Incorporating Active Fingerprinting into SPIT Prevention Systems

Hong Yan[†], Kunwadee Sripanidkulchai[*], Hui Zhang[†], Zon-Yin Shae[‡], and Debanjan Saha[‡]
[†]Carnegie Mellon University [*]NECTEC, Thailand [‡]IBM T.J. Watson Research

ABSTRACT

Voice over IP (VoIP) usage is growing at an astronomical rate. While the economic benefits of deployment are clear, concerns of potential abuse of the technology may hinder its deployment. Emerging security threats such as Spam over Internet Telephony (SPIT) are not far from reality and techniques to detect and prevent SPIT are ongoing research areas. In this paper, we design and implement a firewall to detect SPIT for VoIP devices that use the Session Initiation Protocol (SIP) which is gaining momentum as the de facto signaling protocol. For SPIT detection, we use *fingerprinting* techniques that run protocol analysis on the remote software agent to identify VoIP devices. We build and evaluate a prototype SPIT prevention firewall that leverages fingerprint information to identify call invitations from suspicious user agents. We demonstrate that fingerprinting incurs low overhead and is efficient enough to be incorporated into SPIT prevention systems.

1. INTRODUCTION

Voice over IP (VoIP) technology is gaining popularity as an alternative to traditional telephony in homes and enterprises. Deployment in homes is fueled by lower cost services provided by home broadband Internet services providers, VoIP providers [18] and peer-to-peer networks [16], collectively boasting up to 17.4 million users [20]. Similarly, enterprises are converting their internal PBX systems and contact centers to VoIP. The key motivation for this transition is not just the reduced cost, but the ease of integration of voice services with other network-based applications, particularly in the enterprise environment.

The key functionality required to support VoIP are signaling to establish and terminate calls, media transport for transmission of audio and video signals, and user location registration. In this paper, we focus on VoIP networks that use the SIP protocol [12], an IETF standard for signaling. Compared to the traditional telephony network, SIP is a fundamentally more complex environment. There is tremendous flexibility to interconnect components such as software and hardware-based phones, voice gateways, servers, and other software services in the system. Because SIP is an open standard, there are a large number of vendors of SIP components – there are at least seventy distinct SIP softphones [2] and sixty SIP servers [1].

As the network grows in scale and heterogeneity, potential abuse of VoIP technology may hinder its deployment. An emerging security threat that has already plagued other Internet applications such as email and instant messaging is spam. In this paper, we study mechanisms to prevent Spam over Internet Telephony (SPIT). In particular, we focus on call spam which is defined as a bulk unsolicited set of session initiation attempts (i.e., INVITE requests), attempting to establish a voice, video, or other type of communications session [11]. The state of the art in blocking SPIT varies from using black/white lists to applying machine learning algorithms and social network mechanisms. An area that has yet to be exploited is the additional knowledge of the remote user agent that is involved in the communications.

To draw a parallel with the Web, the communications protocol used by legitimate user agents such as Web browsers are implemented differently from those used by malicious user agents. In particular, malicious user agents often have specialized protocol implementations because (i) they often do not need all protocol features, (ii) smaller code means less complexity, more robustness and faster propagation, and (iii) the simplicity of text-based protocols enables everyone to implement their own version. For example, HTTP-based worms have different sets of HTTP headers and different response behavior when compared to typical Web browsers. Further, email spam bots have specialized built-in SMTP engines. We expect SIP spamming software (or hardware firmware) to have different implementations from legitimate user agents. We can then exploit such differences to identify the remote software agent and then make decisions to forward or drop their corresponding messages.

We implement a SPIT firewall that uses *fingerprinting* mechanisms to identify the remote software in VoIP applications and filter call invitations from suspicious user agents. The firewall may be used in personal computing environments, corporate environments or contact centers, and serves as a step towards a more secure VoIP environment. Our fingerprints are obtained from analysis of the protocol stack running on the SIP device. We demonstrate that there are indeed diverse SIP implementations that can been observed and fingerprinted from normal protocol exchanges in a passive manner (*passive fingerprints*). In addition, we explore *active fingerprints* that are obtained by introducing additional protocol exchanges. Passive fingerprinting has the advantage that it is not detectable by the remote user agent, but it is less robust than active fingerprinting if protocol headers are spoofed. Note that the SIP standard defines an optional *User-Agent* field that can be used to announce the software version of a SIP device. Fingerprints provide a more robust identification of the user agent particularly when the *User-Agent* field is not available or cannot be trusted. Although fingerprinting is used for SPIT detection in this paper, it can be used in many other applications such as classification of incoming SIP messages to provide different levels of service.

This paper is organized as follows: we begin with an overview of the SIP protocol in Section 2. In Sections 3 and 4 we present our SIP fingerprinting technique. We present the resulting fingerprints of twenty different SIP devices in Section 5 . We then present an implementation of a prototype firewall and its performance evaluation in Section 6. We discuss related work in Section 7 and summarize our findings in Section 8.

