J Comput Virol (2010) 6:329–342
DOI 10.1007/s11416-009-0134-4

ORIGINAL PAPER

IpMorph: fingerprinting spoofing unification

Guillaume Prigent · Florian Vichot · Fabrice Harrouet

Received: 5 January 2009 / Accepted: 20 August 2009 / Published online: 3 September 2009

Abstract There is nowadays a wide range of TCP/IP stack identification tools that allow to easily recognize the operating system of foreseen targets. The object of this article is to show that fingerprint concealment and spoofing are uniformly possible against different known fingerprinting tools. We present IpMorph, counter-recognition software implemented as a user-mode TCP/IP stack, ensuring session monitoring and on the fly packets re-writing. We detail its operation and use against tools like Nmap, Xprobe2, Ring2, SinFP and p0f, and we evaluate its efficiency thanks to a first technical implementation that already covers most of our objectives.

1 Introduction

On the attacker's side, information gathering is the preparation phase that determines the creation of an appropriate offensive strategy. At this stage, the point is to have as much reliable and proven information as possible at hand, to give the right direction to the penetration process and to maximize the relevance of all undertaken actions. On the contrary, on the defender's side, the possibility to minimize the visibility of its infrastructure's critical perimeter is a limiting, or at least slowing factor for any potential attack lead against him. To face the increasing number of recognition actions (port scans explosion, massive and automated use of vulnerability analysis tools or fingerprinting agents, target adapted malware), it is now critical to be able to reduce the visible informational perimeter of the infrastructure to protect. At the network level, even if it is almost impossible to hide basic services (IP addresses, TCP ports) supporting the infrastructure's interconnection, it is nevertheless possible to imagine a realistic enough falsification to fool distant recognition actions.

Our understanding is the following:

> "If a computer is able to hide its identity from the recognition tools and to impersonate the identity of a less interesting system, it reduces its appeal to the attacker and disturbs the relevance of attacks targeted to its apparent nature."

The OSFP[1] domain has already been widely discussed in numerous articles [1] in the last 10 years, and nowadays different techniques exist to identify the behavior of operating systems' TCP/IP stacks. The subject of this article is therefore not to once again detail these methodologies from the attacker's (the person trying to identify the personality of the target) point of view. Our objective, in an educational way and from the defender's point of view, is to show that it is possible to create a software component with a role to prevent and to really disturb this remote identification, whether it is

The IpMorph software is distributed under the GPLv3 license.
This independent project is based on our previous works, and mainly derives from a specific need in the "Hynesim" network architecture simulation project (DGA-CELAR/SSI-AMI government contract, http://www.hynesim.org).

G. Prigent (✉) · F. Vichot
Diateam: Architectes de l'information, 41, rue Yves Collet, 29200 Brest, France
e-mail: guillaume.prigent@diateam.net

F. Vichot
e-mail: florian.vichot@diateam.net

F. Harrouet
Laboratoire d'Informatique des Systèmes Complexes (LISyC), Technopôle Brest Iroise, 29280 Plouzané, France
e-mail: harrouet@enib.fr

[1] OSFP: Operating System FingerPrinting.

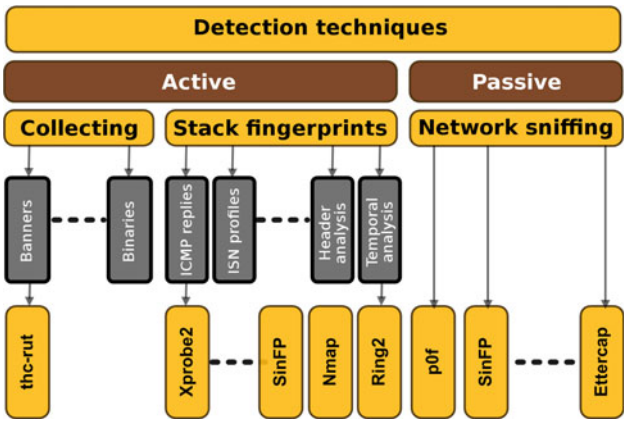

Fig. 1 Classification of a few techniques, illustrated by some associated tools

active or passive. This article is hence intended to be both a pragmatic reflection about what has to be done to fool most currently known tools, and a presentation of the architecture used for the IpMorph tool that we're developing to fool these identifications.

Since we are willing to embrace the wide range of tools implementing different OSFP techniques (Fig. 1), we start the next paragraph by quickly explaining its two main methods, to introduce the tools we want to fool.

2 Reminder

There are two methods for remote fingerprinting an operating system, and each one has advantages and drawbacks.

2.1 Active fingerprinting

This method is considered active because it consists in contacting the target by sending particular stimuli (specially crafted IP and TCP packets) and analyzing its answers in comparison with a previously built knowledge base (signature base). The choice of stimuli, and hence of packets sent from the source, is made following the axiom that several TCP/IP stacks implementations (and therefore operating systems) can answer differently depending on the implementation choices (or even design choices) [2–5].

Major drawbacks By definition, active fingerprinting leaves lots of traces on the studied system, as well as all along the path used (filtering equipment, intrusion detection,…), which makes its potential detection and analysis easier. The modus operandi used by this type of tool is so recognizable that most network intrusion detection system (NIDS) probes contain entries about them. We are often going to use this typical "noise" and this particular traffic to identify the stimuli specific to this type of operation. Moreover, the ability to send these packets to the target for this kind of "intrusive" test requires nominal conditions, whether they are remotely made or on a local network (packets filtering for example). This important limitation depends on the considered tools.

Major advantages Active fingerprinting allows probing of the target as wanted, and therefore testing a wide range of answers. Thanks to this, adaptative tests can be performed (depending on previous answers) to precisely identify a target. When correctly performed, these types of test permit to quickly obtain conclusive results (sometimes, depending on the tools, a few well-chosen packets can be enough).

Tools Nmap, Xprobe2, Ring, SinFP, GFI's Languard Network Security Scanner, Queso,….

2.2 Passive fingerprinting

Passive fingerprinting only performs an analysis of flows coming out of the target, without calling it directly by sending it packets. Most of time, this analysis deals with "legitimate" traffic (which does not deviate from normal use) between the target and a particular host of the local or remote network. The way this method operates offers the attacker the advantage of recognition stealth. Some network protection equipments (like firewalls or NIDS) already use this passive method to correlate potential attacks and operating systems of the computers to protect [3,4,6].

Major drawbacks Passive recognition of a legitimate flow implies to have enough packets to perform identification analysis. This collecting can be long and tedious, and there is no guarantee that the targets at stake in the recorded communications will not change. This temporal aspect is all the more disturbing as it is often hard to access the flows between targeted computers (excluding the trivial case where you are on the filtering element, on a inline probe or directly on a netlink). The techniques that are to be used to capture analysis samples are identical to the ones used for "Man in the middle" attacks.

Major advantages Passive fingerprinting has a read-only access to the flow. Detection of this type of behavior is exactly the same as in a traditional "Sniffing" type capture in "promiscuous" mode on a network. Moreover, the target is never directly contacted. Hence, passive fingerprinting is highly furtive.

Tools p0f, SinFP, Ettercap, Satori,….

	Active fingerprinting	Passive fingerprinting
Advantages	Fast Accurate identification	Stealth
Drawbacks	Often "noisy" Bulky in packets Sensibility to testing conditions	Slow Not very discriminatory

Fig. 2 Advantages and drawbacks of the two methodologies

2.3 Fingerprinting methodologies synthesis

Advantages and drawbacks of the two methodologies are to be seen as a generalization. Some tools may be more or less concerned.

This table (Fig. 2) is a synthesis and a classification based the current state of the art. Some tools are especially "noisy" (Nmap with the sending of a great number of voluntarily distorted packets), while other tools are less noticeable and only imply a few standard packets (e.g. SinFP).

