SinFP, Unification Of Active And Passive Operating System Fingerprinting

Patrice Auffret

Thomson Corporate Research
Security Labs
1, av. Belle-Fontaine - CS 17616
35576 Cesson-Sévigné - France
patrice.auffret@thomson.net

Abstract. Since omnipresence of firewalls using Network Address Translation and Port Address Translation (*NAT/PAT*), stateful inspection, and packet normalization technologies, today's approaches to operating system fingerprinting are showing their limits. With this fact in mind, *SinFP* was developed to attempt to address limitations of nowadays tools. *SinFP* implements new methods, like the usage of signatures acquired by active fingerprinting when performing passive fingerprinting. Furthermore, *SinFP* is the first tool performing operating system fingerprinting on *IPv6* (both active and passive modes). Thanks to its signature matching algorithm, it is near to useless to add new signatures in its current database. Also, its heuristic matching algorithm makes it highly resilient against signatures that have been modified by routing and/or filtering devices in-between, or against *TCP/IP* customization methods. This document presents an in-depth explanation of technics implemented within *SinFP* tool.

1 Introduction

First version of *SinFP* was released on June, 2005 [3]. The code has much evolved since this date and is now quite mature. Why then publish an article three years after ? Just because some publications were made about operating system fingerprinting. Some of them [4][5][6] cite *SinFP* but introduce some false information about its inner workings, or worse, re-invent *SinFP* [7]. Also[1], *SinFP* is not either an active operating system fingerprinting tool or a passive one, but truly both an active and a passive one.

In this article, we will not explain concepts behind operating system fingerprinting. We consider them as known by the reader. The following article [1] introduces these concepts. General principle and supported features of the tool will be covered in section 2, inner workings of active operating system fingerprinting in section 3 and inner workings of passive one in section 4. Matching

[1] We can find on various blogs and/or forums some false information about *SinFP*'s possibilities.

algorithm will be covered in section 5 and deformation masks (the most important concept introduced by *SinFP*) in section 5. Finally, before conclusion (section 7), we will show *IDS* (*Intrusion Detection System*) evasion methods in section 6.

2 Conception principle

SinFP's origin was based upon one question: is it possible, in worst network conditions, to reliably identify a remote operating system ? Worst network conditions being:

1. only one open *TCP* port;
2. all other ports (*TCP* and *UDP*) are dropped by a filtering device;
3. a filtering device with stateful inspection configured on the open port;
4. a filtering device with packet normalization.

In such a configuration scenario only standard frames (that follow *IETF* standards) reach the target, thus eliciting a response frame. In this context, it is mandatory to use only standard probe frames that use only *TCP* protocol to be able to build a reliable signature from the target operating system.

To choose the first probe frame, with the constrain for it to be able to reach the target whatever filtering device configuration there is, we have captured a standard frame. We choosed one generated by a call to `connect()` system call (in our case, the one from a *Linux 2.4.x* operating system). This frame implements many *TCP* options. The second probe frame is a copy and paste of the first, but with *TCP* options removed. These two probe frames will elicit two responses from the target. These responses will be two *TCP SYN+ACK* frames. In order to be able to build a signature with the most caracteristics, we have added a third probe frame with the objective to make the target emit a *TCP RST+ACK* response frame. This third request has no *TCP* options and has *TCP SYN+ACK* flags set. All of these frames are targetted to the same open *TCP* port.

After those useful frames are sent to the target and responses received, the following analyze is done: all fields that may depend over a *TCP/IP* stack implementation or another are analyzed. Some fields are entirely random from a system to another, and are thus not meaningful for our analysis. But all the other are analyzed. This includes all *IP* and *TCP* header fields and sometimes the application layer data[2]. Some fields have random values like values from *TCP timestamp* option, it is thus mandatory to format them in a special way. We will see how in section 3.1.2.

After the three probes are launched and the three replies have been received, a signature is built from the analysis of frames. Then, a signature matching algorithm searches in a database for a corresponding operating system fingerprint. *SinFP* uses an algorithm which could be compared to a Web search engine. The goal being to search for an intersection between multiple domains.

[2] HP-UX 11.x adds the string `No TCP` at the application layer while emitting a *TCP RST+ACK* frame.

Once active fingerprinting has been implemented, a port to passive fingerprinting has been written. Then came the port to *IPv6* [10] fingerprinting, which was a simple search for equivalence between *IPv4* [9] header fields and *IPv6* header fields.

