {
        case TokKeywordNot:
            negated = true
        case TokIdentifier:
    // omitted
```

Note that identifiers are always collected into *NodeSimpleAnd* containers which are in turn wrapped into *NodeSimpleOr* containers though the previously described *Reduce()* method and used to extract a single branch list or convert simple lists into binary tree objects. Likewise, encountering a “X of” keyword will simply set a modifier and extract multiple rule patterns in a subsequent wildcard pattern. Thus, the entire expression can almost be parsed in a single pass. Nested expressions will recursively invoke the same logic, albeit we need to do forward scan until the end of that nested group.

```
case TokSepLpar:
  // recursively create new branch and append to existing list
  // then skip to next token after grouping
  b, err := newBranch(d, extractGroup(rx), depth+1)
  if err != nil {
    return nil, err
  }
  and = append(and, newNodeNotIfNegated(b, negated))
  negated = false
```

A helper function is used to collect all items that belong to the encountered group to construct another *slice* of *Item* types. For that purpose, we pass our receive channel to the function which will pull items while maintaining a balance counter. All child expressions are thus ignored, as they would be handled in recursion.

```
func extractGroup(rx <-chan Item) []Item {
    balance := 1
    group := make([]Item, 0)
    for item := range rx {
        if balance > 0 {
            group = append(group, item)
        }
```

```
        switch item.T {
        case TokSepLpar:
            balance++
        case TokSepRpar:
            balance--
            if balance == 0 {
                return group[:len(group)-1]
            }
        default:
        }
    }
    return group
}
```

The counter is initialised with pre-incremented value as the parent function has already encountered a left paragraph token. Likewise, the function will return all collected items except the last that would be a right paragraph token. The function expects the token sequence to be validated beforehand.

Finally, the tree builder will collect the resulting matcher and return the rule object.

```
    p := &parser{
        lex:       lex(expr),
        sigma:     r.Detection,
    }
    if err := p.run(); err != nil {
        return nil, err
    }
    t := &Tree{Root: p.result}
    return t, nil
```

8. Performance measurements

Golang has a built-in testing and benchmarking functionality. Our library already has several positive and negative test cases to validate matcher results, so reusing them for benchmarking was easy. Go tooling simply executes the same test case with predefined input and the number of test iterations is determined by tooling. The benchmark will simply be executed for a set amount of time and average operation throughput is reported at the end.

```
BenchmarkTreePositive0-12      867567      1363 ns/op
BenchmarkTreePositive1-12      862962      1494 ns/op
BenchmarkTreePositive2-12      795531      1380 ns/op
BenchmarkTreePositive3-12      854679      1393 ns/op
BenchmarkTreePositive4-12      884188      1364 ns/op
BenchmarkTreePositive5-12      809140      1390 ns/op
BenchmarkTreePositive6-12      773706      1410 ns/op
BenchmarkTreeNegative0-12      776173      1385 ns/op
BenchmarkTreeNegative1-12      812887      1481 ns/op
BenchmarkTreeNegative2-12      850477      1401 ns/op
BenchmarkTreeNegative3-12      840723      1390 ns/op
BenchmarkTreeNegative4-12      819126      1417 ns/op
BenchmarkTreeNegative5-12      748514      1416 ns/op
BenchmarkTreeNegative6-12      856683      1382 ns/op
```

Initial results might seem impressive, but several factors can contribute to a more optimistic measurement than can be achieved in practice. Firstly, the test case measures only the *Match* method performance, whereas the raw JSON data is already pre-parsed. Secondly, the test cases are trivial, so map access times are also small. Finally, the built-in benchmark loops over deterministic input and output values. All modern processors do branch prediction and caching to optimise their performance and this benchmarking method is the perfect way to invoke those features. While any good benchmarking utility should clear the cache after each iteration to avoid this leak, we nevertheless decided to design our own measurements to verify the results in realistic use. We designed a small reference utility around our Sigma library with instrumentation to collect various measurements. This binary uses a fan-out worker model with *goroutines* and *channels*. Users can define the number of concurrent workers with a command line argument while a central statistics worker collects measurements from consumers and each individual worker and reports them periodically. A built-in timeout flag was used to stop the consumer after a user-defined amount of time.

