***SYNSCAN*: Towards Complete TCP/IP Fingerprinting**

Greg Taleck
<taleck@nfr.com>

NFR Security, Inc.
5 Choke Cherry Rd, Suite 200
Rockville, MD 20850

ABSTRACT

A tool for TCP stack testing and TCP/IP fingerprinting (a.k.a. OS detection) is introduced. While tools presently exist to do either OS detection[1, 2] or TCP stack testing[3, 4], the methods they employ are limited by the techniques and analysis performed, sometimes resulting in incorrect results or no results at all. We introduce `synscan`, a tool whose objective is to fingerprint every aspect of a TCP/IP implementation. `synscan` is not meant as a proof-of-concept tool; rather, it is a robust and useful tool which can be used in addition to others for TCP/IP stack testing and OS detection. `synscan` incorporates most of the techiques used by the existing tools and introduces a number of new ones. `synscan`'s primary advantage is that each test begins with a TCP SYN segment (hence the name) to an open port, giving it the ability to test and fingerprint even the most fortified hosts. Conclusive data from large network scans and comparisons to results from existing tools are also reported.

1. INTRODUCTION

Since the IP[5] and TCP[6] protocols have become the definitive end-to-end communication protocols for the Internet, the performance of the Internet depends just as much on the performance of these two protocols as on the hardware that carries their payloads. Since its inception in 1981, many algorithms and enhancements have been devised and developed to improve its performance. In 1988, Jacobson introduced the set of algorithms for TCP now commonly known as Tahoe TCP[7]. 1990 the BSD Reno implementation added Fast Retransmit and Fast Recovery. 1996 saw the development of Selective Acknowledgments in [8]. In 1999, the addition of NewReno[9] saught to improve Reno's Fast Recovery algorithm. The timeline also saw the introduction of window scaling, PAWS and timestamping. Even today, working charters exist to further refine the algorithms used by TCP implementations to improve performance in the Internet. All of these changes, refinements and improvements over the years has led to a very diverse set of TCP implementations actively used by hosts on the Internet today.

Uncovering the differences in these many TCP implementations is the primary goal of the new tool introduced in this paper, `synscan`. It is a robust tool that combines a number of different existing analytical methods as well as some newly discovered ones, providing the user with a wealth of information about the TCP implementation analyzed. Additionally, `synscan` uses this data to provide a TCP/IP fingerprint and a guess as to what operating system and version the remote host might be running.

(Possibly) contrary to popular belief, there are a number of legitimate reasons why someone would want or need to know the TCP/IP fingerprint (and hence the operating system and possibly version) of a remote host. Many system administrators manage large networks connected by many routers, firewalls, VPNs and other devices, with physical locations spanning offices, cities or even countries. Auditing, managing and enforcing corporate and network policies is a difficult task. Tools which make scanning network addresses, performing OS detection, and portscanning of hosts on the network make this task easier. Another valuable use for TCP stack fingerprinting is gathering statistical data on the deployment and distribution of different operating systems and tcp implementations in the Internet, as done in [10]. Finally, previous work has shown that is it possible to resolve network and protocol ambiguities by passive OS fingerprinting[11]. This last reason was the primary motivation for writing `synscan`.

The rest of this paper is divided into the following sections: Section 2 describes in detail the design and architecture of `synscan`. In section 3, the analysis techniques employed by `synscan` are described in detail. Then in section 4, the results of some large network scans and comparisons to other tools are described. Section 5 discusses possible countermeasures that could be used to distort or prevent results, and section 6 goes into current limitiations of the tool. Section 7 discusses previous research and work. Section 8 discusses other ideas that could be implemented, and finally section 9 offers some concluding remarks.

2. DESIGN

The basic design of `synscan` is similar to `tbit`, `activemap` (see section 7), `sting`[4] (and probably many other tools) whereby it uses a kernel interface to firewall off certain TCP ports, opens a BPF device to sniff packets off the wire, and create a basic userland TCP stack to talk to remote hosts. Events are created both by packets received off the wire, by packets injected as directed from its configuration files, and by timers and timeouts also based on the command line arguments and configuration files. `synscan` is about 9000 lines of C code.

2.1 Networking and Portability

`synscan` uses `libdnet`[12] for portable networking operations, such as firewalling ports and sending raw packets. It uses `libevent`[13] for portable event handling, and `libpcap`[14] for its portable BPF interface to sniff packets.

2.2 Configuration

Rather than have all the tests and test functionality hardcoded into the tool itself, `synscan` instead parses them from configuration files using a simple grammar. This was crucial to the discovery of many of the implementation differences discussed below as it allows one to easily add, remove and change the parameters of given tests.

Two configuration files are used for running tests: **synscan.services** and **synscan.conf**. The following two sections described the configuration grammer are by no means complete. They are meant to provide the reader with a general sense of how configuration is done and what parameters can be adjusted[1].

2.2.1 Service Configuration

To give `synscan` the ability to speak application-level protocols, it uses a file *synscan.services* which allows one to specify and configure the payloads and states for the type of service at a given port.

For example, listed below is a configuration for the HTTP protocol.

```
    service http {
        proto tcp;
        port 80;
        segment {
             flags = psh|ack;
             outbound;
             payload = "GET / HTTP/1.0\r\n";
        };
        segment {
             inbound;
             payload = "HTTP/1\.. 200 OK";
        };
  };
```

This configuration specifies that a TCP connection made to port 80 of a host first be sent a TCP segment with the HTTP "GET" command as its payload with the TCP flags PUSH and ACK. The second segment specifies that an inbound segment is expected with its payload matching the regular expression "`HTTP/1\.. 200 OK` ".

Note that this says nothing about *how* the TCP session (connection) is negotiated or how TCP segments are packaged for delivery. This merely configured what data should be sent and received at the application level. Lower level specifications are described in the next section.

[1]Please refer to the manual page with the distribution for complete details on how to customize the session and service configurations.

2.2.2 Session Configuration

The file `synscan.conf` contains directives on how each TCP session should be run. The configuration can specify a number of parameters that direct `synscan` in running the session. The example below illustrates a simple connection:

```
session simple {
     3whs;
     seg = -@0;
     close fin;
};
```

The session listed above named "simple" has three components. The directive `3whs` tells `synscan` to properly open the session with a 3-way handshake. The directive `seg = -@0` tells `synscan` to send every segment configured for that service as an entire segment. And finally, `close fin` tells `synscan` to properly close the connection with a 4-way FIN handshake.