2. SIP BACKGROUND

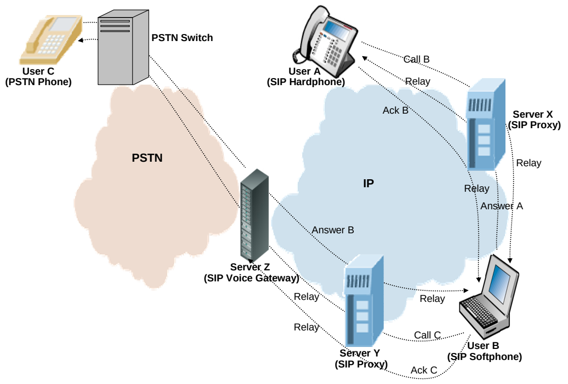


Figure 1: An example SIP infrastructure for voice services.

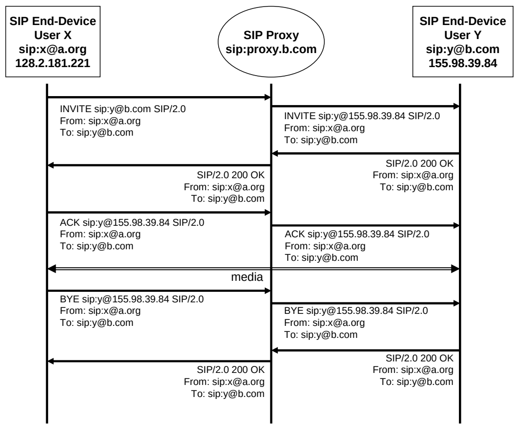


Figure 2: An example of SIP call establishment and tear down.

Our SIP fingerprinting techniques are based on protocol analysis. In this section, we provide a short overview of SIP as a foundation for understanding SIP fingerprinting in Sections 3 and 4.

2.1 SIP-Based System Architecture

SIP [12] is a signaling protocol used to register users, locate users, establish, maintain, and terminate sessions. While there are many competing signaling standards such as H.323 [4] and MGCP [3], SIP is gaining popularity because of its simplicity and lower overall cost. Further, SIP has the potential to go beyond voice services to enable easy integration of network-based applications such as instant messaging, video, games, Web, calendars, applets, and directory services.

Compared to the traditional telephony network, SIP is a fundamentally more complex environment. SIP is supported by a large number of vendors, including key ones such as Cisco, Avaya, and Microsoft. Various SIP hardware and software components from a variety of vendors may be interconnected in many different ways, depending on the services being offered. Because SIP is an application layer protocol, introducing and integrating new applications into the system is a simple and common process. The network may always be evolving with the flux of such new services.

Figure 1 illustrates a simplified SIP infrastructure that provides basic voice services. End-devices or user-agents, such as hardware-based phones (hardphones) and software-based phones (softphones) are used to initiate and listen for incoming calls. End-devices register their location with a registration server, often co-located with their SIP proxy (in this example, user B is registered with SIP proxy server X). SIP proxy servers participate in call signaling by relaying the signaling messages and resolving SIP addresses into IP addresses. Once a call is established, its media session may be transported directly between the end-devices if both are IP-based (between user A's hardphone and user B's softphone), or through a voice gateway (server Z) that translates the media between the IP network and the public switched telephone network (PSTN) between user B's softphone and user C's PSTN phone.

2.2 SIP Message Flow

The SIP protocol is the language that SIP devices, including end-devices and servers, use to exchange requests and responses. There are several types of requests that provide different functionalities. The simplified workflow depicted in Figure 2 illustrates how a SIP user uses a SIP INVITE request to establish a call. To uniquely identify SIP users, an addressing scheme based on SIP URIs (Uniform Resource Identifiers) is used. A typical URI has the format *sip:user@host*. For clarity, only important parts of the exchanged SIP messages are shown.

When user X initiates a call to Y, it sends an INVITE request to $Y's$ proxy server. The request is relayed to the address where Y is located. Y decides to accept the call, and thus responds to the INVITE request with "200 OK", which is then acknowledged by X. At this point, the three-way handshake is complete and a conversation dialog is established. X and Y exchange media, such as voice, until X sends out a BYE request and terminates the session.

2.3 SIP Message Formats

As illustrated in the above example, there are two classes of SIP messages: requests and responses. All the SIP requests (REGISTER, INVITE, OPTIONS, BYE, ACK, etc.) share the same format: a request line followed by a list of headers and an optional message body. For example, consider the INVITE message sent by user X in Figure 2. The request line, *INVITE sip:y@b.com SIP/2.0*, carries information about the request type, the request receiver and the SIP version. The various headers, *From: sip:x@a.org* etc, provide additional information regarding the request, for example the sender of the request. A request might also include a message body to convey more user information, but the structure or content of the message body is not defined by the protocol.

The format of a SIP response is similar to a SIP request, except that the start line is a status line which contains a status code indicating the outcome of the request. Consider the corresponding response from the SIP proxy to user X in Figure 2. As implied by "OK", the response with status code "200" means that the corresponding request has been understood and successfully performed.

3. SIGNS OF IMPLEMENTATION DIVERSITY

As a first step to see if there is sufficient diversity in SIP implementations, we set up a simple experiment using thirteen different hardphones and softphones. We use the phones to send out INVITEs (as though to initiate calls) and log the protocol messages. We then analyze the INVITE messages of these phones by looking at the existence of header fields and their corresponding ordering. We call this analysis passive fingerprinting because we are running the analysis on existing protocol exchanges without introducing any additional messages.