Our benchmarks were executed on Intel i7 8850H CPU with 6 cores and 12 logical threads. Windows events in *ECS* format that we collected during Crossed Swords 2020 were fed into our reference binary. Windows events are well known for their verbosity and large number of distinct fields and the Elastic schema only increases this field count while adding nested JSON substructures. Thus, we believe this accurately reflects a worst case use scenario. Tests were conducted at one-minute intervals and each iteration increased the number of concurrent worker routines. Each event was decoded into a *DynamicMap* object. The Golang standard JSON library is well known for being slow when handling dynamic nested structures as it relies on runtime-type reflection, thus we used a third party library compatible with standard library interfaces. The difference was noticeable, though not significant. Firstly, we established a baseline by measuring pure JSON decode throughput with no Sigma pattern matching.

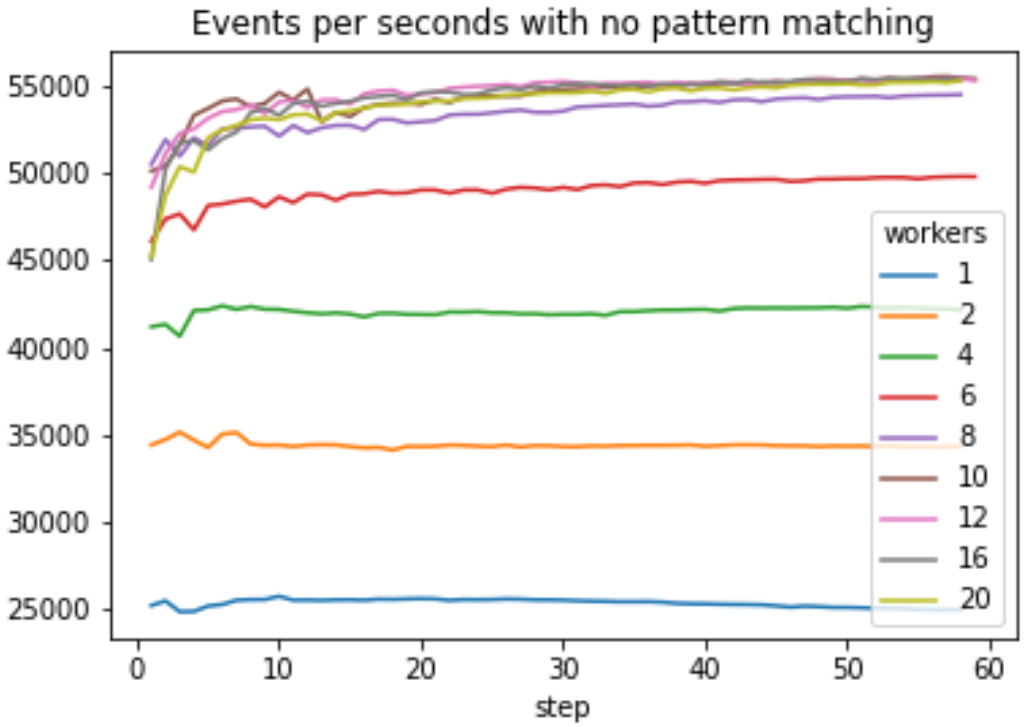

FIGURE 3: DECODE THROUGHPUT

Goroutines are not separate threads. Go binary spawns a configurable number of processes, defaulting to the number of available CPU threads and schedules *goroutines* inside them to achieve balanced system CPU use. We can observe this from Figure 3, as increasing the number of workers does not yield a linear performance gain. We can also see diminishing returns as the number of concurrent workers reaches the number of CPU cores. Effective logical core count is doubled thanks to hyperthreading which allows a single core to execute two tasks at once. However, this concurrency feature comes with a performance penalty, especially in a programming language with a built-in threading model where the user has no control over task scheduling. We can observe diminishing effects as the number of concurrent workers reaches that physical CPU count. We also experimented with worker counts higher than the number of available CPU threads. As expected, this only resulted in more context switches with no observable improvement in message throughput and, at times, even worse performance. The processor was busy switching between tasks whereas the actual work suffered and could have easily achieved the same results with less effort.

