HAL
archives-ouvertes.fr

Advanced Network Fingerprinting

Humberto Abdelnur, Radu State, Olivier Festor

▶ **To cite this version:**

Humberto Abdelnur, Radu State, Olivier Festor. Advanced Network Fingerprinting. Ari Trachtenberg. Recent Advances in Intrusion Detection, Sep 2008, Boston, United States. Springer Berlin / Heidelberg, Volume 5230/2008, pp.372-389, 2008, Computer Science. <10.1007/978-3-540-87403-4>. <inria-00326054>

HAL Id: inria-00326054
https://hal.inria.fr/inria-00326054

Submitted on 1 Oct 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Advanced Network Fingerprinting

Humberto J. Abdelnur, Radu State, and Olivier Festor

Centre de Recherche INRIA Nancy - Grand Est
615, rue du jardin botanique
Villers-les-Nancy, France
<firstname.lastname>@loria.fr
http://madynes.loria.fr

Abstract. Security assessment tasks and intrusion detection systems do rely on automated fingerprinting of devices and services. Most current fingerprinting approaches use a signature matching scheme, where a set of signatures are compared with traffic issued by an unknown entity. The entity is identified by finding the closest match with the stored signatures. These fingerprinting signatures are found mostly manually, requiring a laborious activity and needing advanced domain specific expertise. In this paper we describe a novel approach to automate this process and build flexible and efficient fingerprinting systems able to identify the source entity of messages in the network. We follow a passive approach without need to interact with the tested device. Application level traffic is captured passively and inherent structural features are used for the classification process. We describe and assess a new technique for the automated extraction of protocol fingerprints based on arborescent features extracted from the underlying grammar. We have successfully applied our technique to the Session Initiation Protocol (SIP) used in Voice over IP signalling.

Key words: Passive Fingerprinting, Feature extraction, Structural syntax inference

1 Introduction

Many security operations rely on the precise identification of a remote device or a subset of it (e.g. network protocol stacks, services). In security assessment tasks, this fingerprinting step is essential for evaluating the security of a remote and unknown system; especially network intrusion detection systems might use this knowledge to detect rogue systems and stealth intruders. Another important applicability resides in blackbox devices/application testing for potential copyright infringements. In the latter case, when no access to source code is provided, the only hints that might detect a copyright infringement can be obtained by observing the network level traces and determine if a given copyright protected software/source code is used unlawfully.

The work described in this paper was motivated by one major challenge that we had to face when building a Voice over IP (VoIP) specific intrusion

detection system. We had to fingerprint VoIP devices and stacks in order to detect the presence of a rogue system on the network. Typically, only some vendor specific devices should be able to connect, while others and potentially malicious intended systems had to be detected and blocked. We decided that an automated system, capable to self-tune and self-deploy was the only viable solution on the long run. Therefore, we considered that the ideal system has to be able to process captured and labeled network traffic and detect the structural features that serve as potential differentiators. When searching for such potential features, there are some natural candidates: the type of individual fields and their length or the order in which they appear. For instance, the presence of login headers, the quantities of spaces after commas or the order presented in the handshake of capabilities. Most existing systems use such features, but individual signatures are built manually requiring a tedious and time consuming process.

Our approach consists in an automated solution for this task, assuming a known syntax specification of the protocol data units. We have considered only the signalling traffic - all devices were using Session Initiation Protocol [1] (SIP) - and our key contribution is to differentiate stack implementations by looking at some specific patterns in how the message processing code has been designed and developed. This is done in two main steps. In the first step, we extract features that can serve to differentiate among several stack implementations. These features are used in a second phase in order to implement a decisional process. This approach and the supporting algorithms are presented in this paper.

This paper is organized as follow. Section 2 illustrates the overall architecture and operational framework of our fingerprinting system. Section 3 shows how structural inference, comparison and identification of differences can be done based on the underlying grammar of a given specified protocol. Section 4 introduces the training, calibration and classification process. We provide an overview of experimental results in Sect.5 using the signalling protocol (SIP) as an application case. Section 6 describes the related work in the area of fingerprinting as well as the more general work on structural similarity. Finally, Sect.7 points out future works and concludes this paper.

2 Structural Protocol Fingerprinting

Most known application level and network protocols use a syntax specification based on formal grammars. The essential issue is that each individual message can be represented by a tree like structure. We have observed that stack implementers can be tracked by some specific subtrees and/or collection of subtrees appearing in the parse trees. The key idea is that structural differences between two devices can be detected by comparing the underlying parse trees generated for several messages. A structural signature is given by features that are extracted from these tree structures. Such distinctive features are called fingerprints. We will address in the following the automated identification of them.

If we focus for the moment one individual productions (in a grammar rule), the types of signatures might be given by:

– Different **contents** for one field. This is in fact a sequence of characters which can determinate a signature. (e.g. a prompt or an initialization message).
– Different **lengths** for one field. The grammar allows the production of a repetition of items (e.g. quantity of spaces after a symbol, capabilities supported). In this case, the length of the field is a good signature candidate.
– Different **orders** in one field. This is possible, when no explicit order is specified in a set of items. A typical case is how capabilities are advertised in current protocols.

We propose a learning method to automatically identify distinctive structural signatures. This is done by analyzing and comparing captured messages traces from the different devices. The overview of the learning and classification process is illustrated in Fig.1.

