FiG: Automatic Fingerprint Generation

Juan Caballero
Carnegie Mellon University
jcaballero@cmu.edu

Shobha Venkataraman
Carnegie Mellon University
shobha@cs.cmu.edu

Pongsin Poosankam
Carnegie Mellon University
ppoosank@cmu.edu

Min Gyung Kang
Carnegie Mellon University
mgkang@cmu.edu

Dawn Song
Carnegie Mellon University
dawnsong@cmu.edu

Avrim Blum
Carnegie Mellon University
avrim@cs.cmu.edu

Abstract

Fingerprinting is a widely used technique among the networking and security communities for identifying different implementations of the same piece of networking software running on a remote host. A fingerprint is essentially a set of queries and a classification function that can be applied on the responses to the queries in order to classify the software into classes. So far, identifying fingerprints remains largely an arduous and manual process. This paper proposes a novel approach for automatic fingerprint generation, that automatically explores a set of candidate queries and applies machine learning techniques to identify the set of valid queries and to learn an adequate classification function. Our results show that such an automatic process can generate accurate fingerprints that classify each piece of software into its proper class and that the search space for query exploration remains largely unexploited, with many new such queries awaiting discovery. With a preliminary exploration, we are able to identify new queries not previously used for fingerprinting.

1. Introduction

Fingerprinting is a technique for identifying the differences among implementations of the same networking software specification, be it applications, operating systems or TCP/IP stacks. It is well-known that even when the functionality of a piece of software is detailed in a specification or standard, different implementations of that same functionality tend to differ in the interpretation of the specification, by making assumptions or implementing only part of the optional functionality.

In network security, fingerprinting has been used for more than a decade [15] and it has a variety of applications. Some fingerprinting tools such as Nmap [9] are used to identify hosts running a specific operating system. There are also tools that can be used to identify different versions of the same application such as fpdns [2], Nmap, and Nessus [8]. These tools help network administrators to find version information leaked by a system, inventory the hosts in a network, and check for the existence of hosts running versions with vulnerabilities, or versions that are not allowed under the security policy of a network.

However, identifying the *fingerprints* used by these tools, *the fingerprint generation*, is currently a manual process which is arduous, incomplete, and makes it difficult to keep up-to-date with the numerous new implementations and new version updates. In this paper we propose a novel approach for automatic fingerprint generation. The goal is to automatically produce fingerprints that can differentiate among distinct implementations of the same specification and can be used by different fingerprinting tools.

A fingerprint contains 1) a set of queries, and 2) a classification function. To use the fingerprint to identify the class to which a piece of software belongs, one sends the queries, collects the responses and uses the classification function to classify the response. In the remainder of this paper, we will use the term classifying a host when we refer to classifying a piece of networking software running on the host. Thus, for ease of description, we assume that there is only one relevant piece of networking software running on the host; we can easily remove this assumption by classifying each piece of networking software on the host separately.

In this paper, we demonstrate how to automatically identify useful queries and classification functions. Our fingerprint generation process contains three phases: First, a *Candidate Query Exploration* phase which outputs candidate queries. Second, a *Learning* phase where those candidate queries are sent to different implementations and the responses are gathered and passed to the learning algorithm.

The Learning phase outputs *the query set*, which is a subset of the set of candidate queries that includes only the candidate queries that are useful for fingerprinting, and a *classification function*. The pair formed by the query set and the classification function is the fingerprint. Third, a *Testing* phase where the produced fingerprints are tested over a larger number of different implementations to evaluate its accuracy.

We also study what to do when a host does not match any known fingerprint. One straightforward approach would be to classify the host as unknown, but one could also perform an *approximate matching* to the nearest known class. In this paper, we set out to validate if and when this approximate matching can be performed.

This paper presents a general methodology for automatic fingerprint generation that covers all three phases and generates fingerprints that can be used by different fingerprinting tools. We validate our methodology by applying it on two distinct problems: 1) generating fingerprints to differentiate operating systems and 2) generating fingerprints to differentiate implementations of the same application, namely DNS servers.

Our experimental results show that the fingerprints automatically generated by our approach are accurate. With a preliminary exploration of the search space, we are able to find many novel fingerprints that are not currently used by fingerprinting tools. These novel fingerprints can increase the accuracy and granularity of current fingerprinting tools.

The rest of this paper is structured as follows: Section 2 introduces the automatic fingerprint generation problem. In Section 3 we present the candidate query exploration phase. The learning algorithms are explained in Section 4 and in Section 5 we evaluate their performance. Section 6 presents the related work and we conclude in Section 7.

2. Overview

Problem definition Given a set of k implementation classes $\mathcal{I} = \{I_1, I_2, \ldots, I_k\}$, the goal of fingerprinting is to classify a host H into one of those k classes or to another *unknown* class. The problem of *automatic fingerprint generation* is to output a set of queries Q and a classification function f_Q, such that when we send the set of queries Q to a host H and collect the responses R_Q from H, $f_Q(R_Q)$ can classify the host into one of the k classes or into the unknown class. We refer to the pair $\langle Q, f_Q \rangle$ as the *fingerprint*.

Approach Our automatic fingerprint generation contains 3 steps as shown in Figure 1, namely the *Candidate Query Exploration* phase, the *Learning* phase and the *Testing* phase. First, in the Candidate Query Exploration phase we produce a set of candidate queries Q_c, which could potentially produce different responses from hosts belonging to different classes. This process takes as input the protocol semantics in the form of protocol standards and domain knowledge.

Then, given a set of training hosts T where the implementation class of every host in T is known and is in the set $\mathcal{I}$, the first step in the Learning phase is to send the set of candidate queries Q_c to each of the hosts in T using a packet injection tool and to gather the responses from each host. The responses and the classes are then passed to the learning algorithm, which identifies a set of queries $Q \subseteq Q_c$, that produces different responses from hosts belonging to different classes, and a classification function f_Q.

Finally, in the Testing phase we need to verify that the pair $\langle Q, f_Q \rangle$ generated in the Learning phase has sufficient accuracy. This phase uses a set of testing hosts E, where none of the hosts in E should belong to the training set T, and each of the hosts in E should belong to one of the classes in $\mathcal{I}$. Then, this classification is compared to the known classification for E and if the accuracy satisfies some predefined metrics, the pair $\langle Q, f_Q \rangle$ is considered valid and becomes the output fingerprint. The fingerprints obtained can now be used by any fingerprinting tool to classify unknown hosts by sending the query set Q to this host, and applying the classification function f_Q on the responses.

Besides automatically generating the fingerprints needed by the fingerprinting tools, we also want to validate if it is possible for a fingerprinting tool to perform *approximate matching* when the responses from a host cannot be classified into any known class. Approximate matching could then potentially be used to distinguish between hosts that do indeed belong to a class not yet seen and thus should be classified as unknown, and hosts that have a slightly different behavior (e.g. some networking parameters manually tweaked) but actually belong to one of the known classes. For approximate matching to be meaningful, we need to test if the different implementation classes are well-separated. We do this by clustering, and calculating the distance between implementation classes. If the implementation classes are well-separated, we can, under some natural assumptions, find an approximate match for a given new host. Thus, only if the new host is far from all the known classes do we classify it as unknown.

In Section 3 we describe how to explore the candidate query space, and in Section 4 we present the learning algorithms, and how they are used to find the fingerprints.

3. Candidate Query Exploration

In this section we describe the Candidate Query Exploration phase. This phase needs to select a set of candidate

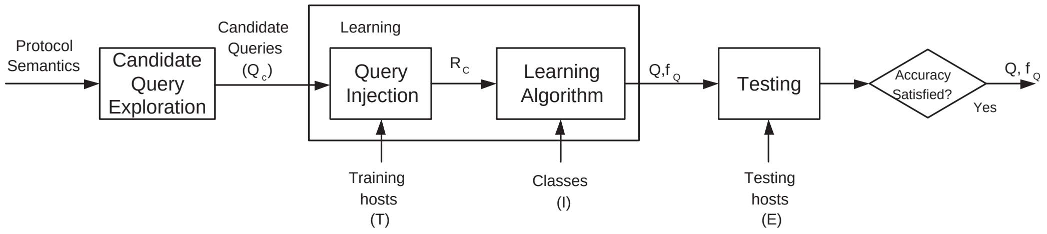


Figure 1. Fingerprint generation process.

queries that can potentially produce distinct responses from the different classes of hosts. During the Learning phase, those candidate queries that indeed produce distinguishing responses will be selected as part of the query set in the final fingerprint.