Having established our baseline, we repeated the experiment with the Sigma engine enabled. We only used the Windows rules in the public Sigma ruleset. Some rules had multiple detection and logsource values separated by YAML document start delimiters. Those rules were split into separate rules in a pre-processing step. Some rules contained expressions that were not yet implemented in our Sigma engine and some were missing field mappings from standard Windows event log format to ECS. These still produced valid Sigma rule trees, but were clearly unable to match any events. Overall, it resulted in 469 distinct rules that were appended to a *slice*. Each event traversed all rules and results, if present, were fed into a separate statistics worker.

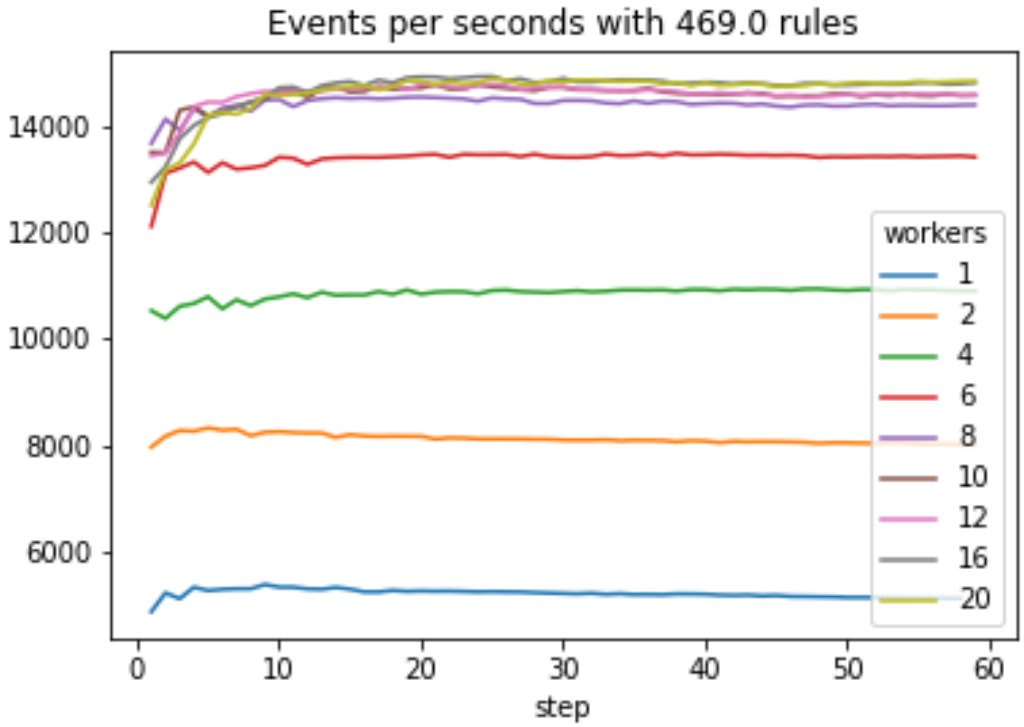

FIGURE 4: MATCH THROUGHPUT

We observed an average of 4.5 time-performance penalties when feeding each event through our engine, with similar per-worker performance gains and diminishing return patterns that we observed without using the Sigma engine. However, this was the result of traversing an entire rule list for each distinct event with no *logsource* pre-filtering. For example, a *sysmon* event would still be evaluated by rules written against security, firewall or powershell events, even when that rule would never match events from those channels. Thus, the effective ruleset size could be reduced significantly in practice; but over time, with new rules being introduced into the set, that benefit could be lost.