Concerning IpMorph, we wish to be able to ensure defeating as many tools as possible, which is the reason why we have integrated both active fingerprinting tools (Nmap, Xprobe2, SinFP, Ring2) and passive fingerprinting tools (p0f, SinFP) to IpMorph. Each one of these tools is further described in Sect. 5, with a technical focus on the specificity of their respective integration to the design of our tool.

3 Fingerprinting defeating state of the art

There is already a wide range of solutions [7] trying, each one in a different way, to disturb the exact identification of the computer to protect. The following section lists most existing approaches, and classifies them according to their predefined way of operating.

3.1 Classification of identification disturbance solutions

We consider that there are three main approach categories using concealment techniques. The ones working on filtering and rewriting of packets, the ones aiming at modifying the native configuration of the TCP/IP stack of the computer to protect, and the ones which substitute a new stack to the existing one.

Filtering

- *Stealth patch* [8]: simple TCP packets identification solution as a GNU/Linux 2.2–2.4 core module patch, allowing to filter and to ignore SYN+FIN packets (QueSO probe), Nmap T2 probe (bit *"reserved"*) and FIN+PUSH+URG packets (Nmap T7 probe).
- *Blackhole* [9]: TCP and UDP packets filtering options, allowing to respectively block RST and ICMP answers on closed ports.
- *IPlog* [10]: *"userland"* application, detecting and answering instead of the native stack to most Nmap probes. Answers are statically created in the code (*iplog_tcp.c*) and allow to prevent Nmap of correctly identifying the target.
- *OpenBSD packet filter* [11]: official OpenBSD firewall, allowing to specify (pf.conf file) most packets options when they are emitted (like default TTL, maximum MSS, IP ID choice policy…). The tools based on matching these fields with their signatures do not find anymore the real signature of the equipment protected by this solution.

TCP/IP stack configuration and modification (*"host based"*)

- *Ip personality*: Netfilter module for Linux 2.4, allowing to configure some parameters of the TCP/IP stack.
- *Fingerprint fucker*: Core module for Linux 2.2, able to read previous Nmap base's signatures and to configure the stack parameters accordingly.
- *FreeBSD fingerprint scrubber* [1]: Developed for FreeBSD, it does not try to emulate the behavior of a particular system, but aims only at not looking like any other system.
- *OSfuscate* [12]: Windows software that modifies keys in the register, to change some TCP/IP parameters.

TCP/IP stack substitution (*"proxy behavior"*)

- *Honeyd* [13]: Probably the most wellknown honeypot software, Honeyd is able to simulate Xprobe2 and Nmap (previous version) signatures for its virtual hosts.
- *Packet purgatory/Morph* [14]: Morph is a process that is able to emulate three personalities by modifying outcoming packets. To do so, it wedges itself between the network interface and the core using PacketPurgatory, a libPcap-based library.

3.2 State of the art synthesis

The most part of presented tools is already old or no longer maintained. Most of them only ensure a partial disturbance of fingerprinting tools. For us, these are not "really" complete identity customization tools, we consider them more as unitary ad hoc disturbance techniques.

The usage of these tools is often difficult and implies to own either the considered filtering equipment or a specific system configuration with regards to the core or firewall of the target to protect. This is exactly what IpMorph refuses to do. We want to offer an entirely user-space application

solution, able to deploy on a wide range of equipments and allowing a complete spoofing depending on the identity chosen by the user.

Beside these general considerations, the solution that at first glance seems to be the one closest to our interests is the Morph tool, based on the "userland" Packet Purgatory [14] library. Though they have similar names, Morph and IpMorph projects strongly differ in their objectives and implementation.

Morph is mainly a "proof of concept" tool, which only partially manages a few fingerprinting tools and does not have any personality or setup mechanism, except three different behaviors directly coded in the sources (OpenBSD 3.3, Linux 2.4, Windows 2000). The packets modification mechanism is not very extensible (if-based implementation, switch/case in C language) and the field values are static. For now, it seems impossible to us to transform it into a generic personalities "engine".

Moreover, the lack of Morph update since 2005 and the compilation problems encountered in executable file creation let us think that the project has been abandoned. The only really functional mode of Morph is the "proxy" mode, but we notice several performance and relevance problems (a Nmap scan forces Morph to use 100% of a CPU and takes up to several minutes, SSH connections take five to ten times longer to establish), as well as a few limitations when the management of a large number of connections is needed (if we launch a Xprobe2 scan with a simultaneous Nmap detection, Xprobe2 returns that the host is not online).

Finally, managed tools versions are outdated: Xprobe2 recognizes a FreeBSD 1.5.1 when the personality coded in the resources is "openbsd", and the latest version of Nmap never recognizes anything. SinFP or RINGv2-type tools are not supported.

4 Design objectives and principles

4.1 Project's context

Our first works in this domain have started with the BridNet [15] project, where our objective was to completely simulate the network behavior of a connected computer. We have then realized our first user-mode TCP/IP stack. This valuable experience has shown us that it was possible to create a complete minimal stack in user space, provided we knew how to correctly handle IP fragmentation and IP session monitoring.

More recently, as part of the Hynesim [16] project, we wanted to be able to "customize" the network behavior of some virtual machines, especially the ones based on OpenVZ. To separate problems and technical components, we have decided to re-use our TCP/IP stack and to adapt it to fingerprinting spoofing. The idea is simple: "how can we make a virtual machine appear like any other operating system for detection tools?". IpMorph project was born.

The following section presents the design and creation goals we have set, followed by the general architecture of the tool we are developing.

4.2 Objectives

IpMorph project has been launched to answer a practical need, which is both conceptual and technical. For this reason, it is meant to be used in several current or upcoming projects. For us, it is not a simple idea, or an elegant theoretical concept, it is much closer to designing and creating an operational, documented, rugged and long-time maintained solution. Beyond the fact that this project is a great occasion to learn more about fingerprinting, it offers us the possibility to re-use several software components that have been validated in previous projects, as well as to perform an innovative work to synthesize different tools signatures.

Our main goal is to design a user-space application that protects a computer by preventing its TCP/IP fingerprint identification, and also allows to customize its appearance to make every testing tool consider it like the one configured in IpMorph, both for active and passive fingerprinting techniques. This is the reason why we talk about unification, for we are able to spoof several different tools with a single solution, IpMorph.

The imperative we have to keep in mind through the whole design stage is that the protected computer must stay fully operational at network level. It is critical that services hosted by the computer under protection continue to normally operate. IpMorph must never introduce an applicative network failure. This challenge implies a special care for the techniques used to manipulate transiting packets, especially during their re-writing. We want to design and create an application allowing a wide range of implementations, to protect either actual computers of a network (Fig. 3) or virtual machines hosted by a host integrating IpMorph (Fig. 4). When implementing this tool, we want to create the core of the application in C++ to associate performance and

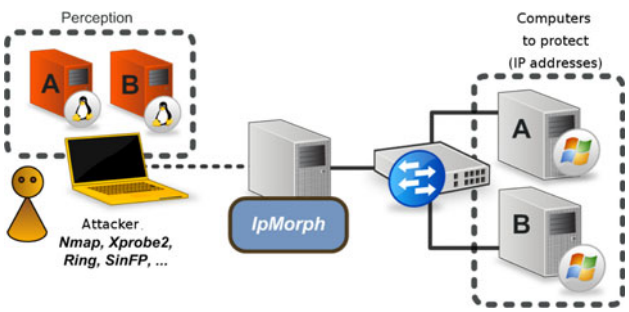

Fig. 3 Use of IpMorph inline to hide "real" hosts

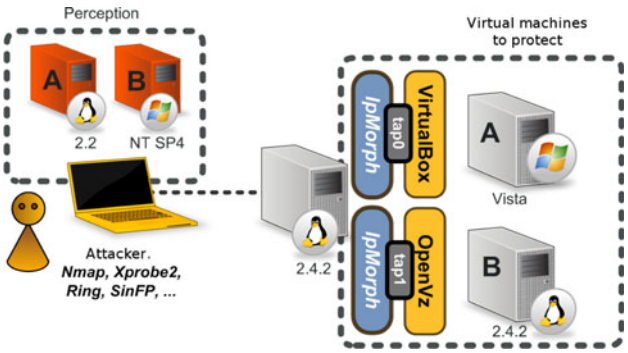

Fig. 4 Use of IpMorph on a host to hide virtual machines hosted on it

object-oriented programming in a portable way and with minimum dependencies on external libraries, thus ensuring its compatibility and "source" portability on a wide range of platforms (for now mainly BSD, Linux, Solaris and MacOS).