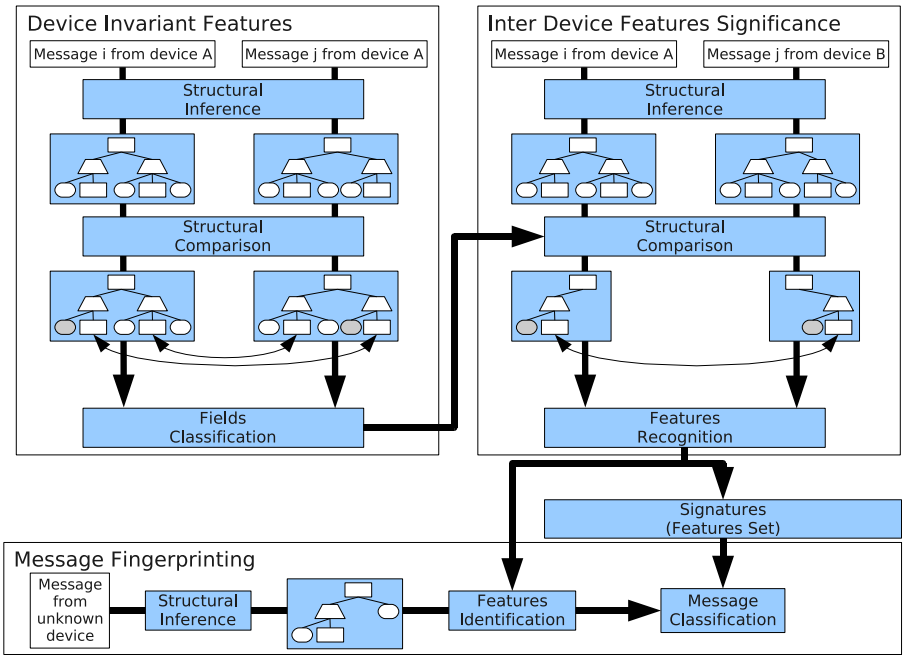


Fig. 1. Fingerprinting training and classification

The upper boxes in Fig.1 constitute the training period of the system. The output is a set of signatures for each device presented in the training set. The lowest box represents the fingerprinting process. The training is divided in two phases:

Phase 1 (*Device Invariant Features*). In this phase, the system automatically classifies each field in the grammar. This classification is needed to

identify which fields may change between messages coming from the same device.
Phase 2 (*Inter Device Features Significance*) identifies among the Invariant fields of each implementation, those having different values for at least two group of devices. These fields will constitute part of the signatures set.

When one message has to be classified, the values of each invariant field are extracted and compared to the signature values learned in the training phase.

3 Structural Inference

3.1 Formal Grammars and Protocol Fingerprinting

The key assumption made in our approach is that an Augmented BackusNaur Form (ABNF) grammar [2] specification is a priori known for a given protocol. Such a specification is made of some basic elements as shown in Fig.2.

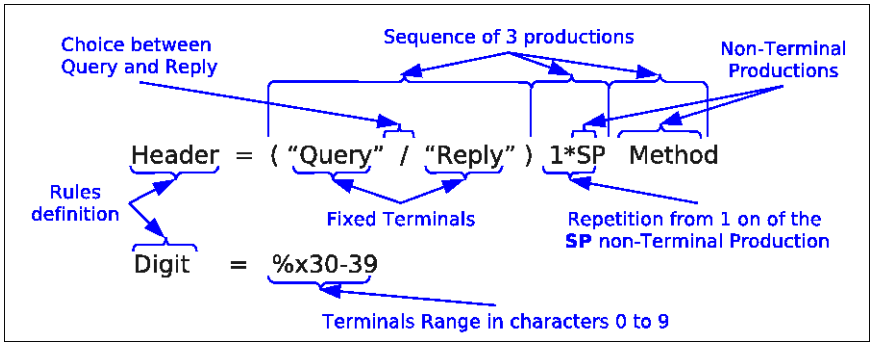


Fig. 2. Basic elements of a grammar

- A **Terminal** can represent a fixed string or a character to be chosen from a range of legitimate characters.
- A **Non-Terminal** is reduced using some rules to either a Terminal or a Non-Terminal.
- A **Choice** defines an arbitrary selection among several items.
- A **Sequence** involves a fixed number of items, where the order is specified.
- A **Repetition** involves a sequence of one item/group of items, where some additional constraints might be specified.

A given message is parsed according to the fields defined in the grammar. Each element of the grammar is placed in an n-ary tree which obeys the following rules:

- A **Terminal** becomes a leaf node with a name associated (i.e. the terminal that it represents) which is associated to the encountered value in the message.
- A **Non-Terminal** is an internal node associated to a name (i.e. the non-terminal rule) and it has a unique child which can be any of the types defined here (e.g. Terminal, non-Terminal, Sequence or Repetition).
- A **Sequence** is an internal node that has a fixed number of children. This number is in-line with the rules of the syntax specification.
- A **Repetition** is also an internal node, but having a number of children that may vary based on the number of items which appear in the message.
- A **Choice** does not create any node in the tree. However, it just marks the node that has been elected from a choice item.

It is important to note that even if sequences and repetitions do not have a defined name in the grammar rules, an implicit name is assigned to them that uniquely distinguishes each instance of these items at the current rule.

Figure 3 shows a Toy ABNF grammar defined in (a), messages from different implementation compliant with the grammar in (b/c) and (d) the inferred structure representing one of the messages in (d).

With respect to the usage, fields can be classified in three categories:

- **Cosmetics Fields**: these fields are mandatory and do not really provide a value added interest for fingerprinting purposes. The associated values do not change in different implementations.
- **Static Fields**: are the fields which values never change in a same implementation. These values do however change between different implementations. Obviously, these are the type of fields which may represent a signature for one implementation.
- **Dynamic Fields**: these fields are the opposite of static fields and do change their values in relation to semantic aspects of the message even in a single implementation.

