Network Protocol System Fingerprinting - A Formal Approach

Guoqiang Shu and David Lee
Department of Computer Science and Engineering, The Ohio State University
{shug, lee}@cse.ohio-state.edu

Abstract — **Network protocol system fingerprinting has been recognized as an important issue and a major threat to network security. Prevalent works rely largely on human experiences and insight of the protocol system specifications and implementations. Such ad-hoc approaches are inadequate in dealing with large complex protocol systems. In this paper we propose a formal approach for automated protocol system fingerprinting analysis and experiment. Parameterized Extended Finite State Machine is used to model protocol systems, and four categories of fingerprinting problems are formally defined. We propose and analyze algorithms for both active and passive fingerprinting and present our experimental results on Internet protocols. Furthermore, we investigate protection techniques against malicious fingerprinting and discuss the feasibility of two defense schemes, based on the protocol and application scenarios.**

Keywords: protocol system; network security; fingerprinting; testing; extended finite state machine; online minimization

I. Introduction

Network protocol system fingerprinting refers to the process of identifying specific features of a network protocol implementation by analyzing its input/output behaviors. Usually these identifiable features may reveal specific protocol versions, vender information, and configurable parameters, and can be stored as the "fingerprint" for matching and comparison. While the original purpose was to identify remotely what Operating System is running on the target host, the applications of fingerprinting techniques nowadays cover a much wider range of areas. It has been shown by the prevalent fingerprinting tools that implementations of most key Internet protocols, such as ICMP, TCP, TELNET and HTTP, can all be targets of fingerprinting [4] [18] [22]. It is interesting to note that fingerprinting techniques by themselves are not necessarily associated with unwelcome behaviors. Network administrators can use remote fingerprinting to collect information to facilitate management, and IDS (Intrusion Detection System) can capture the abnormal behaviors of attackers or worms by analyzing their fingerprints [20].

On the other hand, fingerprinting has been recognized as one of the major threats to cyber-infrastructure security [4] [21]. The main concern is that successful fingerprinting may facilitate attacks, which exploit the vulnerability of certain implementations. In practice, different protocols on one host usually have relations (e.g. from same vendor), which will give attackers more information once the identity of any protocol deployment is revealed. For instance, most reported web server security flaws are operating system specific, while an operating system distribution is also correlated with a specific TCP stack implementation. Therefore, it is convenient for the attacker to identify operating system version first using an active TCP fingerprinting (without involving web server), and then launch the more sophisticated attack on the target web server. Since fingerprinting is lightweight and can be obtained without triggering the Intrusion Detection System, attackers usually prefer to identify the target implementation by fingerprinting first in order to devise damaging attacks. Beside the malicious intruders, commercial advertisers can take advantage of the fingerprints of the hosts of their interest. This is undesirable for some sensitive systems because such implementation details are proprietary.

The presence of protocol system fingerprint is due to a basic fact that most network protocols are not specified completely and deterministically [14]. As a result, there is no unique conforming implementation. This nondeterminism in protocol specification can be from explicit statement of optional features and designer's choices, or from the unspecified behaviors under certain circumstances. In the latter case the implementer has the freedom to decide the response to an unspecified input, which, for instance, could possibly be an error message or no response at all. Given different valid implementations, the goal of fingerprinting is to identify one of them by analyzing the input/output behaviors of an implementation, which is often modeled by a "black-box". Existing methods for obtaining fingerprints can be active or passive [19]. In active fingerprinting process the tester (attacker or administrator) chooses predetermined input sequences for probing the target host, whereas in passive fingerprinting the tester can only observe a trace of input/output messages from the target host without disrupting its normal operations. In general, active approaches are more effective because the tester is capable of selecting "distinguishing" inputs based on the knowledge of the protocol specifications. However, passive approach has the advantage that the target host is completely unaware that it is being fingerprinted.

Most of the recent work about protocol fingerprinting has been focused on the development of software tools for both retrieving the fingerprint (actively and passively) and defending against malicious fingerprinting. While many such tools have demonstrated significant practical value, these ad-hoc approaches have certain limitations. First, as the number of network protocols keeps increasing, we need a general method that is suitable for analyzing most, if not all, of them. Second, most of the current fingerprinting methods use fairly short probing sequences; these simple fingerprints could be erased easily. As we will see in section IV, in general, discovering a fingerprint may need an arbitrarily long probing sequence. Third, future protocol specifications may become too complex for human engineers to analyze manually, and will need automated methods and tools. Finally, with the current ad-hoc fingerprinting methods it is difficult to conduct rigorous proof about the validity and effectiveness of the fingerprinting experiments.

We propose a formal approach for the design, analysis and experiments of protocol fingerprinting. To the best of our knowledge this is the first published work of applying formal methods to protocol fingerprinting. The following are our main contributions. We introduce parameterized extended finite state machine (PEFSM) to formally model protocol specification and candidate conforming implementations. With the classical finite state machine theory this formal approach contributes to a deeper understanding of the nature of protocol fingerprinting. We formally define and categorize fingerprinting problems. Given a finite set of possible candidate implementations modeled by a set of deterministic PEFSMs, we present efficient algorithms for active and passive fingerprinting, and analyze their complexity. Particularly, for active fingerprinting, we design and implement efficient algorithms based on separating sequences of different implementations, using an on online minimization process. This approach does not require complete expansion of PEFSM model and has optimal complexity. We report our results on operating system fingerprinting and TCP congestion control scheme fingerprinting experiments. We also consider countermeasures against malicious fingerprinting. We define a formal model based on the I/O trace of implementations and discuss the feasibility of scrubbing and camouflage approaches for hiding protocol system fingerprints against malicious attackers.

II. Related Works

Protocol fingerprinting works can be dated back to the simple forms of remote operating system (OS) detection. Its task is to determine which OS is running on a remote host by exploiting the unique behavior of various protocols such as TCP and ICMP. Nmap [22] is one of the most popular OS fingerprinting tools. It provides nine special testing packets to determine more than 1000 different versions of operating system. Some other tools apply probabilistic approach to active or passive OS fingerprinting [3] [4], producing a guess rather than a definite conclusion. The author of [18] provides an introduction to identifying popular web servers using an HTTP fingerprinting tool. Its idea of requesting a special (nonexistent) webpage and observing the error message is very similar to Nmap, and in general this could be applied to any application level protocols as well [1].

While OS fingerprinting relies largely on analyzing the behavior of a TCP stack, TCP fingerprinting itself has been a very popular topic [5] [16]. Identifying certain parameters of TCP implementations, such as initial congestion window (ICW) and retransmission timeout (RTO) value can contribute to monitoring and measurement of network performances. One interesting problem is to determine the congestion control algorithm implemented on a TCP stack. In [15] the design of TBIT tool is presented, which is effective in identifying TCP Tahoe, Reno, NewReno, and RenoPlus algorithms, as well as the support of SACK and ECN options. Technically, this is a much harder fingerprinting problem because it requires expertise of TCP to design an effective probing test. For instance, the author of [6] suggests that the difference among NewReno and Reno will be discovered only when multiple packets are dropped within the same congestion window. TBIT tool follows this observation and provides a test involving 20 input data segments.

Various schemes have been proposed to defend against malicious fingerprinting. Since tools like Nmap use very short (usually one packet) tests, it is easy to “patch” the host so that the response to these tests will not reveal its identity [17]. For this reason Nmap has become much less effective now. In [21] remote intrusion based on TCP fingerprinting is studied and the authors design a protocol scrubber that transparently modifies network flow and removes the ambiguity. The Honeynet method proposed by [13] uses a different approach where a network of thousands virtual hosts is emulated and each of them can provide a different version of services to distract fingerprinting tools. To the best of our knowledge, there are no general formal/automated techniques for defending against fingerprinting.

The nature of protocol system fingerprinting problem makes it closely related to protocol testing and identification. There is an extensive literature on applying formal methods to both active conformance testing and passive testing [8] [12]. Formal approaches have been proven to benefit a great deal in testing large and complex protocols [12]. However, fingerprinting problem is different than testing problem in that here we have many rather than one correct implementation. It is also different than protocol identification and learning problems [2] [10] because in the latter we do not have knowledge about the protocol specification to begin with, whereas for fingerprinting problem we have a (nondeterministic) specification, which is equivalent to a class of multiple and in the worst case exponentially many deterministic specifications [14]. Therefore, we need to design different algorithms for fingerprinting than that for testing.