To automate the process of query exploration we could perform an exhaustive search of all possible combinations. Besides being automatic, it is also complete, guaranteeing that it will find all useful queries. However, the problem with this approach is that the search space is very large. For DNS fingerprinting we have at least 16 bytes of header fields in a DNS query, requiring 2^{128} combinations. For OS fingerprinting, the numbers become even more intractable. Even if we restrict the search to the TCP and IP headers there are at least 40 bytes of header information, which require 2^{320} combinations.

Such exhaustive search does not take advantage of the semantics of the protocol, i.e. the field structure in the protocol headers, and it can generate a large number of queries that are useless for fingerprinting, thus wasting both time and resources. For example, a query that spoofs the IP source address becomes useless since the reply will never make it back to the sender.

We combine exhaustive search with the semantics of the protocol by selecting some fields with rich semantics (such as the TCP or DNS flags) and performing an exhaustive search on those, while limiting the search to selected values for other less interesting fields. This greatly reduces the search space and requires little human intervention. It also reduces the time and resources needed to complete the search and our results show that it is still possible to find many useful queries not yet used for fingerprinting.

In Section 5.1 we present the specific TCP/IP and DNS fields that we explore. One example of how combining semantic and exhaustive search reduces the search space is byte 12 of the TCP header [25] which contains the Data Offset (4 bits) and Reserved fields (4 bits). Assuming that the fields are independent, rather than searching the 2^8 combinations in the whole byte, we can fix the value of one field at a time, while performing an exhaustive search on the other. This would require a total of $2^4 + 2^4 = 32$ candidate queries.

4. Learning Algorithms

We formulate the fingerprint generation problem as a *classification* problem: we are given a set of *instances* from different classes, along with the *labels* of the classes they belong to, and we need to find properties that hold within a class and are different across classes. An instance is represented as a collection of values for a set of *features*; thus, an instance is a point in the *feature space*. Given a family of *classification functions* over these features, our goal is to find a good function within this family that separates the classes. In our case the classes are the different implementations of the same functionality and our goal is to output the fingerprint composed by the set of queries and the classification function.

Thus, we need to define the feature space and the family of classification functions, and then learn the classification function, and turn it into our final fingerprint. We describe each of these in turn in this section. First, in Section 4.1 we describe the feature space and how to obtain the instances, needed by the learning algorithms, from the query/response pairs obtained from the training hosts. Then, in Section 4.2 we introduce the classes of functions we use for classification and briefly describe the learning algorithms. In Section 4.3 we describe how to take the output of the learning algorithms and convert it into our final fingerprints. Finally, in Section 4.4 we describe how to obtain an approximate matching when there is no exact matching among the fingerprints.

4.1. Feature Extraction

The first step for using a learning algorithm is to define the feature space, which allows us to convert the query/response pairs into the input for the learning algorithms. Thus, an *instance* is the representation of the query/response pairs from a host in the feature space; and the input to the learning algorithms is a set of instances.

Our feature space For simplicity, we describe our feature space for the case when the following relationships hold between queries and responses: (a) we consider only a single response byte string for each query, and (b) we consider each response to be independent of every other query/response from that host. With these two restrictions, it is sufficient to analyze only responses that come from different hosts to the same query. Therefore, in this section, when we discuss a feature space, we refer to the feature space with respect to a single query. We show later in this subsection how to generalize this feature space.

For a given query, we focus on the position-dependent substrings of responses, which we call *position-substrings*. In particular, we aim to extract the position-dependent byte substrings, that are consistently present in and distinctive to the responses of an implementation class (a similar analysis could also be done at bit level). Features involving position-substrings allow us to exploit the underlying structure of the byte sequences, since we are analyzing network protocols that usually have a well defined field structure.

Specifically, a *position-substring* of a response string is a tuple of three elements: the start and end positions in the original string and the bytes between these positions. So, for example, if the original byte string is *abcdefgabcd*, then we have two distinct position-substrings for the byte string $abcd$: the position-substring [1, 4, $abcd$] is distinct from the position-substring [8, 11, $abcd$]. A position-substring is present in a response if it appears at the proper position in the response. So, for example, the response string $abcdefg$ does not contain the position-substring [5, 7, abc], but the response string $abcdabc$ does.

With this definition of position-substrings, we can now describe the feature sets: for each response, the set of features extracted is the set of all possible position-substrings in the response. For a collection of responses from different hosts to the same query, the corresponding feature set is the union of all the features for each response string. The *feature space of a query* is the feature set of the response strings from all the different hosts to this query. In this section, and in Section 4.2, we will consider each query separately. Later in this section, we illustrate the feature space with an example, and show how individual response strings are represented in the space.

In this feature space, all of the information contained in the response string gets encoded into the features; there is no loss of information when transforming the response string into the feature set. Such a property is good, because the learning algorithm can then decide which information is useful for classification. However, there are other feature spaces which have this property. We choose the position-substring feature space because, in combination with some simple families of classification functions, it provides meaningful fingerprints that are easy to interpret. The classification function we might need to learn over other feature spaces may need to be more complex and thus, harder to interpret.

Generalizing the feature space This feature space could be generalized in many ways. To begin with, we could generalize both the cases mentioned above easily. If we assume that each response depends on the last k query-response pairs of the same host (rather than being independent of the other query-response pairs), we can transform it into our simpler case by concatenating every k responses from each host. If a particular query gets multiple responses from the same host, we can concatenate all responses together to reduce it to the case where there is just one response.

We could also extract more complex relationships between the various regions within a single byte sequence; e.g., we could examine if a certain position-dependent substring is always the sum of another position-dependent substring in the same byte sequence. The study of these more complex relationships within a sequence, and between different byte sequences is left as future work.

Generating instances Given a query, for each training host, we extract all the possible position-substrings from the response of that host. Then, we create the union set U from the position-substrings of all the responses from different training hosts to the same query, ordered lexicographically. Finally, we transform each response string into a vector $v \in \{0,1\}^n$ by setting the i-th bit of v to be one if the i-th position-substring of U is present in the response string, and we call this vector an *instance*.

Figure 2 shows this process for the responses from two different hosts to the same query. The responses from the hosts are the strings ac and ax. The set of position-substrings P_1 obtained from ac is $[1,1,a], [2,2,c], [1,2,ac]$, and the set of position-substrings P_2 obtained from ax is $[1,1,a], [2,2,x], [1,2,ax]$. Taking the union of these two sets P_1 and P_2, in lexicographical order, we get $U = [1,1,a], [2,2,c], [2,2,x], [1,2,ac], [1,2,ax]$. With this U, the instance corresponding to response ac becomes $(1,1,0,1,0)$, since only the first, second and fourth element of U are present in the P_1. Likewise, the instance corresponding to response ax becomes $(1,0,1,0,1)$.

Optimizations We may have very large sets of features if they are generated in this manner, and we might want to reduce the number of features provided there is no information loss. Sometimes (as in the conjunction fingerprints in Section 4.2.1), we do not need to generate all these position-substrings explicitly — our algorithm uses an equivalent set in a more efficient manner. We describe this implicit feature space in Section 4.2.1.