An additional sub-classification can be defined for Dynamic and Static fields:

- **Value Type** relates to the String reduction of the node (i.e. the text information of that node),
- **Choice Type** relates to the selected choice from the grammar,
- **Length Type** corresponds to the number of items in a Repetition reduction,
- **Order Type** corresponds to the order in which items of a Repetition reduction appear.

Even if one implementation may generate different kind of values for the same field, such values could be related by a function and then serve as a feature. Therefore, a **Function Type** can be also defined to be used to compute the value from a node of the tree and return an output useful for the fingerprinting. Essentially, this type is used for manually tuning the training process.

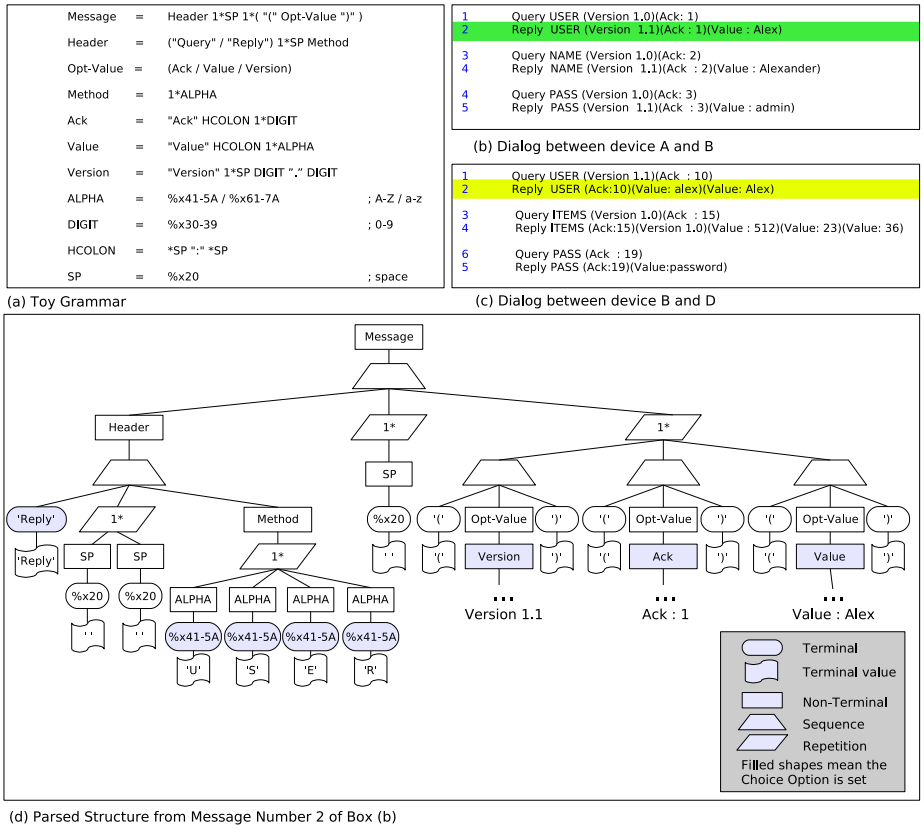


Fig. 3. Parsed Structure Grammar

3.2 Node Signatures and Resemblance

Guidelines for designing a set of tree signatures (for a tree or a sub-tree) should follow some general common sense principles like:

- As more items are shared between trees, the more similar their signatures must be.
- Nodes that have different tags or ancestors must be considered different.
- In cases where the parent node is a Sequence, the location order in the Sequence should be part of the tree signature.
- If the parent node is a repetition, the location order should not be part of the tree signature, order will be captured later on in the fingerprinting features.

The closest known approach is published by D. Buttler in [3]. This method starts by encoding the tree in a set. Each element in the set represents a partial path from the root to any of the nodes in the tree. A resemblance method defined by A. Broder [4] uses the elements of the set as tokens. This resemblance is based on shingles, where a shingle is a contiguous sequence of tokens from the document. Between documents D_i and D_j the resemblance is defined as:

$$r(D_i, D_j) = \frac{|S(D_i, w) \bigcap S(D_j, w)|}{|S(D_i, w) \bigcup S(D_j, w)|} \tag{1}$$

where $S(D_i, w)$ creates the shingles of length w for the document D_i.

Definition 1. *The **Node Signature** function is defined to be a Multi-Set of all partial paths belonging to the sub-branch of the node.*

The *partial paths* start from the current node rather than from the root of the tree, but still goes through all the nodes of the subtree which has the current element as root like it was in the original approach. However, partial paths obtained from fields classified as *Cosmetics* are excluded from this Multi-Set. The structure used is a Multi-Set rather than a Set in order to store the quantity of occurrences for specific nodes in the sub-branch. For instance, the number of spaces after a specific field can determinate a signature in an implementation.

Siblings nodes in a Sequence items are fixed and representative. Sibling nodes in a Repetition can be made representative creating the partial paths of the Multi-Set and using the respective position of a child.

Table 3.2 shows the *Node Signature* obtained from the node *Header* at the tree of Fig. 3 (d).

Definition 2. *The **Ressemblance** function used to measure the degree of similarity between two nodes is based on the (1). The $S(N_i, w)$ function applies the Node Signature function over the node N_i.*

Using $w = 1$ allows to compare the number of items these nodes have in common though ignoring their position for a repetition.

