JOHANNES KEPLER
UNIVERSITY LINZ

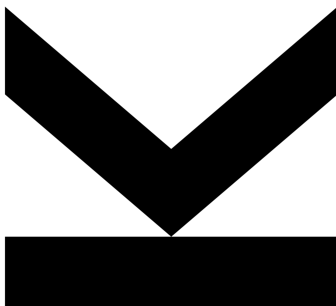
Author
Martin Schwingenschuh
k11803151

Submission
Institute of
Networks and Security

Thesis Supervisor
Dr. **Michael Roland**

March 2022

Android Device Security Database: Network monitoring



Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

Abstract

Smartphones generate an abundance of network traffic while active and during software updates. With such a high amount of data it is hard for humans to comprehend the processes behind the traffic and find points of interest that could compromise the device security. To solve this problem, this thesis proposes a system to automatically monitor the traffic of Android clients, store it in a database and perform a first analysis of the network data. For the capturing and monitoring tasks, we decided to use the full packet capture system Arkime and expand its functionality with a custom tool built in the course of this thesis. To be able to gain relevant insights, the system monitors the traffic over a long time frame, which prevents false data caused by holes in the data stream or one time events. All Android devices are separated from each other by assigning each device to a separate VLAN. For each session the system produces custom tags, low level statistical data and high level classification data. Further, the system provides a solution to apply custom rules in which data from sessions can be freely accessed and modified. Additionally, tags can be set with a matching of host names against custom regular expressions or update information stored in the database. The system uses only the captured data so that changes that can occur later on like the DNS resolution don't affect the accuracy of the outcome.

Kurzfassung

Smartphones erzeugen eine sehr große Menge an Netzwerk-Traffic während Software-Updates und während sie aktiv sind. Mit so einer Menge an Daten sind die Prozesse hinter dem Netzwerktraffic für Menschen nur schwer nachzuvollziehen, so dass die Gerätesicherheit gefährdende Features nur schwer zu erkennen sind. Zur Lösung dieses Problems wird ein System benötigt, welches den Datenverkehr von Android-Clients automatisch überwachen, in einer Datenbank speichern und eine erste Analyse durchführen kann. Für das Capturing und Monitoring bauen wir auf das Full-Packet-Capture-System Arkime auf und erweitern dessen Funktionalität im Zuge dieser Arbeit mit einem eigens entwickelten Tool. Um relevante Erkenntnisse gewinnen zu können überwacht das System den Datenverkehr über einen langen Zeitraum wodurch verhindert wird, dass fehlerhafte Features – die durch Lücken im Datenverkehrsstrom oder einmalige Ereignisse verursacht werden – produziert werden. Jedem Smartphone wird ein eigenes VLAN zugewiesen und damit von anderen Geräten isoliert. Das System generiert für jede Session benutzerdefinierte Tags, Low-Level- und High-Level-Statistiken. Weiters stellt das System eine Funktion bereit mit der benutzerdefinierte Regeln erstellt werden können. In diesen Regeln können die Daten der Session gelesen und bearbeitet werden. Zusätzlich können Tags mithifle eines Abgleiches der Hostnamen, erstellter Regular Expressions und gespeicherte Updatedaten generiert werden. Zur Analyse werden nur die vorhanden Netzwerk-Daten verarbeitet um dynamische Abhängigkeiten wie z.B. die DNS-Auflösung zu vermeiden, welche das Ergebnis verfälschen könnten.

Acknowledgements

This work has been carried out within the scope of ONCE (FFG grant FO999887054) in the program "IKT der Zukunft" and has partially been supported by Digidow (Christian Doppler Laboratory for Private Digital Authentication in the Physical World) and the LIT Secure and Correct Systems Lab. We gratefully acknowledge financial support by the Austrian Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK), the Austrian Federal Ministry for Digital and Economic Affairs (BMDW), the National Foundation for Research, Technology and Development, the Christian Doppler Research Association, 3 Banken IT GmbH, ekey biometric systems GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH & Co KG, Österreichische Staatsdruckerei GmbH, and the State of Upper Austria.

Contents

Abstract ii

Kurzfassung iii

Acknowledgements iv

1 Introduction 1
- 1.1 Motivation 1
- 1.2 System Overview 2
- 1.3 Network Setup 4

2 Arkime / ELK Setup 5
- 2.1 Arkime Setup 5
- 2.2 Indices 7
- 2.3 Custom Index “updates” 8
- 2.4 Custom Index “tagpatterns” 9

3 Analyzer 11
- 3.1 Program Flow 12
- 3.2 JSON Mapping 14
- 3.3 Interaction with Elasticsearch 15
- 3.4 Tagging Rules 15
- 3.5 Update Tagging 16
  - 3.5.1 Potential Methods 17
  - 3.5.2 Implementation 18
- 3.6 DNS Tagging 19
  - 3.6.1 Potential Method 19
  - 3.6.2 Implementation 20
- 3.7 Runtime Options 22
- 3.8 Compile-time Options 24
- 3.9 Status Indication 25
- 3.10 Dependencies 25

4 Future Work 26
- 4.1 Improvements of Core Principles 26
- 4.2 Miscellaneous Improvements 26

References 27

Chapter 1

Introduction

Smartphones generate network traffic with nearly every action. Even when idling and without user input network traffic is generated. This network traffic is an important part of understanding the processes running on a device. Further the traffic is a good basis for security research since most of the security breaches are dependent on transmitting data over the network interface. If the network traffic is captured and monitored such breaches can be detected and prevented in the future. For capturing the network traffic we use the full packet capture system Arkime[1]. And add with a custom tool additional functionality to be able to perform certain rules to derive higher level information and make this information easily readable.

1.1 Motivation

Since the network traffic becomes so large that for a human it is basically an unmanageable amount and thus not human-readable, a system is needed to automatically monitor the network traffic and do a basic analysis on the monitored data and transform it into something humans can work with. That can be done by representing the data on a higher level like sessions instead of single packets and additional information like added custom tags. Those tags should be meaningful for humans and contain derived information like the company associated with the IP address, and information on the encryption. Further we want to use the tags in search queries to increase the level of information gained. With the data flow shown in Figure 1.1 the user always has the automatically processed data available which is enriched with derived information in a human-readable format.

1. The raw network traffic is captured by Arkime.
2. Arkime processes the data, performs a statistical analysis and stores it on a session basis in a database.
3. The Analyzer automatically performs a detailed analysis on the stored data.
4. The user has the analyzed data available and can manually perform further actions.

[1]https://arkime.com

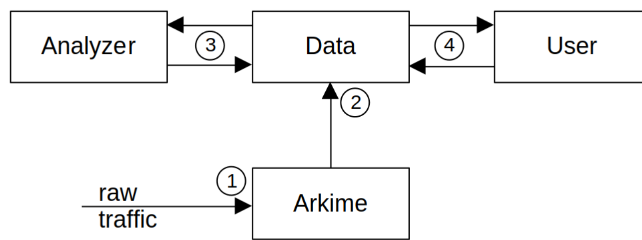


Figure 1.1: Logical data flow

1.2 System Overview

The system uses an OpenWRT[2] Access Point as wireless LAN connection for the smartphone devices seen in Figure 1.2. This access point is configured in a way, that all devices are separated from each other but still have full access to the internet. This is achieved by assigning each device to a different VLAN. The access point then mirrors each VLAN traffic via a L2TP-tunnel into a server for monitoring and analysis.

We decided to use the full packet capture system Arkime for capturing the network traffic and performing low level analysis and simplification. Arkime uses Elasticsearch (from the ELK stack[3]) as database solution and consists of a capture service and a viewer service. Where the capture service fetches the traffic from a network interface, groups packets into sessions and analyzes each session for low level features. When the sessions are complete, the session data is stored in the database as a JSON object and the raw packets are stored in pcap files. The viewer service only presents the data that is stored in the database and pcap files via a web interface. For more information on the Elasticsearch database see section 2.2.

