J Comput Virol (2010) 6:197–205
DOI 10.1007/s11416-008-0107-z

ORIGINAL PAPER

SinFP, unification of active and passive operating system fingerprinting

Patrice Auffret

Received: 27 August 2008 / Accepted: 29 October 2008 / Published online: 26 November 2008
© Springer-Verlag France 2008

Abstract The ubiquity of firewalls using Network Address Translation and Port Address Translation (*NAT*/*PAT*), stateful inspection, and packet normalization technologies is taking its toll on today's approaches to operating system fingerprinting. Hence, *SinFP* was developed attempting to address the limitations of current tools. *SinFP* implements new methods, like the usage of signatures acquired by active fingerprinting when performing passive fingerprinting. Furthermore, *SinFP* is the first tool to perform operating system fingerprinting on *IPv6* (both active and passive modes). Thanks to its signature matching algorithm, it is almost superfluous to add new signatures to its current database. In addition, its heuristic matching algorithm makes it highly resilient against signatures that have been modified by intermediate routing and/or filtering devices in-between, and against *TCP/IP* customization methods. This document presents an in-depth explanation of techniques implemented by *SinFP* tool.

1 Introduction

The first version of *SinFP* was released in June of 2005 [1]. The code has much evolved since this date and is now quite mature. Why then publish an article three years after release date? We want to set the record straight: Some publications about operating system fingerprinting [2–4] cite *SinFP* but introduce erroneous information about its inner workings, or worse, re-invent *SinFP* [5]. It is important to note[1] that *SinFP* is neither exclusively an active operating system fingerprinting tool, nor a passive one, but truly implements both approaches.

In this article, we will not explain the concepts behind operating system fingerprinting. We consider them known by the reader. As a short review, this article may suffice [6]. The rest of the paper is organized as follows: The general principles and supported features of *SinFP* will be covered in Sect. 2. The inner workings of the active and passive operating system fingerprinting is described in Sects. 3 and 4, respectively. Matching algorithm will be reviewed in Sect. 5 and deformation masks—the most important concept introduced by *SinFP*—in Sect. 5. Finally, before reaching the concluding section (Sect. 7), we will review *IDS* (*Intrusion Detection System*) evasion methods in Sect. 6.

P. Auffret (✉)
Thomson Corporate Research Security Labs,
1, av. Belle-Fontaine, CS 17616,
35576 Cesson-Sévigné, France
e-mail: patrice.auffret@thomson.net

2 Conception rationale

SinFP was designed to answer one question: Is it possible, under worst network conditions, to reliably identify a remote operating system? Worst network conditions include following condition:

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.

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

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

We choose the first probe frame, with the constraint that it is to reach the target without any regard to intermediate filtering device configurations. We choose one generated by a `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 build a signature with the most salient characteristics, we add 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 but *TCP SYN+ACK* flags set. All of these frames are sent to the same open *TCP* port.

After those probe frames are sent to the target and responses received, the following analysis is performed: All fields that may depend on *TCP/IP* stack implementation or another are analyzed. Some fields are entirely random from a system to the next, and are thus not meaningful to our analysis. But all the other are; these include all *IP* and *TCP* header fields, and sometimes the application layer data.[2] Some fields have random values like values from the *TCP timestamp* option; it is thus necessary to format them in a special way. We will see how in Sect. 3.1.2.

After a signature is built from the preceding frame analysis, a signature matching algorithm searches in a database for a corresponding operating system fingerprint. *SinFP* uses an algorithm which could be compared to those used in Web search engines: The goal is to search for an intersection of multiple domains.

Once active fingerprinting had been implemented, a port to passive fingerprinting was written. Then came the port to *IPv6* [7] fingerprinting, which was a simple search for equivalence between *IPv4* [8] 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 of launching 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* [9] format;
9. Easy integration with other programs because the tool is written as a *Perl* module.

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

3 Active operating system fingerprinting

We review: the rationale for active fingerprinting is to send requests (or probes, or tests) in a known format specifically targeted towards a system, and to analyze responses received to build the most discriminant signature possible. *SinFP* sends three requests at most, all standard frames, to the same open *TCP* [10] port. The first request is a *TCP SYN* without any *TCP* option (test *P1*) set, the second is a *TCP SYN* with many *TCP* options (test *P2*) set, and the third is a *TCP SYN+ACK* (test *P3*) frame.