III. A Formal Model

Finite state machine models have been proven to be succinct in modeling various network protocols. A protocol specification usually contains both control portion and data portion; therefore we use a parameterized extended finite state machine (PEFSM) to model both:

Definition 1 A Parameterized Extended Finite State Machine (PEFSM) is a 6-tuple M=<S, s_{init}, I, O, X, T>, where

1. S is a finite set of states;
2. s_{init} is the initial state;
3. $I = \{i_0(\vec{v}_0), i_1(\vec{v}_1), \ldots, i_{P-1}(\vec{v}_{P-1})\}$ is the input alphabet of size P; each input symbol i_k $(0 \le k < P)$ carries a vector of parameter values $\vec{v}_k$;
4. $O = \{o_0(\vec{w}_0), o_1(\vec{w}_1), \ldots, o_{Q-1}(\vec{w}_{Q-1})\}$ is the output alphabet of size Q; each output symbol o_k $(0 \le k < Q)$ carries a vector of parameter values $\vec{w}_k$;
5. X is a vector denoting a finite set of variables with default initial values;
6. T is a finite set of transitions. For $t \in T$, $t = < s, s', i(\vec{v}), o(\vec{w}), P(\vec{X}, i(\vec{v})), A(\vec{X}, i(\vec{v}), o(\vec{w})) >$ is a transition where s and s' are the start and end state of the transition, respectively; i and o are the input/output symbols with parameters; $P(\vec{X}, i(\vec{v}))$ is a predicate of the variables and input parameters; the action $A(\vec{X}, i(\vec{v}), o(\vec{w}))$ is an operation on the variables, based on the current variable values, input and output parameter values.

In PEFSM we use parameterized input and output symbols to model the critical content of data packet. For instance, TCP data packets carry Sequence Number and Acknowledgement Number as parameters. PEFSM model has the same computing power as Turing Machine, and it can be used to appropriately model network protocols. The semantic of PEFSM follows the classic EFSM [8] [12] and we do not repeat it here.

Fig.1 shows part of the PEFSM model of a simplified TCP Tahoe implementation. It contains five states: initial state (SYN), slow start (SS), congestion avoidance (CA), retransmission (REX) and finish (Fin). The most important two input/output symbols are *PKT*[*x,y*] and *ACK*[*x*], where *PKT* is a segment of data packets parameterized by starting and ending sequence numbers; and *ACK* is acknowledgement packet parameterized by acknowledgement number. Table I shows one transition of this model. Upon input *ACK* the transition takes the machine from state SS to CA. The guard checks that the acknowledge number is valid and the useful window is larger than zero. The action of this transition reduces the threshold and reset *cwnd* according to the TCP RFC. The output of this transition is *PKT* parameterized by the current useful window. Following [3] we refer to this oversimplified implementation as NoFR later in the paper.

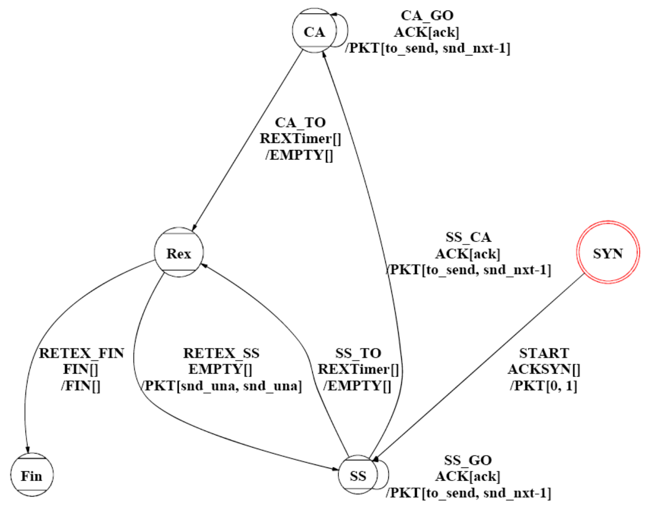


Fig. 1. PEFSM model of TCP NoFR (State variables, guards and actions of transition are omitted)

TABLE I Example of PEFSM Transition in NoFR (Fig.1) machine

Name	*SS_CA*
Start State	*SS*
End State	*CA*
Input	*ACK(ack)*
Output	*PKT(to_send,snd_nxt-1)*
Guard	*(cwnd>=ssthresh && (ack > snd_una) && (ack<=snd_max) && (ack+cwnd> snd_nxt))*
Action	*ssthresh=cwnd/2; if (ssthresh<2) ssthresh=2; cwnd = 1;snd_nxt = snd_una+1;ack_dup = 0;*

A configuration *cfg* of PEFSM *M* is a combination of its state and variable values: $cfg = < s, \bar{x} >$, where $s \in S$. Denote the set of all configurations as *CFG*. A test sequence $seq \in I^*$ is a valid input sequence. A PEFSM is deterministic if any configuration enables at most one transition for any input. Note that it is still possible that there are multiple transitions from a state so long as the guards of them are mutual exclusive. We use deterministic PEFSM to model protocol implementations. The output from a deterministic PEFSM is also deterministic, that is, for an input there is only one output response. The response of machine *M* to a single input is defined by the transition, and we can similarly define the response of *M* to a sequence of inputs. From configuration $cfg_{(0)} = < s, \bar{x} >$, we say machine *M* generates output sequence $O_oO_1 \ldots O_{L-1}$ upon input sequence *seq* = $I_oI_1 \ldots I_{L-1}$ of length *L* if and only if: there exists a sequence of transitions $t_ot_1 \ldots t_{L-1}$, such that, for $0 \le k < L$, t_k has the corresponding input and output symbols I_k and O_k; the parameters of I_k enable the guard of t_k; and the action of t_k updates the configuration from cfg_k to $cfg_{(k+1)}$. Consequently, we can define an output function: $\lambda_M : I^* \rightarrow O^*$ as: $\lambda_M(seq) = O_oO_1 \ldots O_{L-1}$ if *M* generates output sequence $O_oO_1 \ldots O_{L-1}$ upon input sequence *seq* from its initial configuration. Note that we are only interested in the outputs of a machine from its initial configuration, since most protocol implementations have "reset" capability [12] to take the machine to its initial state before conducting an experiment.

To formally define the fingerprinting problem we need to model a set of valid implementations that all conform to a specification, and we call them a Candidate Group.

Definition 2 Given a candidate group of implementation machines, $C = \{M_1, M_2..., M_k\}$, a test sequence $seq \in I^*$ separates M_i and M_j if $\lambda_{Mi}(seq) \neq \lambda_{Mj}(seq)$. A fingerprinting set F for a candidate group C is a set of test sequences, such that for each pair of machines in C, F contains a sequence that separates them. A distinguishing fingerprint of a particular machine M_i is an input sequence that separates M_i from all the other machines in the candidate group.

Without loss of generality we assume that each machine in a candidate group is minimized and that no two machines are equivalent. From the known results in FSM testing theory [12], for any given candidate group there is a fingerprinting set of no more than k test sequences where k is the cardinality of the candidate group. In next section we shall discuss the generalization of this process to PEFSM. Note it is not necessary that each machine has a distinguishing fingerprint.

We have an Implementation machine Under Fingerprinting (IUF) that is supposed to be one of the machines in a candidate group, and we take it as a "black-box", i.e., we can only observe its I/O behaviors without knowing its internal structure. We want to identify it among the candidate group.

The fingerprinting test process can be active or passive. The goal of active fingerprinting is to construct a fingerprinting set for a given candidate group. As for passive fingerprinting, we are given a trace of I/O sequences from IUF for identifying it among the given candidate group. That is, we ask the question whether the given trace is a distinguishing fingerprint for the IUF.

A candidate group can be modeled simply as a set of deterministic PEFSMs, and a pair wise analysis can be easily devised. A more generic approach is to specify all the possible IUF in a candidate group by using one nondeterministic PEFSM. Every derived machine [14] of that specification machine is considered a possible candidate. This is often encountered in practice.