Our reasoning for using these thirteen phones in our experiments is as follows. We use Cisco and Pingtel hardphones for fingerprinting as they are mature products that are widely deployed in enterprises. The eleven softphones, all of which support SIP are selected from a VoIP softphone list published on the VoIP informational Wiki [2]. These are popular softphones that run on a variety of OS platforms and have diverse features, cost and vendors. Among the eleven softphones, four (Adore Softphone, Express Talk, SIPp, and WinSip) only run on Windows, two (KPhone and LinPhone) are for Linux, three (Phoner, sipXphone, and Yate) run on both Windows and Linux, and the other two (eyeBeam and SJPhone) have Windows, Mac OS X, and Pocket PC versions. The features supported by those softphones also vary across a large range. For example, Express Talk has the ability to put calls on hold and do call transfer, Adore SoftPhone and eyeBeam can be used as video phones, eyeBeam and KPhone support Instant Messaging, and WinSip is more of a bulk call generator and testing tool. Some of the phones such as LinPhone, KPhone and sipXphone are Open Source, some such as Phoner are Freeware, and others are commercial products. While most of the softphone vendors are US based, we also pick softphones that have are based outside the US. For example, we use Adore Softphone, Yate and Phoner based in India, Romania, and Germany, respectively.

Table 1 lists the passive fingerprints obtained from analyzing the header fields observed in the INVITE messages of the thirteen different phones. Note that each not all devices have the same set of headers. For example, the Cisco hardphone has fourteen header fields whereas the Yate softphone has only ten. Further, the ordering of the fields varies from device to device. For example, *Via* is the first header from Cisco hardphones whereas *Max-Forwards* is the first header from Yate softphones. Each of the thirteen fingerprinted phones has a distinct passive fingerprint.

Passive fingerprinting can distinguish between the different devices effectively by simply analyzing the contents of one INVITE packet. However, it is fairly easy for user agents who wish to not be identified to spoof passive fingerprinting by including different sets of header fields or reordering the header fields. Next, we present active fingerprinting where we probe the remote user agent with crafted SIP messages to elicit implementation-specific responses which are then used as fingerprints. Active fingerprinting is more robust at identifying the SIP protocol stack as it requires the user agent to implement more complex changes to its protocol stack if it does not wish to be identified. The remainder of the paper leverages active fingerprinting techniques.

4. ACTIVE FINGERPRINTING TECHNIQUES

In active fingerprinting, we probe the remote device using a set of specially crafted SIP messages. To craft SIP messages, we manipulate the SIP request headers in standards-compliant and non-compliant ways. By sending such messages as probes and analyzing the responses, we obtain a unique response sequence which we call a *fingerprint* from each type of SIP device. We note that some of the valuable information is from responses to non-compliant messages for which there is no specified behavior in the RFC. Thus, the differences in responses reflect differences in implementation. Our fingerprints are based on the OPTIONS request, though our techniques do not exclude the use of other types of SIP messages such as INVITE, CANCEL, etc.

To extract the fingerprint, we look at two sources of information: the returned status code and the values returned in certain response header fields. In general, the status codes we have observed fall under the following categories: “200 OK”, “3xx Redirection”, “4xx Request Failure”, “5xx Server Failure”, and no response. To extract status code fingerprints, we probe SIP devices with standards-compliant OPTIONS probe and non-compliant probes as described in Sections 4.1 and 4.2. To extract value-based fingerprints, we look at the content returned in the *Allowed Methods* header in the responses which we describe in Section 4.3. Note that this is not meant to be a comprehensive list as our focus in this paper is to introduce the fingerprinting methodology. Other manipulations to extract additional information to be used as fingerprints may be possible.

4.1 Responses to Standards-Compliant OPTIONS

A SIP OPTIONS request is used to query for the capabilities of a SIP device. When a SIP device receives an OPTIONS request, it responds with the complete set of supported SIP message types.

According to the SIP standard [12], a valid SIP OPTIONS request starts with a request line that contains the keyword ”OPTIONS”, a URI that identifies the recipient of the request, and SIP-Version that indicates the SIP version in use (“SIP/2.0” to be standards-compliant). In addition to the mandatory request line, a SIP OPTIONS request must also contain the following six header fields: Via, CSeq, Call-ID, To, From, and Max-Forwards. The following is an example of a standards-compliant OPTIONS request:

```
OPTIONS sip:128.2.181.221 SIP/2.0
Via: SIP/2.0/UDP 155.98.39.84;branch=z9hG4bKhjhs8as877
CSeq: 1 OPTIONS
Call-ID: a84b4c76e66710
To: <sip:128.2.181.221>
From: Anonymous <sip:anonymous@sipfw.org>;tag=1928301774
Max-Forwards: 70
```

The *Via* header field indicates the transport used for the message and provides the location where the OPTIONS response is to be sent. In the above example, *UDP* is used as the transport. The OPTIONS request is to be sent to *128.2.181.221*, and the response is expected to be sent back to *155.98.39.84*. The *Via* header is also required to carry a branch parameter that identifies the transaction created by the request.