4.3 General architecture

The core principle of our solution is the setup of a TCP/IP stack in user space, which has two sides, among which, one is exposed to fingerprinting tools ("exposed side"), and one is exposed to the computer you want to protect ("protected side"). Each side is directly linked with a network interface of the host system, below the firewall-like mechanisms, whether it is real (eth0,…) or virtual (tap0,…) and is able to read and write the frames it is responsible of (frames to or from the fingerprinting tools for the exposed side and frames to or from the protected computer for the protected side). This reading on each interface exactly corresponds to a "sniff" in "promiscuous" mode, like in numerous network tools. The main work of the application's core is to ensure clever switching between these two sides, depending on the state of flows and actions that are to be implemented.

We want to be put in protection at link layer, which is why we first perform a modification of the Ethernet address at time of ARP and RARP requests (Fig. 5). This way, the exposed side reveals a MAC address which is not really the one of the protected computer. Even if for now it is not managed, it would be useful to ensure coherence between the manufacturer designated by the exposed MAC address and the chosen personality for IpMorph (e.g. it seems quite unlikely to be in front of a Windows Vista computer if the MAC address has the prefix of a VoIP phone manufacturer).

IpMorph only takes care of legitimate traffic to or from the computer to protect, and filters transiting flows at Ethernet and IP level as well. In other words, IpMorph only intervenes in exchanges associated with the protected computer.

Once the work is reassigned to the IP and higher levels, IpMorph supports, for each side, IP reassembly and fragmentation, as well as TCP session monitoring, like most modern "stateful" firewalls do. An IP data packet, whether read or written on one of the sides, compulsory passes through reassembling module (for reading) and fragmentation module (when writing). This way of operating is exactly identical to the one of a router placed between two netlinks with different MTUs. Then, at network layer level, each IP datagram can be directly analyzed by IpMorph (we are sure that this is a complete upper layer packet, e.g. ICMP, TCP or UDP). Once the packet (and hence headers) are complete, we go up the layers until taking care of TCP to ensure session monitoring and relay flows to the opposite side (Fig. 6).

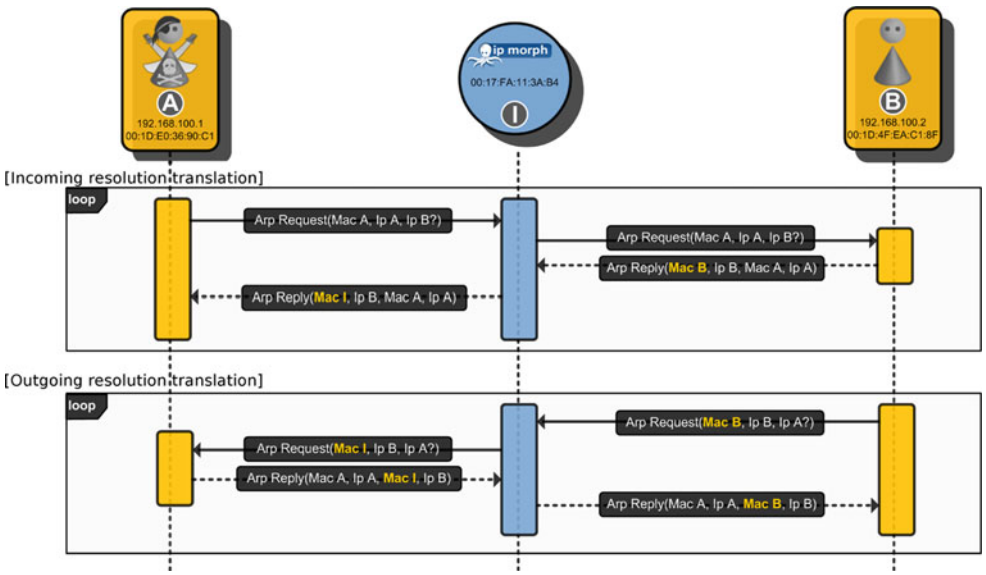

Fig. 5 Bidirectional translation of ARP resolutions

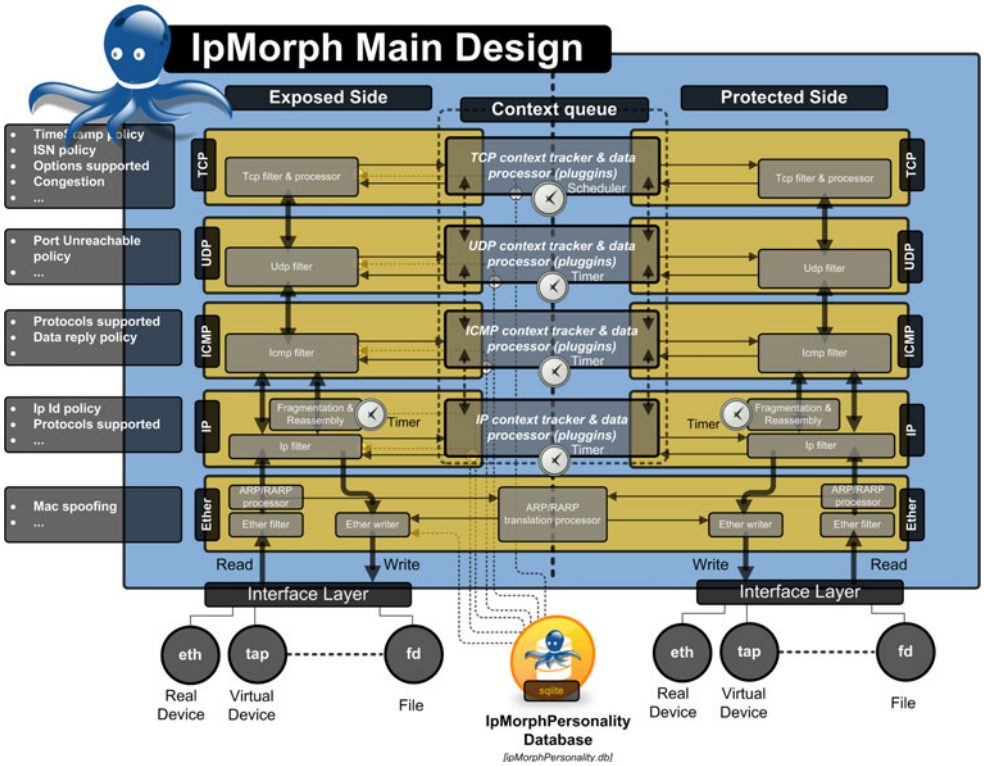

Fig. 6 IpMorph detailed functional architecture

For each flow, an IpMorph context is created and stored to keep in memory sessions, states and possibly data previously exchanged.