TABLE II CLASSIFICATION OF FINGERPRINTING PROBLEMS

	Active Experiment	Passive Experiment
Candidate Group as a set of deterministic PEFSMs	*Problem 1*	*Problem 2*
Candidate Group as a nondeterministic PEFSM	*Problem 3*	*Problem 4*

In summary, we have four cases for fingerprinting as shown in Table II. In this paper we will focus on problem 1 and 2. We present efficient algorithms and analyze their complexity. Problem 3 and 4 are harder since the cardinality of a candidate group is much larger when modeled by a nondeterministic machine [14]; in the worst case there can be exponentially many candidate machines.

IV. ACTIVE AND PASSIVE FINGERPRINTING ALGORITHMS

For fingerprinting Problem 1 and 2, we have a candidate group $C = \{M_1, M_2..., M_k\}$ of PEFSMs. All the candidate machines have the same input and output alphabet I and O with $P = |I|$ input symbols. We present active fingerprinting algorithm first.

A. Active Fingerprinting Algorithm

We construct a fingerprinting set F for a candidate group C as follows. First consider the simple case where all the machines are deterministic finite state machine (FSM) without any variables and parameters and all the machines have no more than n states. Select arbitrary two machines M_i and M_j from C, and construct a separating sequence $seq \in I^*$ such that $\lambda_{Mi}(seq) \neq \lambda_{Mj}(seq)$. Such a sequence always exists since all the machines are minimized and in-equivalent [12], and it takes time $O(Pn\log n)$ to construct. Depending on the different output sequences upon input sequence seq, we partition C into at least two subgroups (one containing M_i and the other containing M_j) We then repeat the same process on each subgroup until the candidate group C is partitioned into k subgroups, each of which is a singleton set. The total number of separating sequences constructed is no more than k, and they consist of a fingerprinting set F for C. Obviously, the total cost is $O(Pkn\log n)$ where P is the number of input symbols, k the number of machines in the candidate group, and n the maximal number of states of all the machines in the group.

However, for the general PEFSM M_i and M_j from C, the algorithm to construct a separating sequence is more involved. A straightforward approach is to compute the reachability graph (FSM) [6] of each of them and then calculate the separating sequence as for FSM. Assume that M_i and M_j each has no more than n states and their reachability graph has no more than N states where $n<<N$. In practice N is large if not infinite and it is often impossible to handle. One might want to minimize them first; however, it is often also impossible. Online minimization [8] was proposed. Suppose that the minimized reachability graph has $N^*<<N$ states. Then an online algorithm based on state block splitting constructs a minimized reachability graph with a cost $O(PN^{*2})$.

For fingerprinting set construction, we can even do better, i.e., there is no need to construct the whole minimized reachability graph for the following reason. As in [11], we can split the blocks of states where no two blocks contain equivalent states while states in a same block may be in-equivalent before the termination of the minimization algorithm. With this observation we can construct separating sequences between the machines in the process of state block splitting. If we can successfully find a separating sequence for two machines even before completing their minimization, then we can terminate the process; this separating sequence remains valid till the splitting completes. Therefore, there is no need to continue until the minimized machines are fully constructed. We have: $n \ll N^{**} \ll N^{*} \ll N$ where n is the original specification machine size, N^{**} the machine size when the fingerprinting set is construct before block splitting is completed, N^{*} the minimized reachability graph size, and N the reachability graph size. We now show that we can construct fingerprinting set in time and space polynomial in N^{**}.

For PEFSM we use the notion of *block* to represent a set of configurations. Initially each state in PEFSM is a block that contains all configurations in that state. Define the *quotient graph* of a PEFSM M as follows: each block is a node, and there is an edge labeled t between blocks b_1 and b_2 if and only if there are configurations $cfg_1 \in b_1$ and $cfg_2 \in b_2$ such that transition t takes cfg_1 to cfg_2 in M. An edge t between b_1 and b_2 is *stable* if t takes all configurations in b_1 to some configuration in b_2. An edge t is *infeasible* if t takes no configuration in b_1 to b_2. Infeasible transitions are transient result of block splitting. An edge is *unstable* if it is neither stable nor infeasible. Cleary if the quotient graph contains only stable transitions then it is exactly the minimized reachability graph of the original PEFSM. Finally we use $t(b)$ and $t^{-1}(b)$ to denote the image and reverse image of b under transition t.

Algorithm 1
(Online Separating Sequence of two PEFSMs)

```
Input: PEFSM $M_1$ and $M_2$;
Output: Separating Sequence $SEQ$;
begin
1.  partition each state in $M_1$ and $M_2$ into blocks such that
    each block enables the guard of one transition;
2.  generate quotient graph $G_1$ and $G_2$ for $M_1$ and $M_2$;
3.  $checksize := max\{|G_1|, |G_2|\}$;
4.  while (true)
5.    if ($max\{|G_1|, |G_2|\} >= checksize$)
6.      find separating sequence $SEQ$ for $G_1$ and $G_2$;
7.      if (found)
8.        return $SEQ$;
9.      $checksize := 2 \times checksize$;
10.   find an unstable edge $t(c \rightarrow b)$ in $G_1$
11.   if (found)
12.     $c' := c \cap t^{-1}(b)$;
13.     split $c$ into two blocks $c'$ and $c - c'$;
14.     remove all resulting infeasible transitions;
15.   repeat 10-13 for $G_2$ and split one block
end
```

Lines 1-2 prepare the quotient graph by initializing the blocks. Since M_1 and M_2 are deterministic, transitions from a same state are mutually exclusive; therefore the operation in line 2 is well defined. Lines 4-15 contain a loop, each iteration of which will split one block for each machine. Line 6 takes two quotient graphs G_1 and G_2 and tries to find a separating sequence. We treat G_1 and G_2 as two nondeterministic finite state machines and apply the classical state partition algorithm (omitted due to limit of space). It is important to note that this does not need to be done in all iterations, but only when the number of blocks is doubled. We assume the candidate group does not contain equivalent PEFSM, therefore separating sequence exists for any M_1 and M_2 and the loop will terminate when the size of G_1 or G_2 reaches N^{**}.

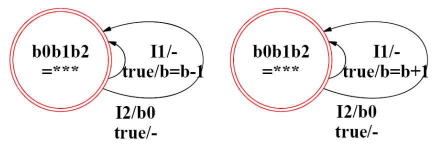


Fig. 2(a). G_1 and G_2 before splitting

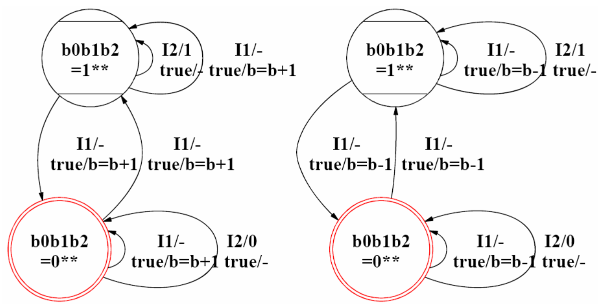


Fig. 2(b). G_1 and G_2 split to two blocks each

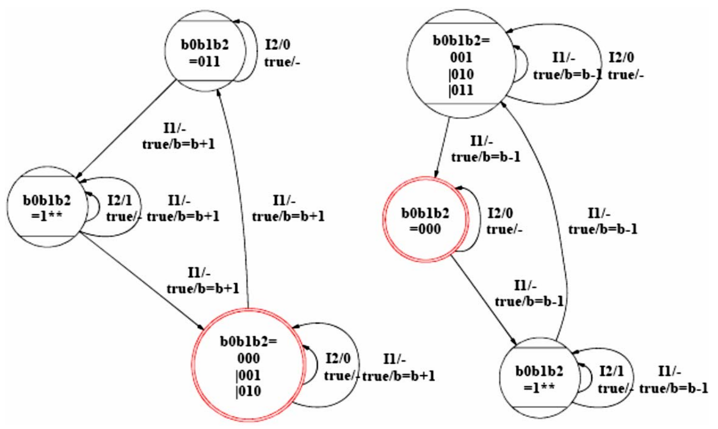


Fig. 2(c). G_1 and G_2 split to three blocks each. I_1I_2 is a separating sequence of the two because it outputs 0 on G_1 and 1 on G_2