To set TCP options and option parameters, one can use the **synopts** and **ackopts** directives:

```
session simple_with_tcpopts {
     synopts=mss 536 timestamp 1,0 nop nop \
           sackok nop nop wscale 4 nop;
     3whs;
     ackopts=timestamp 1,1 nop nop;
     seg = -@0;
     close fin;
};
```

The synopts directive above tells `synscan` to pack the options and option parameters after the initial TCP SYN segment. The ackopts directive tells `synscan` what options should be applied to packets where the ACK bit is set.

Packet loss can be simulated using the "drop" directive as shown below.

```
session drop_example {
     synopts=mss 536 nop;
     3whs;
     seg = -@0;
     drop 5, 8;
     close fin;
};
```

Here, `synscan` will establish a connection (sending the MSS TCP option in the initial SYN) using a three-way handshake. Once established, the entire payload of the segment (configured through the service) is sent to the host. Once (and if) the host sends a fifth or eighth segment, they are pretended to be "lost" by sending a duplicate acknowledgement for the 4th and 7th segments. Upon retransmission by the sender, `synscan` will then acknowledge them.

2.3 Run-Time

Once both configuration files have been loaded, synscan enters an event loop, starting each connection in the order they appear in the configuration and processing the directives accordingly. synscan will delay opening successive connections by 100 milliseconds[2] so as to not trigger rate-limiting behavior on the host.

The tool will exit the event loop when either all the sessions terminate properly, the user interrupts the process with the SIGINT signal, or the global timeout (set on the command line) is reached.

At this point, synscan enters the analysis phase, described in the following section, where it tries to infer its behaviors. Following that, synscan then attempts to match the set of behaviors and characteristics against a database of known characteristics.

3. ANALYSIS

Once synscan finishes running all the configured TCP sessions, the results are then analyzed for a number of different characteristics. Depending on the types and directives within the tests run, synscan will try to analyze a number of different behaviors and values used by the remote host's TCP algorithms.

synscan does this analysis by calling a number of analytical modules passing the results and configuration information of each session run. Below, each module will be described in terms of its input and what characteristics it can determine.

CC: Congestion Control

Determining the congestion control algorithm used by a TCP is dependent on observing a number of different events. The current implementation of synscan is able to replicate the tests performed by TBIT, discussed in section 7.2. For services that will send more that 5 kilobytes of data without special privileges (HTTP, for example), synscan will fake packet-loss by sending duplicates acknowledgements for certain packets using the drop directive. Once the host retransmits the "lost" packet, synscan will acknowledge it (along with other segments above it that have been received).

synscan will first try to detect if the remote host implements the Fast Retransmit algorithm. If a second packet was also dropped (as done in TBIT) then synscan will further try to analyze the behavior to more accurately determine the congestion control algorithm.

By examining the timing and ordering of packets sent by the remote host in the face of the "faked" packet loss, the CC module can make a determination of the congestion control algorithm: Tahoe, Reno, NewReno, or RenoPlus. More information on this technique is available in the TBIT paper[10].

CW: Congestion Window

The Congestion Window (cwnd) is the maximum number of unacknowledged TCP segments a sending TCP can have on the wire. A TCP is also limited by the size of the recievers window (rwnd), where the minimum of these two TCP state variables is the maximum amount of data that can be in the network.

Specifically, RFC2581[15] states in section 3.1 paragraph 4:

> IW, the initial value of cwnd, MUST be less than or equal to 2*SMSS bytes and MUST NOT be more than 2 segments. We note that a non-standard, experimental TCP extension allows that a TCP MAY use a larger initial window (IW), as defined in equation 1 [AFP98]: IW = min (4*SMSS, max (2*SMSS, 4380 bytes)) (1) With this extension, a TCP sender MAY use a 3 or 4 segment initial window, provided the combined size of the segments does not exceed 4380 bytes. We do NOT allow this change as part of the standard defined by this document. However, we include discussion of (1) in the remainder of this document as a guideline for those experimenting with the change, rather than conforming to the present standards for TCP congestion control.

One can easily measure the initial value of a TCPs cwnd by eliciting a TCP payload response and not acknowledging any data the remote TCP sends. The remote host should fill the receiver's window up to the minimum of rwnd and cwnd and wait for data to be acknowledged. Eventually, the retransmit timeout will fire and we should see the first segment resent. The initial value of the cwnd is the number of segments between the first segment and the retransmit of the first segment plus one. This has also been done in TBIT.

DF: DF-bit

The IP header contains a bit labeled Don't Fragment. Senders can set this bit on an IP packet if they wish that the payload not be fragmented at any point along the path to its destination. Other tools also check the value of the DF-bit in response packets.

Beyond testing whether the DF-bit is set on a particular response, synscan is able to detect other patterns. For example, some operating systems echo the value of the DF-bit in a SYN segment in the SYNACK segment. Other operating systems, (some versions of SGI IRIX) will only set the DF-bit if the MSS TCP option is present in the the SYN segment. During testing, it was also observed that some systems will always set the DF-bit, but if a SYN reaches it with a TTL value of one, the reponding SYNACK packet has a DF-bit value of zero. It is unclear if this is a result of the network or operating system and is still being looked into.

FP: Fragment Overlap Policy

[2]The time delay between opening successive connections is configurable via a command line option.

By using the `frag` directive in a session configuration, `synscan` is able to send IP fragments containing different specified data. For example, here is a configuration which sends fragments in a certain order to see if a host accepts them conforming to the BSD reassembly algorithm (Note: This test is derived from the `activemap` test using ICMP echo-requests):

```
    session bsd_frag_check {
      synopts=mss 512 timestamp 1,0 nop nop \
              sackok nop nop;
      3whs;
      ackopts=timestamp 1,1 nop nop;
      maskwidth 8;

      frag = 40@0+;
      delay 50;
      frag = 16@48+ XO;
      delay 50;
      frag = 24@64+;
      delay 50;
      frag = 32@24+ XXOO;
      delay 50;
      frag = 24@64+ XXX;
      delay 50;
      frag = -@88

      policy_name bsd;
      close fin;
  };
```

First, we establish a connection, using the 3whs directive, specifying both the options to append to the SYN segment, the options to append to further ACK segments. Then, we send six overlapping fragments, specifying the length and offset (with len@off), whether to set IP_MF with a plus ('+') sign, and datamask to use for each fragment whereby an 'X' means to overwrite data, and an 'O' means to leave it intact. the `maskwidth` field tells `synscan` to assume each 'X' and 'O' should span 8 bytes of the payload. After each fragment is sent, `synscan` waits 50 milliseconds before sending another to try to prevent packet reordering by hardware queueing. The above example would send out the following fragments for the given text string (assuming each character is 8 bytes of data):

```
string: abcdefghijklmnopqrstuvwxyz

frag 1: abcde
     2:       Xh
     3:         ijk
     4:    XXfg
     5:         XXX
     6:            lmnopqrstuvwxyz
```

A host that reassembles these fragments according to the BSD algorithm will find a valid TCP checksum and a valid TCP segment. Other hosts will not.

