Investigating the Impact of Merging Sink States on Alert-Driven Attack Graphs

The effects of merging sink states with other sink states and the core of the S-PDFA

Jegor Zelenjak[1]

Supervisor(s): Sicco Verwer[1]**, Azqa Nadeem**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Jegor Zelenjak
Final project course: CSE3000 Research Project
Thesis committee: Sicco Verwer, Azqa Nadeem, Asterios Katsifodimos

Abstract

SAGE is an unsupervised sequence learning pipeline that generates alert-driven attack graphs (AGs) without the need for prior expert knowledge about existing vulnerabilities and network topology. Using a suffix-based probabilistic deterministic finite automaton (S-PDFA), it accentuates infrequent high-severity alerts without discarding frequent low-severity alerts. It also captures the context of the alerts with identical signatures and it is an interpretable model. In order to deal with infrequent data, SAGE utilises sink states which are not merged during the S-PDFA learning process. However, this could result in unnecessarily larger AGs. In this study, we have looked at the AGs resulting from merging sink states with other sinks and the core of the S-PDFA after the main merging process. Data from Collegiate Penetration Testing Competitions has been used to compare AGs based on the four metrics: size, complexity, interpretability and completeness. We have shown that the resulting graphs are, on average, slightly smaller, with about the same complexity and the same completeness, but with worse interpretability due to losses of context of attack episodes, which cannot be compensated by the slightly smaller size of the AGs.

Index Terms: SAGE, Attack Graphs, Sink States, Infrequent Data, Context, S-PDFA, FlexFringe

1 Introduction

Security analysts in *security operations centres* (SOCs) have to manually analyse massive volumes of alerts to learn about attackers' behaviour [1,2]. Every day, they receive thousands or even millions of alerts generated by an *Intrusion Detection System* (IDS) [3]. This process is labour-intensive, highly monotonous and stressful, leading to 'threat alert fatigue' [4].

As one of the solutions, *attack graphs* (AGs) are used to visualise the strategies used by attackers when penetrating a network [1]. However, AG generation in general (Topological Vulnerability Analysis or TVA [5]) highly relies on the network topology, expert knowledge and published vulnerability reports, which 'are always one step behind attackers' [6, p. 1]. As a result, even though they are more intuitive, AGs are static and hypothetical: they show what might happen on the network but not what actually happens [1].

To address the aforementioned problems, a tool called SAGE (IntruSion alert-driven Attack Graph Extractor [2, 7, 8]) has been developed. It generates AGs directly from intrusion alerts and does not rely on prior expert knowledge, network topology and system vulnerabilities. The generated AGs aim to show what has actually happened on the network, providing a visual summary of what an IDS observes over time. Under the hood, SAGE uses a *suffix-based probabilistic deterministic finite automaton*, or S-PDFA (learned using FlexFringe [9]) to discover patterns in these intrusion alerts and convert them into interpretable AGs. It is the first tool that uses automaton learning to generate AGs from intrusion data [6].

While SAGE has managed to compress more than 300,000 alerts into less 100 AGs [7], it is difficult to evaluate the quality of SAGE as an unsupervised model, as there is no quantitative metric and no ground truth. Nevertheless, different modelling assumptions can be tested and those that result in the most useful and intuitive AGs for a security analyst can be selected. In this paper, we investigate *what kind of AGs are generated as a result of merging sink states with other sinks and the S-PDFA core*. *Sinks* are states that occur too infrequently to be learned from and are not used in the learning process [2, 6, 9]. We investigate how and why merging sinks affects the size, complexity, interpretability and completeness of the resulting AGs and compare them to the baseline AGs.

The following hypothesis is proposed. The resulting model will produce AGs with, on average, fewer states and lower complexity. On the other hand, unreliability of performing statistical tests on infrequent data traces [9] might negatively affect interpretability, making it more difficult to draw conclusions from the resulting AGs. Finally, completeness might also be negatively affected by merging sinks.

This paper is structured as follows. Section 2 explains AGs, SAGE and S-PDFA. Next, section 3 introduces the problem, presents the hypothesis and the used methodology. Section 4 describes the experimental setup and section 5 analyses the results. Finally, section 6 presents the conclusions and future work, and section 7 discusses the reproducibility of the methods and the ethical aspects of the research.

2 Related Work

Nadeem et al. have introduced the concept of *alert-driven* attack graphs and have developed SAGE, which generates AGs directly from intrusion alerts [2, 7, 8]. Importantly, it does not discard infrequent high-severity alerts, which is the problem of frequency analysis and most machine learning based attacker strategy identification approaches, while summarising frequently-occurring low-severity alerts that lead to high-severity ones. Furthermore, SAGE captures different contexts for alerts with the same signature, e.g. network scans at the start and in the middle of an attack are likely to show different attacker behaviour due to gained experience. Finally, both the SAGE model and the AGs are interpretable: the graphs are concise, give relevant insights into strategic differences and allow for fingerprinting paths (taken by only one attacker) and ranking attackers based on the uniqueness and severity of their actions (see Figure 1 for an example of an AG).

The SAGE pipeline can be summarised into the following three steps [1, 2, 7]:

1. **Intrusion alerts are organised into sequences that represent an attacker strategy**. Noise in the dataset is cleaned and alerts are aggregated into attack episodes, starting from a continuous increase and ending with a continuous decrease in alert frequency, reaching a global minimum. Alerts in an episode are likely to belong to the same attacker action. Time-sorted episode sequences for each attacker-victim pair are partitioned into episode subsequences, starting with a low- and ending with a high-severity episode (a likely attack attempt).

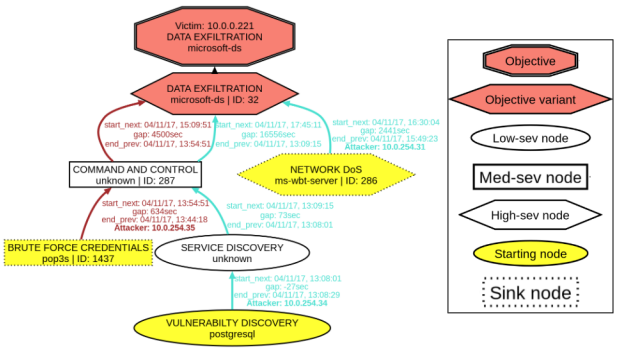


Figure 1: Attack graph for `Data Exfiltration|microsoft-ds` objective on the victim 10.0.0.221 (CPTC-2017) [8]. Edge colours show different team affiliation.

2. **Based on the generated episode subsequences, an S-PDFA is learned using FlexFringe** [9]. Input traces are reversed, i.e. high-severity episodes that are chronologically in the future are closer to the root of the S-PDFA. To summarise the attack paths based on behavioural similarity, FlexFringe uses the evidence-driven red-blue state-merging framework: blue states (merge candidates) are merged with red states (identified parts of the S-PDFA, the core) if they are similar enough (identical futures and similar pasts), starting from the root. The Alergia evaluation function [10] checks if a merge is consistent. Infrequent blue sinks are ignored when testing merge candidates, but they can still be merged during *determinization*, i.e. recursively merging the children of the merged states that have a transition with the same symbol. States of the S-PDFA can be seen as milestones achieved by the attackers. Every episode subsequence is replayed though the learned model, getting assigned a state identifier, after which it becomes a state subsequence (here medium- and high-severity sinks are used).
3. **The generated state sequences are converted into AGs on a per-victim, per-objective basis**. All attackers who have achieved an objective (attack stage, or `mcat`, and the most targeted service, or `mserv`) are shown in one graph, while each successful attempt to achieve an objective is a separate path in the graph. Different paths leading to the same objective have different objective variants (salmon hexagon states, see Figure 1).