Now we use an example to illustrate algorithm 1. We have two simple PEFSM M_1 and M_2. Each of them has only one state, and the variables include three bits $b = b_0b_1b_2$ initialized to all 0. There are two input symbols. I_1 alters b in some manner and I_2 makes the machine output its first bit b_0. Fig. 2(a) shows the two machines. We can tell that M_1 and M_2 respond to I_1 differently; M_1 increments the value of b by 1 while M_2 decrements it. The size of reachability graph of both machines is 8, corresponding to 8 different combinations of $b_0b_1b_2$. In fact, the minimum reachability graph also has 8 states, i.e. $N^{*}=N=8$. Now we want to find a separating sequence of them by gradually splitting the quotient graphs G_1 and G_2. First notice that transitions of I_2 will never be unstable because they do not change state, therefore we only split transitions of I_1. Before algorithm 1 starts we first split the only state of both machines into two blocks (line 1) according to the output on I_2. The purpose is to eliminate the parameters from the outputs so that we can treat outputs as separate symbols. Fig.2(b) shows the graphs after this step. Now *checksize* is 2. In the first iteration of the loop, we will first try to find a separating sequence of G_1 and G_2 but they are not separable now. We set *checksize* to 4. At line 10 suppose the unstable transition from state "b0b1b2=0**" is picked for both machines, and we split this state. The resulting graphs are shown in Fig. 2(c). Note here we omit the steps of removing infeasible transitions. Now G_1 and G_2 are already separable by sequence I_1I_2. However, because of *checksize* algorithm 1 will wait till next iteration to have 4 blocks.

In order to analyze the complexity of this algorithm, we need to assume some basic operations of blocks. The representation and operation for block varies for different protocols. For clarity, we assume a constant time operation for block intersection, difference, inversion and emptiness test. For the initial machine size before splitting n and P, N^{**}, and N^* defined as in the algorithm, assume $N^{**}=n\times 2^k$, the loop for lines 4-15 has at most N^{**} iterations. Lines 1-2 take time $O(N^{**}\times P\times C)$, where C is the constant cost for block operation. In each iteration line 10 has to check every block and every input symbol, therefore the total time this algorithm spends in lines 10-14 is bounded by

$$\sum_{i=n}^{N^{**}} O(i\times P\times C)=O(N^{**2}\times P)$$

The cost of line 6 is $O(n'^2\times P)$, where n' is the size of G_1 and G_2, hence the total time spent on this operation is

$$\sum_{i=0}^{k} O((n\times 2^i)^2\times P)=O(N^{**2}\times P)$$

All added up together, the time complexity of the proposed algorithm is $O(PN^{**2})$. In the worst case $N^{**}=N^*$ and our algorithm costs the same as the classical online minimization algorithm but it may terminate with a separating sequence long before the splitting is completed, hence on the average it performs much better.

Proposition 1 Applying Algorithm 1, the time complexity of calculating a separating sequence of two PEFSM is $O(PN^{**2})$, and the worst case time complexity is $O(PN^{*2})$ where P is the number of input symbols, N^{**} and N^* are the number of blocks when the separating sequence is obtained and that when the online minimization procedure is completed, respectively.

Note that in practice it is very likely that $N^{**}<<N^*$.

Now we use this algorithm to construct fingerprinting set. The main idea is straightforward. We maintain a partition of candidate group. Starting from an empty set, we keep adding separating sequences to refine the partition until all sets in the partition are singletons. The procedure is summarized in algorithm 2 below.

Algorithm 2 (Fingerprinting Set for Candidate Group)

```
Input: candidate group $C = \{M_1, M_2..., M_k\}$;
Output: fingerprint set $F$;
begin
1.  $F := \{\}$;
2.  partition :=$\{\{1,2,...,k\}\}$;
3.  while (partition.size < k)
4.    find machine $M_i$ and $M_j$ in the same set;
5.    calculate separating sequence SEQ for $M_i$ and $M_j$;
6.    foreach set S in partition do
7.      split S according to the output of SEQ ;
8.      $F := F \cup \{SEQ\}$;
9.  return F;
end
```

During every round in lines 3-8 one test is added to the fingerprinting set. Since there are at most k iterations, the total cost of this algorithm is $O(kPN^{**2})$. Given N as the size of the largest minimum reachability graph in $\{M_1, M_2..., M_k\}$, and the resulting set contains no more than k sequences.

Proposition 2 Applying Algorithm 2, the time complexity of constructing a fingerprinting set for candidate group of size k is $O(kPN^{**2})$, and the worst case time complexity is $O(kPN^{*2})$ where P is the number of input symbols, N^{**} and N^* are the number of blocks when the separating sequence is obtained and that when the online minimization procedure is completed, respectively.

Note again that in practice it is very likely that $N^{**}<<N^*$.

B. Passive Fingerprinting Algorithm

In passive fingerprinting we have an explicitly modeled candidate group C and a trace T. The trace contains a vector of input symbol and a vector of output symbol. Since in the PEFSM model we assume each transition only generates one output, the vectors of input and output have the same length. The goal of passive fingerprinting experiment is to decide the machines that could possibly generate T. If T could only be the trace from one machine, then it implies a fingerprinting sequence and we successfully identify the implementation.

To achieve this goal, we follow a passive testing paradigm [12]. We scan the trace only once and conduct testing with all the candidate machines concurrently; whenever all except but one machine are eliminated, we can terminate the algorithm. The key structure needed to maintain during this process is the state uncertainty for each machine. State uncertainty represents the knowledge we have about the current state of a machine M. If M is deterministic, then the size of uncertainty will not increase. There are many different ways to maintain state uncertainty. Here we calculate the reachability graph for each machine first and maintain the set of states in the resulting finite state machine (corresponding to set of configurations in the original PEFSM). This is feasible if we can obtain the finite reachability graph, but not very efficient when the graph is large. In [8] we showed how to maintain uncertainty in EFSM.

Our passive fingerprinting algorithm 3 works as follows. First we calculate the minimal reachability graph of each machine M_i. Then we inspect each pair of I/O symbol starting from the first one in the trace. Initially assume that all machines could possibly be the IUF. The state uncertainty is maintained for each M_i, initialized by all its states (line 4). For each I/O pair passively observed, we update each uncertainty by applying the transition function to the current uncertainty. Particularly, we are interested in the event that the state uncertainty of a machine becomes empty (line 12), which implies that this machine could not be the IUF otherwise there is a contradiction. When the whole trace is processed, the remaining machines are the possible IUF (line 16).

Algorithm 3 (Passive Fingerprinting)

```
Input: candidate group $C = \{M_1, M_2, \ldots, M_k\}$,
trace T=$<<I_0,O_0>,<I_1,O_1>,\ldots,<I_{L-1},O_{L-1}>>$;
Output: possible candidate set $PC$;
begin
1.  $PC := C$;
2.  foreach $i$ in $[1..k]$ do
3.    calculate minimized reachability graph $G_i$ ;
4.    $uncertainty[i]$ := all states in $G_i$;
5.  $id = 0$ ;
6.  while ($PC.size > 1$ and $id < L$)
7.    foreach $M_i$ in $PC$ do
8.      $new\_uncertainty := \{\}$;
9.      foreach $t <S_{src}, S_{dst}, I_t, O_t>$ in $G_i$ do
10.       if ($S_{src} \in uncertainty[i]$ and $I_{id} == I_t$ and $O_{id} == O_t$)
11.         $new\_uncertainty = new\_uncertainty \cup S_{dst}$ ;
12.     if ($new\_uncertainty == \{\}$)
13.       $PC := PC - \{M_i\}$ ;
14.     else $uncertainty[i] := new\_uncertainty$ ;
15.   $id := id + 1$;
16. return $PC$;
end
```

Now we analyze the complexity of this algorithm. Lines 6-15 contain a loop that takes one I/O pair at each step. Lines 7-14 inspect each candidate machine and update the state uncertainty. If the size of a reachability graph is no more than N, then Lines 7-14 terminate in time $O(k \times N^* \times P)$ because the number of transitions that must be considered is less than $N^* \times P$. Since the trace contains L packets and in the worst case they all need to be inspected, the cost of loop 6-15 is $O(L \times k \times N^* \times P)$. As mentioned in the previous section, the cost of calculating minimized reachability graph is $O(N^{*2} \times P)$, therefore the total cost of the algorithm is $O(L \times k \times N^* \times P + k \times N^{*2} \times P)$. When the algorithm terminates, the resulting candidate set may contain more than one machine. In this case we can not identify a single implementation; instead we eliminate some impossible ones. On the other hand, if the algorithm terminates with only one candidate, then it is the implementation, and we can also construct a fingerprinting sequence starting from the initial state by back-tracking the uncertainties.