Partial Paths	Occurrences
Header.0.'Reply'	1
~~Header.0.'Reply'.'Reply'~~	1
Header.1.?	2
~~Header.1.?.SP~~	2
~~Header.1.?.SP.%x20~~	2
~~Header.1.?.SP.%x20.' '~~	2
Header.2.Method.?	4
~~Header.2.Method.?.ALPHA.~~	4
Header.2.Method.?.ALPHA.%x41-5A	4
Header.2.Method.?.ALPHA.%x41-5A.'U'	1
Header.2.Method.?.ALPHA.%x41-5A.'S'	1
Header.2.Method.?.ALPHA.%x41-5A.'E'	1
Header.2.Method.?.ALPHA.%x41-5A.'R'	1

(~~strikethrough~~) Strikethrough paths are the ones considered as cosmetics.
(?) Quotes define that the current path may be any of the repetition items.

Table 1. Partial paths obtained from Fig.3 (d)

Algorithm 1 Node differences Location

procedure NODEDIFF($node_a, node_b$)
  if $Tag(node_a) = Tag(node_b)$ **then**
    if $Type(node_a) = TERMINAL$ **then**
      if $Value(node_a)\ !=\ Value(node_b)$ **then**
        $Report\_Difference('Value', node_a, node_b)$
      end if
    else if $Type(node_a) = NON-TERMINAL$ **then**
      $NODEDIFF(node_a.child_0, node_b.child_0))$ ▷Non_Terminals have ▷an unique child
    else if $Type(node_a) = SEQUENCE$ **then**
      for $i = 1..\#node_a$ **do** ▷In a Sequence
        $NODEDIFF(node_a.child_i, node_b.child_i)$ ▷$\#node_a = \#node_b$
      end for
    else if $Type(node_a) = REPETITION$ **then**
      if not $(\#node_a = \#node_b)$ **then**
        $Report\_Difference('Length', node_a, node_b)$
      end if
      $matches := Identify\_Children\_Matches(node_a, node_b)$
      if $\exists\ (i,j) \in matches :\ i\ !=\ j$ **then**
        $Report\_Difference('Order', node_a, node_b)$
      end if
      forall $(i,j) \in matches$ **do**
        $NODEDIFF(node_a.child_i, node_b.child_j)$
      end for
    end if
  else
    $Report\_Difference('Choice', node_a, node_b)$
  end if
end procedure

3.3 Structural Difference Identification

Algorithm 1 is used to identify differences between two nodes which share the same ancestor path in the two trees,

where the functions *Tag*, *Value*, *Type* return the name, value and respectively the type of the current node. Note that $Tag(node_a) = Tag(node_b) \Rightarrow Type(node_a) = Type(node_b)$.

The function **Report_Difference** takes the type of difference to report and the corresponding two nodes. Each time the function is called, it creates one structure that stores the type of difference, the partial path from the root of the tree to the current nodes (which is the same for both nodes) and a corresponding value. For differences of type 'Value' it will store the two terminal values, for 'Choice' the two different Tags names, for 'Length' the two lengths and for 'Order' the matches.

The function **Identify_Children_Matches** identifies a match between children of different repetition nodes. The similitude between each child from $node_a$ and $node_b$ (with n and m children respectively) is represented as a matrix, M, of size n x m where:

$$M_{i,j} = resemblance(node_a.child_i, node_b.child_j)$$

To find the most adequate match, a greedy matching assignment based on the concept of Nash Equilibrium [5] is used. Children with the biggest similarity are bound. If a child from $node_a$ shares the same similarity score with more than one child from $node_b$, some considerations have to be added respecting their position in the repetitions.

Figure 4 illustrates an example match, assuming that the following matrix was obtained using the *Resemblance* method with the path "`Message.2.?`". The rows in the matrix represent the children from the subtree in (a) and the columns the children from subtree (b).

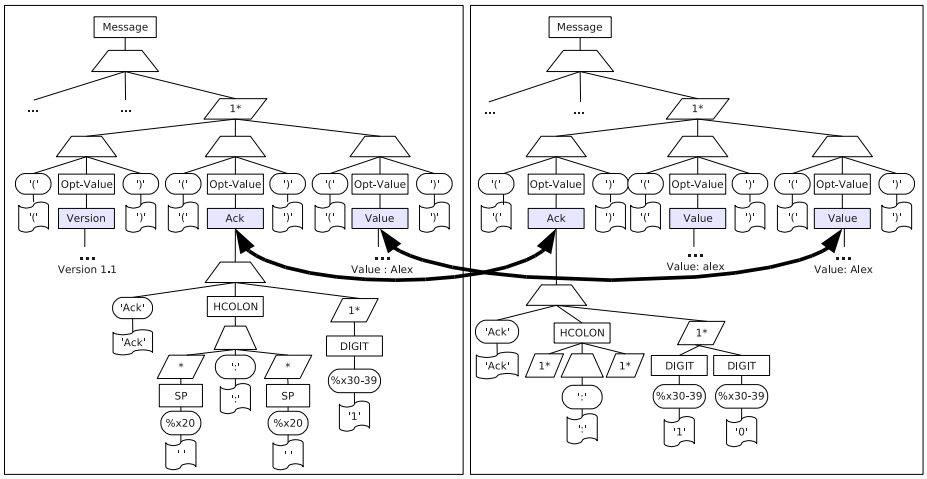
$$M = \begin{pmatrix} .00 & .00 & .00 \\ .33 & .00 & .00 \\ .00 & .61 & .90 \end{pmatrix}$$

All the compared children share some common items besides the choice nodes (colored). Those common items are *Cosmetics* nodes, which are required in the message in order to be compliant with the grammar. Note that, besides the *Cosmetic* fields, the first item of the subtree (a) does not share any similarity with any of the other nodes. It should therefore not match any other node.