The Active Mapping work done described in section 7.3 can perform a similar test, however it is limited to ICMP echo-request packets.

F8: Fragment-length MOD 8 Check

While implementing the above test cases for overlapping fragments, another caveat of fragment reassembly was discovered. In RFC791[5], there is a parenthetical note in the description of how a TCP implemention should fragment a packet.

RFC791, section 2.3 paragraph 13:

> To fragment a long internet datagram, an internet protocol module (for example, in a gateway), creates two new internet datagrams and copies the contents of the internet header fields from the long datagram into both new internet headers. The data of the long datagram is divided into two portions on a 8 octet (64 bit) boundary (the second portion might not be an integral multiple of 8 octets, but the first must be).

It was found that some TCP implementations interpret the last statement (in parenthesis) to mean that a fragment with a length that is not an integral multiple of 8 bytes and has the IP MF bit set is an invalid fragment and should be discarded. Other implementations, however, simply crop the fragment length down to the next integral multiple of 8 bytes and accept it.

For example, a listing below of relatively current linux kernel source which handles ip reassembly shows that if IP MF is set (line 364) and the length is not an integral multiple of 8 bytes (line 365), then it is trimmed accordingly (line 366) and passed on for further reassembly:

Linux source:/usr/src/linux-2.4.22/net/ipv4/ip_fragment.c
CVS version 1.58.2.1

```
352 /* Determine the position of this fragment. */
353 end = offset + skb->len - ihl;
354 /* Is this the final fragment? */
355 if ((flags & IP_MF) == 0) {
      ...
364 } else {
365     if (end&7) {
366         end &= ~7;
367         if (skb->ip_summed !=
                            CHECKSUM_UNNECESSARY)
368             skb->ip_summed = CHECKSUM_NONE;
369     }
370     if (end > qp->len) {
371         /* Some bits beyond end - corruption. */
372         if (qp->last_in & LAST_IN)
373             goto err;
374         qp->len = end;
375     }
376 }
```

However, in the FreeBSD kernel source that handles reassembly, a fragment with both of these properties (line 709,714) is considered invalid and dropped (line 716):

FreeBSD source:/usr/src/sys/netinet/ip_input.c
CVS version: 1.237

```
if (ip->ip_off & IP_MF) {
    /*
     * Make sure that fragments have a data
                                       length
     * that's a non-zero multiple of 8 bytes.
     */
    if (ip->ip_len == 0 ||
              (ip->ip_len & 0x7) != 0) {
        ipstat.ips_toosmall++; /* XXX */
        goto bad;
    }
    m->m_flags |= M_FRAG;
} else
```

Other systems were also found to implement one of these two policies.

FT: FINACK Retransmit Timeout Values

To shut down a TCP connection, the RFC793 specifies a 4-way closing handshake to properly terminate a connection. This test involves performing 3 of the 4 steps in the closing handshake. First, an open connection estabished using a 3-way handshake, then a FIN segment is sent to shut down the connection without transmitting any data. The remote host will acknowledge this FIN, then send a FINACK segment. `synscan`, however, emulates a lost packet by not acknowledging the FINACK. Once the host's FIN RTO is reached, it will resend the FINACK segment. Eventually, the host will timeout the connection completely and either send no further packets, or a RST.

This test was originally devised in `Cron-OS`[16].

HZ: Timestamp Hertz

The timestamp option defined in RFC1323[17] also defines a "timestamp clock" that is used by a TCP to update the values sent in a timestamp header. The RFC only defines loose requirements for the frequency of the timestamp clock.

`nmap` also performs this calculation. However, it is only able to calculate this based on duplicate SYNACK segments. Some TCP implementations were found to not set the TSval in the timestamp header in the initial SYNACK segment, thus preventing measurement. All TCP implementations that supported the timestamp option did correctly set the TSval in FINACK segments. A `synscan` test which uses the *close nolastack* directive tells it to measure the timestamp clock for all observed TCP implementations that implement RFC1323 extentions.

ID: IP Identification Field

TCP stacks use many different algorithms for setting the IP ID. The IPID analysis is able to detect many different observed patterns. Currently, `synscan` can detect the following methods used to set the IPID:

- I: Incremental - The host always increments the IPID using a counter global across all connections.
- IC: Incremental Control - The host has separate global counters for control segments and for payload segments.
- L: Linux - The host sets the IPID to zero on SYNACK segments, and then uses a per-connection incremental IPID for data packets with a randomized start.
- R: Random - The IPID is always a randomized value.
- Z: Zero - The host always sets the IPID to zero;

By opening numerous connections and keeping track of the IPID of each packet, `synscan` is able to observe these patterns.

MS: Default MSS Value

The default MSS value is the the value assumed to be the default MSS of the receiver in the absense of seeing an MSS option on a TCP SYN segment. We can measure this by not sending the MSS option and observing the maximum size of segments sent over the connection. However, this test can only be made over services that will send back more than one frame of continuous data (e.g. ¿ 1500 bytes). For HTTP, this is typical, but for other protocols it is more difficult to get a server to send a large amount of data.

Also note that some systems will filter and drop SYN segments that do not contain the MSS option in the first packet. One explanation given to the author is that many script-kiddie exploits are coded quickly and do not bother to add the MSS option, thus, an administrator can try to make the network more secure by filtering SYNs without the MSS option.