3 Methodology

This section describes the methodology used in this work. Subsection 3.1 defines the problem statement, and subsection 3.2 presents the hypothesis of the experiment results. Next, subsection 3.3 introduces the chosen metrics for comparison of AGs generated by the original and modified (i.e. with merging sinks) SAGE and explains the reasons behind them. Finally, subsection 3.4 explains the experimental workflow of the research.

3.1 Problem

In the S-PDFA learning process, two states are merged if the algorithm cannot find enough evidence to conclude that they are different [2]. Infrequent states, called sinks (occur less than `sinkcount` parameter, currently set to 5), are ignored in the merging process, as performing statistical tests on them is not reliable [9]. The Alergia evaluation function used by SAGE will not find enough evidence for an inconsistency in merging these infrequent states. The resulting merges could be arbitrary and could degrade the model quality and the insights that security analysts might get from the graphs.

On the other hand, it has been claimed that including medium- and high-severity sinks in the post-processing could lead to AGs having different objective variants for similar attack paths, which in turn leads to unnecessarily larger graphs [2]. Furthermore, there may exist some conditions for merging sinks which will result in a better model quality. It is thus still an open problem what to do with these infrequent states. This research analyses what exactly happens when sinks are merged both with other sinks as well as the main core of the S-PDFA after the main merging process, and what the consequences are for the resulting AGs.

3.2 Hypothesis

The analysis of the final S-PDFA before merging sinks shows that blue and white sink states constitute the largest percentage from all states (see Table 1). After a merge, the count of sinks might become high enough for them to be become non-sinks, resulting in a larger core of the S-PDFA. On the other hand, only the medium- and high-severity sinks are included in the AGs. Allowing to merge sinks will result in no sinks for SAGE to be salvaged in the post-processing, thus the state count could decrease because of this (see Table 2).

Considering the above, the following hypothesis is proposed: *merging sinks with other sinks and the SPDFA core will increase the core of the final S-PDFA, while will result in smaller, less complex, but less interpretable and less complete AGs*. The loss of medium- and high-severity sinks in the post-processing and the inability of Alergia to prevent merging these states will result in a lot of merges and, on average, fewer nodes in the AGs. Since the complexity of an AG is directly proportional to the number of nodes, it will also decrease, on average. Finally, due to arbitrary merges, interpretability and completeness are expected to become worse, meaning that security analysts will struggle more to draw conclusions from the resulting AGs, and some important data could be lost.

Table 1: Statistics on the resulting S-PDFA, original SAGE (obtained from FlexFringe output files and the script *stats-ff.sh*).

	CPTC-2017	CPTC-2018
Red states (core)	67 (3%)	56 (3%)
Blue states (sinks)	459 (21%)	241 (13%)
White states (sinks)	1676 (76%)	1533 (84%)
Total sink states (blue + white)	2135 (97%)	1774 (97%)
Total states (red + blue + white)	2202	1830

3.3 Chosen Metrics

This subsection defines the four metrics used for comparing AGs: size, complexity, interpretability and completeness.

Table 2: Statistics on the AGs, original SAGE (computed with the script *stats-nodes-ags.sh*). Root nodes are included in unique nodes.

	CPTC-2017	CPTC-2018
Total number of nodes	1853	1288
Number of unique nodes	329	247
Total number of sink nodes	356 (19%)	250 (19%)
Number of unique sink nodes	136 (41%)	104 (42%)

Size

The chosen metric for the size of a graph is the number of nodes in it, due to its intuitiveness. For a security analyst, it makes a difference whether a graph has 10 or 30 nodes. Furthermore, it has been used in the original SAGE paper [2].

The reason that the number of edges is not included is that it does not affect the size of a graph. Rather, it makes a graph look denser and could make a graph more complex. Hence, it is more useful to include it in the complexity metric.

Complexity

To measure complexity of an AG, the approach proposed by De Alvarenga et al. [11] for process mining has been chosen. The authors defined the boundaries for the number of nodes in a graph, below which a graph is considered not complex and above which the graph is considered complex (v_{min} and v_{max}, respectively). Next, they defined simplicity as $Simplicity(AG) = \frac{|V|}{|E|}$, with $|V|$ is the number of nodes and $|E|$ is the number of edges in a graph. Simplicity quantifies how simple the graph is. If $|V|$ is within the boundaries, its simplicity s is compared to the threshold t_s to limit the number of edges (to allow graphs with more nodes to have more edges). Simplicity has also been used to measure the complexity of AGs in the original SAGE paper [2].

The classification of an AG into complex or non-complex can thus be summarised with the following function:

$$IsComplex(AG) = \begin{cases} No, & \text{if } |V| < v_{min} \\ Yes, & \text{if } |V| > v_{max} \\ No, & \text{if } s \geq t_s \\ Yes, & \text{if } s < t_s \end{cases}$$

where $t_s = A + B \cdot |V|$ is the threshold to be estimated. The values $v_{min} = 15$ and $v_{max} = 30$ are used as in [11].

The threshold t_s is estimated using a linear regression line [11]. Suppose the original and the modified SAGE are compared on one dataset. The AGs generated by both versions are used to avoid bias. From these AGs, those that have fewer than $v_{min} = 15$ or more than $v_{max} = 30$ nodes are picked, and their simplicity s is computed. The regression line $y = A + Bx$ is computed in the following way [12]:

$$A = \frac{(\sum y)(\sum x^2) - (\sum x)(\sum xy)}{n(\sum x^2) - (\sum x)^2}$$

$$B = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2}$$

where $x = |V|$, $y = s$ and n is the number of AGs with $|V| < v_{min}$ or $|V| > v_{max}$. For each AG, $t_s = A + B \cdot |V|$.

Interpretability

'An interpretation is the mapping of an abstract concept (e.g. a predicted class) into a domain that the human can make sense of.' [13, p. 2]. In case of SAGE and AGs, it is interesting to see which conclusions a security analyst can draw by looking at the generated AGs, whether it is easy to draw them and whether the AGs are intuitive enough to understand.

Since interpretability is by definition the mapping of something abstract to something that a human can understand, a qualitative analysis of AGs would be more appropriate. Using a numerical metric will produce another number that has to be interpreted, destroying the entire idea of interpretability. It can also happen that a metric suggests that graphs are interpretable, while in reality they are difficult to analyse. Furthermore, security analysts mostly analyse AGs qualitatively, not quantitatively. It is therefore more important to look at the graphs from their perspective, rather than rely on a number.