Proposition 3 For a candidate group of size k, the worst case time complexity of passive fingerprinting on a trace of length L is $O(L \times k \times N \times P + k \times N^2 \times P)$ where P is the number of input symbols and N the size of the reachability graph traced.

V. Case Study

A. Active Fingerprinting using NMAP Tests

Nmap [22] is the most popular active OS fingerprinting tool. It identifies a TCP stack implementation by using nine test sequences: T_{seq} is for TCP initial sequence number prediction; T_1 to T_7 are seven specially constructed TCP packets; and PU is for probing unreachable port. In the fingerprint database Nmap stores the encoded response to those test sequences of more than 1300 implementations. Those implementations are classified into 33 categories, spanning from general purpose Operating Systems to VoIP phones. In this section we extract one candidate group for each category from the fingerprint database and calculate the fingerprint set. Note that the set of all nine tests forms a fingerprint set of most implementations but for some category not all tests are needed.

Technically, the test T_1 to T_7 and PU are quite different from T_{seq} since the design of T_{seq} and encoding scheme of its output are significantly more involved. For simplicity we still treat T_{seq} as a single "virtual" input symbol. Given this assumption, the modeling of candidate machines is very concise. In fact, they are all FSM. Fig. 3(a) and (b) show the model of Solaris 9.0 and CISCO IOS 11.0 operating system respectively.

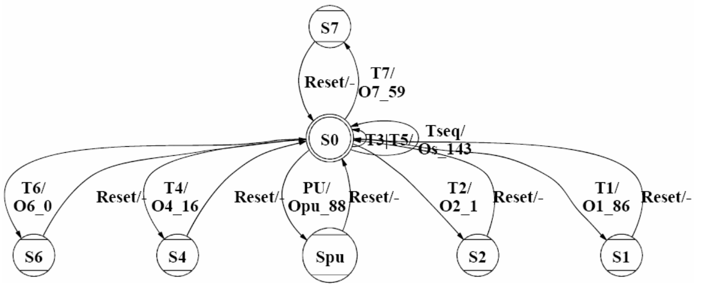


Fig. 3(a). Nmap candidate machine model of Solaris 9.0

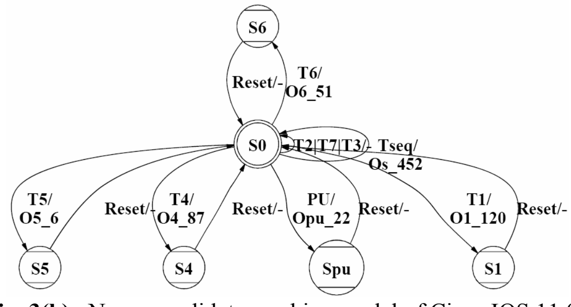


Fig. 3(b). Nmap candidate machine model of Cisco IOS 11.0

Models of Nmap candidate machines have the property that if separating sequence of M_i and M_j exists, then it has length 1. This is due to the large number of different outputs for the test inputs. Devising such inputs certainly requires insight and intelligence, and once those powerful input symbols are known both fingerprinting experiments and potential defense scheme become straightforward. For example, it is obvious to see from Fig. 3 that all inputs except T_3 could be used as separating sequence for the two machines. We use algorithm 2 to calculate fingerprinting sets for the largest 10 categories and show the result below in Table III. In general, larger candidate group requires more tests. Two of the categories do not have an exact fingerprint set (shown by * in the last column) since they include implementations with only minor version differences which will not be distinguished by any Nmap test.

TABLE III FINGERPRINTING SETS IN NMAP

Category	Size	Fingerprinting Set
General Purpose	651	*{Tseq,T1-T7,PU}**
Broadband Router	112	*{Tseq,T1,T2,T3,PU}*
Router	105	*{Tseq,T1,T2,T3,PU}*
Printer	73	*{Tseq,T1-T7,PU}**
Firewall	61	*{Tseq,T1,T2,T3,T4,PU}*
Switch	48	*{Tseq,T1,T2, PU}*
Terminal Server	43	*{Tseq,T1,PU}*
WAP	34	*{Tseq,T1,PU}*
Print Server	27	*{Tseq,T1,PU}*
Webcam	14	*{Tseq,PU}*

B. *Passive Fingerprinting on TCP*

In this section we present our experiment on fingerprinting TCP congestion control algorithms. This is technically a much harder problem than OS fingerprinting using special TCP and ICMP packets as both the model and the test sequences are much more involved. Congestion control schemes implemented in the production TCP stacks include standard TCP Tahoe, Reno, NewReno and so forth. Moreover, it has been reported that some earlier version of Operating Systems deploy nonconforming congestion control scheme such as Tahoe without fast retransmission (NoFR) [15].

We use PEFSM to model four different implementations: M_{NF} (NoFR), M_T (Tahoe), M_R (Reno), and M_{NR} (NewReno), and then conduct a passive fingerprinting experiment. We have already seen NoFR model in section III. In a similar way we model the other three. Two of them, Tahoe and NewReno are shown in Fig. 4(a) and (b). They both have more states than NoFR model. Tahoe implementation uses fast retransmission scheme (FRX) and NewReno implementation uses a variant of fast recovery (FRC) scheme. Note in these models we only have two parameters of output symbol *PKT*, namely the starting sequence number and ending sequence number. This is a simplification for analysis but this will cause some trouble when we model a transition with output of non-continuous packets. For the models in this experiment, we split the transition to handle. However, to model more complicated schemes such as TCP Selective Acknowledgement (SACK), we should add more parameters to the output to allow maximum flexibility.

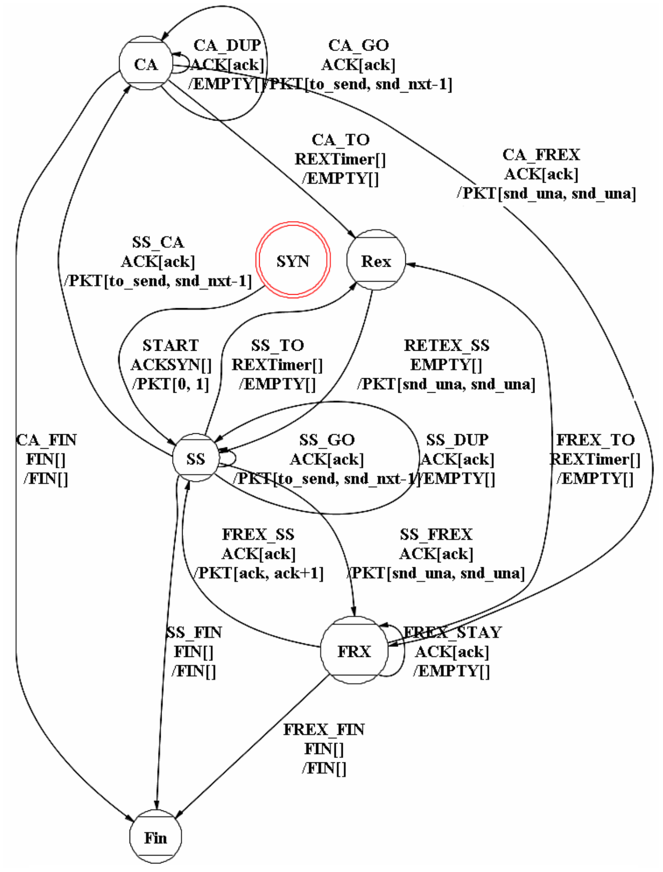


Fig. 4(a). PEFSM model of TCP Tahoe implementation

Another simplification we made is the omission of connection management part of TCP. Since the focus in this experiment is congestion control schemes, we simply use one state (SYN) to represent the initial state and another one (FIN) to represent the connection tear down. It is straightforward to augment these models to include the connection management features.