To better contextualise these measurements against our initial measurements, two measurements were taken for each event. Firstly, the time spent, in nanoseconds, in decoding the JSON byte array into a *DynamicMap* object, and secondly, the time spent on processing that object with the 469 Sigma rules.

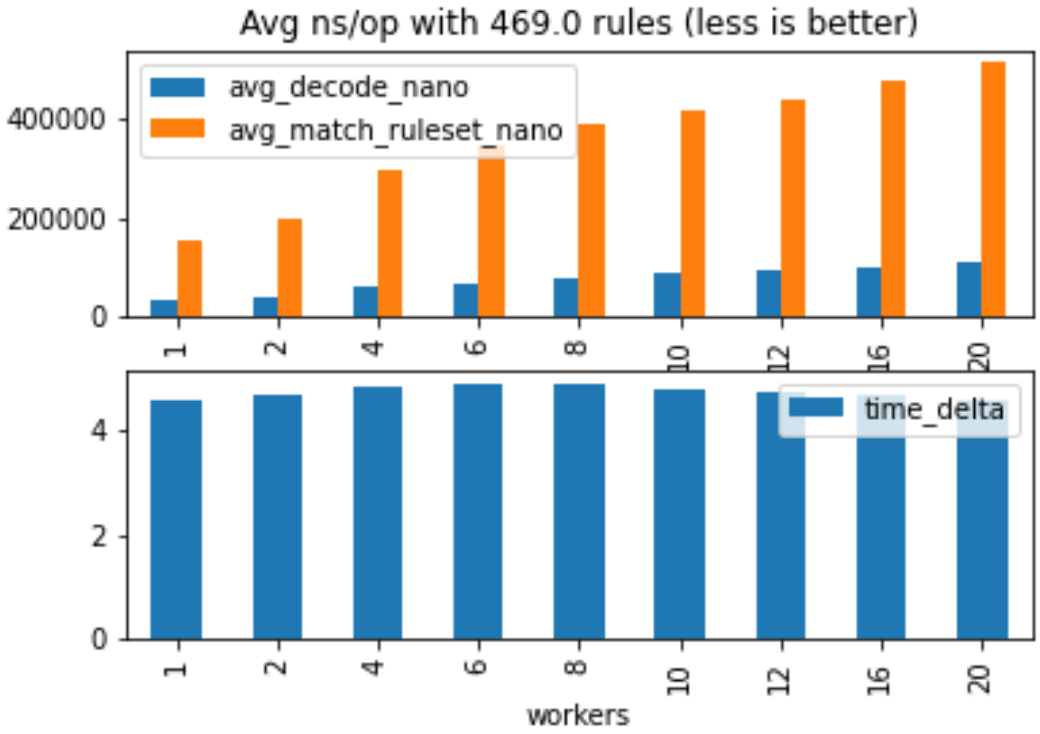

FIGURE 5: WALL CLOCK TIME PER OPERATION

Figure 5 shows the wall clock times as they were taken internally from our application, rather than CPU time that was actually given to our process. The results get progressively worse by increasing the worker count because of competition for CPU clock cycles, so workers are constantly reshuffled between CPU cores, halting execution. Thread pinning could mitigate this efficiency loss if implemented in a lower level language, such as C or Rust. Thus, the efficiency loss is due to the chosen programming language and not to our implementation, as delta values between the two measurements remained consistent across all worker counts. We were observing the natural loss of worker efficiency for Go runtime.

Nevertheless, the results are very promising when measuring individual rule performance. This can be estimated by simply dividing the average time taken for full ruleset evaluation by the total number of distinct rules.