RT: SYNACK Retransmit Timeout Values

To establish a TCP connection, the RFC793 specifies a 3-way handshake to properly syncronize both ends' sequence numbers that will be used to push data across the connection. First, an connection request is made by sending a valid TCP SYN segment to an open port on the host. The host should reply with a SYNACK, and expect to receive an acknowledgement of its sequence number.

Again, `synscan`, emulates a lost packet by not acknowledging the SYNACK. Once the host's SYN RTO is reached, it will resend the SYNACK segment. Eventually, the host will timeout the connection completely and either send no further packets, or a RST.

This test was also originally devised in `Cron-OS`[16], and is similar to both the FIN RTO (FT) and payload RTO (PT) analyses.

PT: Payload Retransmit Timeout Values

In addition to the SYNACK and FINACK retransmit timeout analysis, `synscan` can analyze and record the retransmit timeout values of segments with a data payload.

For this analysis to happen, a test must be configured to illicit data from a remote host and then not send any acknowledgements. Eventually, the remote TCP will resend the initial data segment after the first timeout, and then (usually) exponentially backoff, resending the segment again and again until it either terminates the connection with a reset, or synscan times out and terminates the connection.

Note that the results found for RT, FT, and PT are hardly the same across operating systems and provide useful information in distinguishing between them.

SN: Initial Sequence Number

The initial sequence number (ISN) is used to syncronize both ends of a TCP connection with a common value to send data through the window. Each connection maintained by a TCP will vary the ISN with either some incremental value, or some varied source of randomness.

Other work[18] has looked more closely at the types of randomness used, but such analysis requires hundreds of thousands of connection samples which is beyond the scope of this tool.

This analysis is similar to that of `nmap`.

TL: Default TTL

`synscan` uses a similar (and widely known) method to `traceroute` to determine the default TTL value used by the remote host. TCP packets are sent to the host with each successive packet, setting the TTL one greater than the previous, starting at 1. When a response is finally received from the host, the last TTL value is added to the TTL value that appeared on the wire in the response packet. This sum is the value assumed to be the default TTL set by the host.

The directive `ttlcheck` can be supplied in a session configuration and depending where it is located will cause `synscan` to invoke this algorithm.

- Before the `synopts` directive :: Initial SYN segment
- Before the first payload directive :: First payload segment
- Before the first close directive :: FIN segment

If the directive appears before the `synopts` directive, it will use the initial SYN packet; if it appears after `synopts` but before any payload directives, it will be invoked on the first payload packet.

Some operating systems use different default TTLs for TCP control segments versus segments containing data. `synscan` will detect this and output `S(ttla,ttlb)`, where `ttla` is the default TTL for control segments, and `ttlb` is the default TTL for segments containing data.

TO: TCP Options

As said in [1], "TCP options are a goldmine of information", and `synscan` tries to mine as much information from them as possible. The idea here is to observe how a stack implementation responds to different options' presense and values in a TCP SYN segment. The options supported by `synscan` are MSS, timestamp, wscale, and SackOK. Given enough samples, one can observe a number of differences among TCP implementations.

Once all SYNACK segments are received and analyzed, `synscan` outputs a text string representing how the stack orders, pads, and fills in the values for each option. Left and right brackets are used to indicate that options are only sent when requested in the SYN, and for NOP options (padding) to indicate optional padding bytes. Additionally, parenthesis after options are used to indicate the behavior of the values within those options.

MSS: The MSS value informs the remote TCP the maximum size of a segment to send to avoid fragmentation. Depending on the value sent and the presence of other options, we observed the following behaviors:

- *mss*: The MSS is a constant value *mss*.
- E*mss*: The MSS value is always echoed value of the sent MSS, or *mss* if no MSS option was sent.
- EM536,*mss*: The MSS value is echoed value of the sent if the value sent is greater than the required minimum 536, or *mss* if no MSS option was sent.
- EMM536,*mss*: The MSS value is echoed value of the sent if the value sent is greater than the required minimum 536 AND is less than the default mss *mss*, or *mss* if no MSS option was sent.

These values are printed after the option.

Timestamp: Depending on the values sent in the initial timestamp option, the following behaviors were observed:

- D: The TCP only sends the timestamp option if the TSval in the initial SYN segment is non-zero. I.e., a zero TSval disables timestamping.
- N: The TSval and TSecr fields in the SYNACK segments are always zero regardless of the values sent.
- A: The TSval value is non-zero and the TSecr correctly echoes the value sent in the SYN segment.

These values are printed after the option.

Window Scale: We observed the following behaviors of the wscale value:

- *scale*: The scale value is always a constant *scale*.
- E: The scale value is always echoed.

Here are some samples of observed hosts given the behaviors and options ordering described above:

```
1 M(1460)[[NN]S][[NN]T(D)][NW(0)]
2 M(E)[[NN]S][[NN]T(A)][NW(0)]
3 M(536)[NW(0)][NNT];
4 [M(EM536)][W(0)];
5 [NNT(A)][NW(0)][NNS]M(1460);
6 [NNT(A)][NW(0)][NNS]M(E536);
```

Note examples 1 and 2 above where the NOP padding sent with the sack and timestamp options is optional. Example 4 shows a host that only sends the MSS option when requested, while the rest always send the MSS option.

TP: TCP Overlap Policy

Just as different operating systems handle overlapping and inconsistent fragments differently, so is the case with TCP segments. TCP segments sent out of order with overlapping portions are handled differently from system to system. `synscan` can test the TCP Overlap Policy by sending TCP segments both out of order, and with overlapping data. By masking out certain bytes of the payload, it can create an invalid segment if it is reassembled incorrectly.

Consider this example:

```
session D10 {
    synopts=mss 1460 dfbit;
    3whs;
    policy_name last;
    maskwidth 1;
    seg = 6@2 XXXOOO;
    delay 500;
    seg = -@8;
    delay 500;
    seg = 5@0 OOOOO;
    close fin;
};
```

This test configuration directs `synscan` to first open a connection using the 3-way handshake. Then, it sends a TCP segment 6 bytes long at offset 2, clobbering the first 3 bytes of that segment, followed by a 500 millisecond delay before sending a second segment whose length is the remainder of the payload at offset 8, with the payload intact. Finally, `synscan` sends the first 5 bytes of the segment also intact. If the remote host acknowledges all the data sent, then the TP modules will report the TCP policy “last”.