With these contexts at hand, we perform data handling and make filtering choices on each introspection stage. IpMorph operates most of time as a relay and a packet normalization tool. In some cases, especially during active fingerprinting tools probes detection, IpMorph does not relay these frames but authoritatively (which means that IpMorph decides to answer on behalf of the protected computer) creates each answer awaited by these tools. Put simply, we can consider that our solution works in "interception" mode for active tools and in "relay" mode for passive tools (this relay is of course not a direct copy, because we modify packets to give them the expected characteristics).

The main challenge when you follow this way of operating is to be able to keep a good coherence from end to end (e.g. for TCP) between the exposed side's flows and the associated protected side's flows. Production network traffic must seamlessly go on and each connection state must still correspond on both sides. In an analogical way, it can be seen as a kind of SNAT/DNAT extended to all TCP and IP fields in addition to the source and destination ports. Most of time, we perform translations on packet fields by keeping them in memory in contexts to apply them reversely on return.

5 Implementation and technical focus

IpMorph is made of a core (two-sided virtual TCP/IP stack) and of a set of external operating tools (ipmController, ipmPersonalityDBManager, ipmGui).

Contrary to many open source tools which read and send network packets (e.g. Morph), IpMorph does not use the traditional libpcap/libdnet duo. Even if 1 day we might use dedicated libraries to ensure portability, we have for now chosen not to use them to keep a close control on the code and to unify our design in manipulated objects (for reading, handling or sending as well). We only rely on standard system calls for the underlayers of our solution. No dependency (except the C++ Qt4 library that we use as development facilitator and host structure of our C++ classes) is required in the project (Fig. 7).

5.1 Nmap integration

Nmap is probably the most used active fingerprinting tool, and also the most complete in terms of signatures. That is why, even if we do not consider it the most relevant tool, it is critical to be able to fool it using IpMorph.

For any fingerprinting attempt, Nmap starts by performing a scan of open and closed ports on the target. During this

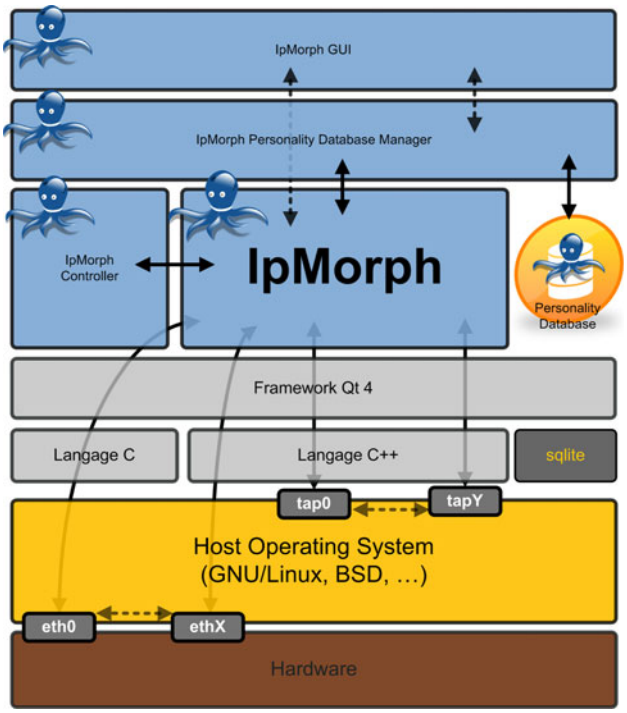

Fig. 7 Software dependencies of IpMorph components

preliminary scanning, IpMorph has no reason to interfere in the exchanges with the protected computer, and hence operates in relay mode. Once this scan is complete, Nmap sends a series of 16 packets that build its discriminative tests to try to identify the remote system. The 16 probes are composed by the sequential sending (average interval: 110 ms) of a series of 6 TCP packets on a port that has previously been detected as open, of two *echo request* ICMP packets, of a explicit congestion notification (ECN) test TCP packet, of a series of 6 TCP packets to open and/or closed ports and of a UDP packet to a closed port. At this stage, IpMorph operates in interception mode and directly answers (or does not answer) on its own initiative without ever relaying packets to the protected side. Based on the desired personality, IpMorph tries to answer precisely what Nmap awaits. Thus, IpMorph's job is to interpret the desired Nmap signature and to react accordingly for the probes.

This interception operating mode is possible because each one of the 16 packets is easily identifiable and widely documented (if a doubt still exists, Nmap source code analysis clearly unveils what it does). Most answers (or the need not to answer) are easily manageable, even if this work has been arduous due to heterogeneous signature elements, and therefore possible answers. However, among tests performed by Nmap, some are harder to fool than others, and we have decided in order to stay concise, to only present below particular points that we consider as non-trivial and that necessitated the most work.

"*Sequence generation* (*SEQ, OPS, WIN and T1*)" *tests* This test deals with the analysis of TCP (ISN) initial sequences numbers chosen by the target stack. This signature element is very discriminative for Nmap for remote system identification. The test consists in emitting a sequence of six TCP packets on an open port. Then, Nmap analyzes and calculates three values from the initial sequence numbers gathered (in SYN+ACK answers to its SYN packets). These three values are the three signature attributes, GCD, ISR and SP. As soon as a new session is detected on the exposed side (receipt of a SYN packet), a TCP context is created, and a new session is initiated on the protected side (sending of a SYN packet). When the SYN+ACK packet returns on the protected computer, an initial sequence number is chosen by IpMorph to replace the one used by the protected side in the exposed side session. This difference is stored in the previous context to apply it on return, and so ensure protected side connection maintenance. The problematic we address here is to correctly create our ISNs, so that the six consecutive ISNs received by Nmap will allow it to calculate the same values for GCD, ISR and SP as in the signature. In other words, the goal is to determine a way to create ISNs only based on three parameters stored in the Nmap signature.

These three parameters are the following:

- *TCP ISN greatest common divisor (GCD)*: the greatest common divisor of the five ISN deltas. Here we call "delta" the differences between two consecutive ISN received by Nmap.
- *TCP ISN counter rate (ISR)*: the ISN average increment per second rate, which corresponds to the speed the ISNs increment at.
- *TCP ISN sequence predictability index (SP)*: dissemination in the creation of ISNs compared to the average, more precisely the standard deviation of the five increment rates with another, compared to the average (ISR above).

While it is easy to create values that respect a given mean and a known standard deviation, it is much harder to make their standard deviations have the same greatest common divisor. This GCD-ruled creation task is not trivial (not to mention the fact that, as ISNs are stored on 32 bits, they can loop) and represented a technical lock for several weeks on IpMorph. Moved by the feeling that it was surely possible (at least by using method from operations research) to find an acceptable solution, we tried several approaches.

Here is the technique we now use to resolve this problematic:

1. We systematically memorize the 6 last created ISNs and their creation date

2. Every time a new ISN is created, we determine, considering what Nmap awaits and the delay we might have, the best ISN rate we will try to keep up with (in other words: what is the ideal ISN we should return to respect the ISR Nmap is going to calculate on our series of answers?)
3. Using the Box–Muller formula, we generate a series a values that respects (for each value) the standard deviation awaited by Nmap. Each one of these values is rounded off to the closest GCD multiple.
4. On the series that has been pre-calculated on previous step, and for each one of the potential values, we calculate, like Nmap would do, the resulting ISR and SP we would have if we kept this value, and then we apply a cost function that is proportional to the squared value of the distance to the signature's ideal (to penalize ISN values that deviate significantly from the ideal value)
5. Finally, we keep the ISN value that minimizes the distance, and we store it with the current creation date in microseconds. This value is the one used in the SYN+ ACK to Nmap.

"*TCP RST data checksum (RD)*" *test* The RD test has been an interesting problem for us, and we will present it below.