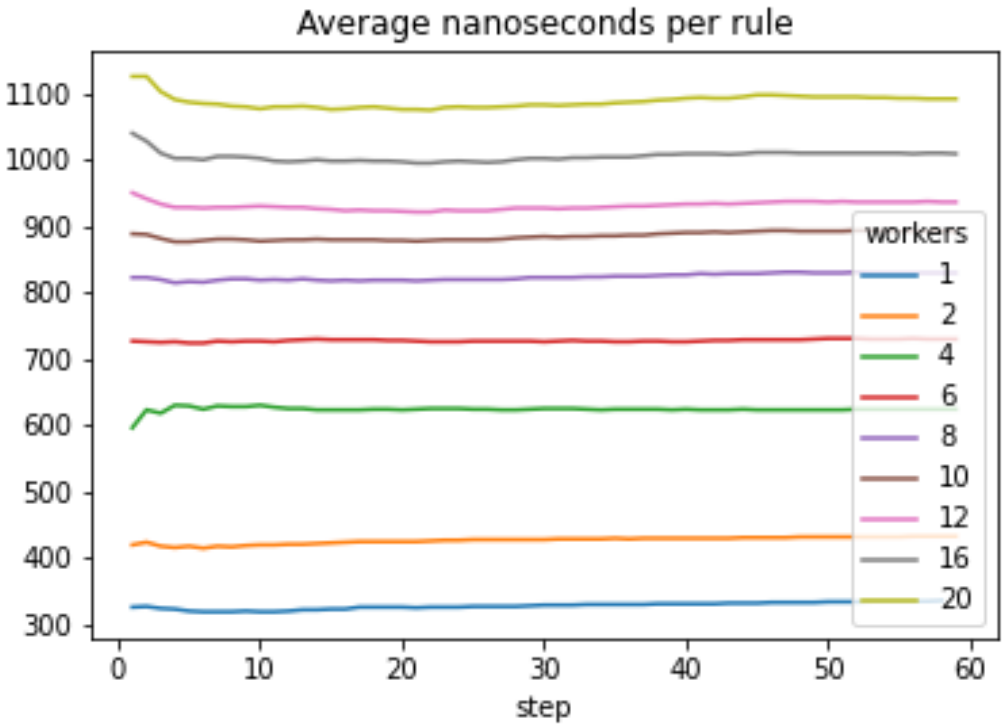

FIGURE 6: WALL CLOCK AVERAGE NANOSECONDS PER RULE

We initially assumed the built-in benchmark would overestimate the results, but this is not true. Average wall clock measurements were roughly 300 to 700 nanoseconds per rule when below maximum CPU core count and still remained below the initial 1,400 nanosecond measurement when going beyond that. However, the results flatten over a large number of measurements and most events never match a rule. Those negative matches could quickly yield a negative result and lower the average.

We did another test to measure each rule execution time to gain a better understanding of overall distribution of these measurements and to see the performance difference between positive and negative results per percentile rank. Collecting measurements for each rule execution consumed a lot of memory and reduced the effective throughput of our application by half, as our one-minute test iteration generated 72 million profiling objects. The benchmark was thus conducted with only one worker, but we doubled the test duration. Other than maintaining the original throughput, this change was the maximum amount of measurements that we could collect in the memory on a system with 32GB of system RAM. While the negative match sample was sufficient, we discovered that a single test iteration only produced 30 to 100 positive matches with the XS 2020 log data, depending on the file. We collected all the positive matches from the 3-day exercise time frame to ensure a sufficient sample size. This produced 15,928 measurements.

