Xprobe v2.0
A "Fuzzy" Approach to Remote Active Operating System Fingerprinting

Ofir Arkin
Founder
ofir@sys-security.com
The Sys-Security Group
http://www.sys-security.com

Fyodor Yarochkin
fygrave@tigerteam.net
Beez!LSD Labs
http://www.notlsd.net

August 2002

Abstract

The tools used today for remote active operating system fingerprinting use a signature database to perform a match between the results they receive from a targeted machine and known operating system fingerprints. Usually, the process is done by utilizing strict signature matching to identify the type of the remote operating system. The operating system fingerprinting tools that rely on strict signature matching face several problems with their way of operation, which when present lead to false identification of the target operating system(s). With this paper we present a different approach to signature matching with remote active operating system fingerprinting. Our approach is one which aims to solve the problems presently faced by remote active operating system fingerprinting tools, as well as providing more accurate results when used against any network topology.

Contents

1.0 Introduction

In August 2001, Fyodor Yarochkin and Ofir Arkin released `Xprobe` version 0.0.1[1]. `Xprobe` is a remote active operating system fingerprinting tool based on Ofir Arkin's "ICMP Usage in Scanning" research project[2]. The tool presents an alternative to other remote active operating system fingerprinting tools which are heavily dependent on the usage of the TCP protocol for remote active operating system fingerprinting. The first versions of `Xprobe` combined various remote active operating system fingerprinting methods using the ICMP protocol, which were derived from the "ICMP Usage in Scanning" research project, into a simple, fast, efficient and a powerful way to detect a target host's underlying operating system.

The first versions of `Xprobe` lacked the support of a signature database and relied on a static decision tree to produce the results. The use of hard coded signatures within `Xprobe` instead of a database holding operating system fingerprinting signatures is one of the main disadvantages of the tool.

This led to the approach for integrating signature support into `Xprobe`. Since the usual strict signature matching approach, taken by other remote active operating system fingerprinting tools, suffers from several design flaws and accuracy problems given the conditions the tools operate against, we considered a different approach with signature matching.

This paper presents a different approach for analyzing the results produced by various remote active operating system fingerprinting tests. We will be explaining how we aggregate different remote active operating system fingerprinting methods in order to identify the type of a remote operating system with a high precision rating utilizing a 'fuzzy' approach to remote active operating system fingerprinting.

[1] Version 0.0.2 is the current official version of the first generation of `Xprobe`.
[2] http://www.sys-security.com

2.0 Strict Signature Matching

The tools used today for remote active operating system fingerprinting (`nmap`[3], `queso`) use a signature database to perform remote operating system recognition by utilizing strict signature matching and a fixed number of fingerprinting tests to identify the type of a remote operating system.

2.1 Problems

The strict signature matching technique, by itself, is not perfect. It is affected from a number of issues reflected by the topology of the targeted system/network and the nature of fingerprinting itself, where we are merely 'guessing' the type of the remote operating system. Among these issues we can identify the following:

- A packet might be affected in different ways while in transit. Several field values within the packet might be changed by a networking device or even by a filtering device for different reasons. We can name several examples:
  - A packet shaping device might change several field values within a packet (forcing TOS values, IP time-to-live values, discarding packets with malformed checksums, calculating checksums for zero-checksumed packets (UDP) etc).
  - A router or a firewall might spoof responses for a targeted system they protect. For example firewalls which spoof ICMP query replies for targeted systems they protect, or even performing the TCP 3-way handshake with an initiating system before handing the connection directly to a protected targeted system (some sort of a denial-of-service protection[4]).
  - A Scrubber[5] may be present between the sending system and the target system.

Potentially, these and other such problems, might affect the results produced by a remote active operating system fingerprinting tool resulting with false and inaccurate results.

If a remote active operating system fingerprinting tool relies on certain IP packet field values, which were changed or affected by the networking environment the packet traverses, it is more than likely that the strict signature matching process will fail (or produce false results). By introducing appropriate signature entries into our signature database we can use this situation to our advantage. The added entries within the signature database will match the modifications, and therefore we will be able to collect extra intelligence and knowledge about certain networking devices, filtering devices, or even networking topologies. Potentially we might be able to recognize a type of a packet filtering device (and sometimes its function, i.e. OpenBSD NAT, or Linux IP masquerade), a packet shaping device, etc.