We can also use domain information to reduce the feature set. We do this in this paper: we use the semantics of the packet header to break the response string by protocol fields; so, each field is treated as a separate position-

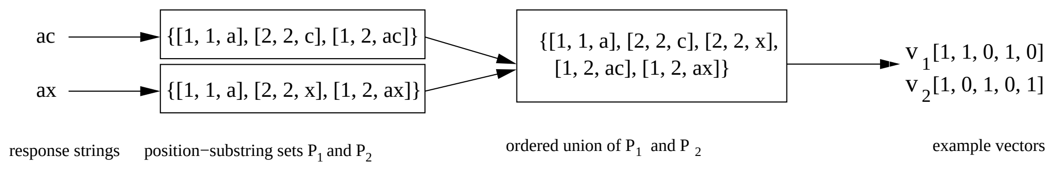

ac
ax
{[1, 1, a], [2, 2, c], [1, 2, ac]}
{[1, 1, a], [2, 2, x], [1, 2, ax]}
{[1, 1, a], [2, 2, c], [2, 2, x], [1, 2, ac], [1, 2, ax]}
v1 [1, 1, 0, 1, 0]
v2 [1, 0, 1, 0, 1]
response strings
position−substring sets P1 and P2
ordered union of P1 and P2
example vectors

Figure 2. Feature generation.

substring. This procedure generates many fewer features than the original feature space, and further, using the packet header semantics to generate features results in fingerprints that are much easier to understand. For this reason, in the analysis of the fingerprints that we present in Section 5, we focus only on this reduced feature set.

Further, not all of the protocol fields contain only implementation-specific information. For example, we term *dynamic fields* to be protocol fields that might change from one instance to another, such as session-specific fields (e.g. session id, time-to-live); host-specific fields (e.g. hostname, IP address, port); or configuration-specific fields (e.g. DNS answer). The values of these dynamic fields are based on external information and are less likely to contain information that helps to identify the implementation.

The training data should include instances of different values for these dynamic fields; if it does not, they might show up in the final fingerprint. For example, if we use the same open TCP port for fingerprinting all Windows hosts (e.g. 139) and a different one for all Linux hosts (e.g. 22), then the conjunction fingerprint will include this field. In this paper, we remove the dynamic fields when generating the fingerprints, but keep them for approximate matching, where we examine the effect of including these dynamic fields.

4.2. Algorithms

Overview For each query, once we have transformed the response from each training host into an instance, we need to find a classification function that separates the implementation classes. We do this on the instances obtained using the training hosts, and test the resulting classification function on instances obtained from the testing hosts. Since we cannot look through all possible classification functions, we search within families of classification functions for a good classification function. The goal of the learning algorithm is to find a good function within a given family that separates the different implementation classes.

We need classification functions that are easy to interpret, for understanding which queries are useful and when they are useful. Therefore, we choose classification functions that are boolean functions of the features, such as conjunctions, decision lists, decision trees, etc. We choose these over classification functions that are real-valued weighted functions of their features (such as linear separators, etc), because the fingerprints generated by such classification functions cannot be interpreted directly, and would need to be converted into a sequence of decision rules in order to be interpreted meaningfully.

In this paper, we consider two families of classification functions: conjunctions and decision lists [17]. We do not use decision trees, since (a) a restricted decision list has lower sample complexity than a similarly restricted decision tree, and (b) a sufficiently complex decision list can represent any decision tree. We discuss these two classes of fingerprints in more detail below. In Appendix A, we present the mistake-bound model for the analysis of these fingerprints, and show improved bounds for the conjunction fingerprints.

Our approach to learn fingerprints is the following: first, we learn a *binary-fingerprint* for each implementation class, which is a fingerprint that can determine whether or not the host belongs to that implementation class. Then we combine many binary-fingerprints to obtain the final fingerprint that can classify hosts into one of multiple implementation classes.

In the rest of this section, we focus on learning fingerprints for a single query only. In the interest of readability, we will use the terms fingerprint and binary-fingerprint to refer to the classification function associated with a single query. We will distinguish explicitly the fingerprint for a single query from the overall final fingerprint when it is not clear from the context.

4.2.1 Learning Conjunction Binary-Fingerprints

Given a query, there may be protocol fields in the response from a host that exhibit values specific to the implementation, and thus are different across implementations. We want to include all of them in the fingerprint. For this reason, we evaluate conjunctions of position-substrings as one class of classification functions.

A *conjunction fingerprint* $C_{q,\mathcal{I}}$ for a set of implementation classes $\mathcal{I} = \{I_m\}_{1 \le m \le k}$ is a query q and a set of position-substrings $\{S_{(q,I_m)}\}_{1 \le m \le k}$ such that for each implementation class $I_m \in \mathcal{I}$, (a) all the position-substrings in $S_{(q,I_m)}$ are always present in the response from every training host in the implementation class I_m, but (b) not all of the position-substrings in $S_{(q,I_m)}$ are present in any response from any host in another class $I_{z \neq m}$ for this query q, i.e., they are distinctive with respect to the responses from an implementation class. Thus, the conjunction fingerprint for a query q can be represented as $C_{q,\mathcal{I}} = \{S_{(q,I_1)}, S_{(q,I_2)}, \ldots S_{(q,I_k)}\}$, when $\mathcal{I} = \{I_1, I_2 \ldots I_k\}$.

Using conjunction fingerprints A conjunction binary-fingerprint for class I_m decides only whether a host belongs to class I_m or not. To use a conjunction binary-fingerprint to classify whether a new host is in I_m, we test if all the position-substrings in S_{q,I_m} in the conjunction binary-fingerprint are present in the response of that host for the query q. If they are, the host belongs to the class I_m, otherwise it does not. We describe how to use the final conjunction fingerprint in Section 4.3.

Learning conjunction fingerprints Note that we describe how to learn conjunction binary-fingerprints here; in Section 4.3, we show how to turn binary-fingerprints into final fingerprints. For this, we use the standard algorithm to learn conjunctions from a set of labeled instances [17], with the following modification for efficiency. Instead of explicitly converting each host's response into an instance as described in Section 4.1, our algorithm first extracts the longest common position-substring for each starting position in each implementation class, while ensuring that there is no overlap between the position-substrings. It then removes common elements of these position-substrings that are present in every implementation class. This way, the algorithm is linear in the length of the response strings.

We illustrate the operation of the algorithm with an example. Figure 3 shows response strings for four hosts in I_1. The position-substrings that we extract are [3, 8, $cdefgh$] and [11, 12, kl]. Figure 4 shows the same process for four hosts in a second class I_2. The position-substrings that we extract are [3, 8, $stufgw$] and [11, 12, kl]. Then, in figure 5 the algorithm removes [11,12,kl] and [6,7,fg] since they are not useful for distinguishing between I_1 and I_2. Thus, we are left with the position-substrings [3,5,cde] and [8,8,h] for class I_1, and [3,5,stu] and [8,8,w] for class I_2.

While the conjunction fingerprints are simple and easy to interpret, they have limited expressivity, and in some cases, they may not be sufficient. There might, for instance, be two kinds of responses generated for a particular query from hosts within an implementation class, e.g., Windows hosts may always have one of two values for the TCP window, neither of which is present in any other implementation, and therefore, we could use the presence of either of these values in a Windows fingerprint. For such a query, if there are no other bytes that uniquely distinguish the implementation classes, we may not be able to find a conjunction fingerprint.

This type of behavior might occur because there may be slight differences in the same implementation class. For example, when collecting learning data one might think that Windows XP SP2 hosts will behave identically but the network behavior is affected by the patch level and language version among others. Thus, in order to capture multiple behaviors in an implementation class, we need to use a more general class of functions. In the next section, we will describe such a class, decision lists.

4.2.2 Learning Decision List Binary-Fingerprints

A decision list can be viewed as an ordered sequence of multiple if-then-else statements, where a conjunction can be viewed as a single if-then-else statement. We will use decision lists to capture the presence of multiple behaviors in a single implementation class, that are not present in other implementation classes.

The standard definition of a decision list [17] is as follows: a decision list over n boolean variables $Y = \{Y_1, Y_2, \ldots, Y_n\}$ and k classes is a sequence L: $(c_1, I_1), (c_2, I_2), \ldots, (c_t, I_k)$, where c_i is a conjunction on Y and I_i denotes the class. We will refer to each pair (c_i, I_i) as a *decision rule*. A decision list is thus an ordered sequence of decision rules – the condition c_i of each decision rule is tested in the order of its appearance in L and the output is the classification I_i corresponding to the first satisfied decision rule. When none of the conditions are satisfied, the decision list outputs *unknown*.