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

Nowadays, a common fingerprinting approach [11] is to send a great number of requests against different ports and different transport layer protocols. This approach is inherently flawed and misleading. If the target is behind multiple filtering devices, each with a different configuration policy, one may end up with a signature that is built using response frames issued by different systems. Imagine the first filtering device responding to *TCP SYN+ACK* frames with *TCP RST+ACK* instead of the true target, by spoofing the *IP* address of the proper target. As a result, a signature is obtained with characteristics of frames issued by the proper target, and some from a wrong operating system. Using this approach, you will never know if the matched signature is the correct one. *SinFP* solves this problem by using only probe frames that will reach the proper target and not some intermediate.

We may encounter, however, a similar problematic situation with our three requests active fingerprinting (default mode in *SinFP*). This is due to the test *P3* which may be answered by an intermediate filtering device. If a firewall

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

[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.

Fig. 1 Example of active fingerprinting

```
% 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
```

is configured with stateful filtering policy and send a *TCP RST+ACK* while spoofing the *IP* address of the target system, we will obtain a signature with the wrong response to the test *P3*. Since this case is not that common, three probe frames remains the default behavior. However, it is possible to only launch some of the three tests.[5,6] For instance, given the previous problematic firewall case, we restrict *SinFP* to *P1* and *P2* test mode, so we are sure that only the correct 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 not factored into a signature. Only responses received are used for the creation of a trusted signature. See Fig. 1 for an example usage of the tool.

We will proceed to explain the request format, as well as the analysis of the responses. A good understanding of *IPv4*, *IPv6* and *TCP* headers is greatly recommended.

3.1 Over *IPv4*

All three requests are fully compliant with standards. They will make the target emit three responses. Once these responses are received, a signature is build from the subsequent analysis. In the following sections, we will see how a signature is obtained by the analysis of request/response tuples. The signature format will be fully explained in Sect. 5.1.

3.1.1 IPv4 *headers analysis*

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 the 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 the request *ID* and response *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 an intermediate filtering device, we will see how the matching algorithm handles this case in a generic way in Sect. 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: A comparison similar to the *ID* field from the *IPv4* header analysis. However, we compare the *TCP* sequence number from the 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 field: The same analysis as for the sequence number is applied to the acknowledgment number.

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

TCP options: They are also copied as-is into the generated signature, but with the following modifications:

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

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

This analysis yields the target's signature (or fingerprint). More about signature format in Sect. 5.1.

3.2 Over *IPv6*

The only difference between fingerprinting *IPv4* and *IPv6* are the *IPv6* headers. The *TCP* analysis part remains exactly the same. In order for *SinFP* to support *IPv6* fingerprinting, we search for equivalent fields 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*.

[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*.

[7] *SunOS* is one of them.