To summarize, here is the list of features supported by *SinFP*:

1. *IPv4* active and passive operating system fingerprinting;
2. *IPv6* active and passive operating system fingerprinting;
3. if an *IPv6* match is not found in the database, it is possible to match against *IPv4* signatures;
4. passive fingerprinting online and offline;
5. active fingerprinting replayable offline against a *pcap* file generated while online;
6. possibility to launch only a subset of probes to avoid *IDS* detection;
7. heuristic matching algorithm able to identify target operating system even with customization on its *TCP/IP* stack;
8. a *SQL* fingerprint database in *SQLite* [21] format;
9. easy integration within another program because the tool is written in the form of a *Perl* module.

Each of these concepts are analyzed in-depth in the following paragraphs.

3 Active operating system fingerprinting

The main principle for active fingerprinting is to send requests (or probes, or tests) in a known format and specially choosed to a target system and to analyze responses received in order to build the most discriminant signature possible. *SinFP* sends at the maximum three requests, all standard frames, to the same open *TCP* [11] port. The first request is a *TCP SYN* without any *TCP* option (test *P1*), the second is a *TCP SYN* with many *TCP* options (test *P2*), and the third is a *TCP SYN+ACK* (test *P3*).

The first two requests will make the target elicit[3] two *TCP* responses with *SYN+ACK* flags. This is the second phase of the establishment of a *TCP* connection. The third and last request will make the target emit[4] a *TCP RST+ACK* response.

Nowadays, the most common approach [8] is to send a great number of requests against different ports and different transport layer protocols. This approach is flawed by design. If the target is behind multiple filtering devices, each with a different configuration policy, you may end up with a signature that is built using response frames issued by different systems. Imagine the first filtering device answers to *TCP SYN+ACK* frames with *TCP RST+ACK* in place of the true target, and also spoofs the *IP* address of the true target, you will have

[3] Some *TCP/IP* stacks or filtering devices do not respond to a *TCP SYN* request which has no *TCP MSS* option.

[4] Except when a filtering device exists and is configured with stateful inspection turned on.

a signature that is built with some frames issued by the good target and some other issued from a wrong operating system. Using this approach, you will never know if the matched signature is the good one, or the wrong one. *SinFP* solves this problem by using only probe frames that will reach the good target and not some device in-between.

But, if an active fingerprinting is launched with the three requests (default mode in *SinFP*), we are in the same problematic situation as described above. This is due to the test *P3* which may be answered by a filtering device in-between. If a firewall in-between is configured with stateful filtering policy and to send a *TCP RST+ACK* by spoofing the *IP* address of the protected system, we will obtain a signature with the response to the test *P3* obtained from the wrong target. Because this case is not that common, this is the default behavior. Thus, it is possible to only launch some of the three tests[5] [6]. For instance, in the previously mentioned problematic case, you must launch *SinFP* in the mode where only tests *P1* and *P2* are sent, so we are sure that only the good target responses will be used in the analysis phase to build a trusted signature.

In all cases, the absence of a response for a request is never used as an element to build a signature. Only responses received are used for the creation of a trusted signature. See figure 1 for an example usage of the tool.

```
% sinfp.pl -ai 10.100.0.22 -p 22
P1: B11013 F0x12 W65535 O0204ffff M1460
P2: B11013 F0x12 W65535 O0204ffff010303000101080affffffff44454144 M1460
P3: B11020 F0x04 W0 O0 M0
IPv4: BH1FH0WH0OH0MH0/P1P2P3: BSD: Darwin: 8.6.0
```

Fig. 1. Example of active fingerprinting

Now, we will explain requests format and how responses are analyzed. A good understanding of *IPv4*, *IPv6* and *TCP* headers is greatly recommanded.

3.1 Over *IPv4*

All three requests are fully compliant with standards. They will make the target to emit three responses. Once these responses are received, a signature is build from the analyze. In the following sections, we will see how we obtain a signature from the analyze of request/response couples. Signature format will be fully explained in section 5.1.

3.1.1 *IPv4* headers analysis

[5] Via `-2` parameter from the command line to only launch tests *P1* and *P2*.
[6] Via `-1` parameter from the command line to only launch test *P2*.

TTL field: some systems[7] do not set the same *TTL* value when they emit a *TCP SYN+ACK* and when they emit a *TCP RST+ACK*. Thus, we analyze the difference between the response's *TTL* from test *P3* with the one from test *P2*. If response's *TTL* from *P3* is different than the one from *P2*, we set a constant value to 0. Otherwise we set it to 1. This constant value is always 1 for the response to *P1* and *P2*.