In our setting, for each query, we use boolean variables to denote the presence of position-substrings in the response, and each condition c_i is equivalent to the presence of a list of position-substrings in that response. Thus, a *decision list fingerprint* is a query and an ordered sequence of rules, where each rule consists of a list of position-substrings and an associated implementation class. To use the decision list fingerprint to classify a new host, we send the query to the host and collect the response. We then test the response with each rule in turn to see if all the position-substrings in that rule are present in the response, and if they are, we output the associated class.

As in the case of the conjunctions, we find a decision list fingerprint for each query separately. We use the standard algorithm to learn a decision list for a given set of responses from different training hosts to a single query [17]. As with the conjunctions, this decision list is a binary-fingerprint; in Section 4.3, we describe how to convert them into the final fingerprint.

Note that decision lists are a more general class of func-

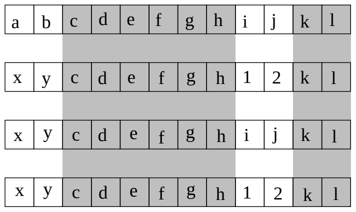


Figure 3. Responses from class I_1.

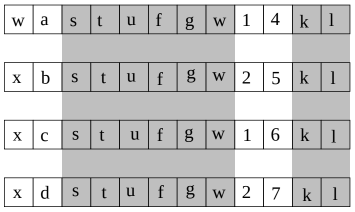


Figure 4. Responses from class I_2.

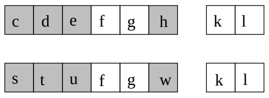


Figure 5. Removing common position-substrings.

tions than conjunctions, and thus, every time we have a conjunction fingerprint for a query we will also have a decision list fingerprint but the reverse is not always true. However, since there is a larger space of candidate functions to be explored, the algorithms for learning decision lists have a higher space and time complexity than the algorithms that learn conjunctions.

4.3. Obtaining the Final Fingerprint

Here we present the steps needed to obtain the final fingerprint from the output of the learning algorithm.

Removing unusable queries As we explore the query space, we may find that some queries do not induce distinguishing behavior among the different implementation classes, and therefore are not useful for fingerprinting. For conjunctions, this happens when a query fails to produce different responses across different implementations and thus an empty conjunction binary-fingerprint is generated. For decision lists, it happens when the binary-fingerprint generated for one query does not cover the complete set of instances in a class, usually because there is not enough distinct behavior that separates all hosts in the class from hosts in the other class. Our approach is to remove these queries and their corresponding binary classification functions from the binary-fingerprints during the Learning phase, leaving only the usable queries.

Final fingerprints for classifying multiple implementation classes The conjunction and decision list algorithms as presented are designed for binary classification of the responses to a single query. However, we can use these binary classification algorithms for multi-class classification by repeating the following procedure for each implementation class I_m. First, separate all hosts that belong to I_m as a single group, and all hosts that belong to *any other* implementation class as a second group, $W_m = \mathcal{I} \setminus \{I_m\}$. Then, generate the binary-fingerprint BF_m for groups $\{I_m, W_m\}_{1 \le m \le k}$, which will distinguish the implementation class I_m from all other classes. Once this procedure has been completed for all implementation classes, the final fingerprint is the set of all the binary-fingerprints.

For some queries, the final fingerprint may not contain a binary-fingerprint for every implementation class. If we have enough queries with complete final fingerprints, we can discard these fingerprints; otherwise, we need to combine multiple queries with partially-complete fingerprints as described below.

To classify a new host using a final fingerprint, we apply each binary-fingerprint BF_m in turn to the responses from the host, and store whether the host belongs in W_m or I_m, for each $m \in [1, k]$. Then, if there is only one m in which the host belongs to I_m, we output that class I_m as the result, otherwise, we report that the class is unknown. We describe an approximate matching we can use in this case in Section 4.4.

Combining multiple queries The conjunction and decision list fingerprints generated in Sections 4.2.1 and 4.2.2 only consider the responses for a single query. Once we have removed the unusable queries, if the number of implementation classes is large, we may not be able to find a single query that can classify all implementation classes simultaneously. In that case, we create fingerprints for multiple queries together, where each query is able to identify some subset of the implementation classes, so that the combination can classify all the implementation classes. We can use a greedy algorithm to create such a fingerprint that adds queries until all the implementation classes can be classified. This algorithm is guaranteed to find a small set of queries, since the problem is equivalent to the Set-Cover problem [28].

Testing the fingerprint Once the final classification function has been generated for each usable query, we test it using a larger set of testing hosts whose implementation classes are known, examining for each fingerprint if it classifies all the hosts correctly. We then discard any fingerprint that does not classify the hosts correctly. This may happen if the training hosts are not representative of the implementation classes; however, in our experiments, this did not happen.

4.4. Approximate Fingerprint Matching

Once shipped in a fingerprint tool, the fingerprints are used to classify new hosts. Sometimes, however, no fingerprint may match a new host and then one simple approach might be to classify the host as unknown. However, it could happen that the host truly belongs to one of the known classes but has slight differences in its responses, e.g. some network configuration parameter has been manually changed from the default. A more elaborate option would be to try to find an approximate match by calculating the distance to all known classes and selecting the nearest class [13, 19]. In this case, only if the new host is far from all of the existing classes would the host be classified as unknown.

Some current fingerprinting tools such as Nmap[1] have an option to do approximate matching, where the tool will print that no perfect match was found, but it was able to find a match with some percentage overlap. However, we need to validate when such a match is meaningful: an approximate matching is only meaningful when the behavior of each implementation class is well-separated from the others. So, we want to answer the following question: given a feature-space, how can we tell when we can do approximate matching, and how can we do approximate matching?

We answer this in the following manner: (a) first we use a clustering algorithm to cluster the responses from the training hosts, (b) then, we examine the implementation classes of the resulting clusters to check if the clusters truly represent the implementation classes, that is, if each cluster consists of a single implementation class and is well-separated from the other clusters. If the implementation classes form well-separated clusters, the training hosts are representative of their respective implementation classes, and any un-represented implementation class is also well-separated from these classes in this feature space, then we can use approximate matching with this feature space to classify new hosts that have no exact fingerprint match.

There might be classes that spread over two or more clusters, for example because hosts in the same class exhibit one of two distinct behaviors. We define a *z-gap property* that takes this into account, and needs to hold for a well-separated clustering: the distance between any two hosts in different implementation classes needs to be at least z times the distance between any two hosts in the same cluster, belonging to the same implementation class.

To do the clustering, we examine two natural feature spaces derived from the set of input candidate queries. In the first case, for each training host, we generate the set of all distinct position-substrings for each field in the response to a single query, as described in Section 4.1. We then remove all dynamic fields and use the remaining position-substrings as features. In the second case, for each training host, we generate the set of all distinct position-substrings of length one byte in the response to a single query, and use those as the features. This second case is independent of the field structure of the protocol and we use it to analyze the impact of dynamic fields contained in the responses. In both cases, the final feature space is the cross product of the feature spaces that we have defined for each query. Once the final feature space is defined, the responses from each host to different queries are then turned into a $\{0,1\}^n$ vector in these feature spaces. The distance metric we use for each feature space is the squared Euclidean distance; so, the distance between two hosts is their squared Euclidean distance when represented in the feature space.

With the z-gap property and a feature space, we can apply any clustering algorithm to test if there is a good clustering of the hosts. Here, we use X-means [24], an extension of the standard k-means algorithm to the case where k is unknown. We choose X-means over k-means because we do not know k, and over hierarchical clustering because we do not need to define a stopping criterion. In Section 5.4, we see that this algorithm performs well when the implementation classes are well-separated.

Given a new host that needs an approximate match, we do the following: we compute the distance from the host to each of the clusters. If the host is within a distance d/z from the nearest cluster, where d is the smallest distance between any two clusters, we classify it into the nearest cluster. If the host is farther away, we classify it as unknown. When the z-gap property holds, this rule will give us the correct matches. In Section 5.4, we show the results of using this rule for classifying new hosts for OS and DNS fingerprinting.