Completeness

Completeness shows how much information is present in a certain dataset [14]. For AGs, the present attack paths and (high-) severity nodes could be investigated. For an absolute comparison, the episodes could be taken as ground truth, since SAGE creates them from the input alerts (whose correctness does not depend on SAGE). However, SAGE discards the episode subsequences of length smaller than three. Furthermore, it excludes from the AGs the state subsequences that have not reached any objective. Hence, the AGs generated by the baseline SAGE already miss some paths and (high-severity) episodes, which is however not part of the modelling assumptions of this research. For this reason, this study focuses on a relative comparison of the completeness of the modified SAGE with respect to the baseline SAGE.

3.4 Method

Figure 2 presents the experimental workflow of the research. First, a literature study has been conducted to get familiar with the background information and metrics for comparing AGs. The chosen metrics and other useful utilities have been implemented in the form of Bash scripts and can be found in the GitHub repository[1]. In the main *README.md*, a description is given for each script, its usage as well as an example use case. Moreover, each script is extensively documented and nuances are mentioned. Experiments have been executed on two datasets on the baseline and modified versions of SAGE (see section 4). Next, AGs have been analysed both quantitatively and qualitatively. Finally, S-PDFA merges have been analysed to investigate the reasons for the results of merging sinks with other sinks and the S-PDFA core.

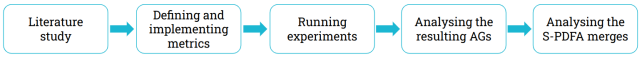


Figure 2: Experimental workflow.

[1] https://github.com/jzelenjak/research-project

4 Experimental Setup

As in the original SAGE paper [2], the Collegiate Penetration Testing Competition dataset from 2017 and 2018 has been used (CPTC-2017 and CPTC-2018, respectively [15], summarised in Table 3). This is to ensure that the results obtained from running SAGE are in line with the results obtained in the original paper. On the other hand, running experiments on another dataset might result in differences that are related to the dataset and not the modelling assumptions.

Table 3: Experimental dataset summary (before filtering) [2].

Dataset/Properties	CPTC-2017	CPTC-2018
# alerts	**43,611**	**330,270**
# teams	**9**	**6**
Duration (hrs)	**11**	**9**
Attacker hosts known?	**No**	**Yes**
Victim hosts known?	**No**	**Yes**

SAGE[2] has been run with the default parameters and the default *spdfa–config.ini* file. The version including the fixes from five pull requests submitted during the research has been used. For the modified version of SAGE, the following two parameters have been added in the config file: `mergesinks=1`, `mergesinkscore=1`. The first parameter allows merging sinks with other sinks after the main merging process, while the second parameter allows merging sinks with the red core after the main merging process, as explained in the *main.cpp* file in the FlexFringe repository.[3]. The experiments have been executed on the processor *12th Gen Intel(R) Core(TM) i7-12800HX* with 16 CPU cores, on Arch Linux.

5 Results and Discussion

This section presents the results of the performed experiments. Subsection 5.1 starts with the general statistics that are not related to the chosen metrics. Then, the results for each of the four metrics are presented in subsections 5.2, 5.3, 5.4 and 5.5, respectively. Next, the insights from the analysis of S-PDFA merges are discussed in subsection 5.6. Finally, subsection 5.7 briefly discusses the bugs discovered during the research process.

5.1 General Statistics

Table 4 shows the general statistics computed after running SAGE before and after merging sinks. Both versions resulted in the same number of AGs for both datasets. In addition, all the generated AGs are the same when comes to their root nodes (i.e. the generated AG filenames were the same, excluding the experiment name part). Furthermore, more than a half of the graphs are the same for both datasets when it comes to the nodes and edges present (excluding the node ID as it is likely to be different for both versions). This means that merging sinks does not miss any victim-objective pair that were present before merging sinks, and the majority of the graphs are not affected (at least, on the used datasets).

[2]https://github.com/tudelft-cda-lab/SAGE

[3]https://github.com/tudelft-cda-lab/FlexFringe

Table 4: General statistics before and after merging sinks (generated using scripts *comp-ag-dirs.sh*, *diff-ags.sh* and *stats-nodes-ags.sh*).

	CPTC-2017		CPTC-2018	
	Original	**Modified**	**Original**	**Modified**
Num. AGs	105	105	75	75
Different AGs	47		21	
Same AGs	58		54	
Total num. nodes	1853	1695	1288	1238
Num. unique nodes	329	223	247	203
Total num. sinks	356 (19.2%)	0 (0%)	250 (19.4%)	0 (0%)
Num. unique sinks	136 (41.3%)	0 (0%)	104 (42.1%)	0 (0%)
Num AGs with sinks	82	0	56	0
Num. start nodes	301	300	203	201
Num. obj. variants	150	128	99	82

Table 4 also shows that merging sinks indeed results in less nodes in total and less unique nodes, even though this decrease is not that substantial. The number of starting nodes is almost the same, and the number of objective variants has slightly decreased. The latter is likely due to the fact that infrequent high-severity sinks that were objective variants have been merged with non-sink objective variants.

The computed statistics on the final S-PDFA (see Table 5) show that the core of the resulting S-PDFA has indeed increased, which confirms the hypothesis. More interestingly, the states added to the red core are all sinks. We have discovered that when sinks are extended (i.e. coloured red and added to the core instead of being merged), they are still considered as sinks, even if their count becomes at least `sinkcount`. This is also the difference between setting `mergesinks=1` and `mergesinkscore=0`. In the latter case, other blue sinks (merge candidates) are only allowed to be merged with these red sinks but not with the red non-sinks that were part of the core when the main merging process has ended, which is however allowed with `mergesinkscore=1`.

Table 5: Statistics on the resulting S-PDFA, original and modified SAGE (from FlexFringe output files and the script *stats-ff.sh*).

	CPTC-2017		CPTC-2018	
	Original	**Modified**	**Original**	**Modified**
Red states (core)	67 (3%)	148 (100%)	56 (3%)	161 (100%)
Blue states (sinks)	459 (21%)	0 (0%)	241 (13%)	0 (0%)
White states (sinks)	1676 (76%)	0 (0%)	1533 (84%)	0 (0%)
Sink states	2135 (97%)	81 (55%)	1774 (97%)	105 (65%)
Total states	2202	148	1830	161

Finally, we have calculated the side-by-side statistics of the resulting AGs, as well as the average node and edge counts of all the AGs (using the script *stats-ags-comp.sh*). For each corresponding pair of AGs, the number of nodes and edges, simplicity, whether the AG is complex and the number of found objective variants are shown, as well as the difference in the node and edge count, simplicity, complexity and the number of found objective variants between the graphs. Full resulting statistics, sorted by the difference in the node count, can be found in the GitHub repository. The following two subsections present the most interesting statistics for the corresponding metrics.

5.2 Results for Size

Table 6 shows that the AGs generated after merging sinks have on average fewer nodes, which confirms the proposed hypothesis. The maximum node count has also decreased. Moreover, there are no increases in the node count. This is because AGs are created from state subsequences, which cannot increase but can decrease if multiple traces have common states. Given that episode traces are now replayed only using the core of the S-PDFA and not the sinks (whose count is very large), it is a reasonable result. Furthermore, only four AGs in CPTC-2017 have a decrease in 10 or more nodes, while in CPTC-2018 only three AGs have a decrease in 5 nodes or more. Finally, for both datasets, more than a half of the AGs have the same node count before and after merging sinks. Therefore, merging sinks does not affect the node count a lot.