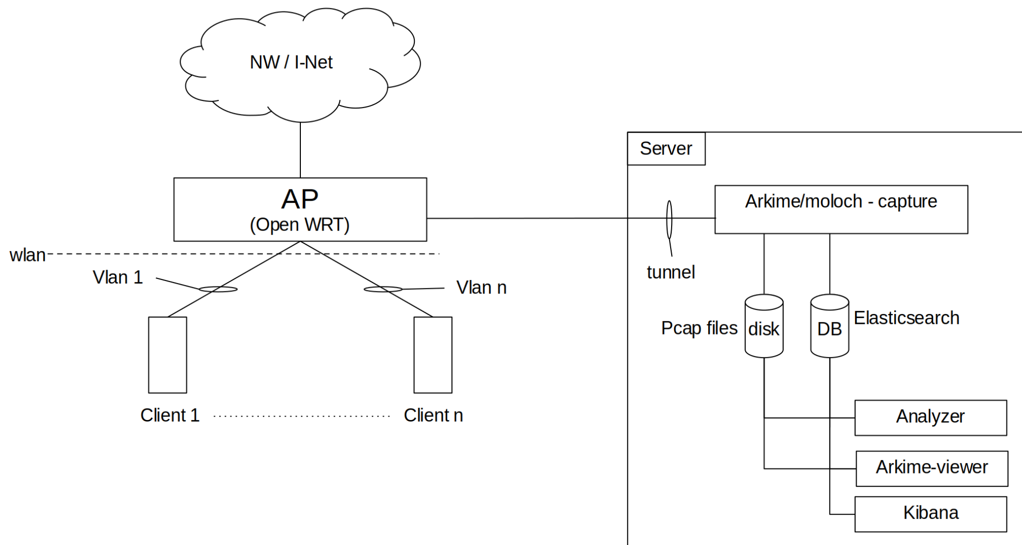


Figure 1.2: Overview of the system

[2]https://openwrt.org/
[3]https://www.elastic.co/what-is/elk-stack

Arkime provides a list of plugins that add additional analysis capability to the system. But we found while testing that those plugins either don't have enough flexibility or are not stable enough for us to rely on. Because of that we designed a custom service called Analyzer. This Analyzer uses only the data stored in the database and pcap files to extract additional features and has no other interactions except write operations on the database. This is to introduce as little dependencies as possible to the system.

Additionally, we set up Kibana from the ELK stack for simpler interactions with the database. Note that this is only for maintenance and completely optional. For the Arkime viewer and the Analyzer it is only necessary for the data to be present in the database. How the data is provided to the database is not relevant.

The Analyzer uses precomputed data that is stored in the database and does not read any files for the tagging at runtime. That is important for the execution time and is explained in detail in section 3. At the current stage of development there are two kinds of information fetched. The information when updates are triggered on devices (see section 3.5) and Tagpatterns used for matching host names with tags (see section 3.4).

Both the update tagging and the DNS tagging are implemented as separate rules. Additionally, the Analyzer allows for custom rules that are executed for each database entry explained in section 3.4.

The captured network traffic is stored in the Elasticsearch database and in pcap files shown in Figure 1.3. For simplification, we merged the different session indices to just one in the following figure, but in reality there are many more session indices. The only automatic write operations are performed by the Arkime capture service which stores the captured data in the session indices and pcap files. The indices "tagpatterns" and "updates" have to be filled manually and are only read by the Analyzer. The Analyzer acts as a mutator for the session indices. At the current state of the development, data is only added and not deleted.

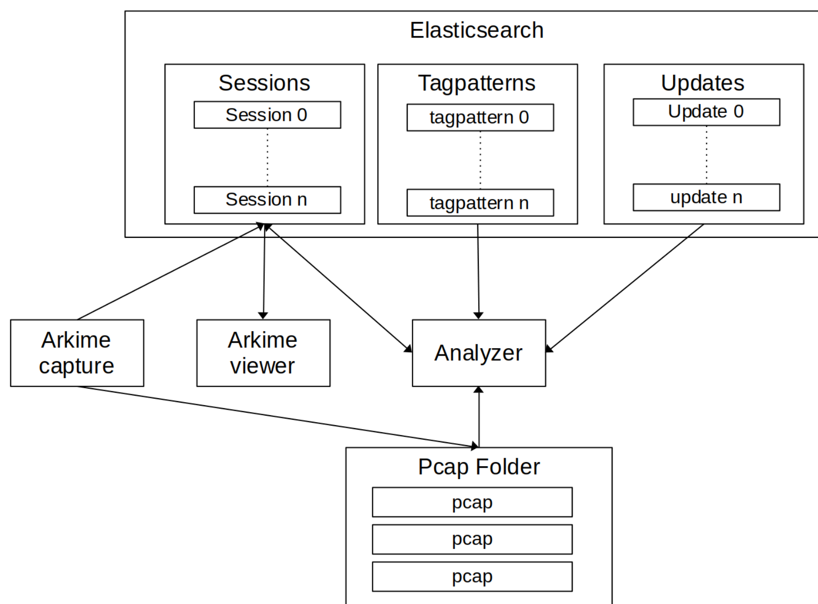


Figure 1.3: Overview of the data storage

1.3 Network Setup

The machine where the system (Arkime, Elasticsearch and Analyzer) is set up has additional network configurations explained in the following. To get traffic into an interface of the machine Arkime can capture, we need a tunnel from the access point to the machine. Since this thesis is only concerned with the virtual machine we will document here only one side of the tunnel. This tunnel was already set up, and we only document the set-up for the sake of completeness.

As implementation a L2TP-Ethernet-over-UDP encapsulation tunnel is used which adds an overhead but is because of its kernel support fairly easy to set up on the virtual machine side. For this implementation we need to add an interface and a systemd service.

Listing 1.1: /etc/network/interfaces

```
1  iface l2tpeth0 inet manual
2      pre-up ip l2tp add tunnel tunnel_id 1
3      peer_tunnel_id 1 udp_sport 5000
4      udp_dport 5000 encap
5      udp local 140.78.100.108 remote 140.78.100.105
6      pre-up ip l2tp add session
7      tunnel_id 1 session_id 1
8      peer_session_id 1
9      mtu 1428
10     down ip l2tp del tunnel tunnel_id 1
```

The interface l2tpeth0 is just added to the network/interfaces config file. To automatically start the tunnel when the machine boots a service is needed. This service file is stored in /etc/systemd/system/l2tp-capture-tunnel.service and has the following content.

Listing 1.2: Unit file for l2tp tunnel

```
1  [Unit]
2  Description=L2TP Capture Tunnel Interface
3  Requires=network-online.target
4  After=network-online.target
5
6  [Service]
7  Type=oneshot
8  RemainAfterExit=yes
9  #StandardOutput=tty
10 ExecStart=/usr/sbin/ifup l2tpeth0
11 ExecStop=/usr/sbin/ifdown l2tpeth0
12
13 [Install]
14 WantedBy=multi-user.target
```

Chapter 2

Arkime / ELK Setup

The system uses an instance of Arkime[1] which is a full packet capture system that captures and stores the incoming network traffic. The version used in this thesis is v3.0.0-GIT.

The ELK stack refers to the Elasticsearch, Logstash and Kibana combination from which the Elasticsearch[2] and Kibana[3] Modules are used. Elasticsearch is a RESTful search and analytics engine used by Arkime as database. The version used in this thesis is v7.9.3-oss. Since Elasticsearch is already needed by Arkime we decided to also use it as database for the Analyzer tool so that all the data is managed in one database solution.

2.1 Arkime Setup

The following additions to the config.ini file are needed for the Analyzer to work properly. This additional configuration creates the custom fields "AnalyzerVersion", "RuleVersion" and "FoundHosts" which are created automatically by Arkime on the next start of the Arkime capture service.

Listing 2.1: Custom fields defined in config.ini

```
1 # Custom field definitions
2 [custom-fields]
3 AnalyzerVersion=db:AnalyzerVersion;kind:termfield;friendly:Analyzer version;count:
      false;help:Version of Analyzer used on this Session
4 RuleVersion=db:RuleVersion;kind:integer;friendly:Rule version;count:false;help:
      Version of Analyzer rules used on this Session
5 FoundHosts=db:FoundHosts;kind:termfield;friendly:Found Hosts;count:true;help:Hosts
       found for this session
```

To make sure that the Arkime-capture module is started on boot of the machine and that the IP tunnel from section 1.3 is working, a service file is needed (see Listing 2.2). The "Requires" fields ensure that Elasticsearch and the L2TP tunnel are running before the capture module starts. Additionally, some configurations are performed by this unit so Arkime always behaves the same between restarts of the service.