ID field: this is a comparison between request's *ID* and response's *ID*. If response's *ID* is 0, we set a constant value to 0. If it is the same as the one from the request, we set the constant value to 2. If it is an increment by 1, we set it to 3. In all other cases, it is set to 1. Because the *ID* may be modified by a filtering device in-between, we will see how the matching algorithm handles this case in a generic way in section 5.3.

Don't Fragment bit: if response has *Don't Fragment* bit set, a constant value is set to 1, otherwise to 0.

3.1.2 *TCP* headers analysis

Sequence number field: it is also a comparison like *ID* field from *IPv4* headers analysis. But we compare the *TCP* sequence number from request with the one from response. If the sequence number from the response is 0, we set a constant value to 0. If it is the same as in the request, we set the constant value to 2. If it is an increment by 1, we set it to 3. In all other cases, the constant value is set to 1.

Acknowledgment number: the same analysis as for the sequence number is applied but with the acknowledgment number.

TCP flags and *TCP* window size: they are copied as-is to the generated signature.

TCP options: they are also copied as-is to the generated signature but with the following modifications applied:

1. *MSS* (*Maximum Segment Size*) value is extracted (if present) in order to create its own signature's element. Also, it is replaced by `ffff` in the option element;
2. if *timestamp* values are greater than zero, we replace them by `ffffffff`.

MSS value is extracted to simplify the process of writing regular expressions (more about this subject in section 5.3). The same is true for *timestamp*, we only need to know if they are different than the value 0.

Once all these analysis are done, we obtain the target's signature (or fingerprint). More about signature's format in section 5.1.

3.2 Over *IPv6*

The only difference with fingerprinting over *IPv4* is the *IPv6* headers because the *TCP* analysis part remains exactly the same. So to make *SinFP* to support

[7] *SunOS* is one of them.

IPv6 fingerprinting, we search which fields may be equivalent between *IPv4* and *IPv6*.

The equivalence we found is as follows:

1. *IPv4 ID* => *IPv6 Flow Label*;
2. *IPv4 TTL* => *IPv6 Hop Limit*;
3. *Don't Fragment* flag => *IPv6 Traffic Class*.

This is the only difference between active fingerprinting over *IPv4* and the one over *IPv6*.

3.3 Active fingerprinting limitations

When there are too few *TCP* options within test *P2*'s response (usually when option's element from *P2*'s response is equal to `0204ffff`), signature's entropy becomes weak. In fact, *TCP* options are the most discriminant caracteristics that compose a signature. That's because nearly no system implements exactly the same *TCP* options, nor in the same order. So when there is only `0204ffff` option in the *TCP* header, we only have one option to analyze (the *MSS* value). In such a case, a risk of mis-identification arises, and *SinFP* tool displays a warning message to the user.

Another problem is that, in most cases, the target sends a response to test *P1* but sometimes the target does not send one. In such a case, either an unknown signature is found or a mis-identification is performed. Thus, it is necessary to re-launch *SinFP* by only using test *P2*. In the vast majority of cases, a correct detection is returned.

In some other cases, no match is found even when all requests have received their responses. It is thus necessary to launch identification process by activating advanced deformation masks[8]. More details about deformation masks in section 5.3. If there is still no match, we have found a new signature.

3.4 Other features

Each time a fingerprinting attempt is launched, a trace file (in *pcap* format) is generated. It allows the matching algorithm to be replayed offline. Thus, it is possible to use a different signature database or to use a custom deformation mask (see section 5.3).

This generated file is anonymized by default. If an unknown signature is encountered you may send the generated file to *SinFP discuss* mailing list [12] with the exact operating system version.

This file also allows to replay the matching algorithm. We have seen it previsouly, sometimes no match is found. Thus, it is possible to re-launch *SinFP* by activating advanced deformation masks and by specifying on which *pcap* file to launch the matching algorithm[9].

[8] Via `-H` parameter from the command line.
[9] Via `-f` parameter from the command line.

4 Passive operating system fingerprinting

This mode works either in offline mode (by analysing a *pcap* file) or in online mode (by sniffing over the network). This is a perfect mode for integration within an *IDS* or a firewall.