Table 6: AG comparison on size (from *stats-ags-comp.sh*).

	CPTC-2017		CPTC-2018	
	Original	**Modified**	**Original**	**Modified**
Average num. nodes	17.65	16.14	17.17	16.51
Max. num. nodes	48	39	34	30
Min. num. nodes	3	3	3	3
AGs with increase	0		0	
AGs with decrease	47		21	
AGs with same count	58		54	
Highest decrease	-15		-7	
Highest increase	N/A		N/A	

5.3 Results for Complexity

The complexity of the AGs has not been affected much (see Table 7). For the vast majority of the AGs, the complexity has not changed. This is because the highest decrease in the number of edges in the corresponding AGs is 3 and 2 for CPTC-2017 and CPTC-2018, respectively, while the decreases in the node count have not been substantial (see subsection 5.2). As a result, the classification function for complexity gives similar results, which could differ when the decrease in the node count crosses v_{min} or v_{max} boundaries (making the graph not complex) or if the decrease in simplicity s crosses the threshold t_s (e.g., when the edge count stays the same and the node count decreases, the simplicity will also decrease, potentially making the graph complex).

Table 7: AG comparison on complexity (from *stats-ags-comp.sh*).

	CPTC-2017		CPTC-2018	
	Original	**Modified**	**Original**	**Modified**
Complex	38	40	30	29
Non-complex	67	65	45	46
Complex ⇒ Not complex	4		2	
Not complex ⇒ Complex	6		1	
Same complexity	95		72	

5.4 Results for Interpretability

A substantial part of the AGs are the same in terms of the nodes and edges present. Hence, the interpretability of these AGs is more or less the same, unless *graphviz* decides to render these graphs differently, which is however not part of the modelling assumptions. From the perspective of node count, AGs are more compact, which should make them easier to analyse from the size perspective.

However, this comes at a cost. As has been mentioned above, SAGE tries to differentiate between the same states in different parts of an attack path (or in different attacks). This is called the context of a node - a different state ID is assigned when the episodes appear in sequences with different futures and pasts, which highlights a different context [2]. A security analyst can clearly see that these episodes correspond to different attacker behaviour.

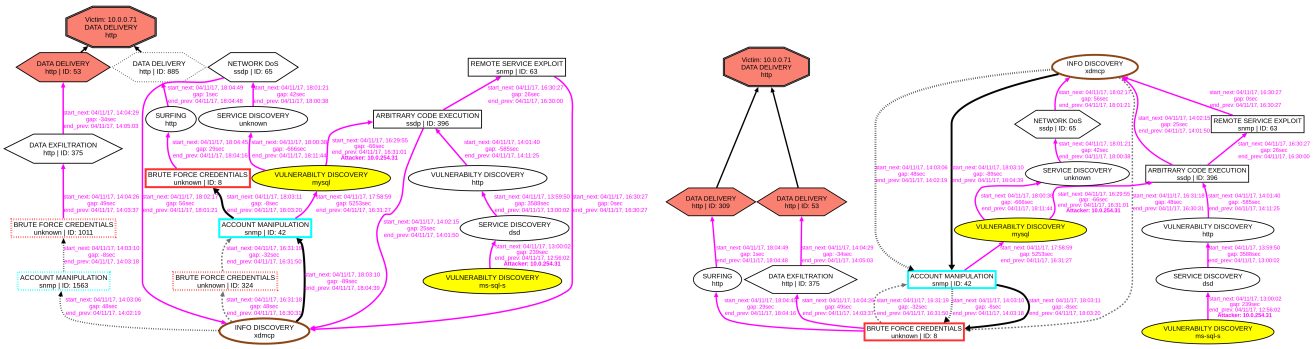
Figure 3 shows a merge which resulted in less states but in worsened interpretability. In the left AG (before merging sinks), there are two occurrences of a sequence `Info Discovery` ⇒ `Account Manipulation` ⇒ `Brute Force Credentials`, in different attacks (Info Discovery should actually be split into three separate nodes, but since SAGE removes ID from low-severity (sink) nodes, they become the same node in the AG, see Figure 4). In the right AG (after merging sinks), there is only one occurrence of this sequence and the separation is not present any more. While the paths are still correct and there is no loss in data, i.e. by following the timestamps a security analyst can still detect two paths, the contextual (behavioural) difference is lost. For instance, the sequence with solid black edges is preceded by `Network DoS|ssdp|ID:65)` and succeeded by `Surfing|http`, while the sequence with dashed black edges is preceded by `Arbitrary Code Execution|ssdp|ID:396` and followed by `Data Exfiltration|http|ID:375`. An analyst might think that after `Network DoS|ssdp` an attacker can reach `Data Exfiltration|http`, whereas it is not the case, as it is a completely different attack. Similarly, `Brute Force Credentials|unknown|ID:324` has been merged with `Brute Force Credentials|unknown|ID:8`, even though their pasts and futures are different.

5.5 Results for Completeness

The total number of incoming edges to all objective variants of an AG is the same for both versions of SAGE. This means that SAGE does not lose any attack paths when merging sinks that were present before. Moreover, while FlexFringe merges do result in less states, there is no loss in data. The difference in edges is always equal to the difference in the number of objective variants, meaning that only the edges from an objective variant to the root node are missing and not the edges from the actual attack paths. This is reasonable, as AGs are created by replaying episode sequences passed to FlexFringe, which are the same for both versions of SAGE. Thus, the completeness of SAGE is not affected by merging sinks.

5.6 Analysis of S-PDFA Merges

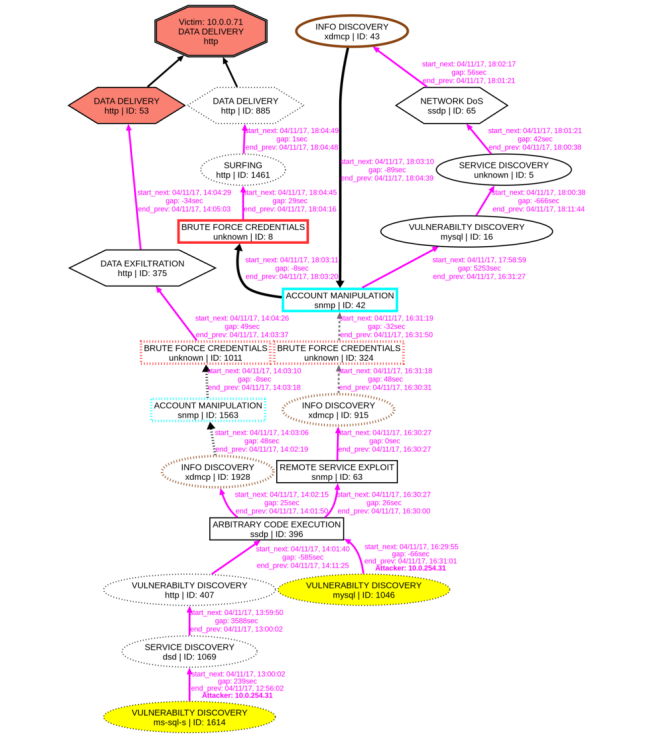
To further investigate the issue with the context, we have looked into the merges of S-PDFA (mostly using CPTC-2017).