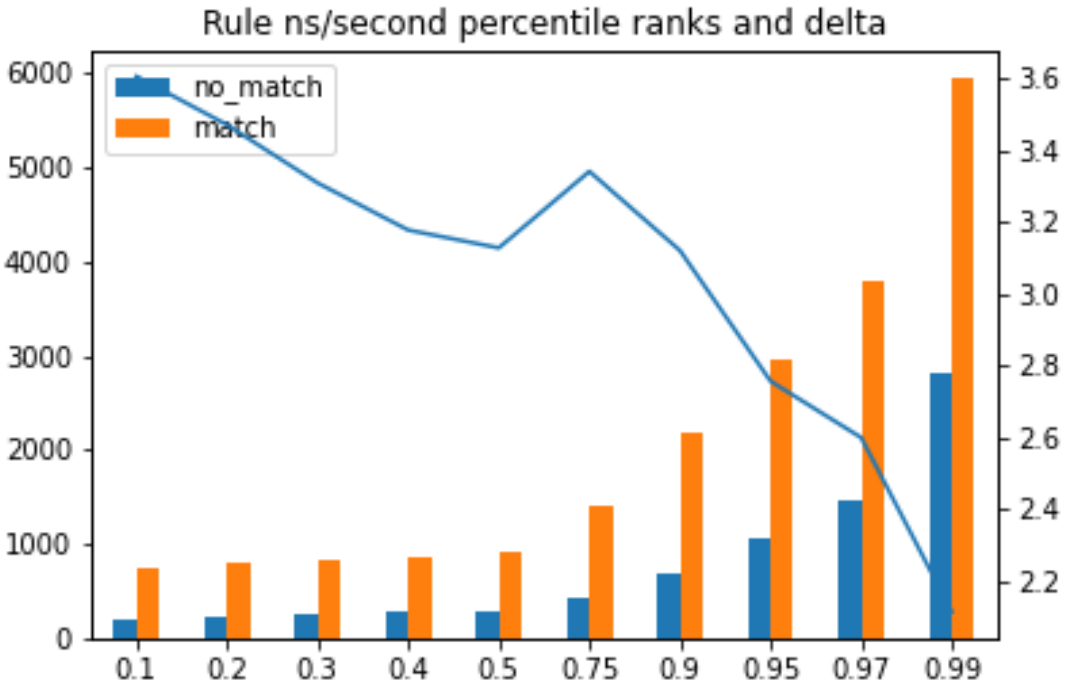

FIGURE 7: MEASURED WALL CLOCK PER RULE PERCENTILES

These results provide insights into realistic rule performance. Nearly 75% of rules yielding positive match were within our initial 1,400 nanosecond benchmark range, 95% of positive matches were within double that estimate and 99% were within 6,000 nanoseconds which is four times our initial benchmark. Few measurements exceeded 18,000 nanoseconds, but Sigma supports regular expressions and more than 40,000 nanosecond execution time is normal for even simple regular expressions that use basic alterations assuming Go standard library, as other implementations could be more efficient. Thus, outliers were still in the acceptable performance range.

By comparison, nearly 97% of negative matches were within our initial 1,400 nanosecond measurement range and 99% within double that estimate. We observed delta values between 2.1 and 3.6 between the percentile ranks for positive and negative matches, with a decreasing trend for subsequent ranks. The vast majority of negative results were achieved with little or no actual pattern matching, because the *EventChecker* interface is designed to indicate field presence. If a field defined in the Sigma rule is missing from the event, then *Matcher* will simply do an early return with a negative result, thus bypassing the expensive pattern operations. Therefore, most negative matches are operating at a highly optimised hash map lookup speed. Only a subset of negative results executes a partial pattern traversal and thus has a smaller performance gap compared to the positive results.

9. Discussion and future work

The following sections discuss the key takeaways from the benchmarks and practical observations made during XS 2020. Some suggestions for future improvement are also proposed.

9.1 Performance

Overall, we found per-rule performance to be adequate, even with no tree optimisation. However, sequentially traversing the whole ruleset had a significant performance effect. Therefore, redesigning the ruleset object to implement proper pre-filtering with Sigma *logsource* field would likely yield the largest performance increase relative to effort needed for coding and testing. Our initial implementation during XS 2020 did that using a Golang *map* object where keys corresponded to our sources, such as Windows event log, Linux syslog, Suricata and Zeek. Values from the Sigma *logsource* field indicated which rulesets store individual rules, but no filtering was done after that. An interesting variation would be ruleset implementation as a binary search tree. Major performance loss was caused by lack of data sharing between individual objects, as each rule executed random access lookups individually, thus causing a large number of redundant operations. Even leaf nodes in a single rule can be affected by this limitation. A binary search tree implementation of the entire ruleset would drastically improve performance, as each event field should ideally be extracted only once. However, this would mean a complete redesign of our Sigma library and incur a fairly complex handling system. Both tasks were out of scope for our reference prototype. Simpler techniques like ensuring proper pre-filtering and rebalancing individual rules should be considered for future developments.