WS: Initial Window Size

Both sides of a TCP connection advertise their current window size in the TCP header. The value represents the size in bytes of the buffer space the sender has for more data. Being a 16-bit unsigned value, it has maximum of 65535 and a minimum of 0 (meaning the sender currently has no buffer space available).

Some TCP stacks simply set the initial window size to a constant value representing the buffer size allocated to each TCP connection. Others, however, use complicated calculations based upon the presence of TCP options, the value of those options and other metrics.

`synscan` is currently able to detect the following observed behaviors of TCP stacks in the wild:

- *win*: A constant window where *win* is the value.
- *E*: An echoed window.
- *T(a,b)*: A window that is a constant, *a*, when the timestamp option is not enabled, and a constant, *b*, when it is enabled
- *M(min,max,mss)*: A window that is a multiple between *min* and *max* of the value sent in the MSS option. If the MSS option is not present, it uses *mss*.
- *MT(min,max,mss)*: A variation of the above *M()* except that the length of the padded timestamp option (12 bytes) is subtracted from the given or default MSS before the multiplication is done.
- *MTT(min,max,mss)*: A variation of *MT()* where the subtraction is only done if both the MSS and timestamp options are present in the SYN segment.

3.1 Fingerprints

The combined set of output from all of the analytical modules describes a synscan *fingerprint*. An example of a fingerprint from the `synscan.fingerprints` file follows:

```
fingerprint "OpenBSD 3.3" {
    CC=Reno;
    CW=2;
    DF=0;
    F8=Y;
    FP=bsd;
    FT=0,1,1,1;
    HZ=2;
    ID=R;
    MS=1460;
    PT=0,1,1,1;
    RT=1,2,4,8;
    SN=R;
    TL=64;
    TP=last;
    TO=M(1460)[NNS][NW(0)][NNT(B)];
    WS=MTT(11,168,1024);
};
```

Fingerprints are loaded from this file upon startup, and when results are received from the analytical modules, each fingerprint from the file is compared to the one retrieved from the remote host. The fingerprint with the most matching characteristics is printed upon exiting.

4. TESTS AND RESULTS

After primary development of `synscan` was complete, extensive tests were conducted to compare the output of `synscan` against that of `nmap` and `xprobe2`. The latest released versions of `nmap` and `xprobe2` were used, v.3.48 and v.0.2 respectively.

4.1 Random Web Servers

Using Yahoo!’s random URL service[19], 4516 hosts were scanned using nmap, xprobe2 and synscan. For all hosts, the following parameters were passed to each program on the command line:

```
nmap -P0 -O -p 21,22,23,25,53,54,80,443 IP
```

```
xprobe2 -P -T 21,22,23,25,53,54,80,443 -U 53,54 IP
```

```
synscan -g complete -t 60 IP 80
```

For nmap, the parameter ’-P0’ tells nmap not to send an initial ICMP echo request to see if the host is up, ’-O’ tells it to attempt OS fingerprinting, and ’-p ...’ indicates the set of TCP ports to use. For xprobe2, ’-v’ tells it to be verbose, ’-P’ tells it to enable TCP and UDP protocol scanning, and ’-T ...’ indicates the set of TCP ports to try to connect, and

Table 1: Total match results for all three tools

percentile	# of hosts	% of total
nmap matches	2968/4516	65.72%
xprobe matches	3218/4516	71.26%
synscan matches	4504/4516	99.73%

Table 2: Outcomes and percentages of `nmap`

`nmap` outcome	# / total	%
One unique result	2968/4516	65.72%
No exact match	1256/1548	81.1%
Multiple guesses	466/1548	30.1%
Too many matches	78/1548	5.0%

'-U ...' the set of UDP ports. For synscan, '-g complete' tells it to use the complete set of tests, '-t 60' sets the global timeout to 60 seconds, and the '80' at the ends tells it to only scan TCP port 80.

For each host was tested at approximately the same time. This was done to reduce the chance of one tool successfully scanning a host, and another scanning the same host at some later time when it could possibly be down or offline. To do this, a perl script was written to fetch a random URL, then three processes were forked off, each executing one of the tools with the parameters above. The output of each scan was redirected to a file., Once all hosts were scanned, the results were correlated and the following statistics were calculated.

Table 1 shows the results of the 4516 hosts scanned. It shows that for 65.7% of the hosts, `nmap` returned a result, 71.2% for `xprobe2`, and 99.7% for synscan[3]. It is difficult to make exact comparisons of these results, and there are some exceptions to the numbers. One primary difficultly is categorizing the output of each tool since the output differs from tool to tool.

Table 2 shows the success of `nmap` in the first row, and then distribution of results for the 1548 hosts that `nmap` could not classify in one unique result. The majority of those hosts returned "No exact match".

While `nmap` returns one unique result for most of the hosts scanned, `xprobe2` always returns a *Primary Guess* with a "Guess Probability", followed by a user configurable number of other possible matches and their corresponding guess probability. Table 3 shows the percentile distribution of the guess probability for all hosts that `xprobe2` was able to make a guess.

Finally, it should be mentioned that `synscan` (currently) will only return one matching OS guess.

Table 4 shows the percentages for which the outcome of one tool identically matched the outcome of another. So,

[3]It should be noted that the 12 hosts synscan did not return results for was because port 80 was closed. This was also the case for nmap and xprobe2

Table 3: "Guess Probability" distribution for the "Primary Guess" determined by xprobe2

percentile	# of hosts	% of total
0-10	0	0.0%
11-20	0	0.0%
21-30	100	3.1%
31-40	16	0.5%
41-50	128	4.0%
51-60	530	16.5%
61-70	533	16.6%
71-80	1692	52.6%
81-90	215	6.7%
91-100	4	0.1%

Table 4: Exact and missed matches between tools

tool output	# of matches	percentage
nmap=xprobe	1543/2232	69.1%
synscan=nmap	1835/2965	61.9%
synscan=xprobe	1655/3213	51.5%
synscan=xprobe=nmap	1314/2229	59.0%
missed	236/1543	15.3%

out of all the hosts for which both `nmap` and `xprobe2` were able to make a guess, 69.1% of the guesses matched indentically. `synscan` matched `nmap` on 61.9% of the hosts, and it matched `xprobe2` on 51.5% of the hosts. All three tools had identical matches for 59.0% of the hosts for which all three hosts returned a guess. Finally, the last row shows the number of hosts where both `nmap` and `xprobe2` returned identical guesses (the quotient for the first row of the table), and `synscan` returned a completely different guess. This occurred for 15.3% of those matches, or 5.2% of all hosts scanned.