4 Structural Features Extraction

4.1 Fields Classification

One major activity that was not yet described is how non-invariant fields are identified. The process is done by using all the messages coming from one device



(a) Message Number 2 of Figure 1 (b)

(b) Message Number 2 of Figure 1 (c)

Partial Path	Type	$Node_a$	$Node_b$
ACK.1.HCOLON.0	Static Length	1	0
ACK.1.HCOLON.2	Static Length	1	0
ACK.2	Static Length	1	2
ACK.2.(?).DIGIT.%x30-39	Static Value	'1'	'1'
ACK.2.(?).DIGIT.%x30-39	Static Value		'0'

(c) Node Signatures from the **Ack** nodes

$S_A \cup S_B$	$S_A \cap S_B$	$\frac{S_A \cap S_B}{S_A \cup S_B}$
1	0	0
1	0	0
2	1	0.5
1	1	1
1	0	0
6	2	0.33

(d) Shingle information from the **Ack** nodes where $S_a = S(N_{a'}1)$ and $S_b = (N_{b'}1)$

Fig. 4. Performed match between sub-branches of the tree

and finding the differences between each two messages using Algorithm 1. For each result, a secondary algorithm (Algorithm 2) is run in order to fine tune the extracted classification.

Algorithm 2 Fields Classification Algorithm

```
procedure FieldClassification(differences_{a,b})
    forall diff ∈ differences_{a,b} do
        if diff.type ==' Value') then
            Classify_as_Dynamic('Value', diff.path)
        else if diff.type ==' Choice' then
            Classify_as_Dynamic('Choice', diff.path)
        else if diff.type ==' Length' then
            Classify_as_Dynamic('Length', diff.path)
        else if diff.type ==' Order' then
            if not (∀ (i, j), (x, z) ∈ diff.matches :
                    (i < x ∧ j < z) ∨ (i > x ∧ j > z)) then
                ▷Check if a permutation exists between the matched items.
                Classify_as_Dynamic('Order', diff.path)
            end if
        end if
    end for
end procedure
```

The **Classify_as_Dynamic** functions store in the global list, **fieldClassifications**, a tuple with the type of the found difference (e.g. 'Value', 'Choice', 'Length' or 'Order') and the partial path in the tree structure that represents the node in the message.

This algorithm recognizes only the fields that are *Dynamic*. The set of *Static* fields will be represented by the union of all the fields not recognized as *Dynamic*.

Assuming a training set *Msg_set*, of messages compliant with the grammar as

$$Msg = \bigcup_{i=0}^{n} msg\_set_i$$

where n is quantity of devices and msg_set_i is the set of messages generated by device i, the total number of comparisons computed in this process is

$$cmps_1 = \sum_{i=0}^{n} \frac{|msg\_set_i| * (|msg\_set_i| - 1)}{2} \tag{2}$$

4.2 Features Recognition

Some features are essential for an inter-device classification. In contrast to the Fields Classification, this process compares all the messages from the training set sourced from different devices. All the *Invariant* Fields -for which different

implementations have different values- are identified. Algorithm 3 recognizes these features. Its inputs are the *fieldClassifications* computed by the Algorithm 2, the Devices Identifier to which the compared message belongs as well as the set of differences found by Algorithm 1 between the messages.

Algorithm 3 Features Recognition Algorithm

procedure featuresRecognition($fieldClassifications,\ DevID_{a,b},\ differences_{a,b}$)
  forall $diff \in differences_{a,b}$ **do**
    if not $(diff.type, diff.path) \in fieldClassifications$ **then**
      if $diff.type == 'Value'$ **then**
        $addFeature('Value',\ diff.path,\ DevID_{a,b},\ diff.value_{a,b})$
      else if $diff.type == 'Choice'$ **then**
        $addFeature('Choice',\ diff.path,\ DevID_{a,b},\ diff.name_{a,b})$
      else if $diff.type == 'Length'$ **then**
        $addFeature('Length',\ diff.path,\ DevID_{a,b},\ diff.length_{a,b})$
      else if $diff.type == 'Order'$ **then**
        if $(\exists\ (x, z) \in diff.matches : x \neq z)$ **then**
          $addFeature('Order',\ diff.path,\ DevID_{a,b},$
            $diff.match,\ diff.children\_nodes_{a,b})$
        end if
      end if
    end if
  end for
end procedure

The **add_Feature** function stores in a global variable, **recognizedFeatures**, the partial path of the node associated with the type of difference (i.e. Value, Name, Order or Length) and a list of devices with their encountered value. However, the 'Order' feature presents a more complex approach, requiring minor improvements.

Assuming the earlier Msg_set set, this process will do the following number of comparisons:

$$cmps_2 = \sum_{i=0}^{n} |msg\_set_i| * \sum_{j=i+1}^{n} |msg\_set_j| \tag{3}$$

From the *recognizedFeatures* only the *Static* fields are used. The recognized features define a sequence of items, where each one represents the field location path in the tree representation and a list of Device ID with their associated value.

The Recognized Features can be classified in:

- Features that were found with each device and at least two distinct values are observed for a pair of devices,

– Features that were found in some of the devices for which such a location path does not exists in messages from other implementations.

4.3 Fingerprinting

The classification of a message uses the tree structure representation introduced in section 3.1. The set of recognized features obtained in section 4 represents all the partial paths in a tree structure that are used for the classification process.