A ruleset-centric approach would also be beneficial for improving the concurrency model. Our current approach is simple – independent workers consume messages from a global thread-safe channel and process rules sequentially. Each worker has a full copy of the entire ruleset and no state is shared between *goroutines*. Results and measurements can be handled locally or produced to another thread-safe channel. In the second case, the results are picked up sequentially by a separate handler routine. Thus, we simply need to avoid passing any data structures via pointers to ensure thread safety. However, the Sigma rule format supports aggregations. For example, the user could define rules that only match if a pattern occurs in N log messages or when multiple events occur in a short space of time. These rules require common memory over sliding window and thus mandate data sharing between workers. Currently, we could only solve this problem by sharing a data structure that has a locking mechanism or by creating a separate set of aggregation workers that are communicated with using channels. While both solutions are perfectly valid, the former could easily cripple performance due to lock contention and the latter would add more worker routines that are difficult to schedule into our existing load-balancing model, thus overcomplicating it and causing potential deadlocks. Instead, the threading model could handle individual messages sequentially while parallelising the rule executions. This would maintain the original message ordering and could prove to be a better platform for creating state sharing across multiple events. Ensuring that this state sharing is designed into our load balancing model also increases code maintainability.

9.2 Post-processing the matched events

Post-processing and visualisation of matched events is not directly part of this work, but still provides a valuable perspective. We employed several methods for improving the detection quality of our Sigma engine during XS 2020. The post-processing of events was performed by a Python script that consumed the stream of events from Apache Kafka, whereas our central logging tool with Sigma functionality was

omitting those high-priority messages. Filtered output was sent directly to our dashboards. Dashboard functionality for red-team feedback was provided by Alerta.

The script parsed JSON formatted events that were already enriched with exercise-specific metadata. Even though the events were structured and mostly processed, we still applied minor normalisation for consistency before transfer to the red team dashboard. However, this step was not performed on original message, but rather on the enriched metadata attached to the event. Essentially, the event already had information on the affected asset, its IP addresses and network zones, Sigma rule matches and event IDs that were mapped to the MITRE ATT&CK enterprise killchain matrix. However, not all fields could always be reliably added by the central engine, so event post-processing functions had to handle these situations, sometimes just dropping the event if there was not enough information for it to be displayed on the dashboard.

One primary step of the post-processing was dealing with the MITRE ATT&CK framework information that was attached to omitted events. We used the information the framework (e.g., phases, techniques and descriptions) contained to standardise everything directed to the dashboard and established an effective baseline to filter periodic low-priority events that were triggered in the target systems before the red team had started its activities. There was another simple filter in place for manually filtering non-interesting events to avoid displaying them on the dashboard. Compiling such filters is normal during real-world use because alerts always need to be contextualised against normal system patterns. The same alert could indicate malicious activity in one environment, yet be triggered by regular benign use patterns in another.

There is definitely room for improvement, but solutions described in this paper were a significant step compared to past exercise iterations. Coming back to the initial discussion on the inefficient use of event correlation, we previously had a substantial number of manually described events which resulted in a disorderly and confusing output for the target audience. We foresee that work conducted in XS 2020 has served as the foundation for developing a proper event correlation ruleset that relies on standardised killchain anchors, rather than having to handle every possible raw atomic message produced by our sensors or target systems. Event correlation is a form of data science whereas understanding atomic security events requires expertise from a particular security domain. Separating those roles enables those domain experts to easily contribute event translations in an easily expressed format that is Sigma, so data scientists can focus on analysing inter-event relationships to detect attack campaigns.