[3] www.insecure.org
[4] For example, Checkpoint FW-1.
[5] A Scrubber is a software logic aimed at "cleaning" several fields within a packet passing through the machine the Scrubber is installed on. This is done by setting several fields within the packet to certain values so malicious parties trying to determine the remote host's operating system that is located behind a Scrubber using a remote active operating system fingerprinting technique(s) will fail in their efforts.

- In a real networking environment systems should be firewalled[6]. If the traffic filtering is done correctly, then some inbound and outbound packets will be dropped by the firewall (i.e. not allowed in or out). If a remote active operating system fingerprinting tool relies on the receipt of particular packet types and those packets were dropped by a firewall protecting the target system(s) chances are high that false results or no results at all will be produced.

- If the packets sent by a remote active operating system fingerprinting tool pass through a load balancing device along their way to the target machine some packets might be routed to a different machine rather than the destined target. This might cause the signature matching process to fail.

- Some characteristics of a TCP/IP stack's behavior can be altered by a user:

  - Tunable parameters of the TCP/IP stack might be changed e.g. the `sysctl` command on the various *BSDs, the `ndd` command on Solaris, etc.

  - Numerous patches exist for some open source operating system's kernels that alter the way the particular operating system's TCP/IP stack responses to certain packets.[7]

  If a remote active operating system fingerprinting tool is using some of the parameters which can be altered as part of its signature base, the signature match will most likely fail.

- If a remote active operating system fingerprinting tool utilizes malformed packets to produce its results, these malformed packets could be easily detected by a properly configured Network Intrusion Detection System (NIDS). If you are ready to sacrifice the quality of the fingerprinting in order to avoid detection, strict signature matching based remote active operating system fingerprinting tools may not allow you to do so if they relay on responses to malformed packets as part of an operating system signature.

- If a remote active operating system fingerprinting tool utilizes malformed packets to produce its results these malformed packets might be dropped by a filtering device, if the filtering device analyzes packets for non-legitimate contents. Therefore fingerprinting tests relying on these packets will fail and no results will be produced.

2.2. Needs

Listed are several abilities and features that we feel should be present with an advanced remote active operating system fingerprinting tool:

- A certain precision of remote active operating system fingerprinting should be maintained even if some particular tests fail or are rendered ineffective by the target network environment/topology.

- The ability to identify networking obstacles such as filtering devices, load balancers, etc.

[6] Firewalled is an acronym to "Protected by a firewall".
[7] One example is the IP Personality patch for Linux Kernel 2.4.x (http://ippersonality.sourceforge.net/).

- The ability to detect modifications made to the targeted machine's TCP/IP stack.

- The ability to detect Scrubber activity.

- If we are unable to identify the operating system type of the targeted machine, we would like to limit the number of possible guesses/matches to a finite number. Having a list of possible matches would allow a knowledgeable auditor, in some cases, to either narrow down the list of possible matches or even take an educated guess at the correct operating system.

- Often, people discover new ways to fingerprint operating systems, or vendors alter their TCP/IP stacks making old-known fingerprints and/or fingerprinting methods fail against new operating systems or the altered TCP/IP stacks. We wish to have a tool which will allow a user to easily add or remove new modules of fingerprinting techniques, based on any protocol, while maintaining and, still being able to use, the original modules and signature database. This means we have to have an API for the tool.

- We wish to maintain control on the ability to use (or not to use) malformed packets in our probes and still be able to gather particular intelligence on a remote operating system's TCP/IP stack type.

- The ability to have full control on each and every aspect of the fingerprinting tests (i.e. number of repeated tests, number of packets sent, parameters used, etc.).

`Xprobe v2` brings about many of these features, and the solution to many of these needs. We attempted to resolve most of these problems, while sticking to our original goal of creating one of the most advanced remote active operating system fingerprinting tool out there.

3.0 A Fuzzy Approach with Operating System Fingerprinting

Several approaches to 'fuzzy' matching can be used with a remote active operating system fingerprinting tool to match received results with a known fingerprints signature database:

- **Fisher's Discriminate Function Analysis**: this is a statistical solution which allows classifying a number of elements into groups based on 'matching factors'. More details are available here: http://www.statsoftinc.com/textbook/stdiscan.html (this could be interesting to implement).