5. Evaluation

We evaluate our results using 128 hosts from 3 different implementation classes for the OS experiments, and 54 hosts from 5 different implementation classes for the DNS experiments. Tables 1 and 2 show the number of hosts in each implementation class for the OS and DNS experiments respectively. For the OS experiment we send queries to open TCP ports, i.e. port 139 on Windows or port 22 on Linux and Solaris.

5.1. Candidate Queries

For OS fingerprinting several protocols such as TCP, UDP or ICMP can be used. In this paper we focus on TCP,

[1]using equivalent options –osscan-guess or –fuzzy

Class ID	Hosts	OS class
Class 1	77	Windows XP SP2
Class 2	29	Linux 2.6.11
Class 3	22	Solaris 9

Table 1. Hosts used in TCP/IP evaluation.

Class ID	Hosts	DNS class
Class 4	10	BIND 8.3.0-RC1 – 8.4.4
Class 5	12	BIND 9.2.3rc1 – 9.4.0a0
Class 6	11	Windows Server 2003
Class 7	10	MyDNS
Class 8	11	TinyDNS 1.05

Table 2. Hosts used in DNS evaluation.

due to its rich semantics. As explained in Section 3, the candidate query exploration phase uses domain knowledge to select some fields to be explored exhaustively and others to be explored only with selected values.

Table 3 shows the 305 TCP/IP candidate queries that were explored in the candidate query exploration phase. We emphasize that this exploration can be easily expanded and is by no means complete, these candidate queries were selected as examples to test the validity of the fingerprint generation process. Three fields in the TCP header were explored using exhaustive search: the TCP flags byte (Byte 12), and Byte 13 which comprises the Data Offset and the Reserved fields [25]. For reference the TCP & DNS headers are reproduced in Appendix B. The reason we performed an exhaustive search on these fields is because they have rich semantics, and because new functionality, such as the flags for Explicit Congestion Notification [26], has not been thoroughly explored. For the other fields, only a few corner cases that could potentially hold interesting information were selected.

For DNS fingerprinting, bytes 2 & 3 of the DNS header were exhaustively searched. These bytes contain the $Opcode$, $Rcode$ and $Flags$ fields. Also the $Qtype$ field in the Question record [21] was exhaustively searched. Like the selected TCP fields, these fields were chosen because they have rich semantics and support numerous options.

5.2. Conjunction and Decision List Fingerprints

As explained in Section 4.2, for each candidate query, the learning algorithm takes two steps in order to find the final fingerprint. First, it generates binary-fingerprints for each implementation class, which determine whether a host belongs to this class or not. Then, the set of all binary-fingerprints for the same query forms the final fingerprint.

In our results, we show the number of binary and final fingerprints found for the cases of OS and DNS fingerprint generation. For the OS experiments the features are extracted from the TCP/IP headers in the response, while for the DNS experiments only the DNS header in the response is used. We run the learning algorithms on the responses of 70% of the hosts in each class and test the resulting fingerprints using the remaining 30% hosts. Any other split of the host set is valid as long as there are sufficient hosts in the training set.

5.2.1 Binary and Final Fingerprints

Table 4 shows the number of binary and final fingerprints identified in both steps of the algorithm for the OS and DNS experiments.

For each experiment, a series of columns show the number of binary-fingerprints for the corresponding implementation classes, while the rightmost column shows the number of final fingerprints. Each binary-fingerprint for an implementation class can decide whether or not a host belongs to that class. The final fingerprint can classify the host into any of the known classes, or state that the class is unknown. As expected, the decision list algorithm outputs a decision list fingerprint in many cases where the conjunction algorithm cannot output a conjunction fingerprint – this happens when the hosts that belong the class under consideration exhibit multiple types of behavior.

The *Final* columns in Table 4 show that there is no final conjunction fingerprint that can separate all the classes in both the OS and DNS experiments. On the other hand, there are 66 decision list fingerprints in the OS experiment that can classify hosts of all 3 classes, and 19 in the DNS experiment that can classify hosts of all 5 classes.

Intuitively, as the number of known classes increases, we expect to find fewer queries that can classify hosts of all known classes simultaneously. For example, when we run the conjunction algorithm using only the Windows and Linux classes, we find 130 final fingerprints that can separate Windows and Linux hosts, but when we add Solaris, we find no final fingerprints that can classify hosts of all of the three classes simultaneously. Note that as the number of classes grows, we can apply the learning algorithms on sets of queries, rather than on a single query. This will generate fingerprints that contain multiple queries, each individually covering some subset of known classes and the whole fingerprint covering all classes.

Testing We evaluate the 66 OS and 19 DNS final decision list fingerprints produced during the learning phase by sending the corresponding queries, in the final fingerprint, to the remaining 30% hosts in each implementation class. Each of the final fingerprints properly classifies all hosts in the testing set into their true OS or DNS class.

Field	Size	Type	# Queries	Tested values
tcp_sport	16	guided	9	0,8,255,1023-4,49151-2,55000,65535
tcp_offset	4	exhaustive	16	all
tcp_reserved	4	exhaustive	16	all
tcp_flags	8	exhaustive	256	all
tcp_window	16	guided	2	0, 65535
tcp_checksum	16	guided	2	good, bad
tcp_urgentPtr	16	guided	4	invalid value with URG flag set, value with URG flag not set

Table 3. Candidate queries for OS fingerprinting. A total 305 queries were tested. The field size is given in bits.

	OS				DNS					
Fingerprint type	Linux	Solaris	Windows	Final	Bind8	Bind9	Microsoft	MyDNS	TinyDNS	Final
Conjunction fingerprints	42	53	53	0	0	0	22	2	9	0
Decision list fingerprints	130	98	98	66	33	28	32	29	41	19

Table 4. Number of binary and final fingerprints output by the learning phase.

5.2.2 Fingerprint Examples

In this section, we show an example of the conjunction and decision list fingerprints for a specific TCP/IP query. First, we show the conjunction binary-fingerprint that separates the Linux class from the other classes (we refer to this case as Linux/NotLinux):

```
Query: tcp_flags=S+P;
if (Response: ip_id=0x0000,tcp_window=0x16d0)
   then Linux
   else NotLinux
```

As shown in the first line of the conjunction binary-fingerprint, this query explores the tcp_flags field and has the SYN+PUSH flags set. This conjunction fingerprint says that if in the response, the IP identification field has a value of zero and the TCP window has a value of 5,840 then the host is Linux, otherwise it is NotLinux. The values of the other fields in the response do not matter.

The conjunction binary-fingerprint for this query exists for the cases of Linux/NotLinux and Solaris/NotSolaris but not for the case Windows/NotWindows. Next, we show the corresponding decision list binary-fingerprint for the Linux/NotLinux case. Note that the decision list algorithm is able to extract more than one rule for the NotLinux case.

```
Query: tcp_flags=S+P;
if (Response: tcp_window=0xffff)
   then NotLinux
else if (Response: tcp_window=0x16d0)
   then Linux
else if (Response: ip_verHdrLen=0x45,
    ip_tos=0x00, ip_len=0x002c,
    ip_flags&offset=0x4000, ip_protocol=0x06,
    tcp_offsetReserved=0x60, tcp_flags=0x12,
    tcp_urgentPtr=0x0000)
   then NotLinux
```

Now, decision list binary-fingerprints exist for all three cases (Linux/NotLinux, Windows/NotWindows, and Solaris/NotSolaris) and the system can generate the following decision list final fingerprint that can classify a host into one of all three classes simultaneously.

```
Query: tcp_flags=S+P;
if (Response: tcp_window=0xffff)
   then Windows
else if (Response: tcp_window=0x16d0)
   then Linux
else if (Response: tcp_window=0xc0a0)
   then Solaris
else if (Response: ip_verHdrLen=0x45,
    ip_tos=0x00, ip_len=0x002c,
    ip_flags&offset=0x4000, ip_protocol=0x06,
    tcp_offsetReserved=0x60, tcp_flags=0x12,
    tcp_window=0x40e8, tcp_urgentPtr=0x0000)
   then Windows
else Unknown
```

This final fingerprint shows that all Solaris hosts set the tcp_window to 49,312 and all Linux hosts set the value to 5,840 but the Windows hosts use two different values for that field: 65,535 or 16,616.