But porting active mode to passive mode has not been as easy as porting *IPv4* active mode to *IPv6* active mode. All analysis that are made by comparing the response with the request are not possible in passive mode because we only have access to responses (we do not send any tests, by definition). Thus, passive signatures and active signatures are not compatible.

Furthermore, the signature database does know only responses to *TCP SYN* requests, that is only *TCP SYN+ACK* or *TCP RST+ACK* responses. In order to be able to analyze *TCP SYN* packets, a modification must be done against captured frame. We will see the detail in section 4.3. For now, see figure 2 for an example of passive fingerprinting.

```
% sinfp.pl -Pf ~/sinfp4-passive.pcap
10.100.0.1:80 > 10.100.0.68:39503 [SYN|ACK]
P2: B10111 F0x12 W5672 O0204ffff0402080affffffffffffffff01030306 M1430
IPv4: BH0FH0WH10H0MH1/P2: GNU/Linux: Linux: 2.6.x
```

Fig. 2. Example of passive fingerprinting

In the following sections, we will detail how we modify a passive signature to become compatible with active signatures.

4.1 *IPv4* headers analysis

Constant value concerning the *TTL* field is always set to 1, because, as explained before, we cannot compare responses versus requests (there are no requests). This is not a problem for the matching algorithm because frames we analyze in passive mode are equivalent to those we get in response to test *P2* in active mode. Those responses also set the constant value from *TTL* field to 1.

Constant value concerning the *ID* field is set to 1 if the *ID* from captured frame is greater than 0 or set to 0 otherwise. In active mode, this constant value may be greater than 1. We will see in section 4.4 how we handle this case.

Don't Fragment bit analysis needs no modification compared to the active mode.

4.2 *IPv6* headers analysis

We have shown that there is an equivalence between *IPv4* and *IPv6* headers. Thus, same modifications are made on *IPv4* headers and on *IPv6* headers.

4.3 *TCP* headers analysis

If the *TCP* sequence number has a value greater than 0, the corresponding constant value is set to 1. The constant value concerning the *TCP* acknowledgment number follows the same logic.

As previously seen, we have only access to responses, thus we cannot compare responses against requests. We need a special handling of this case because constant values obtained in active mode may have values greater than 1. We will see in section 4.4 how this is handled.

TCP flags are modified on the captured frame. If flags are set to *SYN+ACK*, no modification are made. If flags are set to *SYN*, we replace them by *SYN+ACK*. Thus, the captured frame will look like a frame obtained from the response to a request made in active mode and will have a corresponding signature in the database. We must do this modification because we only have *TCP SYN+ACK* and *TCP RST+ACK* in the signature database. By doing this modification, we now can also fingerprint *TCP SYN* packets.

Analysis method for other fields do not change compared to active fingerprinting.

4.4 Passive matching algorithm

Only one last difficulty before passive fingerprinting may work nearly like active fingerprinting. Signatures in the database are active signatures. They are taken from controlled probes, so their format is controlled. In passive mode, we do not send probes. Thus, in passive mode, we do not control analyzed frames. The result is an incompatibility between active fingerprinting signatures against passive fingerprinting signatures. For exampe, some fields in active signatures are issued from a comparison between a request and a response. In passive mode, we cannot compare them. The solution we have adopted is to modify on-the-fly signatures when they are extracted from the database while searching for a match. Modification is just to replace constant values issued from a comparison (values greater than 1) by the value 1.

Now, active signatures taken from the database look like passive signatures and we can compare them. Also, there is no need to modify the matching algorithm, it remains exactly the same for active fingerprinting and for passive fingerprinting and there is also no need to have some specific passive signatures in the database. Port to passive fingerprinting is now complete.

4.5 Passive fingerprinting limitations

As a general fact, passive fingerprinting suffers from a limitation and *SinFP* also suffers from it. Because there are no requests sent to the target, we cannot control how the target builds its responses. As explained before, a response is crafted relatively to a request. In passive mode, some frames that are analyzed are responses to probes we do not have sent. Thus, they are responses crafted relatively to frames we have no control against and the signature for

a same operating system may vary depending on which operating system has attempted a connection to it. In active mode, requests are fully controlled by the fingerprinting tool and this problem does not exist.

We have seen that a response is dependant upon a request's format. In passive mode, responses we will analyze may vary depending on the source system that has elicited these responses (like a source system connecting to a *TCP* port of the target we want to fingerprint). Thus, there may be more deformation in passive mode than in active mode. The heuristic matching algorithm helped by deformation masks are the solution to this problem.