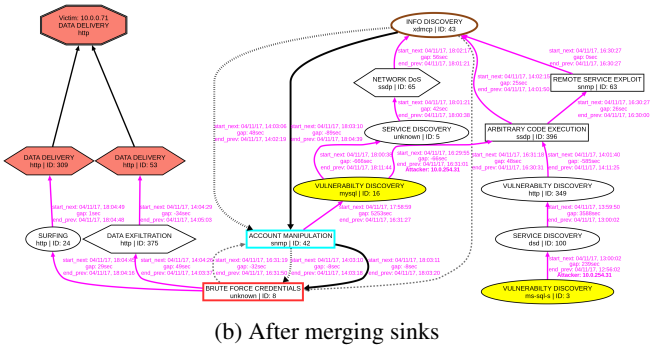
(a) Before merging sinks

(b) After merging sinks

Figure 3: AG `Data Delivery|http` on 10.0.0.71: an example of a loss of context (no IDs for oval nodes and excluding low-sev. sinks).



(a) Before merging sinks



(b) After merging sinks

Figure 4: AG `Data Delivery|http` on 10.0.0.71: an example of a loss of context (with IDs for oval nodes and all sinks).

Problem

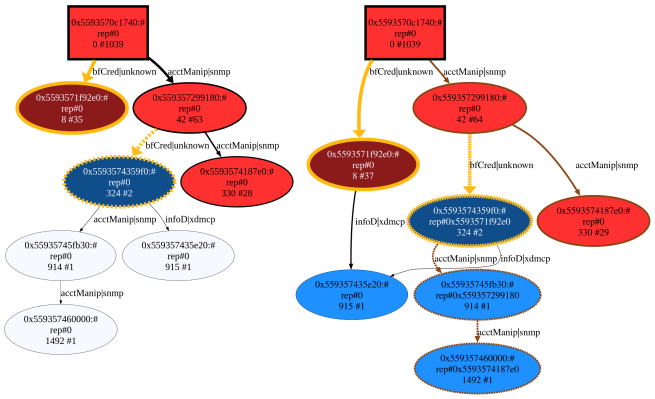
We have executed FlexFringe on the traces generated by SAGE, with `debug=1`, which generates files with the S-PDFA before each refinement (merge or extension). Since the automaton is implemented using Union-Find, the number of nodes always stays the same, but node representatives change. However, the resulting graphs and the number of files are too large to be analysed (see Table 8).

Solution

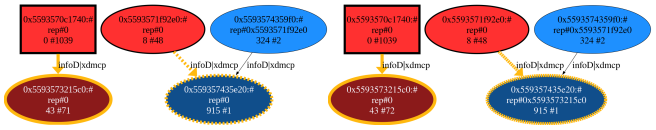
To address this issue, we have created a script *get-merges.sh*, which performs a 'smart' `diff` between each consecutive pair of the S-PDFA files. Extensions (adding a blue (sink) node to the red core) are skipped, since they are not interesting and large. For each merge, the graphs 'before' and 'after' are created and only the affected nodes are selected (i.e. have changed their representatives, count or degree). For each such node, all incoming edges with their endpoints are also included for some context. Other visual enhancements are also added to facilitate the analysis (see Figure 5). Finally, all merges are combined into one PDF file and a log file with all the information is saved (see GitHub repository).

Analysis and Results

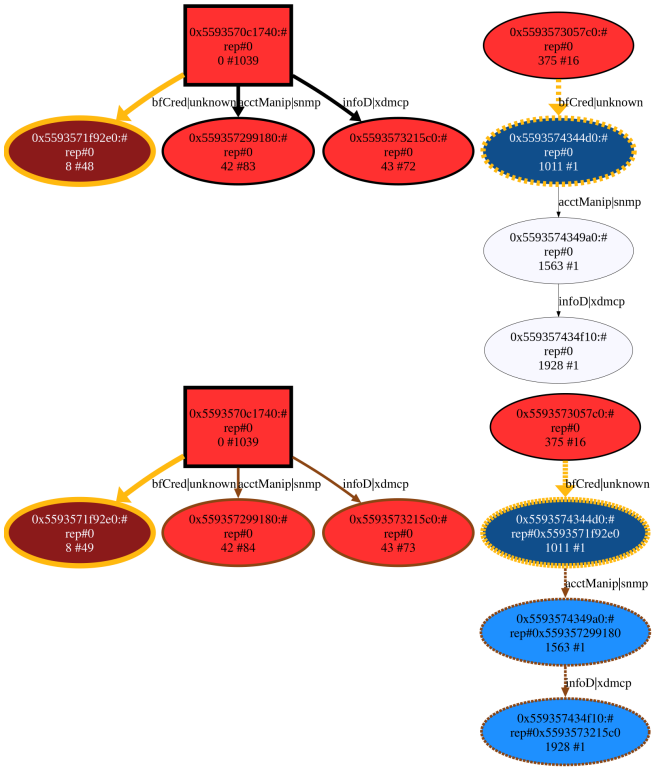
The created script massively reduces the workload of analysing merges (see Table 8) and the generated logs allow quickly finding interesting merges (e.g. by using `grep` on the node ID from an AG). Still, the number of merges is large to thoroughly analyse. We can, however, see a very important observation from the tree merges shown in Figures 5a, 5b and 5c, which correspond to the AG in Figure 4. We can see how three different subtrees with sinks have been merged with the same core states (with the children of the merged red states whose red representatives are shown in the figures), which resulted in the context loss described above. Before, SAGE used these sinks, which resulted in a separate sequence in an AG. Now that sinks have been merged, SAGE replays these attack traces on the same core states. Hence, the assumption that Alergia will not find enough evidence to prevent a merge, which should not occur due to the different context, was correct. Red core states could get more outgoing transitions that come from sinks (see Figure 5a), making subsequent merges even more likely to happen.



(a) Merge iteration 325 (left: before, right: after).



(b) Merge iteration 707 (left: before, right: after)).



(c) Merge iteration 733 (top: before, bottom: after).

Figure 5: S-PDFA merges corresponding to the AG above (generated by `get-merges.sh`). Yellow edges and nodes were initially merged, brown ones have been merged during the determinization. Solid nodes and edges belong to the subtree of the merged red node, dotted ones belong to the subtree of the merged blue node.

Table 8: Statistics on the merges and extensions in the main merging process and after with `mergesinks=1` and `mergesinkscore` set to 0 and 1 (from *get-merges.sh* and FlexFringe files).

	CPTC-2017		CPTC-2018	
mergesinkscore	**1**	**0**	**1**	**0**
# nodes in S-PDFA	2684		2013	
# nodes in get_merges.sh (avg)	10.15	9.80	10.62	10.52
# refinements (total)	924	1150	682	766
# extensions (total)	141	186	158	192
# merges (total)	783	964	524	574
% merges with determinization	65%	55%	68%	64%
# refinements (main)	141		98	
# extensions (main)	63		54	
# merges (main)	78		44	

Since the states 8, 42 and 43 are part of the red core, these merges occur because `mergesinkscore=1`. In contrast, with `mergesinkscore=0` (and `mergesinks=1`) such merges are not allowed, so the context should be better preserved. Appendix A presents the same AG when setting `mergesinkscore=0`. It could be that this setting is an improvement over our value, but it has to be investigated further. Table 8 also shows that our parameter value results in fewer extensions, fewer merges and a higher percentage of merges with additional merges in determinization. This makes sense, as core states are bigger subtrees due to their count, so more states get merged in the determinization. The core states also give more possibilities for merges in every iteration.