For the Arkime-viewer a service is created to start the Arkime-viewer module on boot (see Listing 2.3). This service is only started if the Elasticsearch database is already running.

[1] https://arkime.com/
[2] https://www.elastic.co/elasticsearch/
[3] https://www.elastic.co/kibana/

We decided to use the standard configuration for the IP address and port Elasticsearch listens to, namely localhost:9200. Because of that the curl examples given in this document use this URL. If the address changes the URLs in the listed curl commands need to be updated.

Listing 2.2: arkimecapture.service

```
1  [Unit]
2  Description=Arkime Capture
3  Wants=network.target
4  After=network.target
5  Requires=l2tp-capture-tunnel.service
6  After=l2tp-capture-tunnel.service
7  Requires=elasticsearch.service
8  After=elasticsearch.service
9
10 [Service]
11 Type=simple
12 Restart=on-failure
13 #StandardOutput=tty
14 Environment="ARKIME_INSTALL_DIR=/opt/arkime"
15 EnvironmentFile=-/opt/arkime/etc/molochcapture.env
16 ExecStartPre=-/opt/arkime/bin/moloch_config_interfaces.sh
17 ExecStart=/bin/sh -c '/opt/arkime/bin/capture -c /opt/arkime/etc/config.ini ${
       OPTIONS}'
18 WorkingDirectory=/opt/arkime
19 LimitCORE=infinity
20
21 [Install]
22 WantedBy=multi-user.target
```

Listing 2.3: arkimeviewer.service

```
1  [Unit]
2  Description=Arkime Viewer
3  Wants=network.target
4  After=network.target
5  Requires=elasticsearch.service
6  After=elasticsearch.service
7
8  [Service]
9  Type=simple
10 Restart=on-failure
11 #StandardOutput=tty
12 User=root
13 EnvironmentFile=-/opt/arkime/etc/molochviewer.env
14 ExecStart=/bin/sh -c '/opt/arkime/bin/node viewer.js -c /opt/arkime/etc/config.
       ini ${OPTIONS}'
15 WorkingDirectory=/opt/arkime/viewer
16
17 [Install]
18 WantedBy=multi-user.target
```

2.2 Indices

Arkime creates a lot of indices when Elasticsearch is initialized with the db.pl script provided by the Arkime installation. For more details on the initialization see the documentation[4]. For this thesis, the relevant Arkime indices are the ones where the sessions and the paths of the pcap files are stored. Arkime creates a separate index with the pattern `sessions2-yymmdd` for each day. We saw that in different Arkime versions different prefixes are used, which have to be set in the Analyzer-Options explained in section 3.8. A list of all the stored indices can be queried with the following command.

Listing 2.4: Curl Command - list indices

```
1 curl -X GET http://localhost:9200/_cat/indices
```

Document IDs of the session documents have the pattern `yymmdd-<some hash>` where the timestamp is equivalent to the one in the index.

The mapping of sessions should be checked with the following command when using newer versions of Arkime, since we found that this too changes with the version. The value index-id is to be replaced with the index in question.

Listing 2.5: Curl Command - show session mapping

```
1 curl -X GET http://localhost:9200/index-id/_mapping?pretty
```

If the Arkime version changes and with it the mapping of the indices, the Compile options of the Analyzer also need to change; see section 3.8. It is sufficient to check the mapping of one session index since Arkime uses the same mapping for all session indices. Per default, stored indices are never deleted which can lead to a problem if Arkime is configured to use 100% of disk space or simply a loss of data when no new indices are created to store data to. It is important to prevent that with enough disk space or with the solution recommended by Arkime which is to set up a cronjob to automatically delete data after a set period. The cronjob has to call the script "db.pl" in Arkime included as explained in the Arkime documentation [1]. Of course indices can be deleted manually with the Kibana interface or with curl commands, but it is not recommended unless a tool is used to ensure that only the oldest indices are deleted to prevent gaps in the data. Note that deleting an index does not delete the pcap files stored on disk. The custom indices needed for the Analyzer and any other index stored in Elasticsearch that is not inserted by Arkime are not handled by the deletion script and need a separate solution. We decided that a deletion by query is sufficient since the data has to be managed manually at the current development stage.

[4] https://github.com/arkime/arkime/blob/v3.3.1/release/README.txt

2.3 Custom Index "updates"

The information when a device starts and stops an update is stored in the Update index with one start-update and one optional stop-update document distinguished by the "type" field. A true value of the field "type" stands for a start and a false value for a stop marker. To distinguish different client devices we decided to use the VLANs associated with the devices. The stored data is used by the Analyzer as explained in section 3.5. We decided to omit any restrictions on the document input to be flexible and reduce the overhead. That also means that when adding documents to the index, no checks if the data is reasonable or complete are done. Those checks have to be done by the user or by a query inserting the document.

Listing 2.6: Curl Command - create update index

```
curl -XPUT localhost:9200/updates -H 'Content-Type: application/json' -d'
{
    "mappings":{
        "properties":{
            "vlan":{"type":"keyword","index":true},
            "mac":{ "type":"keyword","index":true },
            "timestamp":{ "type":"date_nanos" },
            "type":{ "type":"boolean" },
            "comment":{ "type":"text"}
        }
    }
}'
```

The mapping can be queried with the following command that shows the custom fields.

Listing 2.7: Curl Command - get mapping of updates

```
curl -XGET localhost:9200/updates/_mapping?pretty
```

With the following command we give an example how to store documents with a curl command. We add an update marker u1 at a certain time for a certain device. The VLAN is a keyword field used for mapping update markers with devices and can be set to arbitrary values. The MAC address is just a string containing the MAC as text. We decided to use the field type "keyword" so that it is indexed by Elasticsearch. The time is set with the field timestamp in epoch format. Since in this case we add a start marker we set the type to true. The comment is just an arbitrary string that is not indexed and with that not searchable.

Listing 2.8: Curl Command - inserting documents

```
curl -XPOST localhost:9200/updates/_doc -H 'Content-Type: application/json' -d'
{
    "vlan":"144",
    "mac":"52:54:00:a0:59:b7",
    "timestamp": 1629732600000,
    "type":"true",
    "comment":"u1"
}'
```

To query all documents stored in the index one can use the following command. If the number of documents exceeds the limit from Elasticsearch the scroll argument has to be added and further queried with the scroll API. For more information see section 3.3 or the Elasticsearch API documentation[5].

Listing 2.9: Curl Command - get all updates

```
curl -XGET localhost:9200/updates/_search -H 'Content-Type: application/json' -d
    '
{
    "query":{"bool":{"must":{"match_all":{}}}}
}'
```

A single document can be queried with the following command. The document ID used by Elasticsearch is not set when storing a document but rather generated automatically by Elasticsearch.

Listing 2.10: Curl Command - get a single update document

```
curl -XGET localhost:9200/updates/_doc/<document-id>
```

2.4 Custom Index "tagpatterns"

The index "tagpatterns" stores the data which regular expression connects to which tags. One entry can have one regular expression (has to be POSIX conform) and one to many tags stored as strings. The identification is handled via an ID automatically generated by Elasticsearch. This index is intended to be written externally with curl commands or via the Kibana interface and is only read from the Analyzer.

We decided that a simple mapping and no restrictions for the inserted documents is best for our use case to have the most flexibility. That means that the documents are not checked for their content whatsoever. So duplicate and missing data is possible and has to be handled when inserting documents. The index was created with the following curl command showing the index mapping for the custom fields.

Listing 2.11: Curl Command - create tagpatterns index

```
curl -XPUT localhost:9200/tagpatterns -H 'Content-Type: application/json' -d'
{
    "mappings":{
        "properties":{
            "regex":{ "type":"keyword","index":true },
            "tags":{ "type":"keyword","index":true },
            "comment":{ "type":"text"}
        }
    }
}'
```

The mapping can be queried with the following command that shows the custom fields.