5 Matching algorithm

This algorithm is similar to a Web search engine [13]. The goal is to find intersection between multiple domains. To better understand its inner workings we need to first introduce signatures format, database format, what deformation masks are, and finally the search for a match.

5.1 Signatures format

A signature (see figure 3) is composed from three responses (*P1(R)*, *P2(R)* and *P3(R)*). Each response is composed by 5 elements. Thus, a full signature is composed by 3 x 5 elements, a total of 15.

```
B11113 F0x12 W65535 O0204ffff M1460
B11113 F0x12 W65535 O0204ffff010303000101080affffffff44454144 M1460
B11020 F0x04 W0 O0 M0
```

Fig. 3. Signature for *Darwin 8.6.0* operating system

15 elements may seem too few to have a unique signature. Some other tools like *nmap* [14] have far more elements [15]. But in practice, 15 elements are enough to correctly identify an operating system in a reliable manner. Furthermore, in most cases, only 5 elements are enough[10].

Each response to a request has the following elements:

1. B: a list of constant values (example: `B11013`);
2. F: *TCP* flags (example: `F0x12`);
3. W: *TCP* window size (example: `W65535`);
4. O: *TCP* options, rewritten to ignore random values like *timestamp*s (example: `O0204ffff010303000101080affffffff44454144`);
5. M: *MSS* size (example: `M1460`).

[10] By only launching test *P2* which gives the most information regarding the target *TCP/IP* stack.

In figure 1, first three lines are the three responses (*P1(R)*, *P2(R)* and *P3(R)*) to the three requests (*P1*, *P2* and *P3*). They compose an operating system signature. The final line is the matched fingerprint from the database.

This last line is composed from multiple elements. The first is the type of match found (`BH1FHOWHOOHOMHO`); this is the deformation mask which has allowed a match to be found. The second element (`P1P2P3`) tells which responses have found a match in the database for the selected deformation mask. In our example, all three responses have found a match. Finally, we have information regarding the target operating system: its class (`BSD`), its name (`Darwin`) and its version (`8.6.0`).

5.2 Signatures database

Each of the 15 elements have an *ID* in a relational database. These elements are common to all database signatures and only the unique attribute of the string determines its *ID*. Thus, each element taken appart is independant from a signature and independant from an operating system.

For example, element `W65535` may be common to many operating systems. Thus, each operating system that have a value of 65535 for its *TCP* window size has the *ID* of `W65535` in its signature. A signature, from the viewpoint of a relational database, is just a list of *ID*s.

Each operating system (with its version) have only one signature in the database. Within *SinFP*, we do not add a signature for a system just because the target has deactivated one *TCP* option. The matching algorithm, thanks to deformation masks, handles such common cases.

A signatures database must be clean. All signatures are not eligible for inclusion. In fact, adding a bad signature may increase the risk for mis-identification. Thus, if a doubt exists in regard to the possible existance of a filtering device while fingerprinting a new system, the signature is simply not added to the database.

To be eligible for inclusion in the database, the following perfect conditions shall be met:

1. no filtering device in-between;
2. no routing device in-between;
3. at least one open *TCP* port.

To summarize, perfect conditions are either link-local access to the target or a target that runs on the local system (like in a virtual machine). All other network conditions brings an unacceptable doubt and will result, sooner or later, in an inconsistant database. *nmap* had this problem and has tried to correct it since its 2nd generation of operating system fingerprinting engine by creating a new and clean database.

5.3 Deformation masks

Response to a request may be modified by a filtering device, be it in-between or directly on the target. Thus, *SinFP* introduces deformation masks. They are

implemented using regular expressions. Each element composing a signature has two regular expressions associated with it, plus the value taken from perfect conditions (we call this last heuristic 0). One is for level 1 (called heuristic 1) and the other for level 2 (called heuristic 2). Thanks to these heuristic values, it is possible to write deformation masks. A deformation mask is applied on a reference's signature taken in perfect conditions (see figure 4) when extracted from the database while searching for a match.