Some stacks add an ASCII message in the data of a TCP RST packet (generally the error message). This case is relatively rare and few Nmap signatures have this RD field, but to ensure the completeness of our approach, we had to perform this test to completely fool Nmap.

To avoid storing the message found in the TCP RST packet, Nmap stores a CRC-32 of the data in its signatures. Unfortunately, this means that IpMorph no longer has access to the original message, which has to be sent again when a TCP RST packet is emitted. We have to create data corresponding to the same CRC-32, which will be calculated on return. Fortunately, the CRC-32 algorithm is "cryptographically" weak, and has the property that each 4 bytes block added to a message can turn the CRC-32 into any desired CRC-32. This directly implies that all CRC-32 inherently have a 4 bytes corresponding message.

Based on this statement, the first approach has naturally been to try brute force, for the range of values to be tested is very narrow (2^{32} combinations) in comparison with the current processors capacities. Beyond the fact that an average of thirty seconds is needed to find a collision (which is too long, even if we memorize the result in the IpMorph personality), this approach is, scientifically speaking, not really elegant.

More subtly, we tried to create CRC-32 collisions in O(1), which means reversing the CRC-32 algorithm.

The calculation of a CRC-32 is like the division of the message by a polynomial, using "modulo 2 polynomial arithmetic", which actually corresponds to binary arithmetic without carry digit (i.e. $1 + 1 = 0$, without carrying the 1 digit on the next bit). In this type of arithmetic, addition and subtraction are identical and correspond to a XOR. A more efficient version of the algorithm (Table driven CRC) uses a pre-calculated table to make this division by 8-bits blocks (see [17]) instead of bit by bit. The CRC is so calculated by shifting the message byte by byte through a "register" the size of the CRC (in our case 32 bits). The outcoming value resulting from the shifting is memorized (it is the *topbyte*). The register is combined by a XOR having the value of the pre-calculated value extracted from the table (on the position equal to the *topbyte*). These manipulations are made once for each byte of the message, and finally the register contains our CRC-32. Reversing this algorithm corresponds to reversing this calculation until the 4 bytes message producing the CRC-32 extracted from the Nmap signature is found.

Our work has been directly influenced by existing works [18].

The principle of this algorithm is the following:

1. First take an 8 bytes table, and fill 0–3 positions with the CRC-32 of the message until now. Fill 4–7 positions with CRC-32 desired value.
2. Take the value from the 7 position and use it to find the complete value back in the table.
3. Make a XOR with this value (4 bytes) with bytes 4–7.
4. Make a XOR of the position in the table with byte 3.
5. Repeat steps 2–4 three times, by decrementing index number by 1 every time (value = 6, 5, 4 for step 2; 3–6, 2–5, 1–4 for step 3 and 3, 2, 1, 0 for step 4)

Finally, data generating the desired CRC-32 are in the positions 0–3 of the table. You can so systematically find 4 bytes which CRC-32 is the one of Nmap signatures, in O(1).

This reverse CRC-32, i.e. the creation of 4 bytes corresponding to the RD of the Nmap signature, is made upon creation of the personality. Fortunately, Nmap does not use any hashing function like MD5 or SHA1, which would have forced us to have access to real operating systems to analyze their specific TCP RST packets, capture their real text field and store it in a base in the IpMorph personality.

"*TCP IP ID sequence generation algorithm (TI) and ICMP IP ID sequence generation algorithm (II)*" *test* During these tests, Nmap tries to determine the IP packets ID creation algorithm, both if it is a TCP packet or an ICMP packet. As for other signature parameters, IpMorph must respect this IP packet ID creation policy. We must especially respect the fact that this ID creation can be either shared between TCP and ICMP or separate (*Shared IP ID sequence Boolean* (SS) parameter). To do so, IpMorph keeps two attributes up-to-date: `_lastIpIdTcp` and `_lastIpIdIcmp`,

which it increments independently or simultaneously upon every new ID creation, depending on the incrementing policy desired by Nmap. The incrementing policy is also tested (TI and II tests), for example systematically equal to zero (TI=Z), always equal to a specific value stored in hexadecimal in the signature (TI=A400), incremented every time (TI=I),…

Taking into account this test has not been difficult, because the call of our `_generateIpIdent()` method only has to scrupulously respect the range of potential cases and the potential sharing of this creation between TCP and ICMP.

The implementation of this IP ID creation spoofing has allowed us to unveil a malfunction of Nmap, due to a "bug" in its code, which was already present in previous versions. During our tests (which mainly consist in launching IpMorph with a random Nmap signature and verifying for each Nmap execution if the detected signature is correct, all this looped on all Nmap signatures), we have noticed that for certain signatures, Nmap did not manage to identify the desired fingerprint. Considering our classification of problematic signatures and Nmap source code analysis (`osscan2.cc` file), we discovered that the Nmap analysis function of the returned IP ID series did not match with its objectives. More precisely, probably because of a copy/paste error in its code, the table containing IDs measured by Nmap was modified before its analysis function. This noticed "bug" has a significant importance for us, because it means that, in many of the last versions of Nmap, several signatures of the Nmap base will never be detected by the tool, even if the remote OS is the one of the signature present in the base (in other words, the base signature is correct, but because of its wrong analysis Nmap will never detect it). We have corrected this malfunction, and we have sent a patch to the developers.

This Nmap code analysis on the last version (current version: 4.85 BETA 4) has allowed us to understand that undocumented tests are already being prepared. As an example, Nmap will soon perform its IP ID creation test simultaneously on open TCP ports, closed TCP ports and ICMP. A new CI field (corresponding to the detected creation policy when the TCP port is closed) should appear in upcoming signature bases, which will add a third parameter for this IP ID test. For us, it is all about adding a specific `_lastIpIdTcpClosed` attribute to differentiate this creation from IpMorph `_generateIpIdent()` method if needed.

5.2 SinFP integration

SinFP [4], developed in Brittany by Patrice Auffret, is the first fingerprinting tool that unifies active fingerprinting with passive fingerprinting and performs stack recognition on IPv6. It is, both in its design and operation (sending of at least three packets in active mode), an interesting tool, which differs significantly from Nmap in terms of stimuli volume and probe characterization (sending of conform packets). Given its characteristics, and mainly its passive analysis mode, it was critical for us to be able to integrate, and thus fool it. Contrary to the easily detectable Nmap probes (where IpMorph operates in interception), in this case our tool operates as a relay between the exposed side and the protected side. Our approach was the following: we first tried to fool SinFP during active fingerprint detection, and then we tried to fool it during passive fingerprinting.

SinFP spoofing in active mode We extract the desired signature from the SinFP signature base (in SQLite format). We only interpret its first heuristic and we do not consider its alteration masks, because we intend to return exactly what SinFP expects to ensure perfect spoofing starting on the first analysis of results by SinFP. The three SinFP probes are standard TCP packets on a port specified as open (SYN, SYN with options and SYN+ACK) that induces respective answers from the protected computer (SYN+ACK, SYN+ACK and RST). As there is no way to differentiate such a SinFP probe from a legitimate connection, IpMorph systematically operates as a relay on return of protected side packets while modifying these upon sending to the exposed side (to SinFP). This relay/re-writing on return only implies SYN+ACK and RST concerned packets, and consists in modifying the IP and TCP headers fields according to the expected SinFP signature. To concretely illustrate this relay, below is an example of re-writing on the P2 test of SinFP:

- When a SYN packet arrives on the exposed side, a context is created and the packet is transmitted to the protected side.
- Upon receipt of the responding SYN+ACK packet of the protected computer, we consult the IpMorph personality. For this example, we consider that this personality has been built from the signature id 140 of SinFP, i.e. an OpenBSD 3.7.
- Depending on the parameters of this personality, we perform a number of alterations:
- We apply directly the initial TTL, Don't Fragment (DF) bit, maximum segment size (MSS), TCP option and Windows Scale values to the packet, like we read them from the signature 140.
- We then apply the right algorithms to generate SEQ, ACK, and IP ID values (once again, based on the SinFP signature).
- The so modified packet is transmitted to the exposed side.