Fig. 2 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: BH0FH0WH1OH0MH1/P2: GNU/Linux: Linux: 2.6.x
```

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

3.3 Active fingerprinting limitations

When there are too few *TCP* options returned by test *P2*'s response (usually when option's element from *P2*'s response is equal to `0204ffff`), the signature's entropy becomes small. Hence, *TCP* options are the most discriminant characteristic of signatures. That is because practically no system implements exactly the same *TCP* options in the same order. So when there is only `0204ffff` option in the *TCP* header, we only have the option to analyze the *MSS* value. In such a case, a risk of misidentification arises, and *SinFP* tool displays a warning message to the user.

Another problem is that though in most cases, the target sends a response to test *P1*, it does not do so always. In such cases, either an unknown signature is found or a misidentification occurs. 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 yet other cases, no match is found even after all request responses have been received. It thus becomes necessary to launch an identification process through advanced *deformation masks*.[8] More details on deformation masks in Sect. 5.3. If there is still no match, we have detected a new signature.

3.4 Other features

Each time a fingerprinting attempt is launched, a trace file (in *pcap* format) is generated. This 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 Sect. 5.3).

The generated file is anonymized by default. If an unknown signature is encountered, one has the option to send the generated file to *SinFP discuss* mailing list [12] with the exact operating system version.

This file also allows replaying of the matching algorithm. We mentioned previously that sometimes, no match is initially found. In this situation, it is possible to re-launch *SinFP* by activating advanced deformation masks and by specifying on which *pcap* file to run 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 analyzing a *pcap* file) or in online mode (by sniffing over the network). Passive mode is thus perfect for integration with an *IDS* or a firewall.

Porting active mode to passive mode was not as straightforward as porting *IPv4* active mode to *IPv6* active mode. 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 knows 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 to the captured frame. We will see the details in Sect. 4.3. For now, we refer the reader Fig. 2 as an example of passive fingerprinting.

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

4.1 *IPv4* headers analysis

The constant value associated with 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.

The constant value associated with 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 Sect. 4.4 how we handle this case.

Don't Fragment bit analysis needs no modification from the active mode handling.

4.2 *IPv6* headers analysis

We have shown an equivalence between *IPv4* and *IPv6* headers. Thus, the 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

Fig. 3 Signature for *Darwin 8.6.0* operating system

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

associated with the *TCP* acknowledgment number follows the same logic.

Since we have only access to responses, we cannot compare responses against requests. Thus, we need special handling of this case, because constant values obtained in active mode may have values greater than 1. We will see in Sect. 4.4 how this is negotiated.

TCP flags are modified on the captured frame. If flags are set to *SYN+ACK*, no modification is 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 perform this modification because we only have *TCP SYN+ACK* and *TCP RST+ACK* in the signature database. Hence, we are now in the position to also fingerprint *TCP SYN* packets.

The analysis method for the other fields is unchanged from the active fingerprinting case.

4.4 Passive matching algorithm

Only one last difficulty remains before passive fingerprinting may work 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 and passive fingerprinting signatures. For example, some fields in active signatures are the result of a comparison between a targeted request and response. In passive mode, we cannot compare them. The solution we have adopted is to modify signatures on-the-fly when they are extracted from the database while searching for a match. Modification served just to replace constant values resulting from a comparison (values greater than 1) by the value 1.

Now, active signatures taken from the database look like passive signatures and thus, we can compare them. Additionally, 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. The port to passive fingerprinting is now complete.

4.5 Passive fingerprinting limitations

By design, passive fingerprinting is more limited than its active counterpart: 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 relative to a request. In passive mode, some frames are responses to probes we did not send. Thus, they are responses to frames we have no control over. The signature for the same operating system may vary depending on which operating system has attempted to connect 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 dependent on a request's format. In passive mode, responses 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. As a solution, we enrich our heuristic matching algorithm by using specific *deformation masks*.

5 Matching algorithm

This algorithm is similar to algorithms used in Web search engine [13]. The goal is to find intersection of multiple domains. To better understand the inner workings, we introduce in order the signature and database formats, deformation masks, and finally the search for a match.

5.1 Signatures format

A signature (see Fig. 3) is built from three responses (*P1(R)*, *P2(R)* and *P3(R)*). Each response is composed of five elements. Thus, a full signature contains 3×5 elements, a total of 15.

Fifteen elements may seem too few for a unique signature. Some other tools like *nmap* [14] have far more elements [15]. But in practice, 15 elements are sufficient to correctly and reliably identify an operating system. In most cases even, only five elements are enough.[10]

Each response to a request contains 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 *timestamps* (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.

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

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

```
B10113 F0x12 W5840 O00204ffff M1460
B10113 F0x12 W5792 O00204ffff0402080affffffff4445414401030306 M1460
B10120 F0x04 W0 O0 M0
```

Reference's signature after the `BH1FH0WH1OH0MH1` mask is applied:

```
B...13 F0x12 W5[789].. O00204ffff M1[34]..
B...13 F0x12 W5[678].. O00204ffff0402080affffffff4445414401030306 M1[34]..
B...20 F0x04 W0 O0 M0
```

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