5.3. Interesting Queries

The final fingerprints generated in our experiments contain some especially interesting queries because we are not aware of any fingerprinting tool that currently uses them. Here, we give some selected examples of these novel queries.

First, we find that the hosts in the Windows and Solaris classes respond to queries with an invalid value in the Data Offset field of the TCP header. This field represents the number of 32-bit words in the TCP header. The candidate query should have a value of 5 (20 bytes) in this field but we deliberately send queries with this field set to smaller and larger values. Both Windows and Solaris hosts reply with a SYN+ACK if the value in the field is less than five, while the Linux hosts do not reply to these incorrect values. No host in any class replies to values larger than five. This reveals that both Windows and Solaris fail to check the TCP header for this simple case.

Second, we see that Windows and Linux hosts ignore the values of the ECN or CWR bits in the queries but certain combinations trigger a different response for Solaris hosts. For example, a query with the SYN+PUSH+ECN+CWR flags all set, gets a SYN+ACK response from both Windows and Linux but a SYN+ACK+ECN response from Solaris.

Finally, we find that Linux and Solaris hosts set the TCP Acknowledgment Number in a RST packet to zero but Windows hosts set it to the value that was sent in the TCP Acknowledgement Number field of the query. This is interesting because a single packet with the ACK flag set, that is sent to a closed port, can distinguish Windows hosts from both the Linux and Solaris hosts. This type of query is very inconspicuous and might be difficult to flag as a fingerprinting attempt.

Among the DNS queries we also find interesting behavior. For example, DNS servers should copy the value of the $Qdcount$ field (i.e. the number of DNS queries) from the query packet to the response packet. This value is usually one, but if the query is not valid, some implementations, depending on the error, will set the field to one in the response while others will keep it to zero. Note that current tools such as fpdns do not test this field because they consider it uninteresting.

Our preliminary exploration of the candidate query space has been able to find multiple novel fingerprints, which reaffirms our intuition that the space of queries that could be used for fingerprinting remains largely unexplored and demonstrates the effectiveness of our automatic approach.

5.4. Clustering

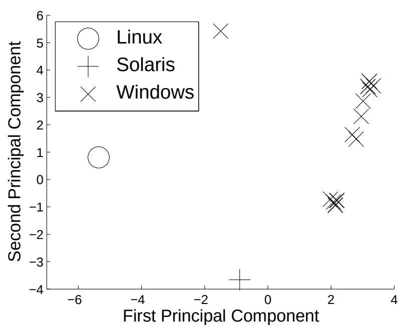
For the clustering experiments, we generate the clusters using 70% of the hosts in each class and then evaluate approximate matching using the remaining 30% hosts, similarly to the fingerprint generation experiments. To generate the clusters, we run the X-Means algorithm on the two feature spaces that we described in Section 4.4, that is, we compute the features either from a selected set of fields that contains implementation-dependent information or from the complete TCP/IP or DNS headers. We name these cases *Chosen Fields* and *Full Header* respectively. The X-means range that we use is from one to twenty. This range is chosen conservatively, and we check that the upper limit is never reached.

Table 5 shows the clustering results for the OS and DNS experiments. As expected, many of the classes defined in Tables 1 and 2 spread over more than one cluster, which indicates the presence of multiple behaviors inside the same class. We manually check a few of the DNS clusters and find that some of them are due to multiple versions in the same class such as BIND 9.2.3 and 9.3.2 being placed in the same class although they behave differently. Also, some BIND tags like 9.3.0 represent up to 8 different versions (3 betas, 4 releases candidates and the final version) [1].

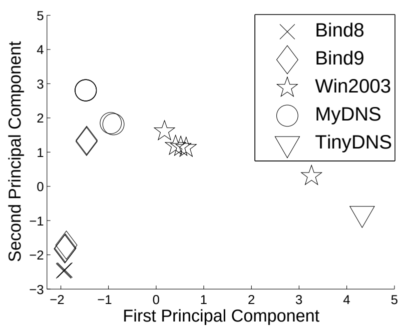
In order to analyze the differences between the Chosen Fields and Full Header cases and to check if hosts that belong to different classes are well-separated, Figure 6 shows the visualization of distances between hosts by projecting the feature space into the first two principal components. All hosts belonging to the same class are plotted using the same icon. This visualization does not reflect the precise distances, as there are a number of less significant principal components, but the first two principal components are significant enough to show the qualitative distances between different clusters.

Figures 6(a) and 6(b) show the results with Chosen Fields. The classes in the OS case are well-separated with only one Windows cluster (6 hosts) farther from the rest of the class but still clearly separated from the other classes. In the DNS case, the classes are more spread and the distances between hosts in the same class are larger. For example, some of the BIND9 hosts are close to the BIND8 hosts but others are close to the MyDNS hosts. This could be due to the evolution of versions of the same implementation that are expected to be close when they share a significant code base and move farther apart as the evolution of the new version progresses. With Chosen Fields, no cluster contains hosts from two implementation classes.

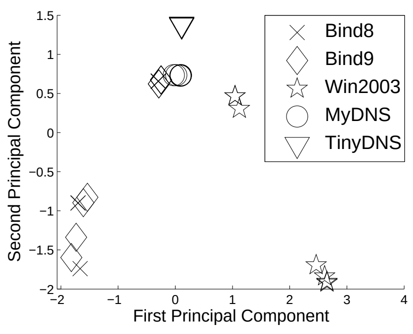
Figure 6(c) shows the results with the full DNS header. Results for the full TCP/IP header are similar and omitted for brevity. Using the full header, the hosts in the same class are further apart and hosts from different classes are closer or even overlap. For example, in this case one cluster contains hosts from the BIND8 and BIND 9 classes, shown with the name *Mixed* in Table 5. These results indicate that using an approach without any domain knowledge, that just considers the complete protocol header, does not obtain well-separated clusters. This is because some protocol fields that include session or host-specific information, such as the DNS ID or the DNS answers, may have more weight than the implementation-specific differences.



(a) Chosen Fields: OS



(b) Chosen Fields: DNS



(c) Full Header: DNS

Figure 6. Principal component plots of the responses from the hosts for both OS and DNS clustering.

	OS			DNS					
Fingerprint type	Linux	Solaris	Windows	Bind8	Bind9	Microsoft	MyDNS	TinyDNS	Mixed
Chosen Fields	1	1	10	2	3	3	2	1	-
Full Header	1	1	3	2	4	6	2	1	1

Table 5. For each implementation class, the number of clusters that contain hosts from this class. Multiple clusters indicate different behaviors inside the implementation class.

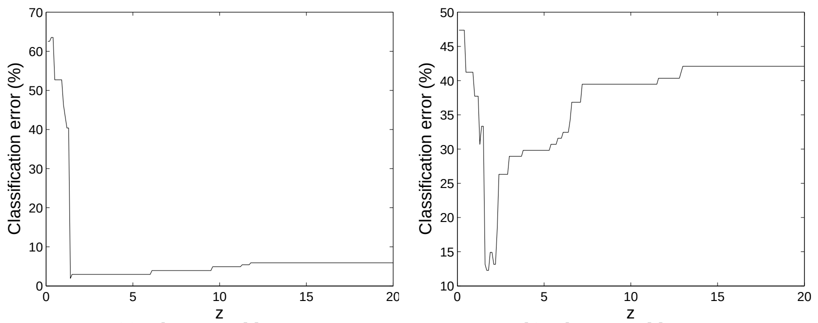
Now, we quantitatively measure how well approximate matching works for the OS and DNS cases with Chosen Fields. We perform the following experiment: from the set of implementation classes, we remove one class and extract the clusters using 70% of the hosts in the remaining classes. Then, we perform approximate matching using the z-gap rule on the remaining 30% hosts from the classes used to extract the clusters, plus all the hosts from the class that was removed. We repeat this process multiple times, each time removing a different class and at the end, we calculate the average classification error for different values of z.