- **OCR recognition**: several **O**ptical **C**haracter **R**ecognition methods have been implemented along the years. Most of them could be applicable to perform 'fuzzy' signature matching.

- Matrix based fingerprints matching based on **statistical calculation of scores for each test** (one of the simplified forms of the OCR methods).

- Other Mathematical algorithms

3.1 The 'Fuzzy' Approach with Xprobe2

We chose to use the matrix based fingerprinting matching approach based on OCR recognition as the method to be used for 'fuzzy' matching with `Xprobe` v2.

The solution is based on a simple matrix representation of the scan (or scans), and the calculation of 'matches' by simply summing up scores for each 'signature' (OS). All tests are performed independently. The following is the abstract matrix we are using with `Xprobe v2`:

OS / **Test**	Operating System 1	Operating System 2	Operating System 3	…	Operating System *i*
Test 1 (TTL)	score	score	score	…	score
Test 2 (IP_ID)	score		score	…	score
Test 3 (ICMP Port Unreachable)	score	score	score	…	score
…				…	
Test n	score	score	score		score
Totals	X	Y	Z	…	D

Table 1: The Results Table

Upon initialization each fingerprinting test, which is implemented as an independent module, builds its own vector of possible 'test matches' for each OS (OS $\rightarrow$ (OS_1, OS_2, …OS_i)). This is done by reading the `xprobe2.conf` configuration file, which holds the fingerprints signature database, and looking for the "`fingerprint`" and "`OS_ID`" entries. Once the fingerprinting test is executed the program examines the packet(s) received as a result of the fingerprinting test and places the appropriate '`score`' into the appropriate OS row.

The `score` value can take one of the following values:

- YES(3)
- PROBABLY_YES(2)
- PROBABLY_NO(1)
- NO(0).

Each test module assigns the appropriate `score` value according to the scheme implemented with the module. Having the `score` parameter able to be assigned different values introduces a certain degree of 'fuzziness' with `Xprobe` v2.

Once all tests are completed, we simply run through all the columns and calculate the summary for each OS. The top-score OS{x} (X, Y, Z, or D) will be declared as the final result.

This approach gives us probabilistic support since the highest `score` given for an OS (or OSs) is the most likely to produce an accurate match.

The other 'possible' results could optionally be listed, as they may be useful to identify:

- A slightly different TCP/IP stack that produced similar test results for some of the fingerprinting tests used.

- The type of an intermediate device which alters some values within the packets sent and/or received (e.g. if you use Linux's IP masquerade abilities, it will overwrite certain settings within the IP headers of packets traversing through. This currently confuses the latest versions of `nmap` and `xprobe`[8]).

- The type of the original operating system, even if the TCP/IP stack has been altered. The alteration can be done by changing the default value of one, or more, tunable kernel parameters, by using a patch for the kernel, or by using a Scrubber. The TCP/IP stack alteration can also be aimed to 'masquerade' as some other operating system.

  The ability to detect the original operating system holds true if some tests are not affected by the TCP/IP stack alternations and some are, taking into account that both signatures for the operating system without the alternation and for the operation system with the alternation should be present with the signature database.

- A filtering device spoofing responses for a system it is configured to defend.

3.1.1 Pluggable Architecture

A pluggable architecture was designed with `Xprobe` v2, where different modules, representing new modules, improved modules, or your own way of TCP/IP stack fingerprinting tests might be introduced by any user using the program's API. The core functionality of `Xprobe` v2 is designed in such a way that each test is independent from each other (the usage of the matrix), and when new

[8] Xprobe version 0.0.2

modules are added, the functionality of the tool does not degrade. Instead, it just adds the appropriate test entry in the matrix.

3.1.2 Overcoming Failures of Certain Tests & Defeating Network Obstacles

Having the ability to choose which fingerprinting tests and modules to use allows us to overcome failures of certain tests, since they will not affect the “global picture” (the final score for each operating system). For example, one can choose to use `NO(0)` for failed tests, and `PROBABLY_NO(1)`, for platforms where responses are unknown. Even if a particular test fails all of the operating systems represented in the matrix will get the same `score`[9]. This also suggests that having more tests might produce a better overall result. The burden of deciding the weight each test has on an operating system lies on each individual module. This gives module writers the freedom to assign `score` values according to their own take on remote active operating system fingerprinting.