Conclusions From the Analysis of S-PDFA Merges
Considering the results above and after further investigating the states of AGs with our scripts, we can draw the following conclusions. For an AG node `mcat|mserv`:

1. All non-sink nodes remain non-sinks
2. If there is at least one non-sink node, all sink nodes will be merged with this non-sink node (or non-sink nodes, if there are multiple non-sink nodes)
3. If there are no non-sink nodes, one of the sinks becomes a non-sink and the remaining sinks are merged with it

The 'loss of context' problem occurs when there are multiple `mcat|mserv` nodes in an AG and at least one of them is a sink. If this is not the case, sinks just become non-sinks, possibly with another ID, and the AG is the same.

5.7 Discovered Bugs

It is rather tricky to verify the correctness of an unsupervised sequence learning tool like SAGE and detect bugs due to the lack of reliable 'test oracles' and since it is not always known what the correct results should be [16]. While performing this research, we have discovered five bugs in SAGE, which were indeed rather difficult to find. Below, we briefly present them in a chronological order, as well as propose tests to detect such problems in the future. More detailed descriptions and the proposed fixes can be found in the corresponding pull requests in the SAGE repository.

Bug 1: Non-determinism in the Most Frequent Service Description: The method `most_frequent` was implemented using Python sets, which are however non-deterministic. As a

result, in case of a tie, the most targeted service for an episode was arbitrarily chosen, leading to anomalies in the resulting AGs and non-determinism.
Proposed tests: The file with the episode traces passed to FlexFringe always has to be the same, no matter what the FlexFringe parameters are. Hence, performing a `diff` on the files with traces after consecutive runs for the same dataset should report that the files are the same.

Bug 2: Duplicate AGs and Attack Paths
Description: Due to some IPs being substrings of other IPs, many duplicate AGs and attack paths were generated as a result of wrong string comparison.
Proposed tests: As proposed by Senne Van den Broeck, attack paths might be assigned unique IDs, so that no two attack paths can have the same ID in different *.dot* files with the correct implementation. In addition, questioning attitude should be used when two AGs appear identical.

Bug 3: Capitalisation in Most Frequent Service
Description: The method `most_frequent` assigned 'Unknown' if a service could not be derived from the IANA mapping, while 'unknown' was used in other places. Due to Windows files and directories being case-insensitive (while being case-sensitive on Linux), AG files with different capitalisation were overwriting each other, leading to a different number of AGs on Windows and Linux.
Proposed tests: Primarily, manual analysis: AGs with different capitalisations were abnormal. In addition, it is important to be consistent in the used naming.

Bug 4: Other Duplicate AGs and Attack Paths
Description: Due to some objectives being substrings of each other (e.g. `DATA EXFILTRATION|http` and `DATA EXFILTRATION|http-alt`), some duplicate attacks were generated as a result of wrong string comparison.
Proposed tests: Same as for bug 2, as these are similar bugs.

Bug 5: Missing Sinks in AGs
Description: After creating the `get-merges.sh` script and analysing the S-PDFA merges, we have noticed that some states that were sinks in the S-PDFA were solid in the AGs. It turned out that when replaying the episode sequences, `traverse` method of SAGE was missing some transitions to sinks and sinks that were at the end of a trace.
Proposed tests: FlexFringe output files can be taken as the ground truth in this case, and it can be checked that dotted states in the AGs are indeed sinks in the `finalsinks.json` file, and that solid states in the AGs are non-sinks in the `final.json` file (see Appendix B for an example test case).

6 Conclusions and Future Work

The motivation behind the topic of this research is to test a particular modelling assumption of SAGE and evaluate the resulting AGs. Here we investigate what happens to the AGs after allowing sink states to be merged with other sink states and the core of the S-PDFA model after the main merging process, and how the size, complexity, interpretability and completeness of the AGs are affected. To conclude, we have demonstrated that merging sink states with other sinks and the core results in, on average, slightly smaller AGs with roughly the same complexity, worsened interpretability due to the loss of context of attack episodes and the same completeness as before merging sinks. Since interpretable AGs are of higher importance for a security analyst, we conclude that it is better not to merge sinks with the red core of the S-PDFA and other sinks, at least not without additional modelling assumptions.

We have developed the `get-merges.sh` script that could be used by other researchers for analysis of FlexFringe's S-PDFA merges. We have shown that the script massively reduces the workload (from more than 2000 nodes of the S-PDFA to, on average, approximately 10 nodes for a 'before', or equivalently 'after', fragment of the S-PDFA). The parameterized visual enhancements as well as extensive logs can facilitate the analysis by allowing quickly pinpointing interesting merges based on the `mcat|mserv` or a state ID in an AG (e.g. using the Unix `grep` command) and finding the corresponding S-PDFA merge on the indicated page in the PDF file. The tool can be further improved, extended and used for other use cases of FlexFringe, not just limited to SAGE.

As a future work, other modelling assumptions could be tested for merging sink states. First, it could be the case that only allowing merging sinks with other sinks and not with the core (i.e. setting `mergesinkscore=0` and `mergesinks=1`) would lead to fewer issues with context, as suggested in section 5.6. Furthermore, we have not found any attempts to tweak the `sinkcount` parameter, so its influence on the AGs could be investigated further. Other studies could try evaluation functions other than Alergia (FlexFringe allows implementing any evaluation function which implements the defined interface), or incorporating victim or attacker information into the data traces passed by SAGE to FlexFringe in order to mitigate the issue with the context. In addition, the influence of 'markovian' property (both with and without merging sinks) on the AGs and the context of episodes can be investigated (see Appendix C for an example of the resulting AG with `markovian=2`). Moreover, merging sinks does not need to happen in FlexFringe. They could be merged in SAGE during post-processing based on some criteria.

We have also noticed that not discarding state IDs from low-severity nodes and including low-severity sinks in the post-processing might lead to more interpretable AGs. Figures 3 and 4 suggest that such graphs, although slightly larger, might give more insights to a security analyst due to more interpretable paths. We think that a parameter could be added to SAGE that allows getting such 'full' version of AGs for further investigation.

Finally, this research is about the modelling assumptions of SAGE and not its correctness. We therefore assume the correctness of SAGE when performing our experiments. Even though during this research five bugs have been found in the code of SAGE (see subsection 5.7), further work might be needed to further investigate the correctness of SAGE. Its limitations such as no support for partial paths and a requirement for sequential learning (no support for parallelized attacks) could also be addressed. SAGE could also be tested on different datasets (potentially on the datasets with more ground truth available) to further analyse its performance in terms of accuracy, completeness and the resulting AGs.

7 Responsible Research

This section discusses the reproducibility of the obtained results (subsection 7.1) and the ethical aspects of the research (subsection 7.2).

7.1 Reproducibility

Reproducibility of research implies that independent (peer-) researchers are able to get the same or at least similar experiment results as the original paper [17]. It also allows assessing whether the results of the research are objective and reliable. It is therefore of crucial importance to consider reproducibility when designing any research.