One possible explanation for this discrepancy is that since all of `synscan`'s tests start with a TCP SYN segment, they will all pass through a stateful firewall to the host behind it, and (usually) get replies directly from the host, passing right out of the firewall unmangled. However, the probe methods used by both `nmap` and `xprobe2` could easily be returned by the firewall. In other words, both `nmap` and `xprobe2` were fingerprinting the firewall (and both were returning identical guesses for it) while `synscan` was fingerprinting the actual host behind the firewall.

Finally, table 5 shows the top 20 OS descriptions returned for each tool for all 4516 scanned. The lone outlier in this set is the fifth entry for `xprobe2`. It shows that 5.4% of its results were for some model of HP printer! We find it highly implausible that a webserver indexed by Yahoo!'s search engine would be running on printer software. This is most likely a bug in the `xprobe2` tool.

5. COUNTERACTIVE MEASURES

The more paranoid system administrators have long sought methods to defeat, confuse, or prevent the cracker underworld, or even the curious hacker, from successfully fingerprinting their hosts[20]. Those methods will now be de-

Table 5: Top 20 OS descriptions for each tool

#	nmap	# (%)	xprobe2	# (%)	synscan	# (%)
1	Linux 2.4,2.5	823 (27.%)	Linux 2.4	840 (26.1%)	Linux 2.4	1098 (24.4%)
2	MS 95/98/ME,NT/2K/XP	522 (17.%)	MS 2000 Server	484 (15.0%)	Linux 2.2	824 (18.3%)
3	Linux 2.1,2.2	268 (9.0%)	Linux 2.2	242 (7.5%)	MS NT/2K/XP	795 (17.7%)
4	FreeBSD 4	265 (8.9%)	MS NT 4 Server	208 (6.5%)	FreeBSD 5	494 (11.0%)
5	Sun Solaris 8	180 (6.1%)	HP JetDirect	173 (5.4%)	Sun Solaris 9	282 (6.3%)
6	MS NT/2K/XP	111 (3.7%)	HP UX 11.0i	121 (3.8%)	MS 2003 .NET	211 (4.7%)
7	Sun Solaris 2,7	109 (3.7%)	MS NT 4,98	98 (3.0%)	Sun Solaris 8	188 (4.2%)
8	FreeSCO 2.0	100 (3.4%)	FreeBSD 4.4	72 (2.2%)	FreeBSD 3	179 (4.0%)
9	IBM AIX 4	89 (3.0%)	FreeBSD 4.8	71 (2.2%)	Sun Solaris 7	100 (2.2%)
10	BSDI BSD/OS 4	42 (1.4%)	FreeBSD 4.5	67 (2.1%)	BSDI BSD/OS 4	80 (1.8%)
11	FreeBSD 2,3,4	42 (1.4%)	Sun Solaris 8	65 (2.0%)	Sun Solaris 2	47 (1.0%)
12	MS 2003/.NET	37 (1.2%)	MS XP SP1a	61 (1.9%)	FreeBSD 4	45 (1.0%)
13	Linux 2.4	37 (1.2%)	OpenBSD 3.0	60 (1.9%)	IBM AIX 4	41 (0.9%)
14	FreeBSD 2,3,4	28 (0.9%)	FreeBSD 4.3	43 (1.3%)	Apple Mac OS X 10	30 (0.7%)
15	FreeBSD 4,5	27 (0.9%)	Sun Solaris 7	42 (1.3%)	Apple Mac OS 7-9	25 (0.6%)
16	Sun Solaris 9	18 (0.6%)	Linux 2.0	41 (1.3%)	IBM AIX 5	21 (0.5%)
17	Cobalt Linux 2.0, Linux 2.0	15 (0.5%)	FreeBSD 2.2	40 (1.2%)	SGI IRIX 6	21 (0.5%)
18	MS 2003/.NET,NT/2K/XP	14 (0.5%)	FreeBSD 4.1	36 (1.1%)	Sun Solaris 8/9	13 (0.3%)
19	Apple Mac OS X 10.1	13 (0.4%)	MS 2003 Server	33 (1.0%)	OpenBSD 3.3	7 (0.2%)
20	Linux 2.4, Panasonic embed.	12 (0.4%)	Sun Solaris 6	31 (1.0%)	IBM AIX 3	2 (0.0%)

scribed, as well as methods that might be successful in defeating probes by `synscan`.

5.1 Firewalling

Stateful firewalls and correct firewall rules can be used to block all TCP traffic for unknown states. This blocks all `nmap` scans except ones beginning with a valid TCP SYN segment. Firewall rules can also be used to block most ICMP traffic[4]. While this may have some impact on operations or network diagnostics, it can be considered an acceptable loss.

IP Personality is a Linux netfilter module that allows one to configure a host's firewall ruleset to mangle TCP and IP packet parameters. For example, here are some of the possible parameters it can mangle[21]:

- TCP Initial Sequence Number (ISN)
- TCP initial window size
- TCP options (their types, values and order in the packet)
- IP ID numbers
- answers to some pathological TCP packets
- answers to some UDP packets

5.2 Kernel-level parameters

Most unix-like operating systems provide an interface to change TCP/IP parameters used by the network stack. On BSD systems, this interface is accessed via the `sysctl` utility, and some of the applicable parameters (on FreeBSD 5.1) are *under net.inet.tcp.**:

- rfc1323 - high performance TCP extensions
- mssdflt - default receiver MSS
- always_keepalive - send TCP keep-alive segments
- keepintvl - interval between keep-alive segments
- sendspace - default initial TCP window size
- delayed_ack - use recommended delayed TCP acks.
- slowstart_flightsize - initial congestion window (cwnd) size
- local_slowstart_flightsize - local initial cwnd size
- newreno - enable TCP NewReno algorithms
- net.inet.ip.ttl - default IP TTL used

The Linux kernel also has changable values for other parameters *under net.ipv4.**:

- tcp_dsack
- tcp_fack
- tcp_fin_timeout
- tcp_synack_retries
- tcp_syn_retries
- tcp_retries2
- tcp_retries1
- tcp_keepalive_intvl
- tcp_keepalive_probes
- tcp_keepalive_time
- tcp_retrans_collapse
- tcp_sack
- tcp_window_scaling
- tcp_timestamps

5.3 Scrubbing

While the above parameters can be used to adjust some of the time-related tests, *packet scrubbing* in a firewall, such as OpenBSD's `pf`, can be used to prevent overlapping IP fragments and possibly TCP segments from entering a host or network. (...More description of pf/other scrubbers here...)