Reference's signature (with a perfect mask `HEURISTIC0`):

```
B10113 F0x12 W5840 O0204ffff M1460
B10113 F0x12 W5792 O0204ffff0402080afffffffff4445414401030306 M1460
B10120 F0x04 W0 O0 M0
```

Reference's signature after the `BH1FH0WH10H0MH1` mask is applied:

```
B...13 F0x12 W5[789].. O0204ffff M1[34]..
B...13 F0x12 W5[678].. O0204ffff0402080afffffffff4445414401030306 M1[34]..
B...20 F0x04 W0 O0 M0
```

Reference's signature after the `BH1FH0WH2OH1MH2` mask is applied:

```
B...13 F0x12 W\d+ O0204ffff M\d+
B...13 F0x12 W\d+ O0204ffff(?:0402)?(?:080afffffffff44454144)?(?:01)?(?:030306)? M\d+
B...20 F0x04 W0 O0 M0
```

Fig. 4. Deformation of reference's signature for the system *Linux 2.6.x* after three different masks have been applied

Each type of element allows some specific deformations. For instance, applicable deformation for O element is not the same as for F element. In F element case, no deformation is applicable at all.

For example, a value often modified by a routing device is the *TCP MSS* value. In perfect conditions, it often has the value 1460. But in many cases, we obtain a value of 1430, problably due to a router that has a *MTU* (*Maximum Transmission Unit*) less than perfect conditions value. Thus, heuristic 0 value (*H0*) is `M1460`. Now we write heuristic 1 value (*H1*) as `M1[34]..` and heuristic 2 value (*H2*) as `M\d+`. *H1* value allows a deformation for *TCP MSS* from 1300 to 1499. In *H2* mode, we simply totally ignore the *TCP MSS* value. Each type of element has its own set of heuristics.

We have seen that each element of a signature is unique in the database and also has a unique *ID*. Deformation masks are linked to an element, thus they are written for a given element and are uncorrelated from a system's signature. But it is still possible to write mask values specifically for a given operating system.

A deformation mask is the association of all masks for all signature's elements. Thus, if we take *Darwin* active fingerprinting example (figure 1), the match found is `BH1FH0WH0O0H0MH0`. A perfect match is `BH0FH0WH0O0H0MH0`, or more

simply wirtten `HEURISTIC0`. This is the most reliable mask. In our example, we do not have found a perfect match but a small deformation on *B* element. This deformation is with heuristic 1 (thus `BH1`). This result is considered as very reliable because other elements have no deformation at all.

In *SinFP*, there are two categories of deformation masks. The first category has a list of deformation masks which are quite reliable because they allow only small deformation in regards to the perfect signature. They are standard masks and are at the number of eight (in *SinFP* version *2.06* [16]). The second category[11] may be used if no match has been found by using perfect or standard masks. But because it allows more deformation for the signature, it also brings more mis-identifications. They are advanced masks and are at the number of fourteen. Using them requires a very good understanding of the tool to be able to say that the matched operating system is reliable or not.

All these deformation masks are obtained from an empirical manner. Each time the tool is used, some deformations on responses are found. If a deformation arise often a specific mask is written and added into *SinFP*'s code. The most common deformations are due to routers that modify *MTU* or some filtering devices that modify some *IP* header fields.

Deformation masks are sorted from the least deformant to the most deformant. `HEURISTIC0` (`BH0FH0WH00H0MH0`) is the mask that accept the least deformations. `HEURISTIC2` (`BH2FH2WH20H2MH2`) is the mask that accepts the most deformations, thus, it is the less reliable mask. In the middle of these two masks, we have the intermediary `HEURISTIC1` (`BH1FH1WH10H1MH1`) mask, which is also classified in advanced masks. Between these three major masks, we have other masks that have been written empirically.

5.4 The search for a match in the database

For a signature to be matched, each element composing a signature needs to find a match in the database. For each element of *P1* (*E1*, *E2*, ..., *E5*), we search the list of *ID*s that match the pattern. Then, the matching algorithm searches signature *ID*s which are common to each obtained lists for each element (the intersection of domains *E1(P1)*, *E2(P1)*, ..., *E5(P1)*). This intersection gives us the domain *I(P1)*. This step is repeated for *P2* and *P3* in order to find respectively *I(P2)* and *I(P3)*.

The final match is the intersection of domains *I(P1)*, *I(P2)* and *I(P3)*, that is the list of signature *ID*s that are common to these three domains. If no match is found, we try by searching the intersection of *I(P1)* et *I(P2)*. If there is still no match, we apply the same search algoithm but with the next deformation mask (from the least deformant to the most deformant). The search is stopped as soon as one or more match are found for a given mask by trying all stored signatures.

In mathematic terms, the algorithm is written:

[11] Activable via `-H` parameter from the command line.