3.1.3 Control

With `Xprobe` v2 a user has full control on which modules, probes and tests the tool will use when targeting a remote machine. This ability gives the experienced user more room to control the tool’s exact behavior as well as flexibility no other remote active operating system fingerprinting tool provides today.

The tool gives its users the ability to be more accurate with matched results, as with the appropriate usage of modules the discovery of “network obstacles” at the targeted network.

3.1.4 Dealing with Yes/No Tests

The only issue that might affect the fingerprinting results, is a fingerprinting test which sends a packet and waits to see if the remote machine answers the probe or not. We define this type of fingerprinting test as “a Yes/No test”. With these type of tests, we have a problem to determine if the remote machine did not produce an answer because it is not answering the particular probe as default, a tunable parameter was set to not answer the particular probe we are using, or because a filtering device is not allowing this type of probe (either inbound or outbound).

Using our fuzzy approach to remote active operating system fingerprinting we have the ability to control the affect of such a test on the overall result scheme.

If we asses that a particular Yes/No test has a high probability of being blocked at the network perimeter, at the host level were the host might have been hardened, or not being answered by the majority of systems we can narrow the range of assigned values for this particular test. Instead of using the YES(3) or No(0) `score` values we will be assigning the `score` values of PROBABLY_YES(2), for a successful attempt, and PROBABLY_NO(1) for an unsuccessful attempt. With this being done we minimize the unreliable test results affect on the overall results, assuming other tests will be taken.

```
                    | ← Less affect on f. score → |
0                   1                             2                   3
| ←                       Wider affect on final score                → |
```

Figure 1: Narrowing the Yes/No affect on the overall results

[9] Please see section 3.1.4 for more information on Yes/No tests.

Although currently we are using only four (4) different values for the `score` parameter, we are planning in the future to use a wider `score` parameter value range with more appropriate values for these type of conditions.

As a rule of thumb we need to understand, and adjust to, the environment(s) we are operating in. Targeting machines over the Internet and auditing your own networks internally are two totally different scenarios.

One also needs to have an intimate knowledge with different TCP/IP stacks in order to fully understand how those will respond when probed with different fingerprinting tests. This person will have the ability to take a full advantage of the modular architecture of `Xprobe` v2 and from its own test results when they are not conclusive.

For example, some operating systems are lacking the appropriate tunable parameter for controlling some behavioral aspects of their TCP/IP stack. We can name Sun Solaris 2.3-2.9 and the lack of ability to configure the TCP/IP stack not to answer ICMP Echo requests and ICMP Address Mask requests. In this case, if, eventually, the remote machine is found to be a Sun Solaris machine but the Address Mask test failed, than there is a filtering device that disallowed this type of messages between us and the target[10].

[10] Can also be a Host-based firewall installed on the remote machine.

4.0 Xprobe2: The Practical Implementation

`Xprobe` v2 is primarily implemented based on operating system fingerprinting tests developed for the original `Xprobe` tool. These are remote active operating system fingerprinting tests based on the ICMP protocol which were discovered during Ofir Arkin's "ICMP Usage in Scanning" research project.

However, new tests have been added based on our own research or other remote operating system fingerprinting tools implementations.

Please refer to the original `Xprobe` article and design notes documentation[11] for more information on the original `Xprobe` remote operating system fingerprinting tests.

A user is not limited to using only the fingerprinting test modules available with the program; he may write his own modules with his own remote operating system fingerprinting tests reflecting his own take on fingerprinting since an API is provided with `Xprobe` v2.

4.1 The API (Modules How-To)

To write a module for `Xprobe` v2, one needs to follow the instructions given within this section. Please refer to the `modules_howto` document and the source code (`target.h`, `xprobe_module.h`) for API details and for the most accurate/updated documentation.