[4]At the very least, one should allow "ICMP Destination Unreachable: Needs Fragmentation" to allow hosts running Path-MTU discovery to work properly

6. LIMITATIONS

Right now, the main limitation to `synscan` is its utter lack of intelligence in detecting and dealing with *true packet loss*. While not all of the analytical techniques suffer from packet loss, the ones which rely on specific timing of the hosts TCP stack will produce incorrect results.

Other areas that might be problematic for `synscan` are *packet reordering*, dynamic routes, asymmetric routes, or large variations in the route-trip time (RTT) to and from the host being scanned.

And, of course, `synscan` requires at least one open TCP port (and preferably one that will send data, or at least not immediately close the connection) to perform any of its tests. Therefore, it is unable to do any scanning of client machines with `synscan`, whereas other tools may be able to if the host responds to ICMP traffic or abnormal TCP packets.

7. RELATED WORK

TCP testing and OS fingerprinting are not new. Here, some of the most common projects which perform TCP testing or OS detection are highlighted. While this section on related work may not be complete, `synscan` has been inspired primarily by the work and tools described here.

7.1 NMAP

The `nmap` tool is well regarded as the swiss army knife of network-scanning[1]. Beyond its capability to do port scans of large networks, scans by a number of different methods, and application version checking among many other things, it can also do OS detection.

`nmap`'s OS detection uses a number of analytical techniques to perform OS detection:

- TCP ISN algorithm prediction
- TCP timestamp hertz calculation
- IP header IPID field algorithm prediction

In addition to these calculations, `nmap` also sends a number of packets to the host and records the values in the responses. These test packets (referred to as T1-T7,PU by nmap) are:

- T1: send a TCP SYN packet with TCP options to open port
  – expect a SYN—ACK packet
- T2: send a TCP NULL packet w/options to open port
  – expect a RST or no response
- T3: send a TCP SYN—FIN—URG—PSH packet with TCP options to open port
  – expect a RST or no response
- T4: send a TCP ACK to open port with TCP options
  – expect a RST or no response
- T5: send a TCP SYN to closed port with TCP options
  – expect a RST
- T6: send a TCP ACK to closed port with TCP options
  – expect a RST
- T7: send a TCP FIN—PSH—URG packet to a closed port with TCP options
  – expect a RST
- PU: send a UDP packet to a closed port
  – expect an ICMP port unreachable message

Once results have been obtained for all the tests, `nmap` tries to match the results against a database of known results for different operating systems and versions. `nmap` reports the entry with matching results, and in the case of a tie or multiple ties, it will report all entries. If not enough responses have been received by `nmap` it will try to guess or fail.

7.2 TBIT

Researchers at ACIRI developed the TBIT test tool and used it to determine which TCP implementation a variety of web hosts on the Internet employed[3]. TBIT was able to detect the Tahoe, Reno and NewReno TCP implementations and a NewReno variant they called "RenoPlus". It was developed on a Linux 2.0 (using the ipfw firewall interface) platform, and at the time of this writing is not presently maintained.

TBIT uses a userland TCP stack, connects to a webserver using a 3-way handshake, sends a "GET / HTTP/1.0" HTTP request string, and simulates "dropping" packets from the HTTP response by sending duplicate ACKs. Depending on the response from the webserver, TBIT is able to categorize the TCP implementation.

The TBIT project had five main goals: To determine whether Internet simulations based on Reno are appropriate; to determine common configurable values used by hosts, such as the Initial Congestion Window (ICW); the determine how widely deployed end-to-end congestion control is in the Internet; to determine TCP implementation correctness (find bugs); and to determine the effectiveness of ECN in the wild.

7.3 Active Mapping

In the field of intrusion detection, work has shown how both the IP and TCP protocols are vulnerable to evasion from network intrusion detection systems (NIDS)[22].

To counter this vulnerability, one can use a tool, called an *Active Mapper*, to assess how the hosts on a network resolve those ambiguities, and then feed that information into a NIDS so it knows the resolution policies of all the hosts on the network[23]. Listed below are the checks the tool makes.

- Hop count to the host
- Path-MTU to the host
- IP fragment reassembly policy
- TCP segment reassembly policy
- TCP RESET acceptance policy

The two interesting checks are IP fragment reassembly and TCP segment reassembly policies. The research reported finding multiple IP fragment reassembly policies (BSD, BSD-right, linux, first and last), and also reported on observed TCP segment reassembly policies in the wild.

7.4 Xprobe2

Xprobe2 is an active OS fingerprinting tool that primarily uses data from ICMP response packets to identify different operating system implementations.

- ICMP Echo request soliciting an ICMP Echo response.
- ICMP Timestamp request soliciting an ICMP Timestamp response.
- ICMP Address Mask request soliciting an ICMP Address Mask response.
- ICMP Info request soliciting an ICMP Info response.
- UDP packet to closed UDP port soliciting an ICMP Port Unreachable response.

A recent version also has the capability to check various fields from a TCP SYNACK reply from an open TCP port.

7.5 Cron-OS

`Cron-OS`, formerly known as RINGv2, is a patched version of `nmap` to measure TCP retransmit timeouts (RTOs)[16]. Patches were then made to nmap's fingerprint file containing results of their tests.

`Cron-OS` has the ability to measure two different TCP RTOs: the SYNACK RTO and the FINACK RTO.

7.6 Passive OS Detection

In addition to these active TCP testing and active stack fingerprinting tools, there also exist passive tools to analyze network traffic and infer TCP behavior, characteristics, or OS information.

One such tool is `p0f`[24]. `p0f` is able to perform OS detection by sniffing TCP SYN and SYNACK packets off of a network wire (through bpf) and examining the header values. These values (DF bit value, TCP window size, TCP options, TTL value, MSS TCP option value, etc.) are matched against a database containing value for common operating systems.