Recall that from the TCP specifications the major difference between FRX and FRC is that when a packet is lost, FRX brings the machine back to slow start (SS) while FRC inflates the congestion window and continues responding to duplicate acknowledgements. In case of multiple packet loss, Tahoe is capable of retransmitting more than one packet per round trip time but some of them could be unnecessary; Reno and NewReno can only retransmit one packet per round trip time. Those features imply the fingerprint of each, however, as studied in [6], it requires great insight of those implementation to see how exactly the difference is manifested by output packets.

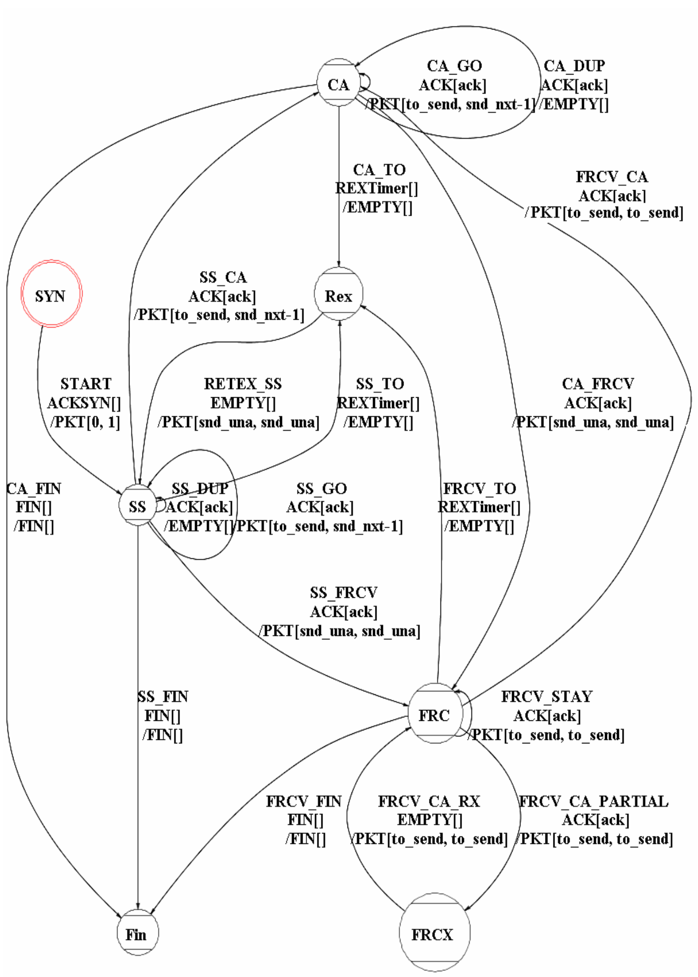

CA
CA_GO ACK[ack] /PKT[to_send, snd_nxt-1]
CA_DUP ACK[ack] /EMPTY[]
CA_TO REXTimer[] /EMPTY[]
FRCV_CA ACK[ack] /PKT[to_send, to_send]
SYN
Rex
SS_CA ACK[ack] /PKT[to_send, snd_nxt-1]
START ACKSYN[] /PKT[0, 1]
RETEX_SS EMPTY[] /PKT[snd_una, snd_una]
SS_TO REXTimer[] /EMPTY[]
CA_FRCV ACK[ack] /PKT[snd_una, snd_una]
CA_FIN FIN[] /FIN[]
SS
SS_DUP ACK[ack] /EMPTY[]
SS_GO ACK[ack] /PKT[to_send, snd_nxt-1]
FRCV_TO REXTimer[] /EMPTY[]
SS_FRCV ACK[ack] /PKT[snd_una, snd_una]
SS_FIN FIN[] /FIN[]
FRC
FRCV_STAY ACK[ack] /PKT[to_send, to_send]
FRCV_FIN FIN[] /FIN[]
FRCV_CA_RX EMPTY[] /PKT[to_send, to_send]
FRCV_CA_PARTIAL ACK[ack] /PKT[to_send, to_send]
Fin
FRCX

Fig. 4(b). PEFSM model of TCP NewReno implementation

We now present our automated process on the models without resorting to any expertise of TCP implementations. Two types of traces are used in the passive fingerprinting experiment. The first type is collected by monitoring regular TCP traffic on internet, and our result shows that those traces normally do not contain distinguishing fingerprint. As already noted in the literature of TCP passive measurement [7], the reason is that most TCP traffic (over 90% of all TCP senders) does not have the interaction pattern that can distinguish those implementations.

In our model, if there is no packet loss, then all four machines will follow the state transition pattern SYN-SS-CA-FIN and the I/O behavior will be the same. Furthermore, if the trace contains one packet loss, then it may distinguish Tahoe and NoFR, while Reno and NewReno still behave the same way.

The second type of trace is collected using TCP Behavior Inference Tool (TBIT). TBIT is essentially an active fingerprinting tool that uses preset strategy and parameters for setting up a TCP connection and simulates packet losses. TBIT aims at characterizing the TCP sender behavior for web servers. Here we use the proposed passive fingerprinting algorithm to validate the fingerprints produced by the tool. Note that these are very special traces that normally will not be observed. The purpose of using them is to illustrate how passive fingerprinting algorithm works.

Both types of traces are captured by Tcpdump program. However, Tcpdump traces are not be directly applicable for our algorithm. In order to translate them to an I/O trace of PEFSM, we develop a packet decoder. The decoder performs the following tasks: (1) Estimate a roundtrip time from the trace and break the trace into sections based on each round trip time. (2) Decode from the packet content the receiver's window (rwnd). (3) Construct one input symbol with parameters using the packets sending from the client to the server. Consecutive incremental acknowledgements are combined, but not duplicated acknowledgements. (3) Similarly construct one output symbol with parameters using the packets sending from the server to the client; multiple packets are combined into intervals (*PKT* [*start*,*end*]). In case they can not be combined, as discussed earlier, we add an extra transition with null input. (4) Detect retransmission timeout and insert a special input symbol to trigger the retransmission.

Table IV shows one run of algorithm 3 on the special trace generated by TBIT. Initially the state uncertainty of each candidate is nonempty. The first four rounds are a typical slow start phase restricted by receiver's window (5 in this case). There is no difference observed. After the duplicated acknowledgement *ACK* [12] is sent four times, we see a fast retransmission without timeout which rules out M_{NF}. Next, *ACK* [15] is a partial acknowledgement, which will make M_T in state SS, M_{NR} in FRC and M_R in CA. In this case M_R can not output *PKT* [15] due to the limit on window size; hence its state uncertainty becomes empty. Similarly, *PKT* [17] will make state uncertainty of M_T empty because M_T can only send one packet. After the whole trace is consumed our algorithm reports NewReno as the only possible implementation, therefore we have verified the fingerprint sequence.

TABLE IV PASSIVE FINGERPRINTING ON A TCP TRACE GENERATED BY TBIT

Decoded Tcpdump Trace		Candidate with Non-empty State Uncertainty
Input	Output	
ACKSYN	*PKT* [0,1]	{ M_{NF}, M_T, M_R, M_{NR} }
ACK [2]	*PKT* [2,5]	{ M_{NF}, M_T, M_R, M_{NR} }
ACK [6]	*PKT* [6,10]	{ M_{NF}, M_T, M_R, M_{NR} }
ACK [11]	*PKT* [11,15]	{ M_{NF}, M_T, M_R, M_{NR} }
ACK [12]	*PKT* [16,16]	{ M_{NF}, M_T, M_R, M_{NR} }
ACK [12]	-	{ M_{NF}, M_T, M_R, M_{NR} }
ACK [12]	-	{ M_{NF}, M_T, M_R, M_{NR} }
ACK [12]	*PKT* [12,12]	{ M_T, M_R, M_{NR} }
ACK [15]	*PKT* [15]	{ M_T,M_{NR} }
-	*PKT* [17]	{ M_{NR} }
ACK [18]	*PKT* [18]	{ M_{NR} }
ACK [19]	*PKT* [19,20]	{ M_{NR} }

VI. Defending Against Malicious Fingerprinting

Up to this point we have discussed the techniques used to fingerprint network protocol systems. Now we take a view from another side. In this section we briefly discuss the techniques used to defeat protocol fingerprinting. As the goal of fingerprinting is to identify a protocol implementation, the goal of a defense mechanism is to hide this information. That is, when a principal conducts an active or passive fingerprinting experiment, the information he gets is insufficient to draw a conclusion or will lead to a wrong conclusion, and ideally this would hold even if the test sequence applied is a fingerprinting set. Toward this goal, a defense system usually involves modification of the I/O behavior of an implementation. Generally speaking there are two methods to hide the identity: scrubbing and camouflage. Scrubbing is used when the fingerprint is caused by some input sequences whose corresponding output is not determined by the specification. Clearly, a communicating peer should not assume any specific output. In this case a scrubber works by modifying the output to such sequences and "erasing" the fingerprint. In contrast, camouflage is used when the outputs to a sequence are specified for each candidate implementation and the communicating peer is expecting one of them. The original output is modified to be that of another candidate implementation. As one could imagine, camouflage is much more expensive than scrubbing because state information needs to be maintained. If the number of fingerprinting sequence is large, this approach is almost identical to redeploying a different protocol implementation.