The results show that the classification error is minimized with values $z \approx 2$ for both the OS and DNS cases and that in the DNS case the error quickly increases for other z values. The OS case does not show such a strong increase. We show the corresponding graphs in Figure 7. Setting the value of z to be 2, the classification error is 3% for the OS case and 12% for the DNS case. This indicates that the class separation for OS with Chosen Fields is more robust and might be used for approximate matching but in the case of DNS with Chosen Fields, approximate matching is unlikely to yield good results.

6. Related Work

Fingerprinting has been used for more than a decade. In 1994 Comer and Lin proposed probing to find differences between TCP implementations [15]. Early application for TCP fingerprinting focused on standard compliance testing to identify flaws, support for optional functionality, protocol violations and design decisions taken by the different implementations [22]. Besides active probing, there has been research on how to passively identify TCP implementations looking at traffic traces [23] and how to passively classify host's operating systems [13, 19]. Franklin et al. [16] proposed a passive fingerprinting technique to identify wireless device drivers on IEEE 802.11 compliant devices. In the context of finding approximate matches, Lippmann et al. [19] proposed to use a k-nearest-neighbor classifier to avoid hosts being classified as unknown when no exact match was found. Our approximate matching differs in that we use a clustering approach and focus on evaluating when such approximate matching is possible. Hardware fingerprinting has also been proposed with applications such as remotely tracking a host in a network [18]. However, to the best of our knowledge, this is the first work to address the problem of automatically generating fingerprints.

TCP/IP fingerprinting can also be used to identify the operating system running on a host [3, 4]. There exists multiple tools for both active and passive OS fingerprinting. The most common active fingerprinting tool in use today is Nmap [9] written by Fyodor, which uses a similar approach to older tools such as Queso [11]. Other active fingerprinting tools include Xprobe [12] that focuses on ICMP probes and Snacktime that identifies hosts based on the TCP timeout and retransmission policy. Passive fingerprinting tools such as p0f [10] and siphon [6] do not need to send traffic



(a) Chosen Fields: OS (b) Chosen Fields: DNS

Figure 7. Classification error of approximate matching using z-gap rule for different values of z.

and can be used to fingerprint hosts that might not reply to a query, such as those firewalled, but require access to the traffic sent by a host

There has also been work on defeating OS fingerprinting. Smart et al. [27] proposed a stack fingerprinting scrubber that sits on the border of a protected network and limits the information gathered by a remote attacker by standardizing the TCP/IP communication. This work is based on the protocol scrubber proposed by Malan et al. [20]. More recent tools such as Morph [7] and IPPersonality [5] operate on the host-level and allow to change the responses to specific queries by faking the behavior of a chosen OS.

7. Conclusion

Fingerprinting is a useful technique that allows us to identify different implementations of the same functionality. But, the fingerprint generation process is at large arduous and manual. In this paper we have proposed a novel approach for automatic fingerprint generation, that produces fingerprints with minimal human interaction.

We have shown how to automatically generate fingerprints and have demonstrated that our approach is flexible and can be applied to different uses. In this paper we have presented its application to two concrete examples: OS fingerprinting and DNS fingerprinting. Our results show that the produced fingerprints are accurate and can be used by fingerprinting tools to classify unknown hosts into given classes. We have also evaluated approximate matching as a technique to assign an unknown host to a known implementation when no exact fingerprint match is available.

In addition, our preliminary exploration of the candidate query space has been able to find new interesting queries, not currently used by fingerprinting tools. This confirms our intuition that the space of candidate queries remains largely unexplored and demonstrates the effectiveness of our automatic approach.

8. Acknowledgements

We would like to thank David Brumley for helpful discussions. We would also like to thank Fyodor, Bruce Maggs and the anonymous reviewers for their valuable comments to improve this paper. Juan Caballero would like to thank *la Caixa Foundation* for the generous support through their fellowship program.

References

[1] BIND. http://www.isc.org/index.pl?/sw/bind/.
[2] fpdns. http://www.rfc.se/fpdns/.
[3] Fyodor. Remote OS detection via TCP/IP fingerprinting (2nd generation). http://insecure.org/nmap/osdetect/.
[4] Fyodor. Remote OS detection via TCP/IP stack fingerprinting. Phrack 54, Vol. 8. December 25, 1998. http://www.phrack.com/phrack/51/P51-11.
[5] IPpersonality. http://ippersonality.sourceforge.net/.
[6] Know your enemy: Passive fingerprinting. identifying remote hosts, without them knowing. Honeynet project. http://project.honeynet.org/papers/finger/.
[7] Morph. http://www.synacklabs.net/projects/morph/.
[8] Nessus. http://www.nessus.org/.
[9] Nmap. http://www.insecure.org/.
[10] p0f. http://lcamtuf.coredump.cx/p0f.shtml.
[11] Queso. http://www.l0t3k.net/tools/FingerPrinting/.
[12] Xprobe2. http://www.sys-security.com/.
[13] R. Beverly. A robust classifier for passive TCP/IP fingerprinting. In *Proceedings of the 5th Passive and Active Measurement Workshop*, 2004.
[14] A. Blum. On-line algorithms in machine learning. In *Online Algorithms*, pages 306–325, 1996.
[15] D. Comer and J. C. Lin. Probing TCP implementations. In *USENIX Summer*, 1994.
[16] J. Franklin, D. McCoy, P. Tabriz, V. Neagoe, J. V. Randwyk, and D. Sicker. Passive data link layer 802.11 wireless device driver fingerprinting. In *Proceedings of the 15th Usenix Security Symposium*, 2006.
[17] M. Kearns and U. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[18] T. Kohno, A. Broido, and kc claffy. Remote physical device fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2005.

[19] R. Lippmann, D. Fried, K. Piwowarski, and W. Streilein. Passive operating system identification from TCP/IP packet headers. In *Proceedings of the ICDM Workshop on Data Mining for Computer Security*, 2003.

[20] G. Malan, D. Watson, and F. Jahanian. Transport and application protocol scrubbing. In *Proceedings of IEEE INFOCOM*, 2000.

[21] P. V. Mockapetris. RFC 1035: Domain names — implementation and specification, 1987.

[22] J. Padhye and S. Floyd. Identifying the TCP behavior of web servers. In *Proceedings of ACM SIGCOMM*, 2001.

[23] V. Paxson. Automated packet trace analysis of TCP implementations. In *Proceedings of ACM SIGCOMM*, 1997.

[24] D. Pelleg and A. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 727–734, San Francisco, 2000.

[25] J. Postel. Transmission control protocol. RFC 793 (Standard), 1981.

[26] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), 2001.

[27] M. Smart, G. R. Malan, and F. Jahanian. Defeating TCP/IP stack fingerprinting. In *Proceedings of the 9th USENIX Security Symposium*, 2000.

[28] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin, 2001.

APPENDIX

A. Mistake Bounds for Conjunctions

We now bound the number of mistakes a fingerprint will make under certain assumptions. We give *mistake-bounds* in an *online* model of learning [14], where the algorithm starts with an initial fingerprint and refines it with every mistake. Sections 4.2.1 and 4.2.2 present *offline* algorithms for learning fingerprints using a set of training hosts T. However, these fingerprints may be too specific to T. Even after testing the fingerprints over the set of hosts E, the fingerprints may not be sufficiently general. If T and E are not large enough or sufficiently representative of the implementation classes, the confidence guarantees we get on the resulting fingerprints might not be very high. This could happen, for example, when one is restricted to hosts within the local network. In this case, the generated fingerprints might be too specific to the local network.

In an online model of learning, an algorithm starts with an initial fingerprint, and keeps refining it every time it makes a mistake; i.e. the algorithm predicts a classification based on the current fingerprint and is then given the right answer, which it uses to update its fingerprint. In our setting, these initial fingerprints could be generated offline. After learning over T and testing over E, we can use these offline fingerprints with online algorithms and guarantee that over the set of *all* hosts classified (i.e., hosts classified by the fingerprint after it was generated using E and T), the number of mistakes we make is bounded. Obviously, small mistake-bounds are what we want.