This concept can also be applied to network intrusion detection systems (NIDS). As done in [11], a database can be built matching TCP/IP values used in `p0f` to operating system descriptions. Then, using active mapping tools, described above, and OS detection tools, the ambiguity resolution policies are added to the database. Then, as a NIDS sniffs a TCP 3-way handshake, the ambiguity resolution policies for both hosts are also known, and any ambiguity the NIDS might see can be resolved correctly.

Another tool is `tcpanaly`[25], that does off-line analysis of packet capture files to infer TCP behavior and correctness. `tcpanaly` is able to measure a number of different characteristics from both the sender's and receiver's perspective, such as if normal ACKing or stretch ACKing is employed, the initial value of the congestion window (cwnd) of the sender, the retransmit timeout value or an implementation of fast retransmit, among other datapoints. The main motivation of `tcpanaly` was to analyze different TCP traces to find bugs or anamalous behavior.

8. FUTURE WORK

While `synscan` performs quite extensive analysis of TCP parameters, there are many more pieces of information it could measure. For example, more detailed examination of how a TCP manages it congestion window in the face of packet-loss. As asked in TBIT, does a TCP correctly reduce its congestion window (cwnd) by half when it detects a lost packet? There are other questions pertaining to congestion control and RFC adherance that `synscan` could also attempt to answer. However, the problem of dealing with real packet loss must first be addressed before trying to answer such questions in a consistent way.

`synscan` currently does not implement the TCP RESET acceptance policy checks as the *active mapper* tool does, described in section 7.3. This would be another useful metric.

The payload division algorithm is also a metric `synscan` could use to distinguish hosts. This is the method used to partition large amounts of data into segments that fill the receiver's MSS. Consider a web server that returns 1018 bytes for a "GET / HTTP/1.0" request (usually in one `write()` system call to the socket). If `synscan` sends an advertised MSS of 100 bytes, some TCP stacks will send ten 100-byte segments and one 18-byte segment; others will send one 18-byte segment *first*, then 10 100-byte segments; while others use different procedures to move the data.

Idle connections are usually preserved on sockets when they have been told to stay *alive* by the program. However, some operating systems aren't careful about *when* they actually send keep-alive TCP segments. During the development and testing of the retransmit timeout analysis code for unacknowledged TCP data, it was observed that some systems will send TCP keep-alive probes every 75 seconds, *in addition* to retransmitting the unacknowledged segments. It is possible for `synscan` to also observe and report this type of behavior.

9. CONCLUSION

This paper introduces a new TCP stack testing and fingerprinting tool, `synscan`, which combines a number of new and old techniques to analyze TCP behavior and provide a TCP/IP fingerprint. The main advantage `synscan` presently has over other tools is its methods for obtaining specific implementation information all come though "normal" TCP client connections.

`synscan` has been demonstrated as a successful OS fingerprinter against other main tools currently being used today. Large scans of hosts on the Internet were performed and their results shed some light on the status of the ability to OS fingerprint hosts on the Internet.

Finally, we hope that the information given on countermeasures to the techniques provided by this tool will possibly aid the more paranoid members of the Internet community.

10. ACKNOWLEDGMENTS

Many thanks to Dug Song's `libdnet`, Niels Provos' `libevent`, and to those who provided feedback on drafts of this paper and developent of `synscan`.

11. REFERENCES

[1] Fyodor. Remote OS Detection Via TCP/IP Stack FingerPrinting. *Phrack*, 54(8), December 1998.

[2] F. Yarochkin O. Arkin. Xprobe v2.0, A "Fuzzy" Approach to Remote Active Operating System Fingerprinting. *http://www.sys-security.com*, August 2002.

[3] J. Padhye and S. Floyd. On Inferring TCP Behavior. In *Proc. of SIGCOMM 2001*, pages 287–298, Aug 2001.

[4] S. Savage. Sting: A TCP-Based Network Measurement Tool. In *1999 USENIX Symposium on Internet Technologies and Systems*, page 7179, October 1999.

[5] J. Postel. Internet Protocol, September 1981.

[6] J. Postel. Transmission Control Protocol. Internet RFC 793, September 1981.

[7] V. Jacobson. Congestion Avoidance and Control. In *Proc. of SIGCOMM '88*, Aug 1988.

[8] S. Floyd M. Mathis, J. Mahdavi and A. Romanow. TCP Selective Acknowledgment Options. Internet RFC 2018, October 1996.

[9] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. Internet RFC 2582, April 1990.

[10] S. Floyd and K. Fall. Promoting the Use of End-to-End Congestion Control in the Internet. *IEEE/ACM Trans. Networking*, August 1999.

[11] G. Taleck. Ambiguity Resolution Via Passive OS Fingerprinting. *6th International Symposium on Recent Advances in Intrusion Detection*, September 2003.

[12] D. Song. libdnet: A Portable Networking Library. *http://libdnet.sourceforge.net/*, 2001.

[13] N. Provos. libevent. *http://www.monkey.org/ provos/libevent/*, 2001.

[14] C. Leres S. McCanne and V. Jacobson. libpcap. *http://www.tcpdump.org/*, 1994.

[15] V. Paxson M. Allman and W. Stevens. TCP Congestion Control. Internet RFC 2581, April 1999.

[16] O. Courtay F. Veysset and O. Heen. New Tool and Technique For Remote Operating System Fingerprinting. *Website Publication*, April 2002.

[17] R. Braden V. Jacobson and D. Borman. TCP Extensions for High Performance. Internet RFC 1323, May 1992.

[18] M. Zalewski. Strange Attractors and TCP/IP Sequence Number Analysis. *http://razor.bindview.com/publish/papers/tcpseq.html*, 2001.

[19] Yahoo! Random URL Service. *http://random.yahoo.com/fast/ryl/*, 2001.

[20] D.B. Berrueta. A Practical Approach for Defeating Nmap OS-Fingerprinting. *http://voodoo.somoslopeor.com/papers/nmap.html*, 2002.

[21] G. Roualland and J. Saffroy. IP Personality: A Linux netfilter module. *http://ippersonality.sourceforge.net/*, 2001.

[22] T. Ptacek and T.N Newsham. Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection. January 1998.

[23] U. Shankar. Active Mapping: Resisting NIDS Evasion Without Altering Traffic. *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003.

[24] M. Zalewski. p0f: Passive OS Fingerprinter. *http://lcamtuf.coredump.cx/p0f.shtml*, 2001.

[25] V. Paxson. Automated Packet Trace Analysis of TCP Implementations. *SIGCOMM*, 1997.