There is one important principal in the defense of fingerprinting: the modification should be transparent to all regular users. A regular user's behavior is defined in the protocol specification and it must be retained all the time. Note that it is possible that regular user's behavior is distinct for different implementations, as we have seen in the TCP example. Moreover, the peers of our target host may or may not assume a particular type of behavior. These factors to a large extent decide what defense method should be used and what the result would be. We discuss different cases below.

There are various issues of designing and deploying a fingerprint defense system. First of all, it could be deployed online or offline. An offline defense system is a patch to the original protocol implementation that statically changes its response to some input sequences; while an online system is usually installed as a component of the firewall and transparently modifies the inbound and outbound traffic. Moreover, an online defense system could be synchronous, meaning that it will respond immediately after it receives an input packet and no delay will occur. It could also be asynchronous, where a buffer holds a small number of packets. Buffering some packets will help make better decisions of defense; however, it is not always possible to delay the response. Those design details are out of the scope of this paper. [17] and [21] discuss the detail about how a defense system could be implemented on a typical protocol stack. In this paper we focus on the theoretical model of defense and introduce general solutions.

A. A Formal Model

In order to see the nature of fingerprint defense problem, we again use a formal model. Similar to the discussion in section IV, we have a candidate group $C = \{M_1, M_2..., M_k\}$. Define the trace set $T_i \in (I^* \times O^*)$ of implementation M_i as follows. $T_i = \{t \mid t = <Seq, \lambda_{Mi}(Seq)>, Seq \in I^*\}$. In short, T_i is the set of I/O traces that could possibly be generated by M_i. Note if a trace $t \in T_i$, then its prefixes are all in T_i. We define the attacker trace set T_i^a by one of the following two ways. (1) $T_i^a = \{t \mid t \in T_i, \textit{ and } t \notin T_j, \ \forall j \neq i\}$, or (2) $T_i^a = \{t \mid t \in T_i, \textit{ and } \exists j,k \ (j \neq k \textit{ and } t \in T_j \textit{ and } t \notin T_k)\}$. The first one defines an attacker trace as one that distinguishes an implementation from all others in the group, and the second defines it as one that distinguishes any two implementations. Depending on the application environment and the requirement for defense, either of these two could be more appropriate. For both definitions, if $t_1 \in T_i^a$ and t_1 is the prefix of t_2, then $t_2 \in T_i^a$. Finally, a subset of T_i, the user trace set contains the traces that are required by the user and therefore must be included in any implementation. We denote the user trace set of M_i as T_i^u.

Let us consider a scenario of three implementations M_1, M_2 and M_3. Their trace sets T_1, T_2 and T_3 are shown by the Venn diagram in Figure 5. We adapt the first definition of attacker trace, i.e. $T_1^a = T_1 - T_2 - T_3$. Suppose the protected host is running implementation M_1. Initially we have an empty trace, which belongs to every trace set. The goal of a fingerprinting experiment is to apply a sequence leading the trace to any of the attackers trace set, so the implementation could be identified. For instance, in (a) the trace $<I_1I_2, O_1O_2>$ are common to all implementations; however, the next input makes the trace $<I_1I_2I_3, O_1O_2O_3>$ in T_1^a and therefore identifies M_1.

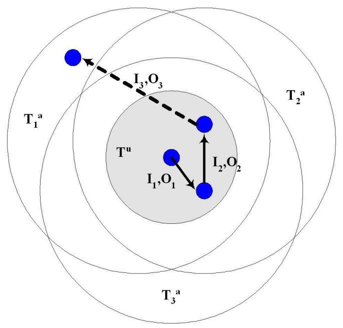
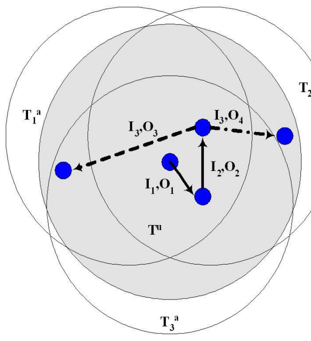
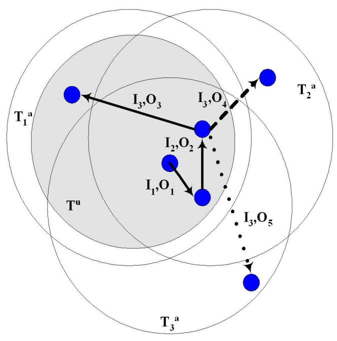


Fig. 5(a) Defense against fingerprinting by scrubbing



(b)



(c)

Fig. 5(b) Defense against fingerprinting by camouflage
Fig. 5 (c) A case with no solution

B. Defense Schemes

To prevent attacker from identifying a machine, usually the defense system has to change the trace. The goal here is either not to produce any trace in attacker trace set, or produce an attacker trace of a different implementation. Based on users trace set we classify the solutions into two different cases.

Case 1 (Common User Trace Set) $T_1^u = T_2^u = \ldots = T_k^u$. The user trace set is the same for all implementation. This implies $T_i^u \subseteq \bigcap_{i=1}^{k} T_i$. This also implies $T_i^u \cap T_i^a = \{\}$. Fig. 5(a) shows this case. This is a very typical case where the specification only defines a unique set of "core" behavior (T^u) shared by all implementations. The trace outside users set is not essential and could be implemented arbitrarily. Scrubbing is effective in this case. We want to modify M_1 to only include the core features and shadow the difference on other traces. Therefore, the goal is to build M_1^* whose trace is T^u, i.e. $T_{1*} = T^u$. The basic strategy is to monitor the incoming sequences, and whenever the next input symbol (in this case I_3) will take the trace into T_1^a, the defense system simply discard the input. This will not affect regular users because $I_1I_2I_3$ is not in T_3^u.

Case 2 (Unique User Trace Set) $T_i^u \neq T_j^u$. The user trace sets are different in this case. This implies $T_i^u \supset \bigcap_{i=1}^{k} T_i$ and also $T_i^u \cap T_i^a \neq \{\}$. Fig. 5(b) shows this case. The grey circle represents the union of all user sets T^u. This case is characterized by the fact that some implementations have unique traces, and the regular user expect the trace from any implementation. Consequently our goal is to modify M_1 to M_1^* whose trace is a subset of T^u and it must not include any trace in T_1^a, that is, $T_{1*} \subseteq T^u$ and $T_{1*} \cap T_1^a = \{\}$ In contrast to case 1, we can not use scrubbing alone to achieve the goal. The first two inputs are the same with case 1. When I_3 is observed, we are not allowed to scrub it because it is a user trace. On the other hand, if we follow the original response of M_1 and output O_3, the resulting trace could be in T_1^a. To defend in this case, we have to use camouflage. The main idea is to pretend to be another implementation. This could be done statically if we decide the disguise implementation beforehand. For instance, in Fig. 5(b) we choose M_2 as the disguise. Hence upon receiving I_3, the new implementation will follow M_2's response, and output O_4. This trace will lead to a wrong conclusion of the fingerprinting algorithm. As the result, the defense system generates a user trace set T_2^u. Note here that camouflage is not always practical. Some times it is imaginable that regular users expect only the trace from T_i^u. This could be because some communicating peers are aware of the special implementation feature of M_i, and they indeed make use of it. Fig. 5(c) demonstrates this scenario. The grey circle represents the user sets T^u and it is also the trace of our modified implementation M_{1*}. Obviously now neither scrubbing nor camouflage is effective. From the graph we can see that upon receiving I_3, we have three alternatives O_3 O_4 or O_5. However, O_4 and O_5 do not belong to an acceptable user trace T_i^u, and O_3 lead the trace to T_i^a.