All the code and data used for this research are open-source: SAGE, FlexFringe and created scripts can be cloned and executed by anyone who wants to verify the obtained results. Furthermore, both FlexFringe and SAGE are deterministic algorithms. As a consequence, the same results can be obtained after multiple consecutive runs of SAGE. Finally, the used Bash scripts have been extensively documented and the main *README.md* provides assistance in deciding which script to use in which case.

7.2 Ethical aspects

When performing any kind of research, it is important to adhere to ethical norms in research [18]. Namely, they promote the goals of research, such as knowledge transfer and truth, and the values crucial to collaborative work, such as trust and fairness. Furthermore, they usually promote other moral and social values, for instance, human rights, respecting the law and safety.

One ethical issue to be considered is the correctness of SAGE. While manually analysing intrusion alerts is a very labour-intensive process, it is still based on the incoming alerts. They might be false if an IDS is faulty, but they are still present. On the other hand, SAGE compresses all these alerts into a smaller visual representation to facilitate the work of security analysts. However, if SAGE wrongly summarises (some of) the alerts, then security analysts might draw incorrect conclusions, which might result in serious consequences, depending on the nature of the security operations centre. Nevertheless, this ethical implication is outside the scope of this research, as this paper focuses on the modelling assumptions of SAGE and not its correctness.

Acknowledgements

I would like to acknowledge my supervisor Azqa Nadeem and my responsible professor Sicco Verwer for their guidance and help during this research project. Furthermore, I would like to thank my peer researchers Ioan Oprea, Alexandru Dumitriu, Senne Van den Broeck and Vlad Constantinescu for help, insights and feedback throughout the process of this research. In particular, I would like to thank Ioan Oprea and Alexandru Dumitriu for their input and contribution in developing the methodology for the comparison of attack graphs, and Senne Van den Broeck for helping in the debugging process.

References

[1] Azqa Nadeem. The anatomy of Alert-driven Attack Graphs. https://cyber-analytics.nl/blogposts/2022-01-03-attack-graphs/, January 2022.

[2] Azqa Nadeem, Sicco Verwer, Stephen Moskal, and Shanchieh Jay Yang. Alert-driven Attack Graph Generation using S-PDFA. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2021.

[3] Shalom Tuby. Survey: 27 Percent of IT professionals receive more than 1 million security alerts daily, June 2019.

[4] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage. *Network and Distributed Systems Security Symposium*, February 2019.

[5] Steven Noel, Matthew Elder, Sushil Jajodia, Pramod Kalapa, Scott O'Hare, and Kenneth Prole. Advances in Topological Vulnerability Analysis. In *2009 Cybersecurity Applications & Technology Conference for Homeland Security*, pages 124–129, 2009.

[6] Dennis Mouwen, Sicco Verwer, and Azqa Nadeem. Robust Attack Graph Generation, 2022.

[7] Azqa Nadeem, Sicco Verwer, and Shanchieh Jay Yang. SAGE: Intrusion Alert-driven Attack Graph Extractor. In *Symposium on Visualization for Cyber Security (Vizec)*. IEEE, 2021.

[8] Azqa Nadeem, Sicco Verwer, Stephen Moskal, and Shanchieh Jay Yang. Enabling visual analytics via alert-driven attack graphs. In *SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2021.

[9] Sicco Verwer and Christian Hammerschmidt. FlexFringe: Modeling Software Behavior by Learning Probabilistic Automata, 2022.

[10] Rafael C. Carrasco and Jose Oncina. *Learning stochastic regular grammars by means of a state merging method*. Springer Science+Business Media, January 1994.

[11] Sean Carlisto De Alvarenga, Alessandro Ulrici, Rodrigo Sanches Miani, Michel Cukier, and Bruno Bogaz Zarpelão. Process mining and hierarchical clustering to help intrusion alert visualization. *Computers & Security*, 73:474–491, March 2018.

[12] Christopher Wanamaker. How to Create Your Own Simple Linear Regression Equation. Available: https://owlcation.com/stem/How-to-Create-a-Simple-Linear-Regression-Equation, July 2022.

[13] Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. Methods for interpreting and understanding deep neural networks. *Digital Signal Processing*, 73:1–15, February 2018.

[14] Subhi Issa, Onaopepo Adekunle, Fayçal Hamdi, Samira Si-Said Cherfi, Michel Dumontier, and Amrapali Zaveri. Knowledge Graph Completeness: A Systematic Literature Review. *IEEE Access*, 9:31322–31339, February 2021.

[15] RIT. CPTC dataset. Available: http://mirror.rit.edu/cptc/, 2018.

[16] Christian Murphy, Gail E Kaiser, and Marta Arias. An approach to software testing of machine learning applications. 2007.

[17] David B. Resnik and Adil E. Shamoo. Reproducibility and Research Integrity. *Accountability in Research*, 24(2):116–123, 2017. PMID: 27820655.

[18] D. B. Resnik. What is ethics in research and why is it important? *National Institutes of Health*, December 2015. https://www.niehs.nih.gov/research/resources/bioethics/whatis/.

A An Example of an Attack Graph With `mergesinkscore=0`

As discussed in subsection 5.6, not allowing the merges of sinks with the core of the S-PDFA might better preserve the context. Due to their high count, the red core states are likely to appear in more AG, potentially leading to the loss of context when sinks are also present in the AG and `mergesinkscore=1`. Figure 6 shows the same AG as in Figure 3 generated by SAGE when setting `mergesinkscore=0` and `mergesinks=1`. The merge of the sink subtree with the core non-sink subtree described in subsections 5.4 and 5.6 does not occur, and these sequences are correctly separated, resulting in a more interpretable AG.

B An Example Test Case for Sinks in the AGs

With the following pipelines, we can verify that all dotted states in the AGs are indeed sinks in FlexFringe output files. First, we get all unique sinks in CPTC-2017 (analogously for CPTC-2018) from the AGs. Then, using `jq`, we take all sink states from `orig-2017.txt.ff.finalsinks.json` file generated by FlexFringe (the name might be different), and extract their IDs. Finally, we take the intersection between the sink IDs found in the AGs and the sink IDs found in the FlexFringe output file (`comm -12 ...`) and count them (`... | wc -l`). The result is equal to the number of unique sink IDs found in the AGs, which means that all states defined as sinks in the AGs are also sinks in the FlexFringe file with the sinks.