[5] https://www.elastic.co/guide/en/elasticsearch//reference/current/scroll-api.html

Listing 2.12: Curl Command - get mapping of tagpatterns

```
curl -XGET localhost:9200/tagpatterns/_mapping?pretty
```

To add documents to the index we used curl with the following command. Here we add an example tag pattern with the regular expression "`google`" which matches all strings where the string "google" occurs. The Analyzer does not add anything to the regular expression so if a full text search is preferred then the control characters "^" and "$" are needed. For example "`^.*google.*$`" would check the whole string. For a more detailed description on how to write these expressions refer to the GNU documentation[6]. Here we add just one tag into the tags array, but there is no limit on the number of tags stored in the document. The comment field is a text field where arbitrary strings can be stored but not searched in queries since this field is not indexed.

Listing 2.13: Curl Command - insert tagpattern

```
curl -XPOST localhost:9200/tagpatterns/_doc -H 'Content-Type: application/json'
    -d'
{
    "regex":".*google.*",
    "tags":["google"],
    "comment":"everything google"
}'
```

The following command lists all documents in the index. At some time the number of documents will exceed the limit Elasticsearch returns. In this case the query has to be expanded with a scroll argument and queried with the scroll-api.

Listing 2.14: Curl Command - get all tagpatterns

```
curl -XGET localhost:9200/tagpatterns/_search -H 'Content-Type: application/json
    '
-d'
{
    "query":{"bool":{"must":{"match_all":{}}}}
}'
```

A single document can be queried with the following command. The document ID used by Elasticsearch is not set when storing a document but rather generated automatically by Elasticsearch.

Listing 2.15: Curl Command - get a single tagpattern document

```
curl -XGET localhost:9200/tagpatterns/_doc/<document-id>
```

Alternatively one could also use more sophisticated methods provided by Elasticsearch e.g. search-api or update-by-query. We found that the sql-api does not work with the index layout we chose. We will not explain those methods in detail and refer to the Elasticsearch documentation[7].

[6] https://www.gnu.org/software/findutils/manual/html_node/find_html/Regular-Expressions.html#Regular-Expressions
[7] https://www.elastic.co/guide/en/elasticsearch//reference/current/rest-apis.html

Chapter 3

Analyzer

The Analyzer is a C program written in the course of this thesis that enriches data stored by the Arkime system with custom tags. Those tags are set by applying custom written rules. The Analyzer has the custom indices updates and tagpatterns as dependencies which are automatically checked but not created if absent. We decided that we develop this program with a clean separation between functionality and data in mind. For data storage we use the already existing Elasticsearch database which has the advantage that the storage is very efficient and there is only one logic place where data is stored and not in different files. That means that the Analyzer has to fetch the needed data like the tagpatterns from Elasticsearch for each execution at runtime. Which is from a performance point of view better than computing data at runtime. The decision of the programming language is purely by preference, any other language that can perform bitwise operations on the raw traffic data is sufficient. If the low level packet analysis is not needed at all one could port this program into a high level language. Figure 3.1 shows the file structure of the source files.

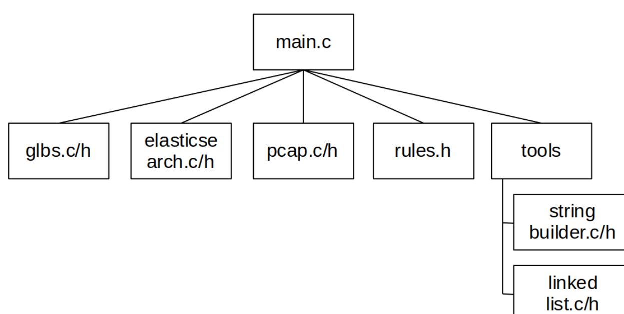


Figure 3.1: Source File structure

- glbs.c/h
  Contains the compile options explained in section 3.8 and data management. Additionally, this file defines some structs with some helpful functionality usable in rules.
- elasticsearch.c/h
  Handles the read and write interactions with Elasticsearch especially the execution of curl commands.
- pcap.c/h
  Contains all the functionality for the interactions with pcap files, primarily searching for and in pcap files.
- rules.h
  In this file the custom rules are defined as functions.
- tools.c/h
  Contains some useful data structures like a string builder and a linked list.

3.1 Program Flow

The main function handles the preparations needed for a run, like parsing the arguments and getting the update data and tag patterns, and executes all rules defined in the file rules.h. Since we decided to rely on the host system to periodically call this program there is no need for a main loop. When called, the program handles preparations, sets the scope and analyzes the data just once. The steps "parse Arguments", "check Dependencies" and "read Matchdata from Elasticsearch" are preparation steps while analyzeSession applies the defined rules to the set scope. The Matchdata is a list of elements extracted from the index "tagpatterns" and contains the regular expressions and the corresponding tags. A block is a set of sessions defined by start and stop time used to determine the scope to work on.

Algorithm 1 Analyzer main program flow

```
 1: procedure main(argc, argv)
 2:     parse Arguments
 3:     check Dependencies
 4:     read Matchdata from Elasticsearch
 5:     if argScope = single id then
 6:         analyzeSession(session)
 7:     else if argScope = block then
 8:         calculate scope of block
 9:         for each session in scope do
10:             analyzeSession(session)
11:         end for
12:     else if argScope = all then
13:         for each session in database do
14:             analyzeSession(session)
15:         end for
16:     end if
17:     cleanup
18: end procedure
```

The function "analyzeSession" analyzes only one session and is just called more often for the different scopes. We decided to design the program flow this way so that all the functionality concerning the data is centralized in one place and not scattered across different functions. This means of course that some optimizations could not be implemented since the function handling all the logic has only access to one session, but we decided that in this case a simple and a more stable execution of the program is more important than the execution time. This design also has the advantage that a multithreaded implementation is easily possible in the future, since the "analyzeSession" function can be seen as an atomic block. The scope is set with the program arguments when the program is called and once started can not be changed. See section 3.7 for more information on the possible arguments. It is possible for different instances of the Analyzer to have overlapping scopes. That poses no problem if the same version is used since the actions on the Elasticsearch database are atomic. But we recommend to only have one running instance because the Analyzer is designed in a way that does not increase the performance using multiple instances. If a performance increase is needed it would be better to update the Analyzer to support multithreading.

The function “analyzeSession” handles the analysis of a single session which includes fetching data from the Elasticsearch database, applying the custom rules and storing inferred information into the Elasticsearch database. This function can be seen as an atomic and is called for each session in the working scope. We decided to design it this way to make future work implementing multithreading easier. Since this function is such an integral part of the program we describe it in more detail.

Algorithm 2 AnalyzeSession program flow

```
 1: procedure analyzeSession(session_id)
 2:     check session id
 3:     extract session from elasticsearch and pcap files
 4:     if ruleVersion > old ruleVersion then
 5:         remove old hosts
 6:         for each rule in rule_ptrs do
 7:             execute rule
 8:         end for
 9:         remove old tags in ES
10:         for each tag in tags do
11:             add tag to ES
12:         end for
13:         set rule/analyzer version in ES
14:     end if
15: end procedure
```

The step “check session id” is a check against a regular expression describing the prefix “`^[0-9]\{6\}\-.*`” of the session ID which is always a six-digit number representing the date the session was captured. This check does not guarantee that the given session ID is valid, but it is a good indicator since we found that most of the errors that lead to an invalid ID, change the String in such a way that the prefix is too short or too long.

The “extract session from elasticsearch and pcap files” step searches the Elasticsearch database for the given session ID and stores the returned JSON object. Further the raw data packets from the session are read from the pcap files if those files exist. Since Arkime does not guarantee that the pcap files for a session exist we too can not guarantee that the packets are provided for the rules. The pcap files are stored on disk and the mapping from session to pcap file is stored in an index in Elasticsearch. That means that in the next step where each rule is executed the JSON object is guaranteed and rules are not executed without it. If packets are dependent on the packets a null check needs to be present in the rule function itself. If a rule produces an error the program stops. There is no error handling in the current version.

The session is only processed further if the current rule-version, set with the option “RULEVERSION”, is higher than the rule-version stored in the session. This prevents a second analysis with the same rule set. This check can be deactivated with the “--ignoreversion” argument.

Executed rules add their tags into a list that is the same for all rules. When all rules are executed, the old tags stored in Elasticsearch are deleted to prevent a growth in the “tags” field if the Analyzer is executed on the same session more than once. Then the collected tags are written to Elasticsearch. After that the program and rule versions set in the compile options are written. For more information on the options see section 3.8.