In some cases, the features are of type *'Value'*, *'Choice'* or *'Length'*. Their corresponding value is easily obtained. However, the case of an *'Order'* represents a more complex approach, requiring some minor improvements

Figure 5 illustrates some identified features for an incoming message.

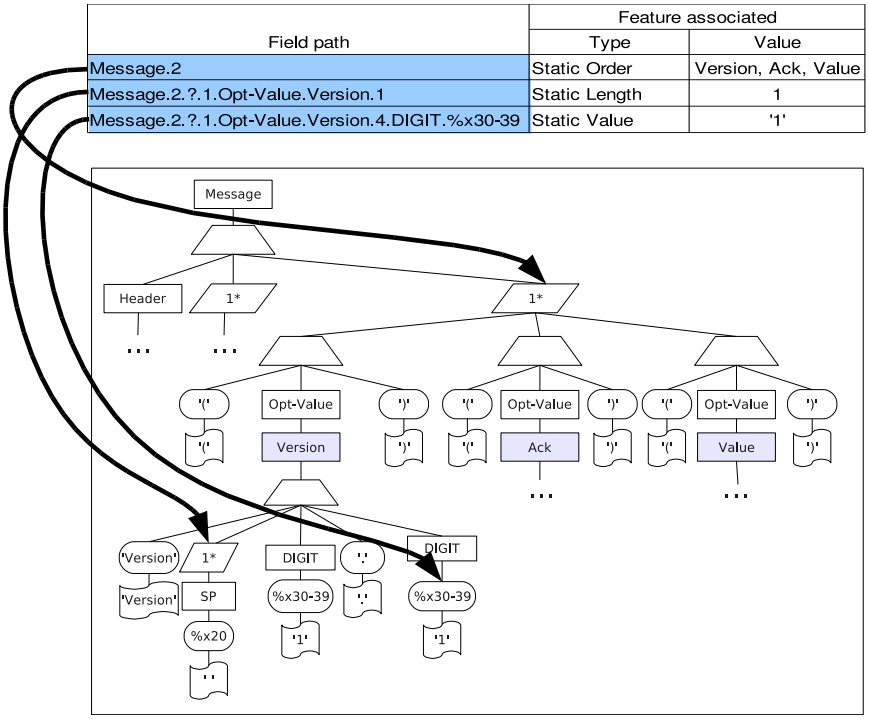


Fig. 5. Features Identification

Once a set of distinctive features is obtained, some well known classification techniques can be leveraged to implement a classifier. In our work, we have leveraged the machine learning technique described in [6].

5 Experimental Results

We have implemented the Fingerprinting Framework approach in Python. A scannerless Generalized Left-to-right Rightmost (GLR) derivation parser has been used (Dparser[7]) in order to solve ambiguities in the definition of the grammar. The training function could easily be parallelized.

We have instantiated the fingerprinting approach on the SIP protocol. The SIP messages are sent in clear text (ASCII) and their structure is inspired from HTTP. Several primitives - REGISTER, INVITE, CANCEL, BYE and OPTIONS - allow session management for a voice/multimedia call. Some additional extensions do also exist -INFO, NOTIFY, REFER, PRACK- which allow the support of presence management, customization, vendor extensions etc.

We have captured 21827 SIP messages from a real network, summarized in Table 2.

Device	**Software/Firmware version**
Asterisk	v1.4.4
Cisco CallManager	v5.1.1
Cisco 7940/7960	vP0S3-08-7-00
	vP0S3-08-8-00
Grandstream Budge Tone-200	v1.1.1.14
Linksys SPA941	v5.1.5
Thomson ST2030	v1.52.1
Thomson ST2020	v2.0.4.22
SJPhone	v1.60.289
	v1.60.320
	v1.65
Twinkle	v0.8.1
	v0.9
Snom	v5.3
Kapanga	v0.98
X-Lite	v3.0
Kphone	v4.2
3CX	v1.0
Express Talk	v2.02
Linphone	v1.5.0
Ekiga	v2.0.3

Table 2. Tested equipment

The system was trained with only 12% of the 21827 messages. These messages were randomly sampled. However, a proportion between the number of collected messages and the number used for the training was kept; they ranged from 50 to 350 messages per device. Table 3 shows the average and total time obtained for the comparisons of each training phase and for the message classification process (i.e. message fingerprinting). During both Phase 1 and 2, the comparisons were distributed over 10 computers ranging from Pentium IV to Core Duo. As it was expected, the average comparison time per message in Phase 2 was lower than in the previous phase, since only the invariant fields are compared. To evaluate the training, the system classified all the sampled messages (i.e. 21927 messages) in only in one computer (Core Duo @ 2.93GHz).

Type of Action	Average time per action	Number of actions computed	Total computed time
Msg. comparisons for Phase 1	632 milisec	296616	5 hours(1)
Msg. comparisons for Phase 2	592 milisec	3425740	56 hours(1)
Msg. classification	100 milisec.	21827	40 minutes

(1) Computed time using 10 computers.
Table 3. Performance results obtained with the system

172 features were discovered among all the different types of messages. These features represent items order, different lengths and values of fields where non protocol knowledge except its syntax grammar had been used. Between two different devices the distance of different features ranges between 26 to 95 features, where most of the lower values correspond to different versions of the same device. Usually, up to 46 features are identified in one message.

Table 4 summarizes the sensitivity, specificity and accuracy. The results were obtained using the test data set.