$I(P1) = E1(P1) \bigcap E2(P1) \bigcap ... \bigcap E5(P1)$
$I(P2) = E1(P2) \bigcap E2(P2) \bigcap ... \bigcap E5(P2)$
$I(P3) = E1(P3) \bigcap E2(P3) \bigcap ... \bigcap E5(P3)$
$I = I(P1) \bigcap I(P2) \bigcap I(P3)$
If I is null:
$I = I(P1) \bigcap I(P2)$

In passive mode, the algorithm is written:

$I = E1(P2) \bigcap E2(P2) \bigcap ... \bigcap E5(P2)$

The matching algorithm is the same in *IPv4* and in *IPv6*. We have seen it previously, there is an equivalence between *IPv4* header and *IPv6* header fields. Thus, it is directly possible to use *IPv4* signatures when doing *IPv6* fingerprinting. If no match is found for the target *IPv6* signature, it is possible to use *IPv4* ones[12] while searching for a match. While experimenting with this feature, we have confirmed that this "compatibility" mode is very reliable. The explanation comes from the fact that the *TCP* stack is nearly the same from *IPv4* to *IPv6*.

5.5 Deformation masks advanced usage

We have seen it, deformation masks are written empirically[13]. For example, the mask `BH0FH0WH2OH0MH0` was[14] necessary to correctly identify `www.openbsd.org` operating system (figure 5).

Operating system that ran on `www.openbsd.org` had a response very near to the one for a *SunOS 5.6* system but with a different *TCP* window size (536 for *P1* and 1460 for *P2*). By using a custom deformation mask, we ignore values for *TCP* window size (mask `WH2`) contained in responses. Thus, we find a match with a low heuristic, and the target system is correctly identified as *SunOS 5.6* (see figure 5). `www.openbsd.org` server is not the only one requiring such a specific deformation mask, thus we have added the mask to *SinFP*'s code. We suppose that a filtering and/or routing device in-between modifies *TCP* window sizes, or that *TCP/IP* stack has been customized.

As for signatures, it is highly important to choose deformation masks judiciously. If deformation masks accepting huge deformation are added to the code, all stored signatures may look the same and a match will display many different operating systems. The choice wether to add or not a new deformation mask is a manual process which requires a strong expertise of the tool.

---

[12] Via `-4` parameter from the command line.

[13] `-A` parameter from the command line allows to test new deformation masks before adding them to the code.

[14] Was, because it is no more useful today, the targetted server seems to have changed its network architecture.

SinFP launched in active mode with a custom deformation mask to be able to identify `www.openbsd.org` operating system:

```
% sinfp.pl -ai www.openbsd.org -p 80 -A BH0FH0WH2OH0MH0
P1: B11113 F0x12 W536 O0204ffff M536
P2: B11113 F0x12 W1460 O0101080affffffff4445414401030300020 4ffff M1460
P3: B01120 F0x04 W0 O0 M0
IPv4: BH0FH0WH2OH0MH0/P1P2P3: Unix: SunOS: 5.6
```

SunOS 5.6 reference's signature as stored in the database:

```
B11113 F0x12 W9112 O0204ffff M536
B11113 F0x12 W10136 O0101080affffffff4445414401030300020 4ffff M1460
B01120 F0x04 W0 O0 M0
```

Fig. 5. Fingerprinting `www.openbsd.org` server

6 *IDS* evasion methods

SinFP uses standard requests, it is thus hard to write *IDS* rules to detect the use of *SinFP* on a network. It may be possible because when *SinFP* is used in the default mode it sends two *TCP SYN* and one *TCP SYN+ACK* packets in a short timeframe. These events may be put in an *IDS* rule.

But other modes activable from *SinFP*'s command line may be used to bypass *IDS*s:

1. `-3`: launch all requests (default mode);
2. `-2`: launch only request one and two;
3. `-1`: launch only request two.

Launching requests one and two remains identifiable by an *IDS* but may introduce some false positives. Launching only request two is far harder to detect for an *IDS* without many false positives because a *TCP SYN* with some *TCP* options is a standard packet seen every time a *TCP* connection is established. But the operating system launching the fingerprinting will send a *TCP RST* packet after it receives the response to request two. That's because the *TCP/IP* stack has no knowledge of the packet sent by *SinFP* (this is a manually crafted frame, not one sent by the operating system *TCP/IP* stack). In passive mode, we can avoid that because we use the operating system *TCP/IP* stack to establish a *TCP* connection that will be used to fingerprint the target.