For debugging and further development, we point out that this function is a core component of the analysis process and is called from all scopes. Meaning that if this function works properly for the session scope it works for all scopes. Additional logic deriving new information from the data should be put in a custom rule to have a clean separation between code working on the data and code needed as overhead handling the execution of rules and the communication with Elasticsearch.

3.2 JSON Mapping

Arkime stores sessions as JSON objects in Elasticsearch which are parsed by the Analyzer. Those JSON objects have a certain structure (mapping in Elasticsearch) that can change with the version of Arkime. Meaning that this mapping has to be checked after an update of Arkime. The Analyzer extracts the most common fields with the function `es_parse_session` in the Elasticsearch module and provides a session object to rules which can access the data without parsing the JSON again. If fields are not provided in the session struct, a rule can parse the original JSON object on its own. The `es_parse_session` function uses a hardcoded static mapping that can lead to problems if not manually compared with the objects stored in Elasticsearch. If objects are not where they are expected, the used library just returns a NULL value that can lead to a program crash later in the execution of rules if not checked correctly. Out of time constrains we did not provide a dynamic solution, but one could prevent this error source by implementing a dynamic mapping in the Analyzer. This could be achieved by defining a file where the mapping from Elasticsearch is stored and used by the Analyzer to get the structure and paths to objects. Alternatively, a flattening approach would also solve this issue. If the JSON object is transformed into a flat table with field names as keys, rules could just search in this table and don't have to consider the mapping at all.

Listing 3.1: Mapping example of older Arkime version

```
_source:{
    "srcIp": "10.111.222.114",
    "srcMac": [ "58:cb:52:17:e0:06" ],
    "dstIp": "10.111.222.115",
    "dstMac": [ "58:cb:52:17:e0:07" ],
    ....
}
```

Listing 3.2: Mapping example of newer Arkime version

```
"_source":{
    "source":{
        "ip": "10.111.222.114",
        "mac": [ "58:cb:52:17:e0:06" ],
        ....
    },
    "destination":{
        "ip": "10.111.222.115",
        "mac": [ "58:cb:52:17:e0:07" ],
        ....
    }
    ....
}
```

3.3 Interaction with Elasticsearch

Since the Analyzer is built in a way that, at runtime, data (except pcap files) is not fetched from files in the file system but exclusively from Elasticsearch and also the network traffic is stored in Elasticsearch, this interaction is a core functionality. The function handling all interactions between the Analyzer and Elasticsearch is called `es_execute` and is located in the "elasticsearch" module. This function is an additional abstraction layer for the curl library functions. That means that some options are hardcoded especially for this use case and can't be changed.

Listing 3.3: Signature of curl wrapper function

```
MemoryStruct* es_execute(char *url, char *payload)
```

The function takes two parameters, "url" and "payload" which are strings defining the commands to execute, The parameter "url" has to be set and the parameter "payload" can be NULL when no payload is required for the command. Each call of es_execute is equivalent to a curl command in a terminal with the url-parameter being the url argument and the payload-parameter being the data (-d) parameter. The returned struct is a memory area that is filled with the result of the curl execution which is a JSON object in text format. The memory for the returned struct is allocated by the function but not managed whatsoever, meaning the caller has to free the struct later on. Since this function is only used by the Analyzer we decided to keep the interaction as simple as possible resulting in some options hardcoded. If the command produces output larger than Elasticsearch has set as maximum return value the query has to be traversed in batches. This function has, at the current state of development, no checks built in if the result returned by Elasticsearch contains all documents or not. This has to be checked by the caller but can be done by checking the returned JSON object. If this happens the original command has to be expanded with the scroll parameter and all following calls have to set the scroll parameter in the payload instead of a query. If a scroll parameter is given, the associated query is automatically executed by Elasticsearch returning the next batch of results.

3.4 Tagging Rules

The rules are functions in the "rules" module that are applied by the Analyzer for each session in the working scope. Since those rules will not change frequently we decided to add them in a module which will be compiled with the Analyzer. This means that the rules are not entirely separated from the applying program but somewhat interleaved. A rule has to use the following signature.

Listing 3.4: Signature of a rule and type definition

```
void rule_name(Session *session, LinkedList *tags);
typedef void (*rule)(Session*, LinkedList*);
```

Where the arguments session and tags are set by the Analyzer. The session argument holds the JSON object, some parsed values and the raw packets for the current session. Since the raw packets are not always stored by Arkime it is not guaranteed to be populated in the session struct e.g. `session->packets` is NULL. If a rule wants to add a tag to the current session the tag needs to be added to

the tags argument with the following call, which only adds the tag if it is not already stored to prevent multiple instances of the same tag. Further, rules can delete tags stored by other rules already executed.

Listing 3.5: Add/Delete a tag in a rule

```
int ll_add_string_distinct(LinkedList *taglist, char *tag);
int ll_delete_string(LinkedList *taglist, char *tag);
```

Rules can access the global matchdata and the global arguments which are guaranteed to be set by the Analyzer before applying the rules. The variable `extern MatchData *matchdata` contains the regular expressions with the corresponding tags extracted from the Elasticsearch index "tagpatterns" in a simple list. The variable `extern Arguments glb_args` contains the arguments struct with the arguments which the Analyzer was called.

After defining a rule, the function name has to be added to the rule_ptrs array defined in the rulee module. This is necessary for the Analyzer to know which rules to apply. Rules that are not in the array will be ignored by the Analyzer.

Listing 3.6: rule_ptrs array

```
rule rule_ptrs[] = {
    rule_encrypted,
    rule_updates,
    rule_dns,
    rule_nosource,
    rule_dns_error,
    rule_test,
};
```

The tags list is the same object per session for all the rules. Since the rules are applied in the order given in the rule_ptrs array, rules can delete the tags from its predecessors or use the tags as a form of communication between rules to create dependent rules.

3.5 Update Tagging

The goal of the update tagging is to automatically detect updates in the network traffic and mark the corresponding sessions with a tag. There exists work for clustering of network traffic focusing on a wider view that could be adapted to detect updates [2, 6]. Most of the newer methods for clustering and classification are based on machine learning algorithms requiring a training set. Since we do not have a sufficiently large training set we focus here on solutions that produce data that can be used in future work to implement such a machine learning algorithm. Android updates are fetched from an OTA-server[1] over HTTPS. That means that we loose features useful for the detection. Some features lost are for example the host headers, request path, and parameters, basically everything above the OSI-layer where TLS is implemented (e.g. Application layer) is lost because the TLS communication channel is encrypted. Some features, like the hostname, are potentially recovered by the DNS-tagging functionality described in Section 3.6 or other custom rules while most of them, especially the content, are lost.

In the following we describe potential methods to discover updates and build a data set for future work. The mentioned methods are not an exhaustive list of all

[1]https://source.android.com/devices/tech/ota/ab

possibilities but only a selected set of methods we considered. All the methods mentioned could be used, but we decided to only implement one solution.

3.5.1 Potential Methods

Hostname Zaman et al. [13] proposed a method to detect malware by comparing the URLs with a blacklist and Saidi et al. [9] proposed a method to detect IoT devices. Both methods are similar and can also be used to detect updates by maintaining a list of known update URLs the devices connect to. The main problem with this approach is the static behavior of the list. Meaning this list has to be created by hand and changes to the hostnames are not detected which makes a periodical check necessary. Further, the reliability of this approach is relatively low because one hostname is potentially not only used for updates.

IP address Similar to the hostname approach, a list of update servers is maintained, just instead of using the hostnames, the IP addresses are used. This approach has all the disadvantages of the hostname approach plus the problem that the IP addresses can change frequently. Further, the URL to the OTA-server is stored on the device, which makes the hostname approach better suited. For example in HTC devices the URL is stored in the service "HTCOTAClient.apk" or "HTCCheckin.apk"[2].

Data size The packet size is stored by Arkime and used in this approach to identify updates. Sessions exceeding a certain size threshold are marked as updates. The problem with this approach is that a good threshold is not easy to determine because updates have different sizes. Additionally, this method guarantees false positives (normal large packets) and false negatives small updates.