The *CSeq* header, which consists of a sequence number and keyword OPTIONS, serves as a way to identify and order transactions. The RFC requires that the sequence number value be expressible as a 32-bit unsigned integer.

The *Call-ID* is a unique identifier that is selected when a new request is generated and remains the same in the request-response exchange dialog.

The *To* header field specifies the recipient of the OPTIONS request, or the address of the request target (*128.2.181.221* in the example).

The *From* header field indicates the sender of the OPTIONS request. The sender can choose to hide the identity, as shown in the example, by using the display name “Anonymous” along with a syntactically correct but otherwise meaningless URI. A tag parameter should be appended to the end of the *From* header field such

SIP Component	Header Fields in INVITE Message
Hardphones	
Cisco Phone	Via,Record-Route,Via,From,To,Call-ID,Date,CSeq,User-Agent,Contact,Expires,Content-Type,Content-Length,Accept
Pingtel Phone	Via,Record-Route,From,To,Call-ID,CSeq,Contact,Content-Type,Content-Length,Accept-Language,Allow,Supported,User-Agent,Date,Via
Softphones	
Adore Softphone	Via,Max-Forwards,From,To,Call-ID,CSeq,Contact,User-Agent,Content-Type,Content-Length
Express Talk	Via,To,From,Call-ID,CSeq,Max-Forwards,User-Agent,Contact,Allow,Supported,Content-Type,Content-Length
eyeBeam	Via,Max-Forwards,Contact,To,From,Call-ID,CSeq,Allow,Content-Type,Supported,User-Agent,Content-Length
KPhone	Via,CSeq,To,Content-Type,From,Call-ID,Subject,Content-Length,User-Agent,Contact
LinPhone	Via,From,To,Call-ID,CSeq,Max-Forwards,User-Agent,Subject,Expires,Allow,Content-Length
Phoner	Via,From,To,Call-ID,CSeq,Contact,Max-Forwards,User-Agent,Allow,Content-Type,Content-Length
Sipps	Via,From,To,Call-ID,CSeq,User-Agent,Expires,Accept,Content-Type,Content-Length,Contact,Max-Forwards,Allow
sipXphone	From,To,Call-ID,CSeq,Contact,Content-Type,Content-Length,Date,Max-Forwards,User-Agent,Accept-Language,Allow,Supported,Via
SJPhone	Via,Content-Length,Contact,Call-ID,Content-Type,CSeq,From,Max-Forwards,To
WinSip	Via,Max-Forwards,From,To,User-Agent,Call-ID,CSeq,Contact,Allow,Accept,Accept-Language,Content-Type,Content-Disposition,Content-Length
Yate	Max-Forwards,Via,From,To,Call-ID,CSeq,User-Agent,Allow,Content-Type,Content-Length

Table 1: Passive fingerprints obtained from INVITE messages of SIP hardphones and softphones.

that when combined with the *Call-ID* can uniquely identify a conversation.

The *Max-Forwards* header field contains an integer that limits the number of hops a request can transit on the way to its destination. The integer, set to *70* in our example, is decremented by one at each SIP hop.

The response to an OPTIONS message, from the eyeBeam [5] softphone is shown in the following example.

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP 155.98.39.84;branch=z9hG4bKhjhs8as877
Contact: <sip:128.2.181.221:5060>
To: <sip:128.2.181.221>;tag=7c3d124c
From: Anonymous <sip:anonymous@sipfw.org>;tag=1928301774
Call-ID: a84b4c76e66710
CSeq: 1 OPTIONS
Accept: application/sdp
Accept-Language: en
Allow: INVITE,ACK,CANCEL,OPTIONS,BYE,REFER,NOTIFY,MESSAGE,
       SUBSCRIBE,INFO
```

The status code returned from the eyeBeam softphone for a compliant probe is “200 OK”. Our fingerprints our obtained from collecting such status codes from responses to the various probes that we send. Each device has a distinct fingerprint.

4.2 Responses to Non-Compliant OPTIONS

Next, we describe different manipulations of the OPTIONS request in non-compliant ways by changing individual header fields. Note that this is not meant to be a comprehensive list.

Invalid Version

The next fingerprint is extracted from the status code to an OPTIONS message that has an invalid SIP version. The RFC requires all SIP messages to carry the version information “SIP/2.0”. We replace the RFC-required version number with a nonexistent one, e.g. “99.9” in the following example.