4.1.1 The Buildup

First, one needs to create a `class` which will inherit `Xprobe_module`:

```
class Your_Module: public Xprobe_Module {
    private:
    public:
        Test_Mod(void) : Xprobe_Module(XPROBE_MODULE_ALIVETEST,
"YOUR_MODULE NAME") { /*your constructor*/ }

        /* or just
        Test_Mod(void) : Xprobe_Module("YOUR_MODULE NAME")  { your
constructor } */
        ~Test_Mod(void) { return; }
        int init(void);
        int parse_keyword(int os_id, char *keyword, char *value);
        int exec(Target *tg, OS_Matrix *os);
        int fini(void);
};
```

The first argument in the `constructor` signifies the type of function which your module is going to perform:

- "`XPROBE_MODULE_ALIVETEST`" are modules which test a remote system's reachability (perform a "reachability test").

[11] X – Remote ICMP based OS fingerprinting Techniques, Ofir Arkin & Fyodor Yarochkin August 2001. Available from: http://www.sys-security.com.

- "`XPROBE_MODULE_OSTEST`" are modules which perform an actual remote active operating system fingerprinting test.

The following function needs to be created:

```
int your_module_init(Xprobe_Module_Hdlr *pt) {

    int mod_id;
    Your_Module *mod = new Your_Module;

    xprobe_mdebug(XPROBE_DEBUG_MODULES, "Initializing the YOUR_MODULE
module\n");
    pt->register_module(mod);
    /* keywords which you will be parsing in parse_keyword routine */
    pt->add_keyword(mod->get_id(), "keyword1");
    pt->add_keyword(mod->get_id(), "keyword2");
    pt->add_keyword(mod->get_id(), "keyword3");

return OK;
}
```

The following methods of `class` you will have to write yourself:

init()
The `init()` function will prepare all the data, sockets whatever you will need to use.

parse_keywords()
`parse_keywords(int os_id, char *keyword, char *value);`

We will call this function if we will see keywords which you registered for your module, appearing for OS fingerprint with id `OS_ID`. It is intended that your module will parse the `keyword` and store the result in an internal format (whichever format you wish to use) which will represent `OS_id`, your test results according to the `keyword` you have parsed for this OS, and `score`, which ranks how the given 'fingerprint' matches the signature if the test results match. Possible variants for the score are:

- XPROBE_MATCH_YES
- XPROBE_MATCH_NO
- XPROBE_MATCH_PROBABLY_YES
- XPROBE_MATCH_PROBABLY_NO

exec()
`exec(Target *tg, OS_Matrix *os)`

When the exec() function is being called, you perform the actual tests (send packets, receive response, or just analyze data in `Target()` stored by other modules). Then you should go through your pre-parsed signature data, and fill in `OS_vector` for every OS which you have parsed, signifying how "well" every OS signature matches what you have absorbed with the packet/target:

```
OS_Matrix->add_result(this->get_id(), int OS_id,int score);
```

All the information regarding the target system you obtain through a class '`Target`' (see `target.h` for details), more over the communications with the core system you perform through target methods as well. You should be able to get all the data you need, if you are missing something please let us know.

Here are some of the most useful functions:

- `tg->get_addr()`, returns `in_addr struct` of target
- `tg->get_interface()`, returns `char ptr` to interface name
- `tg->get_inetface_addr()`, returns address of the interface through which you connect to the target
- tg->get_port(IPPROTO_TCP, XPROBE_TARGETP_OPEN) (or CLOSED, or FILTERED), will give you a port number (or -1 if unknown) which you asked for
- You can also use `tg->add_port()` to set port status, if you found it during the test
- `tg->get_ttl()` and `tg->set_ttl()` will return or set (for use of other modules) `ttl` distance to the target system

Please Note:
For reachability test modules, OS_id, should always be 1, since we don't have signatures to be parsed. The `parse_keywords()` routine should be a dummy and you don't have to register any particular keywords unless you expect parameters from config file.

fini()
de-allocate all the memory which you allocated here. Close all the sockets. Clean up everything. Destructor for your module will be called right after. You don't have to delete your module object instance itself, which you allocated in init function, we will do that for you.

User Interface

Currently only command line/text interface is implemented. However for further compatibility we have an `Interface()` class which has particular methods for data output. We insist on using these in your code.

In the future other interfaces might be developed (or you could develop this yourself), and it will save the hassle of porting each module to every new interface.