Update DB In this approach a supervisor triggering the device updates stores the start and stop time of each update per device in a database. This could also be done by the device if changes to the device are allowed. The tagging is done by the Analyzer with the information provided by the update database and the traffic. This approach has the disadvantage that the database for the update data has to be filled by a system which has to be implemented.

We evaluate the approaches with three dimensions depicted in Table 3.1, manual work being the effort for personnel building and maintaining the solution, accuracy representing a measure how many false positives and false negatives we expect and reliability the long term stability. Since we could not find a satisfying approach to detect updates only from the traffic data we decided to implement the Update DB Approach since it is the only method with a high accuracy and reliability. Further, a large collection of training data can be acquired helping the implementation of more sophisticated classifiers.

Table 3.1: Comparison of update tagging solutions

Approach	Manual work	Accuracy	Reliability
Hostname	medium	medium	medium
IP address	medium	medium	low
Data size	low	low	low
Update DB	medium	high	high

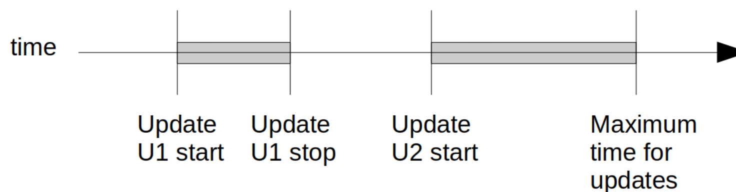
[2] https://www.droidwiki.org/wiki/OTA-Server

3.5.2 Implementation

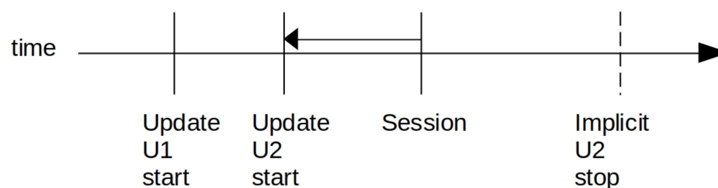
The Analyzer implements the update detection with a custom rule in the "rules" module. This rule checks for every session if an update is stored in the database and marks the session with the tag "update". This tag is stored in Elasticsearch and with that automatically shown in the Arkimeviewer.

The required database is realized as a custom index "updates" in the already existing Elasticsearch database. This index needs to be filled with data from an external source. Here we assume that this part will be done in follow-up projects. The external source could be a supervisor that triggers the updates on devices or the devices themselves if modifications of the devices are allowed. All that is needed is an execution of the curl command shown in Listing 2.8 at the beginning and optionally the end of an update to provide the necessary information.

An update is linked to a client with its VLAN and timestamp. For the Analyzer an update for a device is valid, if the VLAN matches with an update entry in Elasticsearch and the timestamp from the session is in the scope of the most recent update. The scope of an update is the time it takes to be performed marked with a start marker and terminated with an end marker (see Figure 3.2). Markers are documents in the Elasticsearch index (see Section 2.3). The end marker is optional and the scope of those updates can be set via the compile-time options described in section 3.8. If more than one update is in the scope of one session only the nearest is actually considered. At the current state it makes no difference since a static tag is used for all updates, but can be relevant when introducing dynamic tags with more information in them. This case should not happen in practice since devices do not run updates that frequently, but this can be an error source in the future if the data in the index "updates" is not accurate.



(a) Update range



(b) Update example

Figure 3.2: Update tagging

3.6 DNS Tagging

Arkime stores only the source and destination IP addresses for a session which makes the analysis of the traffic harder. To improve the usability the goal of the DNS tagging is to link those IP addresses a client connects to with potential host name candidates stored as tags. We decided to condense the hostnames to simple tags with a mapping of regular expressions to tags.

There already exists work on analyzing network traffic utilizing DNS data [5, 10, 12]. We come to the same conclusion that a static approach described in 3.6.1 is not viable, and instead perform the analysis on the real captured DNS traffic.

DoH (DNS over HTTPS) [3] and DoT (DNS over TLS) [4] would compromise our approach so much that the Analyzer could not perform the DNS tagging anymore for obvious reasons. For simplicity, we do not consider those cases in the current solution and reference to already existing work that solves this problem with ML classifiers [11].

3.6.1 Potential Method

One solution we considered is a static dataset containing known hostnames and the resolved IP addresses computed either at each execution of the Analyzer or precomputed at some point in time. Additionally, IP addresses not stored in the list are resolved to hostnames with rDNS queries at analysis time. With such an approach certain problems arise that are reducing the reliability to such an extent that the usability of this solution is very limited. The problems we identified are caused by the nature of the domain name system [7] and are as follows.

- A records which link a hostname to an IP address can change over time causing the stored data to differ from the actual records. This difference is bound to happen at some point in time since the list is only expanded with unknown hostnames at analysis time. This on-the-fly resolution just assures that all the occurring hostnames are stored, but does not fix the underlying problem of changing A records.
- PTR records which link IP addresses to hostnames, which can again point to A records, CNAME records or even no record, can also change over time resulting in a similar difference.
- PTR records resolved at analysis time may not always match the A record resolved by the device at communication time. This would result in a difference between the mapping stored by the Analyzer and the mapping the device used.
- An A record can resolve to multiple IP addresses and one DNS node can have multiple A records resulting in a many-to-one mapping. Similar to that, there can also be multiple CNAME records pointing from multiple different hostnames to one CNAME resulting in a many-to-one mapping overall. This many-to-one mappings increase the complexity of a static solution.
- PTR records are one to many mappings making it difficult to determine which hostname was requested originally.

3.6.2 Implementation

To avoid this kind of behavior we decided to work with the hostnames requested by the clients via DNS requests extracted directly from the real traffic stored in Elasticsearch. We use the data prior to the session so that we know what the actual data was the client had at hand. To extract the hostnames we start with a session (the session the Analyzer currently processes) as seen in Figure 3.3 and extract the IP address. Then all the sessions prior to the session in work that are DNS requests and are in the time scope defined by the compile options are extracted. We extract more than the most recent request to get all hostnames associated with the IP address. This is necessary because A records are a many-to-one mapping meaning that multiple hostnames can point to the same IP address. Which means that we cannot determine which DNS request provided the IP address for a specific session since the device can cache DNS responses and reuse older requests. This list of DNS requests is additionally filtered for the IP address the session in work had to limit the list to the relevant requests. The hostnames extracted from the sessions are resulting in a list of hostnames for each session in work. Further, we only consider requests from the same device, which is identified by the VLAN, to abide the device separation.

To get the tags we have to extract the tagpatterns from the custom index "tag-patterns" described in section 2.4. Each tagpattern contains a regular expression and a list of corresponding tags. Each hostname is then matched against each tagpattern. If the tagpattern matches, the corresponding tags are stored in a list. Those tags are stored distinctively so that duplicate tags in the tag-patterns are not a problem. We are aware that this procedure results in a bad runtime performance, but it is very stable and guarantees that all the tags are applied. This function is implemented as a custom rule in the file rules.h namely "rule_dns". If the argument "hosts" is set, all the found hostnames are stored in the field "FoundHosts". That makes it easier to identify missing tag patterns and false positives.