Classification	**True Positive**	**False Positive**	**Positive Predictive Value**
	18881	20	0.998
	False Negative	**True Negative**	**Negative Predictive Value**
	2909	N.A.	0.993
	Sensitivity	**Specificity**	**Accuracy**
	0.866	0.999	0.993

Table 4. Accuracy results obtained with the system

In this table we can observe that the results are very encouraging due to the high specificity and accuracy. However, some observations can be made about the quantity of false negatives. About 2/5 of them belong to only one implementation (percentage that represents 50% of its messages), 2/5 belongs to 3 more device classes (representing 18% of their messages), the final 1/5 belongs to 8 classes (representing 10% of their messages) and the 7 classes left do not have false negatives. This issue can be a consequence of the irregularity in the quantity from the set of messages in each device. Three of the higher mentioned classes had been used in our test-bed to acquire most features of SIP. A second explanation can be that in fact many of those messages do not contain valuable information (e.g. intermediary messages). Table 5 shows all the 38 types of messages collected in our test with information concerning their miss-classification (i.e. False Negatives).

Finally, we created a set of messages which have been manually modified. These modifications include changing the User-Agent, Server-Agent and references to device name. As a result, deleting a few such fields did not influence

Type of Message	False Negatives	Message quantity	Miss percentage
200, 100, ACK	1613 (710, 561, 347)	9358 (4663, 1802, 2893)	17% (15%, 31%, 11%)
501, 180, 101 BYE, 486	824 $\begin{pmatrix} 257, 215, 148 \\ 104, 100 \end{pmatrix}$	3414 $\begin{pmatrix} 385, 1841, 148 \\ 892, 176 \end{pmatrix}$	24% $\begin{pmatrix} 65\%, 11\%, 100\% \\ 11\%, 67\% \end{pmatrix}$
489, 487, 603 202, 480, 481 380, 415, 400	213 $\begin{pmatrix} 84, 57, 28 \\ 21, 13, 6 \\ 2, 1, 1 \end{pmatrix}$	636 $\begin{pmatrix} 84, 230, 118 \\ 52, 42, 18 \\ 2, 38, 51 \end{pmatrix}$	33% $\begin{pmatrix} 100\%, 24\%, 23\% \\ 40\%, 30\%, 33\% \\ 100\%, 2\%, 2\% \end{pmatrix}$
INVITE, OPTIONS REGISTER, CANCEL SUBSCRIBE	117 $\begin{pmatrix} 38, 34 \\ 25, 19 \\ 1 \end{pmatrix}$	5694 $\begin{pmatrix} 3037, 628 \\ 1323, 297 \\ 409 \end{pmatrix}$	2% $\begin{pmatrix} 1\%, 5\% \\ 1\%, 6\% \\ .00\% \end{pmatrix}$
INFO, REFER PRACK, NOTIFY PUBLISH	0	2223 $\begin{pmatrix} 1830, 163 \\ 117, 77 \\ 36 \end{pmatrix}$	0%
11 other Response Codes	0	492	0%

Table 5. False Negative classification details

the decision of the system; neither did it changing their banners to another implementation name. However, as more modifications were done, less precise the system became and more mistakes were done.

6 Related Work

Fingerprinting became a popular topic in the last few years. It started with the pioneering work of Comer and Lin [8] and is currently an essential activity in security assessment tasks. Some of the most known network fingerprinting operations are done by NMAP [9], using a set of rules and active probing techniques. Passive techniques became known mostly with the P0F [10] tool, which is capable to do OS fingerprinting without requiring active probes. Many other tools like (AMAP, XProbe, Queso) did implement similar schemes.

Application layer fingerprinting techniques, specifically for SIP, were first described in [11, 12]. These approaches proposed active as well as passive fingerprinting techniques. Their common baseline is the lack of an automated approach for building the fingerprints and constructing the classification process. Furthermore, the number of signatures described are minimal which leaves the systems easily exposed to approaches as the one described by D. Watson et al. [13], that can fool them by obfuscation of such observable signatures. Recently, the work by J. Caballero et al. [6] described a novel approach for the automation of Active Fingerprint generation which resulted in a vast set of possible signatures. It is one of the few automatic approaches found in the literature and it is based in finding a set of queries (automatically generated) that identify different responses in the different implementations. While our work addresses specifically the automation

for passive fingerprinting, we can imagine this two complementary approaches working together.

There have been recently similar efforts done in the research community aiming however at a very different goal from ours. These activities started with practical reverse engineering of proprietary protocols [14] and [15] and a simple application of bioinformatics inspired techniques to protocol analysis [16]. These initial ideas matured and several other authors reported good results of sequence alignment techniques in [17], [18], [19] and [20]. Another major approach for the identification of the structure in protocol messages is to monitor the execution of an endpoint and identify the relevant fields using some tainted data [21], [22]. Recently, work on identifying properties of encrypted traffic has been reported in [23, 24]. These two approaches used probabilistic techniques based on packet arrivals, interval, packet length and randomness in the encrypted bits to identify Skype traffic or the language of conversation. While all these complementary works addressed the identification of the protocol building blocks or properties in their packets, we assumed a known protocol and worked on identifying specific implementation stacks.

The closest approach to ours, in terms of message comparison, it is the work developed by M. Chang and C. K.Poon [25] for collection training SPAM detectors. However, in their approach as they focus in identifying human written sentences, they only consider the lexical analysis of the messages and do not exploit an underlying structure.

Finally, two other solutions have been proposed in the literature in this research landscape. Flow based identification has been reported in [26], while a grammar/probabilistic based approach is proposed in [27] and respectively in [28].