```
OPTIONS sip:128.2.181.221 SIP/99.9
Via: SIP/99.9/UDP 155.98.39.84;branch=z9hGbKhjhs8as877
...
```

While the RFC does not define how the SIP stack should react to invalid versions, the most appropriate status code is “505 Version Not Supported”. However, we find that among the twenty types of SIP devices we probed, two of them respond with “505 Version Not Supported” while the others either process the request as if it were valid, do not respond, or respond with some different status code. We list all those responses in Section 5.

Invalid Via Address

As discussed in Section 4.1, the *Via* header field contains the address where the OPTIONS response should be sent. Instead of using the correct IP address, we fill the *Via* header field with the text “localhost”. This tells the probed SIP device to send the response back to itself rather than the person issuing the probe. Upon receiving such requests, again different SIP implementations react differently.

Incorrect Content-Length

The next probe uses an incorrect content length for the message. *Content-Length* is an optional header field that indicates the size of the message body in decimal number of octets. When no body is present in a message, the *Content-Length* field value must be set to zero. In our “Incorrect Content-Length” probe, we add a non-zero *Content-Length* value to an OPTIONS request that does not have a body.

Malformed CSeq

Next we describe the malformed CSeq probe. A well-formed *CSeq* header field in an OPTIONS request contains both a single decimal sequence number and the request method keyword “OPTIONS”. However, we remove the “OPTIONS” keyword and only include the sequence number in our probe.

Missing Call-ID

Next, we remove the mandatory *Call-ID* field from the SIP-compliant request. Although the RFC requires that user agents respond to such requests with “400 Bad Request”, different SIP implementations respond differently.

Incompatible Transport Protocol

The RFC requires the *Via* header field to include the transport protocol that is used to send the request. The transport protocols supported by SIP are UDP, TCP, TLS, and SCTP. Our “Incompatible Transport Protocol” probe manipulates the *Via* field to introduce a mismatch between the claimed transport protocol and the one actually used. For example, we use UDP to send an OPTIONS request but claim to have used a TCP transport.

4.3 Allowed Methods

In addition to looking at status codes in the responses to active probing, we also extract values from the response headers as part of the SIP fingerprint to improve its identification capabilities.

The OPTIONS response message contains information regarding the methods that are supported by the component. Upon receiving a compliant OPTIONS request, a user agent is expected to respond with a list of supported SIP methods in the *Allow* field. An example of a SIP response with an *Allow* header can be found in Section 4.1.

The values in the *Allow* list provide useful information for inference of the SIP stack because different user agents support different requests. Further, the ordering of the supported requests is also different. However, some user agents may choose to not provide

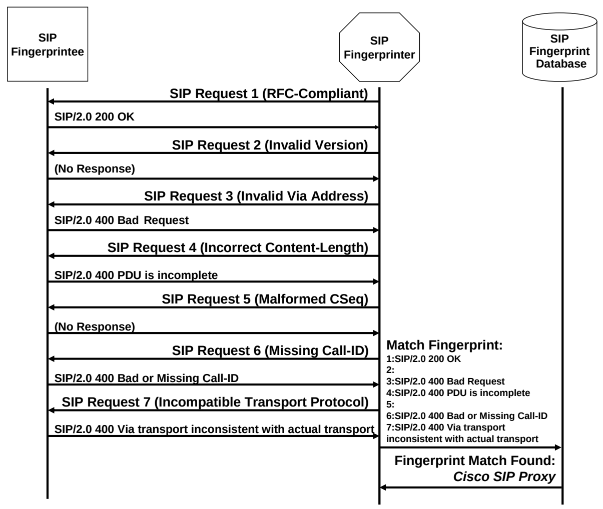

SIP Fingerprintee
SIP Fingerprinter
SIP Fingerprint Database
SIP Request 1 (RFC-Compliant)
SIP/2.0 200 OK
SIP Request 2 (Invalid Version)
(No Response)
SIP Request 3 (Invalid Via Address)
SIP/2.0 400 Bad Request
SIP Request 4 (Incorrect Content-Length)
SIP/2.0 400 PDU is incomplete
SIP Request 5 (Malformed CSeq)
(No Response)
SIP Request 6 (Missing Call-ID)
SIP/2.0 400 Bad or Missing Call-ID
SIP Request 7 (Incompatible Transport Protocol)
SIP/2.0 400 Via transport inconsistent with actual transport
Match Fingerprint:
1:SIP/2.0 200 OK
2:
3:SIP/2.0 400 Bad Request
4:SIP/2.0 400 PDU is incomplete
5:
6:SIP/2.0 400 Bad or Missing Call-ID
7:SIP/2.0 400 Via transport inconsistent with actual transport
Fingerprint Match Found:
Cisco SIP Proxy

Figure 3: An example of active fingerprinting workflow.

any information about the requests that it supports for security purpose. For example, the OPTIONS response from a softphone called sipXphone [15] does not have an *Allow* field. Nevertheless, its response is valid; it does not mean that sipXphone does not support any method.

5. FINGERPRINTS OF SIP DEVICES

Next, we report the resulting fingerprints obtained by active fingerprinting twenty different SIP devices. We find that each device has a distinct fingerprint that can be used for identification.