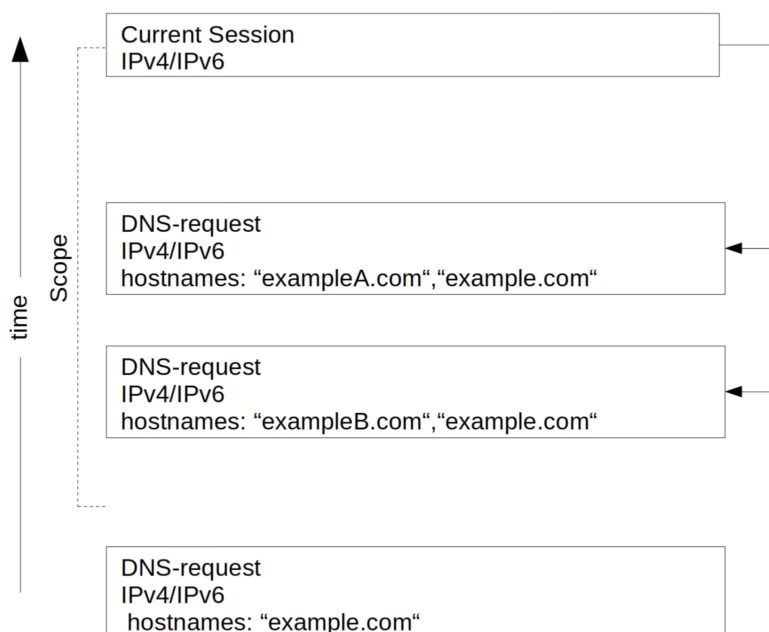


Figure 3.3: Hostname extraction

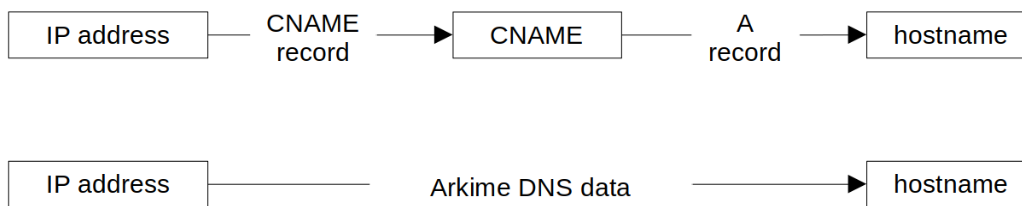


Figure 3.4: DNS resolution with intermediate steps

In the resolution of hostnames it can happen that a CNAME record is returned which points to another record (see Figure 3.4). This would mean for the Analyzer that the reverse DNS resolution would have to be performed in a cascade until an A record was found resolving the IP address to a hostname. We saw that we only have DNS requests that span over a single session which means that we only have to search for stored DNS data and can treat them as A records, mapping hostnames directly with IP addresses. It is possible that DNS queries span over multiple sessions, creating DNS sessions in Arkime that do not have the expected format. There are further exceptions like multicast DNS (mDNS) records which are not supported at the current version and DNS requests with no IP stored. This can happen if the query type has no IP like HINFO, SOA and NS, or the query was refused indicated with the status code REFUSED. We mention those cases which obviously have no IP address because they are stored the same way as normal DNS request are, which could lead to problems if rules access the JSON data of the session directly. The DNS data stored by Arkime contains the hostnames and, if present, the CNAMEs of requests. If such a case where a DNS request has no IP address stored is found the session is marked with the tag “error-DNS-noIp”. All of those entries are used by the Analyzer and matched against all tagpatterns present in the index “tagpatterns”.

The static scope, meaning the timespan in which DNS requests are searched is the same for all requests, of this solution proves to be a problem because with only the session in work we do not know when the DNS request was sent and if the request was cached. The problem is that we cannot consider all stored DNS requests because it would slow down the analysis too much, although it would be an option that would produce the most complete set of requests. If we only look at a short time span we could miss requests that are older. We had to make a compromise on the scope since we had to balance performance with completeness of the results and decided to set the time to look into the past to one hour. With that time frame most of the longer lived requests should be in the scope and analyzed.

The better solution would be a per-device DNS cache similar to the solution proposed by Woodruff et al. [12], which we did not implement due to time constraints. Such a cache would not only increase the performance of the Analyzer but also provide a complete list of DNS requests. The performance increase is achieved by replacing the search of older DNS requests by a simple query from the cache.

Since we use Elasticsearch for data storage it would make sense to implement such a cache as custom index. Cache entries would be documents which hold the DNS information like IP address, hostname, timestamp, and TTL. An implementation directly into the Analyzer does not work as long as the architecture is not changed to a non-terminating program flow.

The Analyzer would have to be extended with an update and fetch functionality

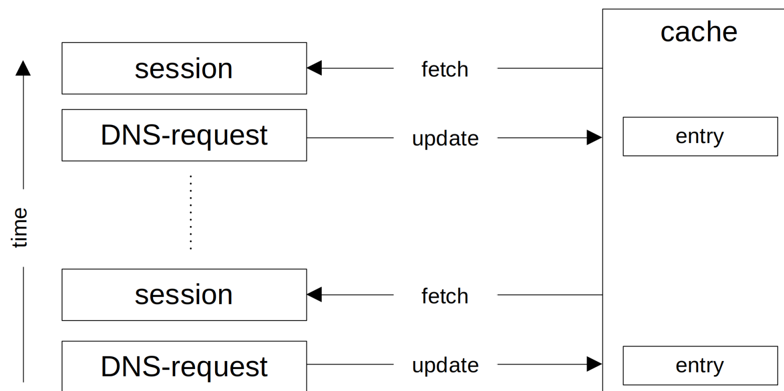


Figure 3.5: DNS device cache

for the cache. Where the update function stores the DNS request as a cache entry as the Analyzer progresses through the network traffic. When the Analyzer works on a session, the DNS data is not searched in the data traffic but fetched from the cache. This works as long as the traffic is progressed in order, from oldest to most recent session. Since the used Arkime version does not store the TTL from DNS requests it has to be extracted through a second channel when a cache update is performed which could lead to a problem if older sessions are analyzed where the DNS entries already changed in the time between the capture-time and analysis-time. Since there are no guidelines on how to set the TTL this is especially a troubling problem [8]. This could be solved with a caching solution proposed by Woodruff et al. [12] directly storing the DNS data when queries are processed, instead of relying on post-processing of data stored by Arkime.

3.7 Runtime Options

The runtime options have to be set for each call of the Analyzer. The options are checked automatically for syntax and use errors with the gnu lib module argp[3]. The program expects some arguments and terminates if no arguments are set. This behavior should prevent an execution with the wrong scope set because of default values. An example execution would be as follows:

Listing 3.7: Example Analyzer call

```
./Analyzer -o --hosts --id=210823-MBplZzu0XOxHtbAUOmkk4BN3
```

All options, listed in Table 3.2, have a long and a short form that set the same option. We designed this part to behave like other standard Linux programs. Some listed options override the default values of compile-time options.

[3]https://www.gnu.org/software/libc/manual/html_node/Argp.html

Table 3.2: Runtime options

Long form	Short	Argument	Description
--help	-h	—	print help
--all	-a	—	set scope to all sessions
--cutoff	-c	time in hours	set scope to a block of sessions
--id	-i	id of session	set scope to one single session
--hosts	-h	—	store the found hosts
--verbose	-v	—	print additional information
--nopcpap	-p	—	Activate the nopcap mode, in which the pcap files will not be extracted and not set in the session argument used by rules
--esurl	-e	URL to Elastic-search db	Overrides the default value of ES_URL
--pprefix	-P	prefix	Overrides the default value of PCAPPREFIX
--ignoreversion	-I	—	Disables the rule version check

- The help option prints information on the usage of the different options like in other Linux command line programs.
- The all option sets the scope for this run to the scope is every session found in the database.
- The cutoff option sets the scope for this run to the scope is a block of sessions between the time of the call and the given parameter in hours. This option needs an argument like --cutoff=1 for the last hour.
- The id option sets the scope for this run to the scope is a single session. For example --id=210823-MBplZzu0XOxHtbAUOmkk4BN3.
- The hosts option is a flag that determines if the found hosts for the DNS tagging should be stored in the Elasticsearch db.
- The verbose option is a flag that determines if additional information should be printed on the command line. This option is primarily for debugging and should only be used in combination in the id-scope since printing to the terminal is slow for a larger scope.
- The esurl option overrides the value set by the compile option ES_URL.
- The pprefix option overrides the values set by the compile option PCAPPREFIX used for communication with Elasticsearch.
- The nopcap option is a flag that sets the Analyzer into the nopcap mode in which no pcap files are read and sessions are not filled with packets.
- The ignoreversion option is a flag that disables the rule version check for sessions.

3.8 Compile-time Options

The compile-time options listed in Table 3.3 are located in the module glbs and have to be set before compiling the program. These options should not change frequently since most of the options determine the interaction with Elasticsearch, which is important if the Arkime version changes.