SinFP spoofing in passive mode Contrary to what we could think at the beginning, the supporting of SinFP spoofing in passive mode is almost instantaneous if you also know how

to modify outgoing SYN packets, i.e. coming from the protected computer. This statement mainly derives from the fact that SinFP uses a unified approach both in active and passive mode (only a few details differ) on acquired packets analysis.

In passive mode, SinFP can detect the communicating computer's identity both on its answers to SYNs (i.e. SYN+ACK packets) of a client and on initial connections of a computer to a server (i.e. SYN packets). For us, the first case is automatically managed because we ensure to send a SYN+ACK answer, like SinFP awaits. It is the same case as active mode spoofing.

For example, in passive mode and for a setup of IpMorph with the 140 signature (OpenBSD 3.7) of the SinFP, here is the detection performed:

```
 $ sudo/usr/local/sinfp/bin/sinfp.pl -P -d eth0
192.168.100.110:80 > 192.168.100.73:47979 [SYN|ACK]
P2: B11111 F0x12 W16384 O0204ffff010104020103030001010 80
affffffffffffffff M1460
IPv4: HEURISTIC0/P2: BSD: OpenBSD: 3.5
IPv4: HEURISTIC0/P2: BSD: OpenBSD: 3.6
IPv4: HEURISTIC0/P2: BSD: OpenBSD: 3.7
IPv4: HEURISTIC0/P2: BSD: OpenBSD: 3.8
IPv4: HEURISTIC0/P2: BSD: OpenBSD: 3.9
IPv4: HEURISTIC0/P2: BSD: OpenBSD: 4.0
```

The 192.168.100.110 computer (which is an Ubuntu 8.04), protected by IpMorph, appears correctly on SYNs/ACKs as an OpenBSD 3.5–4.0-family computer (this uncertainty range is due to the fact that these SinFP signatures are identical for the P2 test and that it is the only identification information the tool has access to in passive mode).

In the second case, and when SinFP only has access to the SYN packets coming from the protected computer, we simply have to add support for rewriting of connection made from the protected computer. In this detection mode, SinFP analyzes the SYN packet by normalizing it its way to make it match with a P2 test analysis (several fields of the received packet and of the in-base signature are deactivated because the P1 or P2 initial comparison packet has not been sent by SinFP). Our task consists then in only modifying the SYN packet on tested fields, i.e. the order and values of TCP options given by the SinFP signature, the MSS and the window size. The initial sequence number is chosen and modified as usual in IpMorph (see Nmap), the acknowledgement number is set to zero and the TTL is the one from IpMorph's personality (coming from other tools signature).

In this very case, SinFP can not legitimately base itself on IP header's fields (ID, TTL, bit DF) or on the sequence and acknowledgement numbers of the packet, for it does not have any corresponding signature in base (the acquired patterns are often differences from a request and only on SYN+ACK answers). Aware of this, upon packet analysis in passive mode, the author does not consider most of fields for its matching in signature base.

Unfortunately for SinFP, some analysis and pattern matching fields are kept, though we think they should not be, because they make identification much less precise at first time (without activating the advanced alteration masks). To be sure of this, see below the detection performed in passive mode for the same IpMorph's setup with the 140 signature (OpenBSD 3.7) when SinFP does not access to the SYN packets:

```
 $ sudo/usr/local/sinfp/bin/sinfp.pl -P -d eth0
192.168.100.110:35366 > 192.168.100.73:80 [SYN]
P2: B11110 F0x12 W16384 O0204ffff010104020103030001010 80
affffffff00000000 M1460
IPv4: unknown
```

If we now activate the advanced alteration masks (which equals to allowing any value used for B11110 analysis and thus ignore these analysis criteria), the new detection is the following:

```
 $ sudo/usr/local/sinfp/bin/sinfp.pl -P -d eth0 -H
192.168.100.110:59893 > 192.168.100.73:80 [SYN]
P2: B11110 F0x12 W16384 O0204ffff010104020103030001010 80
affffffff00000000 M1460
IPv4: BH2FH0WH0OH0MH1/P2: BSD: OpenBSD: 3.5
IPv4: BH2FH0WH0OH0MH1/P2: BSD: OpenBSD: 3.6
IPv4: BH2FH0WH0OH0MH1/P2: BSD: OpenBSD: 3.7
IPv4: BH2FH0WH0OH0MH1/P2: BSD: OpenBSD: 3.8
IPv4: BH2FH0WH0OH0MH1/P2: BSD: OpenBSD: 3.9
IPv4: BH2FH0WH0OH0MH1/P2: BSD: OpenBSD: 4.0
```

Willing to participate in the quality of major tools like Nmap or SinFP, we told the author about our analysis. According to us, there is no need to perform comparison tests when the reference sample is too weak and when no comparison with reference values is possible.

5.3 p0f integration

p0f has been designed and created by Michal Zalewski and is historically the first operational and efficient passive mode fingerprinting tool (other pre-existing tools as Siphon did not make it further than the proof of concept step). As a key passive mode detection and fingerprinting tool, it should integrate with our spoofing tool.

Default p0f performs a SYN packets analysis, i.e. the "outgoing" SYN packets for us (coming from the protected computer). Not only is this for us the same connection diagram (relay and re-writing) as in SinFP passive mode on SYN packets, but, moreover, the re-writing of packets is the same because p0f's signatures, and thus analyzed fields, are almost the same as the ones of SinFP (and vice versa).

No p0f specificity has had to be considered because we have already handled SinFP integration. In other words, the operating process and analysis criteria of SinFP completely "overlap" the ones of p0f.

To be sure of this without having modified IpMorph since complete SinFP integration, we have performed a few tests by configuring IpMorph with signatures coming from SinFP, that correspond to stacks identified in p0f base. Here is an

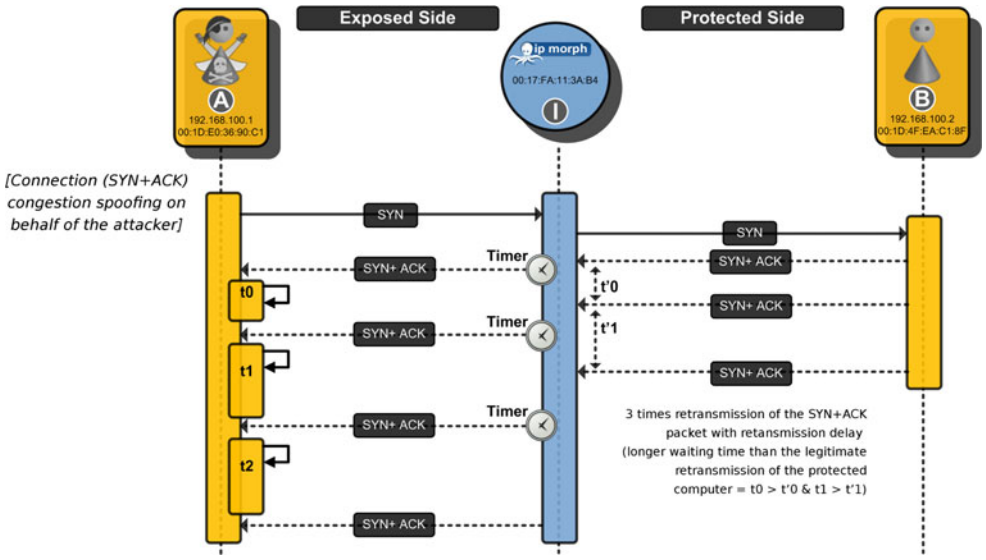

Fig. 8 Implementation of congestion spoofing on a connection on behalf of the "exposed side" computer (i.e. "client")

example for the same personality as above (SinFP signature N° 140: `OpenBSD 3.7`):

```
 $ sudo ./p0f -i eth0
p0f - passive os fingerprinting utility, version 2.0.8
(C) M. Zalewski <lcamtuf@dione.cc>, W. Stearns <wstearns@pobox.com>
p0f: listening (SYN) on 'eth0', 262 sigs (14 generic, cksum 0F1F5CA2), rule: 'all'.
192.168.100.110:35365 - OpenBSD 3.0-3.9 (up: 0 hrs)
-> 192.168.100.73:80 (distance 0, link: ethernet/modem)
```

The IpMorph-protected 192.168.100.110 computer really appears like an OpenBSD 3.0–3.9 operating system.