7 Conclusions

In this article we described a novel approach for generating fingerprinting systems based on the structural analysis of protocol messages. Our solution automates the generation by using both formal grammars and collected traffic traces. It detects important and relevant complex tree like structures and leverages them for building fingerprints. The applicability of our solution lies in the field of intrusion detection and security assessment, where precise device/service/stack identification are essential. We have implemented a SIP specific fingerprinting system and evaluated its performance. The obtained results are very encouraging. Future work will consist in improving the method and applying it to other protocols and services. Our work is relevant to the tasks of identifying the precise vendor/device that has generated a captured trace. We do not address the reverse engineering of unknown protocols, but consider that we know the underlying protocol. The current approach does not cope with cryptographically protected traffic. A straightforward extension for this purpose is to assume that access to the original traffic is possible. Our main contribution consists in a novel solution to automatically discover the significant differences in the structure of

protocol compliant messages. We will extend our work towards the natural evolution, where the underlying grammar is unknown.

The key idea is to use a structural approach, where formal grammars and collected network traffic are used. Features are identified by paths and their associated values in the parse tree. The obtained results of our approach are very good. This is due to the fact that a structural message analysis is performed. Most existing fingerprinting systems are built manually and require a long lasting development process.

References

1. Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., Schooler, E.: SIP: Session Initiation Protocol (2002)
2. Crocker, D.H., Overell, P.: Augmented BNF for Syntax Specifications: ABNF (1997)
3. Buttler, D.: A Short Survey of Document Structure Similarity Algorithms. In Arabnia, H.R., Droegehorn, O., eds.: International Conference on Internet Computing, CSREA Press (2004) 3–9
4. Broder, A.Z.: On the Resemblance and Containment of Documents. In: SEQUENCES '97: Proceedings of the Compression and Complexity of Sequences 1997, Washington, DC, USA, IEEE Computer Society (1997) 21
5. Nash, J.F.: Non-Cooperative Games. The Annals of Mathematics **54**(2) (1951) 286–295
6. Caballero, J., Venkataraman, S., Poosankam, P., Kang, M.G., Song, D., Blum, A.: FiG: Automatic Fingerprint Generation. In: The 14th Annual Network & Distributed System Security Conference (NDSS 2007). (February 2007)
7. : DParser. `http://dparser.sourceforge.net/`
8. Douglas Comer and John C. Lin: Probing TCP Implementations. In: USENIX Summer. (1994) 245–255
9. : Nmap. `http://www.insecure.org/nmap/`
10. : P0f. `http://lcamtuf.coredump.cx/p0f.shtml`
11. Yan, H., Sripanidkulchai, K., Zhang, H., yin Shae, Z., Saha, D.: Incorporating Active Fingerprinting into SPIT Prevention Systems. Third Annual VoIP Security Workshop (June 2006)
12. Scholz, H.: SIP Stack Fingerprinting and Stack Difference Attacks. Black Hat Briefings (2006)
13. Watson, D., Smart, M., Malan, G.R., Jahanian, F.: Protocol scrubbing: network security through transparent flow modification. IEEE/ACM Trans. Netw. **12**(2) (2004) 261–273
14. : Open Source FastTrack P2P Protocol. `http://gift-fasttrack.berlios.de/` (2007)
15. Fritzler, A.: UnOfficial AIM/OSCAR Protocol Specification. `http://www.oilcan.org/oscar/` (2007)
16. Beddoe, M.: The Protocol Informatics Project. Toorcon (2004)
17. Gopalratnam, K., Basu, S., Dunagan, J., Wang, H.J.: Automatically Extracting Fields from Unknown Network Protocols. In: Systems and Machine Learning Workshop 2006. (2006)

18. Wondracek, G., Comparetti, P.M., Kruegel, C., Kirda, E.: Automatic Network Protocol Analysis. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08). (2008)
19. Newsome, J., Brumley, D., Franklin, J., Song, D.: Replayer: automatic protocol replay by binary analysis. In: CCS '06: Proceedings of the 13th ACM conference on Computer and communications security, New York, NY, USA, ACM (2006) 311–321
20. Cui, W., Kannan, J., Wang, H.J.: Discoverer: automatic protocol reverse engineering from network traces. In: SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, Berkeley, CA, USA, USENIX Association (2007) 1–14
21. Brumley, D., Caballero, J., Liang, Z., Newsome, J., Song, D.: Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In: SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, Berkeley, CA, USA, USENIX Association (2007) 1–16
22. Lin, Z., Jiang, X., Xu, D., Zhang, X.: Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In: 15th Symposium on Network and Distributed System Security. (2008)
23. Bonfiglio, D., Mellia, M., Meo, M., Rossi, D., Tofanelli, P.: Revealing skype traffic: when randomness plays with you. SIGCOMM Comput. Commun. Rev. **37**(4) (2007) 37–48
24. Wright, C.V., Ballard, L., Monrose, F., Masson, G.M.: Language identification of encrypted VoIP traffic: Alejandra y Roberto or Alice and Bob? In: SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, Berkeley, CA, USA, USENIX Association (2007) 1–12
25. Chang, M., Poon, C.K.: Catching the Picospams. In: ISMIS, Springer, Berlin, ALLEMAGNE (2005) 641–649
26. Haffner, P., Sen, S., Spatscheck, O., Wang, D.: ACAS: automated construction of application signatures. In: MineNet '05: Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data, New York, NY, USA, ACM (2005) 197–202
27. Borisov, N., Brumley, D.J., Wang, H.J.: Generic Application-Level Protocol Analyzer and its Language. In: 14th Symposium on Network and Distributed System Security. (2007)
28. Ma, J., Levchenko, K., Kreibich, C., Savage, S., Voelker, G.M.: Unexpected means of protocol inference. In: IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement, New York, NY, USA, ACM (2006) 313–326