Declare following:

```
extern Interface *ui;
```

Use our user interface routines to generate output:

```
xprobe_debug(XPROBE_DEBUG_MODULES, "fmt string: %s", arg); to print
debug messages.
xprobe_mdebug(XPROBE_DEBUG_MODULES, "something"); to print single string
debug messages.

ui->msg("message: %s", args); to print messages (prefix them with [x]
[module])
ui->log("something"); to log data
```

```
ui->error("...", blah); to report errors..
ui->perror("something"); use instead of perror();
ui->verbose(verbosity_level, "msg",..) -- (controlled with -v option,
1,2,3 .. are possible levels)
```

4.1.2 The Compilation

To include the module into our source you should do the following:

1. Add your init function to `xpmodules/static_modules.h` like the following:

```
...
typedef int(* xprobe_module_init_t)(Xprobe_Module_Hdlr *);

xprobe_module_init_t mod_init_funcs[]= {
    test_mod_init,
    your_module_init,
    NULL};
```

2. Place your module into the `alive_probe` directory or the `os_probe` directory. Then create `Makefile.in` to compile your modules and edit `xpmodules/Makefile.in` to link your objects to the `modules.a` library file. You can create your own archive of objects with `ar`, just don't run `runlib` on it and let us add your archive to ours.

3. Test that everything builds properly.

5.0 Sample Xprobe2 Run[12]

The sample run was produced utilizing a Linux kernel 2.4.18-based machine running `Xprobe2` targeting a Microsoft Windows XP Professional machine on the same local LAN.

The following is the sample run `Xprobe2` have produced:

```
carman:~/tmp/xprobe2/src # ./xprobe2 -v 192.168.1.200

XProbe2 v.0.1 Copyright (c) 2002 fygrave@tigerteam.net, ofir@sys-security.com

[+] Target is 192.168.1.200
[+] Loading modules.
[+] Following modules are loaded:
        [x]ICMP echo (ping)
        [x]TTL distance
        [x]ICMP echo
        [x]ICMP Timestamp
        [x]ICMP Address
        [x]ICMP Info Request
        [x]ICMP port unreachable
[+] 7 modules registered
[+] Initializing scan engine
[+] Running scan engine
[+] Host: 192.168.1.200 is up (Guess probability: 100%)
[+] Target: 192.168.1.200 is alive
[+] Primary guess:
[+] Host 192.168.1.200 Running OS: "Microsoft Windows 2000/2000SP1/2000SP2"
(Guess probability: 68%)
[+] Other guesses:
[+] Host 192.168.1.200 Running OS: "Microsoft Windows XP Professional" (Guess
probability: 68%)
[+] Host 192.168.1.200 Running OS: "Microsoft Windows ME" (Guess probability:
63%)
[+] Host 192.168.1.200 Running OS: "Microsoft Windows NT 4 Service Pack 4 and
Above" (Guess probability: 59%)
[+] Host 192.168.1.200 Running OS: "NetBSD 1.5.2" (Guess probability: 59%)
[+] Cleaning up scan engine
[+] Modules deinitialized
[+] Execution completed.
carman:~/tmp/xprobe2/src #
```

The first two modules to be initialized and used, ‘`ICMP Echo`’ and ‘`TTL distance`’, are reachability tests. With the first test, an ICMP echo request is sent to the target machine. The goal is to elicit an ICMP echo reply back from the target system:

```
11:40:37.046355 192.168.1.13 > 192.168.1.200: icmp: echo request (ttl 64, id
477, len 41)
                         4500 0029 01dd 0000 4001 f4d1 c0a8 010d
                         c0a8 01c8 0800 b6be dd59 0000 5850 524f
                         4245 322d 7072 6f62 65
11:40:37.046587 192.168.1.200 > 192.168.1.13: icmp: echo reply (ttl 128, id
1817, len 41)
                         4500 0029 0719 0000 8001 af95 c0a8 01c8
```

[12] libpcap 0.7.1 must be used with `Xprobe2`

```
                         c0a8 010d 0000 bebe dd59 0000 5850 524f
                         4245 322d 7072 6f62 6500 0000 0000 fce5
                         84a7
```