Active fingerprints are obtained by probing SIP devices with the protocol message manipulations and analyzing the results as discussed in Section 4. Figure 3 shows an example of active fingerprinting workflow. The fingerprinter sends seven crafted SIP OPTIONS requests to the fingerprintee. The fingerprintee returns “200 OK” only to the RFC-compliant OPTIONS, and returns “400” with different error messages such as “Bad Request” and “PDU is incomplete” to OPTIONS requests with invalid via address, incorrect content-length, missing Call-ID, and incorrect transport protocol. And it does not respond to OPTIONS requests with invalid version or malformed CSeq. The resulting status codes are logged and stored in a database of SIP fingerprints for future reference. For example, a device’s fingerprint may be compared with fingerprints of known devices to find a match.

We built a tool called SIPProbe [21] that uses the above workflow to implement active fingerprinting and probed twenty different SIP components, including voice gateways, SIP proxies, softphones and hardphones. The phones are the same as those in Table 1. The servers that we probe are chosen based on access and availability: we have access to an operational Cisco-based deployment with voice gateways and proxies, and the remaining SIP proxies are public SIP servers that are available for testing purposes [13]. Their fingerprints are listed in Table 2, with each row corresponding to a component. The columns denote the seven manipulations described in Section 4 and the observed status codes to each manipulation.

For example, the Cisco voice gateway listed in the first row returns a “200 OK” response to RFC-compliant and incorrect Via address OPTIONS requests, but returns a “400” to invalid version, missing Call-ID, and incorrect transport protocol requests. In addition, it does not respond to OPTIONS messages with incorrect content length or malformed CSeq.

Each SIP component has its own distinct fingerprint. Even similar components from the same vendor, such as the two different versions of Cisco voice gateways have significantly different fingerprints. This indicates that they are not implemented using the same SIP stack. Further, none of the other Cisco components (SIP proxy and hardphone) have the exact same fingerprint. We note that in general, active fingerprinting is difficult to forge. This is because there are many possible manipulations of OPTIONS messages that can be used as probes in addition to ones we present in this paper. If the fingerprinting algorithm selects a random subset of such probes to use, a malicious user agent cannot predict ahead of time which probes it will receive and cannot rely on returning predetermined responses to look like good user agents. In fact, a malicious user agent cannot have the same fingerprint as a good user agent unless they have the same protocol logic and implementation – they use the same protocol stack. We argue that this is not expected to be the case in Section 1.

Next, we present the fingerprinting results for the $Allow$ field. Table 3 lists the $Allow$ fields of the user agents that responded with an $Allow$ field. Of the twenty fingerprinted devices, thirteen responded with the $Allow$ field. The other user agents from Table 2 that are not listed did not respond with an $Allow$ field or did not respond at all. For example, the Cisco voice gateway listed in the first row supports thirteen different methods whereas the Cisco phone supports only eight. Also, note that the ordering of the supported methods is also different. The first Cisco voice gateway lists INVITE as the first method, but the Cisco phone lists OPTIONS as the first method. For those user agents that populate their $Allow$ fields, the methods together with their ordering are incorporated as part of their SIP fingerprints.

We note that a particular weakness with the $Allow$ field is that as SIP devices become more mature, they will likely support more and potentially *all* methods. Therefore, the usefulness of this field will decrease over time. However, the ordering of methods may still provide useful information.

6. ACTIVE SPIT PREVENTION FIREWALL

Leveraging active fingerprinting, we develop a SIP firewall called *sipfw* to detect and filter SPIT calls. In contrast to traditional firewalls, our firewall *actively* fingerprints callers and uses fingerprints to make allow or deny decisions. Section 6.1 describes how we incorporate fingerprints into the design and implementation of *sipfw* and discusses a number of implementation issues to improve the efficiency of the firewall. Section 6.2 evaluates the performance of the firewall in terms of response time and throughput.

6.1 System Design and Implementation