```
B...13 F0x12 W\d+ O00204ffff M\d+
B...13 F0x12 W\d+ O00204ffff(?:0402)?(?:080affffffff44454144)?(?:01)?(?:030306)? M\d+
B...20 F0x04 W0 O0 M0
```

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

This last line is composed of multiple elements. The first is the type of match found (`BH1FH0WH0OH0MH0`); this is the deformation mask that allows a match to be found. The second element (`P1P2P3`) indicates which responses have been matched to 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 are mapped to 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 apart is independent of a signature and independent of an operating system.

For example, element `W65535` may be common to many operating systems. Thus, each operating system that has 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) has 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. Not all signatures are eligible for inclusion. In fact, adding a bad signature may increase the risk for misidentification. Thus, if doubt exists with regard to the possible existence 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 intermediate filtering device;
2. No intermediate routing device;
3. At least one open *TCP* port.

Perfect conditions are met in either link-local access to targets or targets that run on the local system (like in a virtual machine). All other network conditions introduce unacceptable ambiguity and will result, sooner or later, in an inconsistent database. *nmap* had this problem and has tried to correct it since its second generation of operating system fingerprinting engine by creating a new and clean database.

5.3 Deformation masks

Responses to a request may be modified by a filtering device, be it intermediate or directly on the target. Thus, *SinFP* introduces deformation masks. They are implemented using regular expressions. Each element which make up 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 to a reference's signature taken under perfect conditions (see Fig. 4) when extracted from the database while searching for a match.

Each element type allows for 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. Under perfect conditions, it often has the value 1460. But in many cases, we obtain a value of 1430, probably due to a router that has a *MTU* (*Maximum Transmission Unit*) set to less than the perfect condition 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 ignore the *TCP MSS* value. Each element type has its own set of heuristics.

We have seen that each element of a signature is unique in the database with an unique *ID*. Deformation masks are linked to an element, thus they are written for a given element and are uncorrelated to a system's signature. However, 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 (Fig. 1), the match found is `BH1FH0WH0O-H0MH0`. A perfect match is `BH0FH0WH0OH0MH0`, or written more simply `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 was performed with heuristic 1 (thus `BH1`). This result is considered very reliable because other elements have no deformation at all.

In *SinFP*, there are two categories of deformation masks. The first category contains a list of deformation masks which are quite reliable because they allow only small deformation with regard to the perfect signature. There are eight such standard masks (in *SinFP* version *2.06* [16]). The second category[11] may be used if no match is found with perfect or standard masks. But because it allows more deformation to the signature, it also runs the risk of more misidentifications. There are 14 such advanced masks. Using them requires a very good understanding of the tool in order to ascertain the reliability of the matched operating system.

All these deformation masks are obtained empirically. Each time the tool is used, some deformations on responses are found. If a deformation arise often enough, a specific mask is written and added to *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 deforming to the most deforming. `HEURISTIC0` (`BH0FH0WH0OH0MH0`) is the mask that accept the least deformations. `HEURISTIC2` (`BH2FH2WH2OH2MH2`) is the mask that accepts the most deformations, thus, it is the less reliable mask. In the middle, we have the intermediary `HEURISTIC1` (`BH1FH1WH1OH-1MH1`) mask, which is also classified as an advanced mask. In addition to these three major masks, we may obtain other masks empirically.

5.4 The search for a match in the database

For a signature to be matched, each constitutive signature element 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 these element lists (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 *I*(*P2*) and *I*(*P3*), respectively.

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 searching the intersection of *I*(*P1*) et *I*(*P2*). If there is still no match, we apply the same search algorithm but with the next deformation mask (from the least deforming to the most deforming). The search is stopped as soon as one or more match are found for a given mask by trying all stored signatures.

The algorithm is given in mathematical notation below:

$$I(P1) = E1(P1) \bigcap E2(P1) \bigcap \cdots \bigcap E5(P1)$$
$$I(P2) = E1(P2) \bigcap E2(P2) \bigcap \cdots \bigcap E5(P2)$$
$$I(P3) = E1(P3) \bigcap E2(P3) \bigcap \cdots \bigcap E5(P3)$$
$$I = I(P1) \bigcap I(P2) \bigcap I(P3)$$

If I is null:

$$I = I(P1) \bigcap I(P2)$$

In passive mode, the algorithm is written:

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

The matching algorithm is the same for *IPv4* and *IPv6*. We have previously noted 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 reason is that the *TCP* stack remained nearly unchanged from *IPv4* to *IPv6*.

5.5 Deformation masks advanced usage

We have seen that deformation masks are generated empirically.[13] For example, the mask `BH0FH0WH2OH0MH0` was[14] necessary to correctly identify the http://www.openbsd.org operating system (Fig. 5).

Operating systems that run http://www.openbsd.org had responses very similar to the ones 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 ignored values for the response' *TCP* window size (mask `WH2`).

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

[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] Emphasis on 'was', because it is no more useful today: The targeted server seems to have changed its network architecture.

Fig. 5 Fingerprinting http://www.openbsd.org server

SinFP launched in active mode with a custom deformation mask able to identify the `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 O0101080affffffff44454144010303000204ffff 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 O0101080affffffff44454144010303000204ffff M1460
B01120 F0x04 W0 O0 M0
```