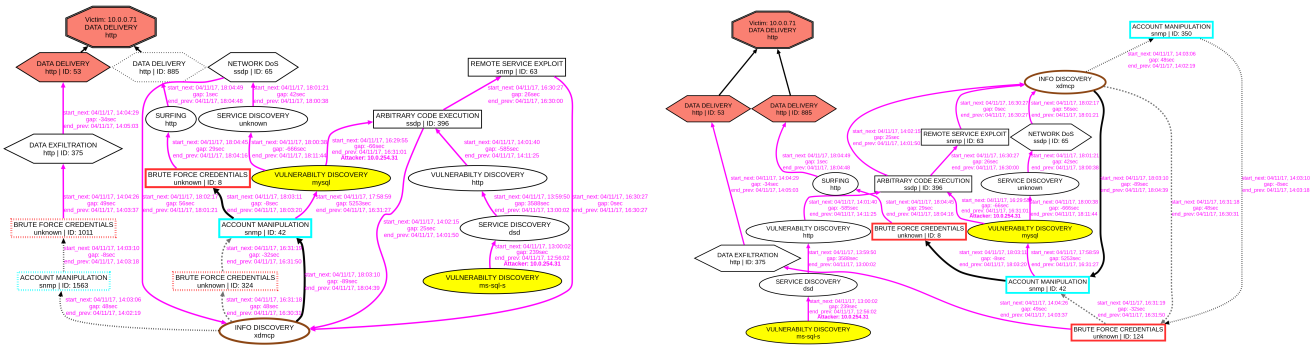
```
# All found sinks in 2017 are indeed sinks
$ sinks_orig_2017=$(find orig-2017AGs/ -type f
    -name '*.dot' | xargs gvpr 'N [ index($.style,
    "dotted") != -1 ] { print(gsub(gsub($.name,
    "\r"), "\n", " | ")); }' | sort -u)
$ all_sinks_2017=$(jq '.nodes[] |
    select(.issink==1) | .id'
    orig-2017.txt.ff.finalsinks.json | sort)
$ echo -e "$sinks_orig_2017" | wc -l
136
$ comm -12 <(echo -e "$sinks_orig_2017" | sed
    's/^.*ID: \([0-9-]\+\)$/\1/' | sort) <(echo -e
    "$all_sinks_2017") | wc -l
136

# All found sinks in 2018 are indeed sinks
$ sinks_orig_2018=$(find orig-2018AGs/ -type f
    -name '*.dot' | xargs gvpr 'N [ index($.style,
    "dotted") != -1 ] { print(gsub(gsub($.name,
    "\r"), "\n", " | ")); }' | sort -u)
$ all_sinks_2018=$(jq '.nodes[] |
    select(.issink==1) | .id'
    orig-2018.txt.ff.finalsinks.json | sort)
$ echo -e "$sinks_orig_2018" | wc -l
104
$ comm -12 <(echo -e "$sinks_orig_2018" | sed
    's/^.*ID: \([0-9-]\+\)$/\1/' | sort) <(echo -e
    "$all_sinks_2018") | wc -l
104
```

Similarly, we can verify that all non-sinks with IDs are indeed non-sinks (using `orig-2017.txt.ff.final.json` file, potentially with a different name).

```
# All non-sinks with IDs in 2017 are indeed
    non-sinks
$ non_sinks_with_ids_orig_2017=$(find
    orig-2017AGs/ -type f -name '*.dot' | xargs
    gvpr 'N [ index($.style, "dotted") == -1 ] {
    print(gsub(gsub($.name, "\r"), "\n", " | "));
    }' | sort -u | grep 'ID: ')
$ echo -e "$non_sinks_with_ids_orig_2017" | wc -l
28
$ all_non_sinks_orig_2017=$(jq '.nodes[] |
    select(.issink==0) | .id'
    orig-2017.txt.ff.final.json | sort)
$ comm -12 <(echo -e
    "$non_sinks_with_ids_orig_2017" | sed
    's/^.*ID: \([0-9-]\+\)$/\1/' | sort -u) <(echo
    -e "$all_non_sinks_orig_2017") | wc -l
28

# All non-sinks with IDs in 2018 are indeed
    non-sinks
$ non_sinks_with_ids_orig_2018=$(find
    orig-2018AGs/ -type f -name '*.dot' | xargs
    gvpr 'N [ index($.style, "dotted") == -1 ] {
    print(gsub(gsub($.name, "\r"), "\n", " | "));
    }' | sort -u | grep 'ID: ')
$ echo -e "$non_sinks_with_ids_orig_2018" | wc -l
16
$ all_non_sinks_orig_2018=$(jq '.nodes[] |
    select(.issink==0) | .id'
    orig-2018.txt.ff.final.json | sort)
$ comm -12 <(echo -e
    "$non_sinks_with_ids_orig_2018" | sed
    's/^.*ID: \([0-9-]\+\)$/\1/' | sort -u) <(echo
    -e "$all_non_sinks_orig_2018") | wc -l
16
```



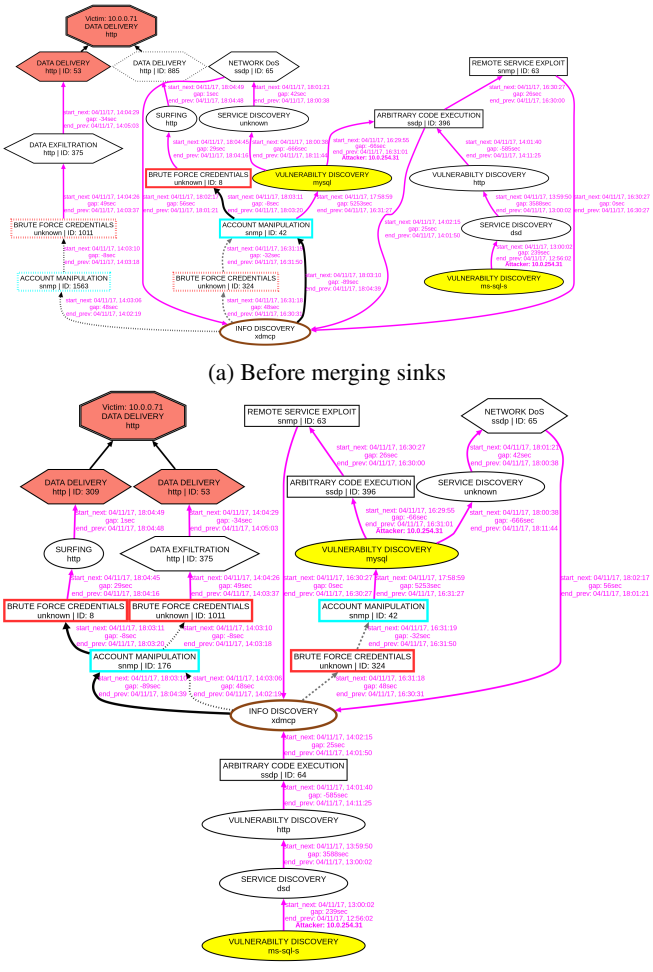
(a) Before merging sinks with other sinks.

(b) After merging sinks with other sinks.

Figure 6: AG `Data Delivery|http` on 10.0.0.71: an example of less loss of context with `mergesinkscore=0`.

C An Example of an Attack Graph With `markovian=2`

The `markovian` property of FlexFringe ensures that the incoming edges have the same label [9]. Setting `markovian=1` (the value used by SAGE) checks one edge above, setting `markovian=2` checks two edges above, etc. It is possible that this property could help preserve the context when merging sinks. Figure 7 shows the same graph as in Figure 3 generated by SAGE with `markovian=2`, `mergesinks=1` and `mergesinkscore=1`. It can be seen that the context is better preserved than in the graph in Figure 3.



(a) Before merging sinks

(b) After merging sinks

Figure 7: AG `Data Delivery|http` on 10.0.0.71: an example of less loss of context when setting `markovian=2`, `mergesinks=1` and `mergesinkscore=1`.