The goal of the SPIT prevention firewall is to determine whether or not to forward incoming INVITEs to its users. Upon seeing an incoming INVITE message, the firewall must perform the following operations: (i) identify the remote caller, (ii) classify the remote caller as a spammer or a normal user, and (iii) allow or deny the INVITE message.

Figure 4(a) illustrates the operations when calls are denied. The caller sends an INVITE message to the SIP address of the callee, but the message is intercepted by the firewall, which first sends back a Ringing message (according to the SIP protocol) to prevent the caller from thinking that the INVITE is lost and retransmitting it, and then identifies the caller by actively fingerprinting it using the techniques described in Section 4. Once a fingerprint is obtained, the firewall checks with its allow/deny policy and decides that the call should be blocked. It then drops the INVITE message

	SIP Fingerprint						
Component	RFC-Compliant	Invalid Version	Incorrect Via Address	Incorrect Content Length	Malformed CSeq	Missing Call-ID	Incorrect Transport Protocol
SIP Servers							
3Com SIP Proxy (siphappens.com)	405	405	NR	405	NR	NR	NR
Cisco Voice Gateway (Cisco-SIPGateway/IOS-12.x)	200	400	200	NR	NR	400	400
Cisco Voice Gateway	400	400	400	NR	NR	400	400
Cisco SIP proxy	NR	NR	400	400	NR	400	400
MCI SIP Proxy (sipaccount.mci.com)	302	400	NR	302	NR	NR	400
Microappliances SIP Proxy (zdots.com MA-1000-2.1)	403	400	403	403	NR	NR	400
SIP Express Router Proxy (iptel.org 0.0.0udpfifo i386/linux)	404	NR	404	404	NR	404	NR
Hardphones							
Cisco Phone (cisco.com)	200	NR	200	400	NR	400	NR
Pingtel Phone (pingtel.com)	200	505	200	200	NR	NR	NR
Softphones							
Adore Softphone (adoresoftphone.com)	200	481	NR	400	NR	400	NR
Express Talk (nch.com.au)	200	200	200	200	NR	200	200
eyeBeam (counterpath.com)	200	200	200	200	405	NR	200
KPhone (wirlab.net)	200	200	NR	200	200	200	NR
LinPhone (linphone.org)	200	200	200	200	NR	NR	NR
Phoner (phoner.de)	200	200	200	200	NR	NR	200
Sipps (nero.com)	200	200	200	200	400	NR	NR
sipXphone (sipfoundry.org)	200	505	200	200	NR	NR	200
SJPhone (sjlabs.com)	405	NR	405	NR	NR	NR	NR
WinSip (touchstone-inc.com)	200	NR	200	200	NR	481	481
Yate (yate.null.ro)	501	NR	NR	501	NR	501	501

Table 2: Fingerprints of various SIP components. "NR" denotes no response.

Component	SIP Allowed Field in OPTIONS Response
SIP Servers	
Cisco Voice Gateway (Cisco-SIPGateway/IOS-12.x)	INVITE, OPTIONS, BYE, CANCEL, ACK, PRACK, COMET, REFER, SUBSCRIBE, NOTIFY, INFO, UPDATE, REGISTER
Hardphones	
Cisco Phone (cisco.com)	OPTIONS, INVITE, BYE, CANCEL, REGISTER, ACK, NOTIFY, REFER
Pingtel Phone (pingtel.com)	INVITE, ACK, CANCEL, BYE, REFER, OPTIONS, NOTIFY, REGISTER, SUBSCRIBE
Softphones	
Adore Softphone (adoresoftphone.com)	INVITE, BYE, OPTIONS, MESSAGE, ACK, CANCEL, NOTIFY, SUBSCRIBE, INFO, REFER
Express Talk (nch.com.au)	INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY
eyeBeam (counterpath.com)	INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY, MESSAGE, SUBSCRIBE, INFO
KPhone	INVITE, OPTIONS, ACK, BYE, MSG, CANCEL, MESSAGE, SUBSCRIBE, NOTIFY, INFO, REFER
LinPhone (linphone.org)	INVITE, ACK, OPTIONS, CANCEL, BYE, SUBSCRIBE, NOTIFY, MESSAGE, INFO
Phoner (phoner.de)	INVITE, ACK, CANCEL, BYE, NOTIFY
Sipps (nero.com)	INVITE, ACK, CANCEL, BYE, REFER, OPTIONS, NOTIFY, INFO
SJPhone (sjlabs.com)	INVITE, ACK, CANCEL, BYE, REFER, NOTIFY
WinSip (touchstone-inc.com)	INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, INFO
Yate (yate.null.ro)	ACK, INVITE, BYE, CANCEL

Table 3: Allow fields of various SIP components. "N/A" indicates that the Allow field is not included in the response.

and declines the call. When the caller acknowledges the Decline message, the dialog ends. In this scenario, SIP messages are exchanged only between the caller and the firewall. The callee is unaffected since no SIP traffic is leaked to it.

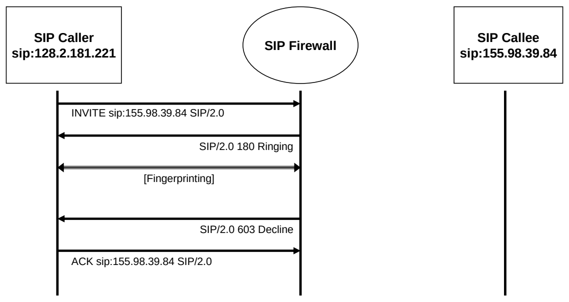
Figure 4(b) shows the call flow in the scenario where calls are accepted by *sipfw*. In contrast to the "deny" scenario, the call invitation is allowed to pass through the firewall to the callee. The callee sends back a sequence of Trying, Dialog Establishment, and Ringing messages, and eventually answers the phone by sending OK according to the SIP protocol. The OK is acknowledged and media exchange starts. To end the call, the caller and callee exchange BYE and OK, and successfully complete the call.