Fig. 6 Fingerprinting example using active/passive mode

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: BH1FH0WH0OH0MH0/P2: GNU/Linux: Linux: 2.6.x
```

Thus, we found a match with a low heuristic, and the target system was correctly identified as *SunOS 5.6* (see Fig. 5). The http://www.openbsd.org server was not the only one requiring such a specific deformation mask, thus we added the mask to *SinFP*'s code. We assumed that an intermediary filtering and/or routing device modifies the *TCP* window sizes, or that the *TCP/IP* stack had been customized.

As for signatures, we stress the importance of choosing 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 whether or not to add a new deformation mask is a manual process which requires strong tool expertise.

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 still be possible, however, because when *SinFP* is used in default mode, it sends two *TCP SYN* and one *TCP SYN+ACK* packets in a short timeframe. These events may be modeled by an *IDS* rule.

Optional modes 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 remain identifiable by an *IDS*, but may introduce some false positives. Launching only request two is far harder to detect for an *IDS* without generating many false positives: This is 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's *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 Fig. 6). In this example, the result is very reliable because only a minor deformation exists on the *B* element, a deformation in heuristic 1 (`BH1`).

7 Conclusion

In this article, we have described the design philosophy and in-depth implementation of *SinFP*. We have shown how to unify active and passive fingerprinting with a unique category of signatures taken in an active way. *SinFP* is also the first public tool to implement fingerprinting over *IPv6*, both active and passive.

The matching algorithm—similar to those used in Web search engines—gives excellent results, especially when enhanced by deformation masks. Nevertheless, some misidentifications will occur; but we have some palliative in mind.

These solutions have not been implemented yet in the tool, but may be the subject of another publication.

In the meantime, if you wish to compare *SinFP*'s active fingerprinting against *nmap* active fingerprinting, you may consult the following sites [17–20]. Finally, some useful *SinFP*'s tips and tricks for your daily usage may be found at [21,22].

References

1. `Net::SinFP 0.92`. http://search.cpan.org/~gomor/Net-SinFP-0.92/
2. Stateful Passive Fingerprinting for Malicious Packet Identification. http://www.andrew.cmu.edu/user/xsk/XenoKovahThesis.pdf
3. IPv6 Neighbor Discovery Protocol based OS Fingerprinting. http://hal.inria.fr/docs/00/16/99/90/PDF/technical_report_fingerprinting.pdf
4. A Hybrid Approach to Operating System Discovery using Answer Set Programming. http://ieeexplore.ieee.org/iel5/4258513/4258514/04258556.pdf?tp=&isnumber=&arnumber=4258556
5. Toward Undetected Operating System Fingerprinting. http://www.usenix.org/events/woot07/tech/full_papers/greenwald/greenwald.pdf
6. Prise d'empreinte active des systèmes d'exploitation. http://www.gomor.org/bin/view/GomorOrg/Misc7
7. Internet Protocol (version 6). ftp://ftp.rfc-editor.org/in-notes/rfc2460.txt
8. Internet Protocol (version 4). ftp://ftp.rfc-editor.org/in-notes/rfc791.txt
9. SQLite Home Page. http://www.sqlite.org/
10. Transmission Control Protocol. ftp://ftp.rfc-editor.org/in-notes/rfc793.txt
11. Remote OS Detection using TCP/IP Fingerprinting (2nd Generation). http://insecure.org/nmap/osdetect/
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. Tips and Tricks. http://www.gomor.org/bin/view/Sinfp/DocTipsAndTricks
22. SinFP OS fingerprinting tool. http://www.gomor.org/bin/view/Sinfp/WebHome