5.4 Ring2 integration

Ring2 [19], developed in Brittany by Franck VEYSSET, Olivier COURTAY and Olivier HEEN, tries to perform a remote fingerprinting by focusing on temporal specificities of TCP/IP stacks during an induced simulation of TCP congestion. "SYN relay" type solutions only rarely handle transmissions on behalf of protected computers [19] (upon establishment of the session, and thus sending by protected computer of a SYN packet and waiting of the SYN+ACK packet). This characteristic is all the more obvious as it implies a disconnection (session closing by a FIN and waiting for ACK as answer) on behalf of the protected computer. Since these are two of the main tests of RINGv2 (temporal measures on a congestion simulation) and we consider this approach as groundbreaking and elegant, we have decided to integrate these types of relays in IpMorph.

The consideration of these tests forces us to extend our sessions monitoring to the phases when the TCP retransmission is supposed to take place. It can sometimes be compulsory to answer ahead of time, depending on the defined personality, and to emit the SYN or FIN packet on behalf of IpMorph before the protected computer itself. The same phenomenon has to be considered for the opposite case (introduction of a retransmission delay). Moreover, in these cases, you have to block legitimate retransmissions of the protected computer and correctly manage this protected side congestion depending on the exposed side results. Figures 8 and 9 schemas graphically illustrate these use-cases.

At implementation level, we have added timers on the TCP session monitoring IpMorph context. In every concerned case, and both at SYN packet emission level and at FINs level, we systematically launch a time counter, which then launches the re-emission of the last packet stored in the IpMorph context (traffic memory), with defined intervals and with as many occurrences as needed depending in Ring2 signatures. As said above, it is especially important to take care of filtering legitimate re-emission flows of the protected computer as long as the retransmission simulation is not over on the exposed side to Ring2.

5.5 Signature unification

IpMorph's behavior is defined by a set of parameters that concerns some reply options as well as the algorithmic choices to make on certain stimuli. All these parameters are customizable and this is what we define as the IpMorph personality.

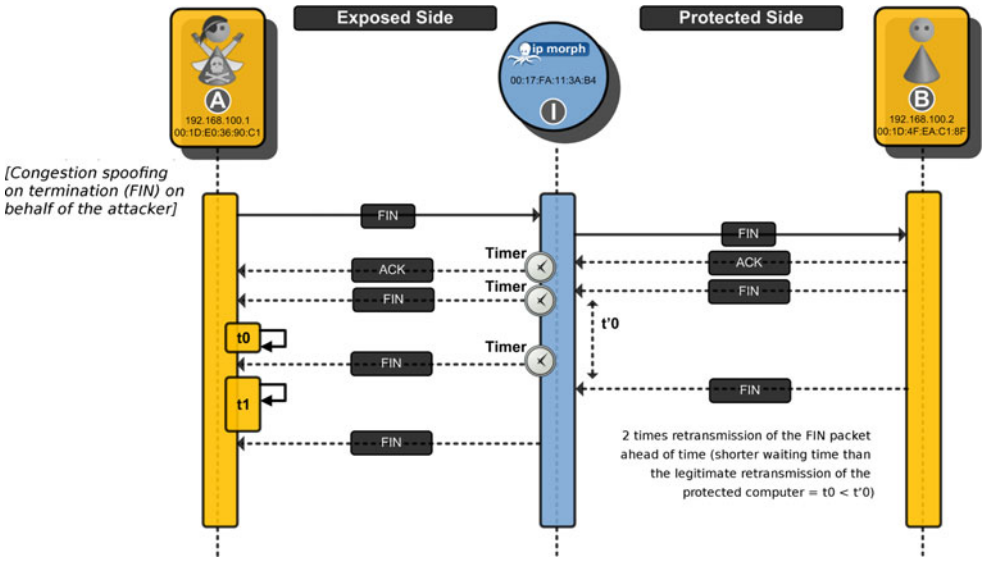

Fig. 9 Implementation of congestion spoofing on a session termination on behalf of the "exposed side" computer (i.e. "client")

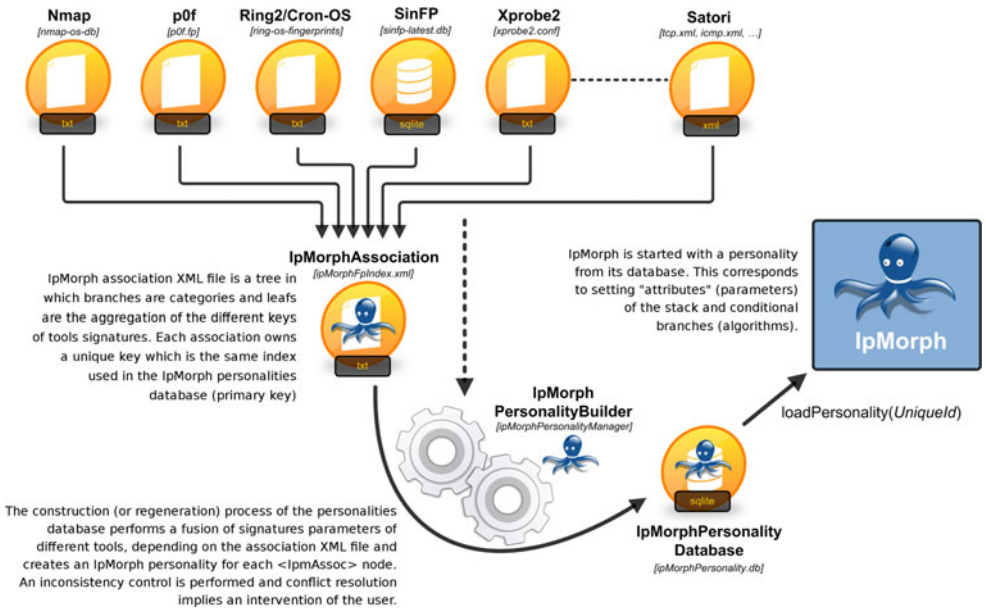

Fig. 10 Indexation/association principle for the creation of the personalities database and their use

This personality concept is simply the granular setup of the application according to the signatures of the tools. For us, an IpMorph personality is the union (or aggregation) of multiple signatures coming from different tools and parameters of the application itself.

Inside an indexation format (Fig. 10), we textually describe (using a simple text editor or the IpMorphGui with its PersonalityDBManager module) associations structured in categories and subcategories of different signatures of observed tools.

Based on this indexation schema, on the analysis and on the extraction (IpMorphFpParser class), we build (or re-build if the user asks us to) a SQLite database of IpMorph personalities (in a format close to the C++ structures/classes we use then in the virtual stack).