With the second reachability test a TCP SYN packet is sent to the target system. The goal is to elicit a response from the target system, a TCP SYN/ACK (when the TCP port is opened) or a TCP RST packet (when the TCP port is closed):

```
11:40:37.049343 192.168.1.13.5557 > 192.168.1.200.65535: S [tcp sum ok] 1:1(0)
win 512 (DF) (ttl 80, id 5774, len 40)
                         4500 0028 168e 4000 5006 901c c0a8 010d
                         c0a8 01c8 15b5 ffff 0000 0001 0000 0000
                         5002 0200 1407 0000
11:40:37.049581 192.168.1.200.65535 > 192.168.1.13.5557: R [tcp sum ok] 0:0(0)
ack 2 win 0 (ttl 128, id 1818, len 40)
                         4500 0028 071a 0000 8006 af90 c0a8 01c8
                         c0a8 010d ffff 15b5 0000 0000 0000 0002
                         5014 0000 15f4 0000 0000 0000 0000 57de
                         88ff
```

If no answer is received, a second TCP SYN packet will be sent targeting a different TCP port on the targeted system, with the same idea in mind. Only if these attempts will not produce a reply from the target system, a UDP datagram will be sent to the target system in an attempt to elicit an ICMP Port Unreachable error message.

All test within the second reachability module work in a `traceroute` like manner.

The rest of the modules which are currently available with Xprobe v2 are fingerprinting modules which are based on the fingerprinting tests we have used for the original Xprobe v1[13].

An ICMP Echo fingerprinting test:

```
11:40:37.053629 192.168.1.13 > 192.168.1.200: icmp: echo request (DF) [tos
0x6,ECT]  (ttl 64, id 0, len 41)
                         4506 0029 0000 4000 4001 b6a8 c0a8 010d
                         c0a8 01c8 087b b643 dd59 0000 5850 524f
                         4245 322d 7072 6f62 65
11:40:37.053825 192.168.1.200 > 192.168.1.13: icmp: echo reply (DF) (ttl 128, id
1819, len 41)
                         4500 0029 071b 4000 8001 6f93 c0a8 01c8
                         c0a8 010d 0000 bebe dd59 0000 5850 524f
                         4245 322d 7072 6f62 6500 0000 0000 e035
                         5b51
```

An ICMP Timestamp yes/no test:

```
11:40:37.054630 192.168.1.13 > 192.168.1.200: icmp: time stamp query id 56665
seq 0 (ttl 64, id 478, len 41)
```

[13] For more information regarding these fingerprinting tests please refer to: Arkin Ofir & Fyodor Yarochkin, "X – Remote ICMP based OS fingerprinting Techniques", August 2001. Available from: http://www.sys-security.com/archive/papers/X_v1.0.pdf

```
                          4500 0029 01de 0000 4001 f4d0 c0a8 010d
                          c0a8 01c8 0d00 b1be dd59 0000 5850 524f
                          4245 322d 7072 6f62 65
11:40:37.054855 192.168.1.200 > 192.168.1.13: icmp: time stamp reply id 56665
seq 0 : org 0x5850524f recv 0x79760104 xmit 0x79760104 (ttl 128, id 1820, len
40)
                          4500 0028 071c 0000 8001 af93 c0a8 01c8
                          c0a8 010d 0e00 7511 dd59 0000 5850 524f
                          7976 0104 7976 0104 0000 0000 0000 80bc
                          e732
```

An ICMP Address Mask yes/no test:

```
11:40:37.055567 192.168.1.13 > 192.168.1.200: icmp: address mask request (ttl
64, id 479, len 41)
                          4500 0029 01df 0000 4001 f4cf c0a8 010d
                          c0a8 01c8 1100 adbe dd59 0000 5850 524f
                          4245 322d 7072 6f62 65
```

An ICMP Information Request yes/no test:

```
11:40:42.053602 192.168.1.13 > 192.168.1.200: icmp: information request (ttl 64,
id 480, len 41)
                          4500 0029 01e0 0000 4001 f4ce c0a8 010d
                          c0a8 01c8 0f00 caaf c268 0000 5850 524f
                          4245 322d 7072 6f62 65
```