Thus, another mode exist: mixed mode. It is active and passive at the same time. To use it, we start *SinFP* in passive mode and we establish a *TCP* connection to a target we want to fingerprint, for example by using a Web browser (see figure 6 for an example). In this example, the result is very reliable because only a minor deformation exists on the *B* element, a deformation in heuristic 1 (`BH1`).

SinFP launched in passive mode within a terminal, while a connection is established to `www.sstic.org` using a Web browser:

```
% sinfp.pl -PF 'host www.sstic.org and src port 80'
88.191.41.247:80 > 192.168.0.101:60623 [SYN|ACK]
P2: B11111 F0x12 W5792 O0204ffff0402080affffffffffffffff01030305 M1460
IPv4: BH1FHOWHOOHOMHO/P2: GNU/Linux: Linux: 2.6.x
```

Fig. 6. Fingerprinting example using active/passive mode

7 Conclusion

In this article, we have described in-depth choices and implementations of *SinFP*. We have shown how it was possible to unify active and passive fingerprinting in the same tool with a unique category of signature taken from an active manner. Furthermore, *SinFP* is the first public tool to implement fingerprinting over *IPv6*, both active and passive.

The matching algorithm looking like a Web search engine gives excellent results, especially associated with deformation masks. Nevertheless, some cases still give some mis-identifications and we have some possible solutions to limit them. These solutions are not yet implemented in the tool but may be the subject of another publication.

In the meantime, if you wish to compare *SinFP* active fingerprinting against *nmap* active fingerprinting, you may consult the following sites [17][18][19][20]. Finally, some useful *SinFP*'s tips and tricks for your daily usage may be found at [22].

References

1. Prise d'empreinte active des systèmes d'exploitation
   `http://www.gomor.org/bin/view/GomorOrg/Misc7`
2. SinFP OS fingerprinting tool
   `http://www.gomor.org/bin/view/Sinfp/WebHome`
3. `Net::SinFP 0.92`
   `http://search.cpan.org/~gomor/Net-SinFP-0.92/`
4. Stateful Passive Fingerprinting for Malicious Packet Identification
   `http://www.andrew.cmu.edu/user/xsk/XenoKovahThesis.pdf`
5. IPv6 Neighbor Discovery Protocol based OS fingerprinting
   `http://hal.inria.fr/docs/00/16/99/90/PDF/technical_report_fingerprinting.pdf`
6. A Hybrid Approach to Operating System Discovery using Answer Set Programming
   `http://ieeexplore.ieee.org/iel5/4258513/4258514/04258556.pdf?tp=&isnumber=&arnumber=4258556`
7. Toward Undetected Operating System Fingerprinting
   `http://www.usenix.org/events/woot07/tech/full_papers/greenwald/greenwald.pdf`

8. Remote OS Detection using TCP/IP Fingerprinting (2nd Generation)
http://insecure.org/nmap/osdetect/
9. Internet Protocol (version 4)
ftp://ftp.rfc-editor.org/in-notes/rfc791.txt
10. Internet Protocol (version 6)
ftp://ftp.rfc-editor.org/in-notes/rfc2460.txt
11. Transmission Control Protocol
ftp://ftp.rfc-editor.org/in-notes/rfc793.txt
12. sinfp – News about SinFP
http://lists.gomor.org/mailman/listinfo/sinfp
13. Analyse fine : bornes inférieures et algorithmes de calculs d'intersection pour moteurs de recherche
http://www.cs.uwaterloo.ca/~jbarbay/Recherche/Publishing/Publications/these.pdf
14. Nmap - Free Security Scanner For Network Exploration and Security Audits
http://insecure.org/nmap/
15. TCP/IP Fingerprinting Methods Supported by Nmap
http://insecure.org/nmap/osdetect/osdetect-methods.html
16. Net::SinFP 2.06
http://search.cpan.org/~gomor/Net-SinFP-2.06/
17. SinFP vs Nmap
http://www.computerdefense.org/2006/12/04/sinfp-vs-nmap/
18. Nmap vs SinFP
http://www.computerdefense.org/2006/12/08/nmap-vs-sinfp/
19. Introduction and Comparison with Nmap 4.10, Part I
http://www.phocean.net/?p=13
20. Comparison with Nmap 4.20, Part II
http://www.phocean.net/?p=14
21. SQLite Home Page
http://www.sqlite.org/
22. Tips and Tricks
http://www.gomor.org/bin/view/Sinfp/DocTipsAndTricks