We derive improved mistake bounds for learning conjunction fingerprints in this online model of learning. For the conjunction fingerprint, Theorem 1 shows that the mistake bound is small: when the initial fingerprint has n position-substrings and the true fingerprint has t, the mistake bound is $n-t$. Under certain assumptions, it is $\lceil \log \frac{n}{t} \rceil$, where n is the number of position-substrings considered, and t is the number of position-substrings in the conjunction fingerprint. This implies that we will make only $\lceil \log \frac{n}{t} \rceil$ mistakes (under certain assumptions) before reaching the right conjunction fingerprint. The mistake bound for the decision lists is much larger [14] and therefore, not practically useful.

We now present the theorem for the mistake-bounds for conjunction fingerprints. To do so, we need the following definitions and notation, so that we can represent the fingerprints and the response strings from the hosts as boolean functions and boolean vectors respectively.

Since our fingerprints denote the presence of position-substrings corresponding to pre-specified queries, we will define an *element* of a fingerprint to be a single position-substring along with the corresponding query identifier. Let U be the set of all the elements in the fingerprints of all the implementation classes under consideration, and let $|U| = n$. An *instance* X_j represents the response strings of a host that needs to be classified, and is a vector in $\{0,1\}^n$ where the ith coordinate is 1 if the ith element of U is present in the response strings and 0 otherwise.

Next, we describe how to represent a fingerprint as a boolean function. Let y_i be a boolean variable that denotes the presence of the ith element in U (e.g., if the ith element of U must be present in the fingerprint, y_i is in the corresponding boolean function.) Let $Y = \{y_1, \ldots, y_n\}$. Let H be the class of monotone conjunctions over Y (so, no negative literals of Y are allowed in the conjunctions). Let $h_1, h_2 \in H$ be the conjunctions that represent classes 1 and 2 respectively. Let A_1 be the set of boolean variables present in h_1 and let A_2 be the set of boolean variables present in h_2.

We give bounds under two cases: first, with no further assumptions; second, under the following two assumptions: (1) A_1 and A_2 are disjoint, and (2) any instance that belongs to h_2 contains no variable in A_1 and vice versa. The first assumption is that A_1 and A_2 are disjoint; no variable present in h_1 is also present in h_2 and vice versa. In our setting, this implies that the position-substrings present in one conjunction fingerprint are not present in the other. This is not an unreasonable assumption; we see this in the testing, especially when there are only two implementation classes under consideration. The second assumption is that no instance that belongs to h_2 contains the variables in A_1 and vice versa. We might, for example, expect this to be true when all the position-substrings consist of distinct values for the same fields of the underlying packet headers.

Theorem 1. *Assume that there are two implementation classes, each of which has fingerprints that can be represented by a conjunction of position-substrings. Let H be the class of monotone conjunctions over Y, and let $h_1, h_2 \in H$ denote the conjunction fingerprints with t variables for class I_1 and I_2 respectively. Let A_1 and A_2 denote variables present in h_1 and h_2 respectively. With no further assumptions, h_1 and h_2 have a mistake bound of $n - t$. When A_1 and A_2 are disjoint, and when every instance that is consistent with h_1 does not contain any variable in A_2 and vice versa, we can learn a conjunction fingerprint with t variables with a mistake bound of $\lceil \log(\frac{n}{t}) \rceil$ on instances that belong to I_1 and I_2.*

We now present the proof of this theorem.

Proof. We will show how to use a conjunction fingerprint to get a bounded number of mistakes for each case in the theorem statement. Let S denote the set of variables in the current conjunction hypothesis for I_1. Let $X_j \in \{0,1\}^n$ denote the current instance. Let $True_S(X_j)$ denote the set of variables in X_j that are set to true and are also present in S. Let $False_S(X_j)$ denote the set of variables in X_j that are set to false are also present in S. Let $Ones(X_j)$, $Zeros(X_j)$ denote the sets of variables in Y that are set to true and false respectively in the instance X_j. Note that $True_S(X_j) = Ones(X_j) \cap S$, while $False_S(X_j) = Zeros(X_j) \cap S$.

The proof for the first case is well known but we sketch it for completeness. In the first case, we will begin with the most specific conjunction over Y: the conjunction $y_1 \wedge y_2 \ldots \wedge y_n$. So, we begin with $S = Y$. We do the following: every time we make a mistake on an instance $X_j \in h_1$, we remove all the variables in $False_S(X_j)$ from S. We never make a mistake on an instance $X_j \notin h_1$ since we start with $S \supseteq A_1$ and never remove a variable in A_1. Thus, the number of mistakes we can make is bounded by $n - t$.

Next, we outline the proof for the second case. We will analyze the number of mistakes made to reach the correct conjunction for the implementation class I_1, on the instances that come from I_1 and I_2. We begin with the most specific conjunction over Y: the conjunction $y_1 \wedge y_2 \ldots \wedge y_n$. So, we begin with $S = Y$.

When we get a new instance X_j (from I_1 or I_2) that needs to be classified, we do the following: If the number of variables in $True_S(X_j)$ is greater than the number of variables in $False_S(X_j)$, we classify X_j as true, otherwise we classify it as false. If we make a mistake on an instance which does not belong to h_1 (so we report “true” when we should have reported false), we remove the variables in $True_S(X_j)$ from S. If we make a mistake on an instance that belongs to h_1 (so we report false when we should have reported true), we will remove all the variables in $False_S(X_j)$ from S.

This procedure will give us a bound of at most $\lceil \log(\frac{n}{t}) \rceil$ mistakes, since each mistake causes us to remove at least half the variables that are present in S, but are not present in the true hypothesis. So, if we make a mistake on an instance that belongs to h_1, at least half the variables in S must have been false in X_j. All of these will belong to $False_S(X_j)$. Now, none of these variables will be present in A_1: since X_j belongs to h_1 and h_1 is a monotone conjunction, all variables in A_1 must be set to true in X_j (i.e., $A_1 \subseteq Ones(X_j)$). So, A_1 is disjoint from $Zeros(X_j)$, therefore, none of the variables in A_1 will be in $False_S(X_j)$. Therefore we can remove all of the variables in $False_S(X_j)$ from S.

Likewise, if we make a mistake on an instance that does not belong to h_1, at least half the variables in S must have been true in X_j. Let Y_{rem} denote the set of variables in Y that are not in A_1 or A_2; so $Y_{rem} = Y - (A_1 \cup A_2)$. Since this instance X_j belongs to I_2, by assumption, $Ones(X_j) \subseteq A_2 \cup Y_{rem}$. None of these variables can be present in A_1 (since A_1 and A_2 are disjoint), so they can be discarded from S. Therefore, since $True_S(X_j) \subseteq Ones(X_j)$, we can discard the set $True_S(X_j)$ from S.

Thus, since we reduce the set of variables in the conjunction by at least half with every mistake, we will make $\lceil \log(\frac{n}{t}) \rceil$ mistakes when we start with a conjunction of size n, and our true conjunction is of size t.

□

There are also mistake-bounds for learning decision lists in the literature [14], however, they are quite loose and therefore not of practical use.

B. Headers

TCP header from [25] with added ECE and CWR flags.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |       |C|E|U|A|P|R|S|F|                               |
| Offset|Reserv.|W|C|R|C|S|S|Y|I|            Window             |
|       |       |R|E|G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

DNS header from RFC 1035 [21].

```
Header
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  +-------------------+
|                      ID                       |  |      Header       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  +-------------------+
|QR|   Opcode  |AA|TC|RD|RA|   Z    |   RCODE   |  |     Question      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  +-------------------+
|                    QDCOUNT                    |  |      Answer       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  +-------------------+
|                    ANCOUNT                    |  |    Authority      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  +-------------------+
|                    NSCOUNT                    |  |    Additional     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+  +-------------------+
|                    ARCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
Question
                                1  1  1  1  1  1
  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
/                     QNAME                     /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     QTYPE                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     QCLASS                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```