10. Conclusion

Log management and signature-based threat detection have been security operation cornerstones for decades, but using both has historically meant heavyweight SIEM deployment. The Sigma project has emerged in recent years to create a common vendor-agnostic rule language and public community ruleset for security monitoring. However, it is still missing real-time IDS capability, something that has long been the norm for network security. Relying on log management solutions or SIEM systems is not always ideal, for example in cyber exercises or security research.

This paper presents a novel real-time pattern matching engine that functions as an IDS for logs. It was initially implemented as an experimental Golang library for use in the Crossed Swords 2020 yellow team logging engine. Since then it has been rewritten to serve as a reference for anyone interested in integrating real-time detection capability into their logging stream.

As Sigma is a loose rule language that has evolved naturally in real-world use, we firstly analysed existing rule expressions and condition constructs in the public Sigma repository. We then proceeded to define a common interface that users should implement to extract fields defined in the Sigma rules. A rule object was then constructed as polymorphic tree where each node adhered to common Boolean match gateway. We described our method for lexically analysing and parsing the Sigma rule condition into that tree object, along with pre- and post-processing steps we took to handle issues that were not yet managed within the engine. Finally, extensive performance testing was done to study the performance characteristics of our current implementation and to identify needed developments.

Overall, individual rules performed relatively well, even in their current unoptimised form. However, the ruleset as a whole can be developed to eliminate redundant operations. Finally, our chosen programming language proved to be a good choice for prototype implementation. However, high throughput production implementation should consider systems programming languages that provide more control over thread scheduling.

11. References

Apache Kafka. (n.d.). Available: https://kafka.apache.org/.

Elastic Common Schema Overview. (2020). Available: https://www.elastic.co/guide/en/ecs/current/ecs-reference.html.

Elastic stack. (2020). Available: https://www.elastic.co/.

Eve JSON Output. (2020). Available: https://suricata.readthedocs.io/en/latest/output/eve/eve-json-output.html.

Go programming language. (2020). Available: https://golang.org/.

Go Sigma Rule Engine. (2020). Available: https://github.com/markuskont/go-sigma-rule-engine.

Kont, M., Pihelgas, M., Maennel, K., Blumbergs, B., & Lepik, T. (2017). Frankenstack: Toward real-time red team feedback. In *MILCOM 2017 - 2017 ieee military communications conference (milcom)* (pp. 400–405). https://doi.org/10.1109/MILCOM.2017.8170852

Lonvick, C. (2001). *The bsd syslog protocol* (RFC No. 3164). RFC Editor; Internet Requests for Comments; RFC Editor.

MITRE ATT&CK. (2020). Available: https://attack.mitre.org/.

Pike, R. (2011). Lexical scanning in Go. Available: https://talks.golang.org/2011/lex.slide.

Python. (2020). Available: https://www.python.org/.

Roth, F. (2020a). Sigma github repository. Available: https://github.com/Neo23x0/sigma.

Roth, F. (2020b). Sigma specification. Available: https://github.com/Neo23x0/sigma/wiki/Specification.

Satterly, N. (n.d.). alerta. Available: http://alerta.io/.

Snort. (2020). Available: https://www.snort.org/.

Splunk. (2020). Available: https://www.splunk.com/.

Suricata. (2020). Available: https://suricata-ids.org/.

Vaarandi, R., Blumbergs, B., & Çalişkan, E. (2015). Simple event correlator - best practices for creating scalable configurations. In *Cognitive methods in situation awareness and decision support (cogsima)*, *2015 ieee international inter-disciplinary conference on* (pp. 96–100). https://doi.org/10.1109/COGSIMA.2015.7108181

Windows Sysinternals - Sysmon. (2020). Available: https://technet.microsoft.com/en-us/sysinternals/sysmon.

YARA. (2020). Available: https://virustotal.github.io/yara/.