Instead of deciding the disguise implementation offline, we have another option. We want to confuse the attacker as long as possible. On every input, we inspect the output of the current set of possible implementations and follow the maximum overlapping subset. This heuristic process is repeated until there is only one implementation possible. For example, in Fig. 5(b), when we see I_3, we could follow any of M_1, M_2 or M_3 because at this point the trace is in $T_1 \cap T_2 \cap T_3$. If we follow T_2, we immediately jump into T_2^a. On the other hand, if we follow M_1 and M_3 (they generate same output) the attacker still gets a nondeterministic result. We call the set of possible implementations confusion set and describe the algorithm as follows.

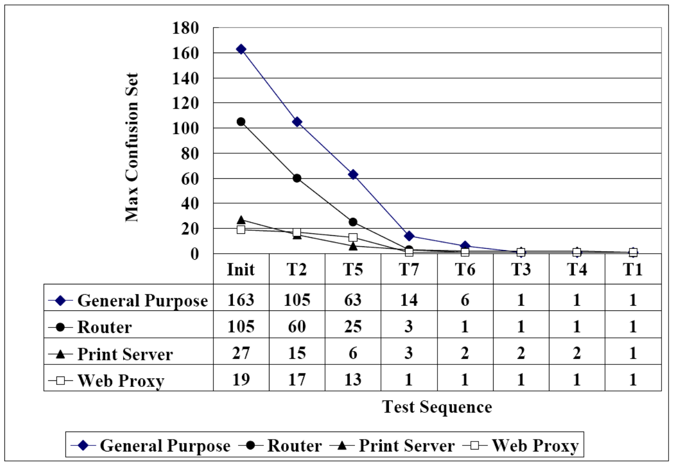
The selection of C is up to the designer as long as it is a subset of all implementations. The size of C directly affects the complexity of the algorithm. Finally, we note that the maximum confusion trace approach is just another alternative, and it is not absolutely better than the static disguise approach because eventually both systems will camouflage as one implementation.

Algorithm 4 (Maximum Confusion Trace)

```
Input: candidate group $C = \{M_1, M_2, \ldots, M_k\}$,
Input sequence $Iseq = \langle I_0, I_1, \ldots, I_{L-1} \rangle$;
Output: Output sequence $Oseq$;
begin
1.    $CS := C$; $Oseq := \langle\,\rangle$; $id := 0$;
2.    while ($id < L$)
3.      foreach $M_i$ in $CS$
4.        $O_i := \lambda_{Mi}(\langle I_o, I_1, \ldots, I_{id} \rangle)$;
5.      partition $CS$ by $O_i$ and select the set of largest
      cardinality $CS_{next}$;
6.      $CS := CS_{next}$;
7.      $Oseq := Oseq + O_i$;
8.      $id := id + 1$;
9.    return $Oseq$;
end
```

Fig.6 shows the result of maximum confusion set algorithm on selected Nmap candidate groups. We use four of the largest categories of Nmap implementations: Router, Print Server, Web

Proxy, and a subset of General Purpose Operating System. We use a particular fingerprinting sequence <T_2, T_5, T_7, T_6, T_3, T_4, T_1>. We could see how the confusion set shrinks as more input symbols are consumed. For the majority of candidate groups after a sequence of length 5, the disguise implementation is pin down to a specific one.



	Init	T2	T5	T7	T6	T3	T4	T1
◆ General Purpose	163	105	63	14	6	1	1	1
● Router	105	60	25	3	1	1	1	1
▲ Print Server	27	15	6	3	2	2	2	1
□ Web Proxy	19	17	13	1	1	1	1	1

Fig. 6. Example of maximum confusion set camouflage algorithm

VII. CONCLUSION

Understanding fingerprinting problems for complex network protocol systems is crucial yet challenging for both network security and management. To complement and enhance current experimental study on fingerprinting, a formal approach is proposed. Parameterized Extended Finite State Machine is used to model protocol specification and implementation. Similar to protocol testing, fingerprinting problems are categorized as active and passive. For a candidate set of deterministic PEFSMs, efficient active fingerprinting algorithm is proposed based on the computation of pair wise separating sequences. This algorithm is based on the online minimization and avoids the generation of complete minimal system. Passive fingerprinting is conducted using concurrent passive testing on all candidate machines. We use the data set of popular fingerprinting tools NMAP and TBIT to verify our model and algorithms. To protect against malicious fingerprinting attack, it is desirable to modify the trace generated by an implementation to confuse and defend against fingerprinting experiments. We discuss the criteria of designing such defense systems and propose two plausible approaches: scrubbing and camouflage.

ACKNOWLEDGMENT

This work was supported in part by the U.S. National Science Foundation (NSF) under grant awards CNS-0403342 and CNS-0548403.

REFERENCES

[1] Amap Project, http://thc.org/thc-amap/.

[2] D. Angluin. Computational learning theory: survey and selected bibliography. Proceedings of the 24th ACM STOC, 351-369,1992.

[3] O. Arkin and F. Yarochkin. Xprobe2 - a 'fuzzy' approach to remote active operating system fingerprinting. http://www.sys-security.com, 2002.

[4] R. Beverly. A robust classifier for passive TCP/IP fingerprinting. In Passive and Active Network Measurement, 5th International Workshop, PAM 2004.

[5] D. Comer and J. C. Lin. Probing TCP implementations. In USENIX Summer, pages 245--255, 1994.

[6] K. Fall and S. Floyd. Simulation-based comparisons of Tahoe, Reno and SACK TCP. Computer Communication Review, 26(3):5–21, July 1996.

[7] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose and D. Towsley: Inferring TCP connection characteristics through passive measurements, Proc. Infocom, 2004.

[8] D. Lee, D. Chen, R. Hao, R. E. Miller, J. Wu, and X. Yin. A formal approach for passive testing of protocol data portions. In 10th IEEE International Conference on Network Protocols (ICNP 2002), IEEE Computer Society, pages 122–131, 2002.

[9] D. Lee, A. N. Netravali, K. K. Sabnani, B. Sugla, and A. John, Passive testing and its applications to network management, Proc. ICNP, October 1997.

[10] D. Lee and K. Sabnani, Reverse engineering of communication protocols, Proc. ICNP, pp. 208 – 216, October 1993.

[11] D. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In STOC '92: Proceedings of the twenty-fourth annual ACM Symposium on Theory of Computing, pages 264–274, New York, NY, USA, 1992.

[12] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In Proceedings of the IEEE, pages 1090–1123, August 1996.

[13] J. Levine, J. Grizzard, and H. Owen, Using honeynets to protect large enterprise networks, in IEEE Security & Privacy, November 2004.

[14] R. Miller, D. Chen, D. Lee, and R. Hao. Coping with nondeterminism in network protocol testing. In 17th International Conference on Testing of Communicating Systems (TestCom), IFIP, 2005.

[15] J. Padhye and S. Floyd. On inferring tcp behavior. In SIGCOMM, pages 287–298, 2001.

[16] V. Paxson. Automated packet trace analysis of TCP implementations. In SIGCOMM, pages 167–179, 1997.

[17] G. Roua and J. Saffroy. IP personality. http://ippersonality.sourceforge.net/, 2001.

[18] S. Shah. An introduction to HTTP fingerprinting. http://net-square.com/httprint/httprint paper.html, 2004.

[19] G. Shu and D. Lee, Defending against internet host fingerprinting - toward an outermost barrier of cyberspace security. In Working Together: Research & Development (R&D) Partnerships in Homeland Security, 2005.

[20] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In Operating Systems Design and Implementation (OSDI), pages 45–60, 2004.

[21] D. Watson, M. Smart, G. Robert Malan, and F. Jahanian. Protocol Scrubbing: network security through transparent flow modification. IEEE/ACM Transaction. Networking, 12(2):261–273, 2004.

[22] F. Yarochkin. Remote OS detection via TCP/IP stack fingerprinting. http://www.insecure.org, 1998.