A UDP packet masquerading as a DNS query result which is sent to elicit an ICMP Port Unreachable error message from the target machine:

```
11:40:47.053811 192.168.1.13.53 > 192.168.1.200.65535:  22178% q: A?
www.securityfocus.com. 1/0/0 www.securityfo (70) (DF) (ttl 255, id 1, len 98)
                          4500 0062 0001 4000 ff11 f763 c0a8 010d
                          c0a8 01c8 0035 ffff 004e 0c14 56a2 81f0
                          0001 0001 0000 0000 0377 7777 0d73 6563
                          7572 6974 7966 6f63 7573 0363 6f6d 0000
                          0100 0103 7777 770d 7365 6375 7269 7479
                          666f
11:40:47.054164 192.168.1.200 > 192.168.1.13: icmp: 192.168.1.200 udp port 65535
unreachable for 192.168.1.13.53 > 192.168.1.200.65535:  43200 op5 [b2&3=0x2a06]
[28525q] [256n] [259au][|domain] (DF) (ttl 255, id 1, len 98) (ttl 128, id 1822,
len 56)
                          4500 0038 071e 0000 8001 af81 c0a8 01c8
                          c0a8 010d 0303 f065 0000 0000 4500 0062
                          0001 4000 ff11 f763 c0a8 010d c0a8 01c8
                          0035 ffff 004e 0c14 a8c0 2a06
```

The results are being examined and compared with the fingerprinting database, and `scores` are being calculated according to each fingerprinting module.

The Operating System Fingerprint that `Xprobe2` returns is the most probable match, taken from the sum of all `scores`. The answer also reflects other possibilities:

```
[+] Primary guess:
[+] Host 192.168.1.200 Running OS: "Microsoft Windows 2000/2000SP1/2000SP2"
(Guess probability: 68%)
[+] Other guesses:
[+] Host 192.168.1.200 Running OS: "Microsoft Windows XP Professional" (Guess
probability: 68%)
[+] Host 192.168.1.200 Running OS: "Microsoft Windows ME" (Guess probability:
63%)
[+] Host 192.168.1.200 Running OS: "Microsoft Windows NT 4 Service Pack 4 and
Above" (Guess probability: 59%)
[+] Host 192.168.1.200 Running OS: "NetBSD 1.5.2" (Guess probability: 59%)
```

With our example, Microsoft Windows XP and Microsoft Windows 2000 have received the same `score`. This is due to the fact that they share a very similar TCP/IP stack.

6.0 Conclusion

We believe that using a fuzzy approach to remote active operating system fingerprinting produces better and more accurate results than with the traditional strict signature matching approach taken today with other remote active operating system fingerprinting tools. The fuzzy approach is more resistant to network topology issues, filtering devices, external failures, and 'noise' reduction methods with remote TCP/IP stack probing.

7.0 Related Work and Reference

Arkin Ofir, “ICMP Usage in Scanning” research project
http://www.sys-security.com

Arkin Ofir, “ICMP Usage in Scanning” version 3.0, June 2001
http://www.sys-security.com/html/projects/icmp.html

Arkin Ofir & Fyodor Yarochkin, “X – Remote ICMP based OS fingerprinting Techniques”, August 2001 (This paper describes the first generation of `Xprobe`).
http://www.sys-security.com/archive/papers/X_v1.0.pdf

Arkin Ofir & Fyodor Yarochkin, “ICMP based remote OS TCP/IP stack fingerprinting techniques”, Phrack Magazine, Volume 11, Issue 57, File 7 of 12, Published August 11, 2001.
http://www.sys-security.com/archive/phrack/p57-0x07

Fyodor[14], “Remote OS detection via TCP/IP Stack Finger Printing”, October 1998
http://www.phrack.com/show.php?p=54&a=9

Franck Veysset, Olivier Courtay, Olivier Heen, “New Tool and Technique for Remote Operating System Fingerprinting” (Response timing calculation based fingerprinting), April 2002
http://www.intranode.com/site/techno/techno_articles.htm

Xprobe’s Homepage
http://www.xprobe.org
http://www.xprobe2.org

8.0 Credits

We would like to thank the following people for contributing time, code and effort into `Xprobe` v2:

- Meder Kydyraliev (meder@areopag.net)
- Ilya Levin

We would like to thank the following people for reviewing the paper:

- Saumil Shah
- Mike Poor

[14] This is a different Fyodor.