This personalities extraction and construction mechanism is able to detect some inconsistencies between signatures, whether they are due to an inconsistency of signatures between the tools or to a bad indexation in the association XML file. We think for example of "unique" attributes of the

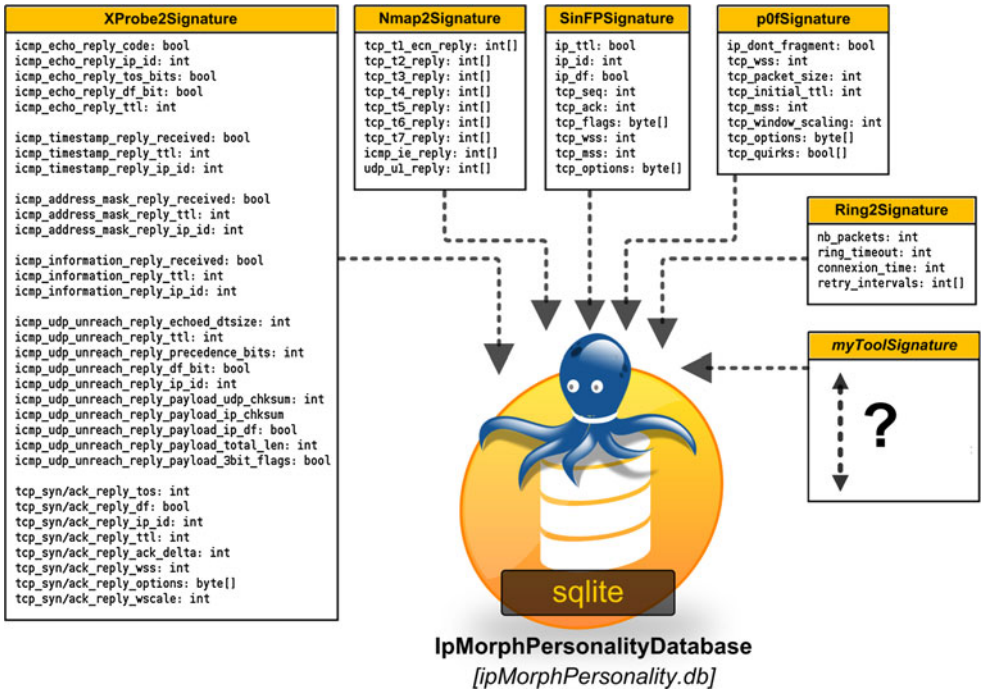

Fig. 11 Detailed example of reading and storing structures of signature extraction tools (IpMorphFpParser abstract class derived depending on the tools)

IpMorph stack (e.g. icmp.ip.defaultTtl or icmp.ip.idPolicy), that can be indicated or described with different values in different signatures (Fig. 11). Depending on the importance of these differences, the personalities creation tool may need some interaction (conflict resolution) with the user.

6 Conclusion and perspectives

We have demonstrated in this article that it was possible to design and create a fingerprint scrubber preventing most fingerprinting tools from working, whether they are in active or passive mode. The most significant work has been to precisely analyze each tool's modus operandi, and has offered us the ability to correct some of its malfunctions (e.g. Nmap) or to suggest changes to make these tools more relevant (e.g. SinFP). The necessary consideration of multiple signatures of different tools has enabled us to perform a synthesis work on different parameters of signatures and has naturally led us to the unified personality concept. Finally, the implementation of a TCP/IP stack substitute inside our tool and the extent of parameters and connections that are not currently used in fingerprinting tools signatures let us think that multiple OSFP techniques can appear.

IpMorph does not go any further than TCP for now, which is where, according to us, personality detection future starts. It appears critical to us to be able to go up into most of the basic network applicative layers to spoof the "real" content of traffic (DNS, DHCP, SMB,…) as well. This perspective seems all the more relevant as certain tools, like Satori, already perform this type of analysis on flows to identify probed computers fingerprint.

However, even if our design principles have to make IpMorph more scalable (to manage new versions of existing or upcoming tools as well as new fingerprinting methods), this kind of approach highly depends on evolutions of tools we want to fool, and often remains an ad hoc work, which can hardly be generalized. It seems to us especially obvious that it is quite easy to detect the presence of a tool like IpMorph. This last statement has to be put in context and moderated, because in the end people will only know they must not trust the performed fingerprinting and that a filtering mechanism prevents them from doing what they want.

Finally, beyond what IpMorph really is for now (a simple promising prototype resulting of a feasibility study), it is critical to perform an experimentation and measurement phase on this tool in a production environment, as well as, similarly as most prototypes, a new implementation phase to make IpMorph rugged, documented, distributable and sustainable.

However, the frame of this application (spoofing protected resources real identity), potential stakes, evolution perspectives and first feedback elements of people who we presented our designs to, let us think this project might quickly make a important contribution to the information systems security field.

References

1. Smart, M., Malan, G.R., Jahanian, F.: Defeating TCP/IP stack fingerprinting. In: Proceddings of the 9th USENIX Security Symposium. http://www.usenix.org/events/sec00/full_papers/smart/smart_html/index.html
2. Fyodor: Remote OS detection via TCP/IP stack fingerprinting. http://www.insecure.org/nmap/nmap-fingerprinting-article.txt
3. Spangler, R.: Analysis of remote active operating system fingerprinting tools, ettercap, Nmap and other OS detection tools. http://www.packetwatch.net/documents/papers/osdetection.pdf (2008)
4. Auffret, P.: SinFP, unification de la prise d'empreinte passive et active des systèmes d'exploitation, SSTIC 2008. http://www.gomor.org/bin/view/GomorOrg/ConfSstic2008
5. Veysset, F., Courtay, O., Heen, O.: New tool and technique for remote operating system fingerprinting. http://www.ouah.org/ring-full-paper.pdf (2002)
6. Smith, C., Grundl, P.: Know your enemy: passive fingerprinting. http://old.honeynet.org/papers/finger/ (2002)
7. Berrueta, D.B.: A practical approach for defeating Nmap OS-fingerprinting. http://nmap.org/misc/defeat-nmap-osdetect.htm (2003)
8. Trifero, S., Callaway, D.: Linux stealth patch. http://www.innu.org/~sean/ (2002)
9. Rehmet, G.: FreeBSD blackhole. http://www.gsp.com/cgi-bin/man.cgi?section=4&topic=blackhole
10. McCabe, R.: IPlog. http://ojnk.sourceforge.net/stuff/iplog.readme (2001)
11. Hartmeier, D.: OpenBSD packet filter. http://www.openbsd.org/faq/pf/index.html
12. Crenshaw, A.: OSfuscate: change your windows OS TCP/IP fingerprint to confuse P0f, NetworkMiner, ettercap, Nmap and other OS detection tools. http://www.irongeek.com/i.php?page=security/osfuscate-change-your-windows-os-tcp-ip-fingerprint-to-confuse-p0f-networkminer-ettercap-nmap-and-other-os-detection-tools (2008)
13. Provos, N.: Honeyd: a virtual honeypot daemon. http://www.citi.umich.edu/u/provos/papers/honeyd-eabstract.pdf (2003)
14. Wang, K.: Frustrating OS fingerprinting with morph. http://www.synacklabs.net/projects/morph/Wang-Morph-TheFifthHOPE.pdf (2004)
15. BridNet SSTIC 2005. http://www.bridnet.fr/files/23/sstic2005_bridnet.pdf
16. Hynesim http://www.hynesim.org
17. A painless guide to CRC error detection. http://www.repairfaq.org/filipg/LINK/F_crc_v3.html
18. CRC and how to reverse it. http://www.codebreakers-journal.com/downloads/cbj/2004/CBJ_1_1_2004 Anarchriz_CRC_and_how_to_Reverse_it.pdf
19. Veysset, F., Courtay, O., Heen, O.: Détection des systèmes d'exploitation avec RINGv2 Actes SSTIC 2003