Table 3.3: Compile-time options

Option	Type	Description
DEBUG	Bool	Print additional debug information
PROG_VERSION	String	Version to store in database
RULEVERSION	Integer	Version of the rules used for preventing double analysis of sessions
MAXUPDATELENGTH	Integer	Defines the scope for update-tagging
MAXDNSTIME	Integer	Defines the scope for dns-tagging
ES_URL	String	URL to Elasticsearch db
ES_DOC	String	Document endpoint from Elasticsearch
ES_INDEXPREFIX	String	Prefix used from Arkime for sessions ids
ES_FILEINDEX	String	Name of the index where Arkime stores the pcap files
ES_TIMESTAMP	String	Name of the Timestamp field used by Arkime
ES_UPDATE	String	Update endpoint
ES_SEARCH	String	Search endpoint
ES_ALL	String	All endpoint
ES_TAGPATTERNS	String	Name of the custom index where the tagpatterns are stored
ES_UPDATES	String	Name of the custom index where the update data is stored
ES_INDEXCHECK_ERROR	String	String for Index check
ES_INDEXCHECK_OK	String	String for Index check
PCAPPREFIX	String	Prefix used by Arkime for pcap filenames
PCAPFOLDER	String	Path to the folder where Arkime stores the pcap files

- DEBUG: Set this to true if more detailed debug information should be printed.
- PROG_VERSION: Simply a string used for indicating with which version a session was analyzed.
- RULEVERSION: Set the version of the current rule set. This option is used by the Analyzer to prevent a double analyzation of the same session.
- MAXUPDATELENGTH & MAXDNSTIME: Set the scope of the update- and DNS-tagging respectively. The time is set in milliseconds.

- ES_*: All options with the prefix "ES_" configure parts of the curl commands used by the Analyzer. Set this options according to the Elasticsearch mapping and overall configuration.
- PCAPPREFIX & PCAPFOLDER: Set the prefix used by Arkime for naming the pcap files and the folder the files are stored in. Extract this information from Arkime and set it accordingly.

3.9 Status Indication

The Analyser uses four different information messages which are printed to a stream defined in the module glbs. Per default, we use stdout.

- OK: Indicating that a step is successfully completed.
- INFO: A marker for general information on the current instance of the Analyzer.
- WARNING: A warning means that something went wrong, but the program can still continue. A common example is when packets in the pcap files are missing. Sometimes warnings can be fixed with a second execution at a later time so possible missing data is written by Arkime.
- ERROR: An error indicates a problem that cannot be solved automatically. The program terminates in this case.

A successful execution prints the following lines indicating that everything went fine.

Listing 3.8: Analyzer Status Output

```
1 [  OK   ] arguments parsed succsessful
2 [  OK   ] dependencies checked
3 [  OK   ] extracting of matchdata succsessful
4 [  INFO ] execution time: 19.00 seconds
```

3.10 Dependencies

This program has the two custom indices updates and tagpatterns described in section 2.3 and section 2.4 respectively as dependencies. Further there are two libraries needed for compiling the program. Namely, libcurl[4] for the Elasticsearch interactions and libjson[5] for parsing JSON objects. Further an Elasticsearch database with the Arkime indices is necessary.

[4] https://curl.se/libcurl/
[5] https://json-c.github.io/json-c/json-c-current-release/doc/html/index.html

Chapter 4

Future Work

4.1 Improvements of Core Principles

Update detection The update detection is at this stage of development not automated in the sense that updates are detected without the need of human work. The approaches of this work were all static solutions that have significant drawbacks, especially the long term maintenance of static rules and host lists pose a significant problem. To solve that problem of static solutions we propose a machine learning approach with a supervised machine learning algorithm. Of course there is the open question, if the available data provides enough information for the algorithm to work. Since most of the traffic is encrypted, basically only metadata is usable as input. The first step to this approach would be to acquire a sufficiently large training set. In this case that means using the implemented solution to tag sessions.

DNS cache As mentioned in section 3.6 a per-device cache would be a good improvement in terms of runtime, and quality of the used data. This cache could be implemented as an update to the Analyzer or as a stand-alone project.

4.2 Miscellaneous Improvements

The following ideas improve the overall usability of the Analyzer but don't change the core principles on which the solution builds upon:

Dynamic rules At the current version, rules are compiled into the Analyzer. It could make sense to separate the logic of the Analyzer from the rules, so that changes to one do not affect the other. With that, one could introduce a scripting support to increase flexibility.

Yara support A yara rule support would enhance the usability because rules would not need to define a logic how to analyze a packet but only provide a rule to apply. That would not only enhance the readability but also reduce the probability of errors. There is already a project that could be used[1].

JSON mapping Arkime versions change the mapping in the Elasticsearch database for sessions and the naming scheme of indices. Since the Analyzer works directly on the database those changes lead to a crash. To prevent this, it would be useful to support a dynamic mapping or option files where the different names and paths can be set without compiling the Analyzer.

[1] https://yara.readthedocs.io/en/v3.4.0/capi.html

References

[1] Arkime. 2022. *Arkime FAQ*. https://arkime.com/faq#data-never-gets-deleted.

[2] Jeffrey Erman, Martin Arlitt, and Anirban Mahanti. 2006. Traffic Classification Using Clustering Algorithms. In *Proceedings of the 2006 SIGCOMM Workshop on Mining Network Data* (*MineNet '06*). ACM, Pisa, Italy, pp. 281–286. doi: 10.1145/1162678.1162679.

[3] Paul E. Hoffman and Patrick McManus. 2018. DNS Queries over HTTPS (DoH). RFC 8484. (October 2018). doi: 10.17487/RFC8484.

[4] Zi Hu, Liang Zhu, John Heidemann, Allison Mankin, Duane Wessels, and Paul E. Hoffman. 2016. Specification for DNS over Transport Layer Security (TLS). RFC 7858. (May 2016). doi: 10.17487/RFC7858.

[5] Jason Kim, Hyojoon Kim, and Jennifer Rexford. 2021. Analyzing Traffic by Domain Name in the Data Plane. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*. ACM, New York, NY, USA, pp. 1–12. doi: 10.1145/3482898.3483357.

[6] Yingqiu Liu, Wei Li, and Yunchun Li. 2007. Network Traffic Classification Using K-means Clustering. In *Second International Multi-Symposiums on Computer and Computational Sciences (IMSCCS 2007)*, pp. 360–365. doi: 10.1109/IMSCCS.2007.52.

[7] Paul V. Mockapetris. 1987. Domain names - concepts and facilities. RFC 1034. (November 1987). doi: 10.17487/RFC1034.

[8] Giovane C. M. Moura, John Heidemann, Ricardo de O. Schmidt, and Wes Hardaker. 2019. Cache Me If You Can: Effects of DNS Time-to-Live. In *Proceedings of the Internet Measurement Conference* (*IMC '19*). ACM, Amsterdam, Netherlands, pp. 101–115. doi: 10.1145/3355369.3355568.

[9] Said Jawad Saidi, Anna Maria Mandalari, Roman Kolcun, Hamed Haddadi, Daniel J. Dubois, David Choffnes, Georgios Smaragdakis, and Anja Feldmann. 2020. A Haystack Full of Needles: Scalable Detection of IoT Devices in the Wild. In *Proceedings of the ACM Internet Measurement Conference* (*IMC '20*). ACM, Virtual Event, USA, pp. 87–100. doi: 10.1145/3419394.3423650.

[10] Sadegh Torabi, Amine Boukhtouta, Chadi Assi, and Mourad Debbabi. 2018. Detecting Internet Abuse by Analyzing Passive DNS Traffic: A Survey of Implemented Systems. *IEEE Communications Surveys Tutorials*, 20, 4, 3389–3415. doi: 10.1109/COMST.2018.2849614.

[11] Dmitrii Vekshin, Karel Hynek, and Tomas Cejka. 2020. DoH Insight: Detecting DNS over HTTPS by Machine Learning. In *Proceedings of the 15th International Conference on Availability, Reliability and Security* (*ARES '20*). ACM, Virtual Event, Ireland, 8 pages. doi: 10.1145/3407023.3409192.

[12] Jackson Woodruff, Murali Ramanujam, and Noa Zilberman. 2019. P4DNS: In-Network DNS. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 1–6. doi: 10.1109/ANCS.2019.8901896.

[13] Mehedee Zaman, Tazrian Siddiqui, Mohammad Rakib Amin, and Md. Shohrab Hossain. 2015. Malware detection in Android by network traffic analysis. In *2015 International Conference on Networking Systems and Security (NSysS)*, pp. 1–5. doi: 10.1109/NSysS.2015.7043530.