Next, we describe the design of *sipfw* to support these operations. Figure 5 shows the architecture of our SIP firewall. The Packet Filter passes SIP packets from the kernel to the user space. After parsing the SIP header, the Packet Filter puts call invitations into the INVITE Queue. The Packet Filter also writes probe responses to a global Probe Response Table when responses are received. The Scheduler gets the call invitations from the INVITE queue and sets up Probe Timers and Verdict Timers. When triggered, the Probe Timer sends a probe to the call initiator, and the Verdict Timer decides to allow or deny the call. In addition, a Result Cache is used to store recent verdict outcomes, and an Aging Thread periodically scans the Result Cache to clean up expired entries.

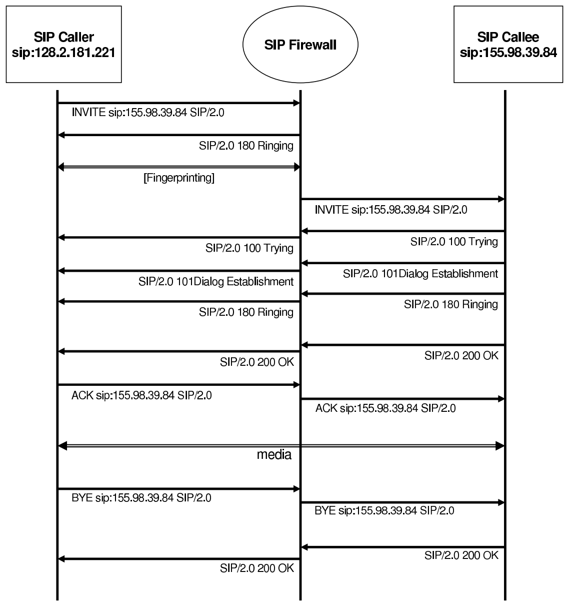
We now describe four important system components: the Packet Filter, Scheduler, Timer, and Result Cache.

Packet Filter: We prototype our firewall in Linux, where it runs as a Linux user-space program and intercepts packets from the kernel using the *netfilter* packet filter library. The Linux *netfilter* framework supplies an interface to pass network packets from the kernel to a user process, modify it, and reinject it to the kernel. We use the packet interception and reinjection functionality encapsulated in the *libnetfilter_queue* library which is supported in kernel versions 2.6.14 and higher.

Scheduler and Timer: The scheduler is a thread that fetches INVITE messages from the INVITE queue and sets up two timers:



9(a) Basic call flow for a call denied by sipfw.


SIP Caller
sip:128.2.181.221
SIP Firewall
SIP Callee
sip:155.98.39.84
INVITE sip:155.98.39.84 SIP/2.0
SIP/2.0 180 Ringing
[Fingerprinting]
INVITE sip:155.98.39.84 SIP/2.0
SIP/2.0 100 Trying
SIP/2.0 100 Trying
SIP/2.0 101Dialog Establishment
SIP/2.0 101Dialog Establishment
SIP/2.0 180 Ringing
SIP/2.0 180 Ringing
SIP/2.0 200 OK
SIP/2.0 200 OK
ACK sip:155.98.39.84 SIP/2.0
ACK sip:155.98.39.84 SIP/2.0
media
BYE sip:155.98.39.84 SIP/2.0
BYE sip:155.98.39.84 SIP/2.0
SIP/2.0 200 OK
SIP/2.0 200 OK

9(b) Basic call flow for a call accepted by sipfw.

Figure 4: Basic call flows with sipfw.

probe timer and verdict timer. The probe timer gets a time interval from the scheduler, and sends out fingerprinting probes to the caller according to that interval. The verdict timer obtains a waiting time from the scheduler, and waits until the waiting time has passed before reading the probe response table for the resulting fingerprint. After a fingerprint is obtained, the verdict timer matches the fingerprint with a database of known fingerprints (as described in Figure 3) and determines whether the INVITE message should be allowed or denied. Based on the decision, the verdict timer instructs the packet filter to either reinject the INVITE message to the kernel or to drop the message.

The value of the verdict timer directly affects call establishment response time as a call can only be established after fingerprinting has completed (when the verdict timer expires). We optimized the verdict timer by making it adapt to the round-trip-time estimate as follows: when the first probe is received, we estimate the round-trip time and adjust the timer to expire when all probe responses are expected to arrive.

Result Cache: Another optimization that can help reduce response time is to avoid repeatedly probing a caller. We use a result cache to store recent verdicts that have been made. If a verdict already exists, we can make an allow/deny decision immediately without having to probe the caller again.

The result cache is implemented as follows. Each entry in the result cache contains the IP address of the caller, the time when the verdict was made, and a boolean indicating the allow/deny decision. An aging thread periodically scans the result cache to purge old cache entries. An entry is created in the result cache when fingerprinting starts. And the verdict result is written to the result cache when the fingerprinting completes. Before fingerprinting, the scheduler checks the result cache for existing entries. If a match is found, the scheduler does not create a probe timer so that no new probes are issued. However, fingerprint results may not be available yet if the fingerprinting is still in progress. If the results are not yet available, the scheduler creates a verdict timer with the same expiration as the pending verdict timer for this caller. Thus, if a malicious caller floods the system by sending many “INVITE”s we only probe the caller based on the first “INVITE” and automatically block the following calls if they occur before the preceding one expires from the result cache.

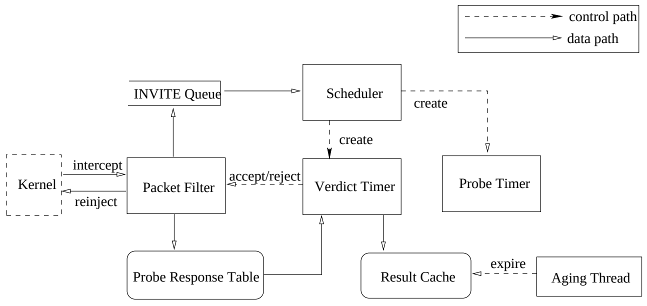


Figure 5: SIP Firewall Architecture

6.2 System Evaluation

The goal of our evaluation is to understand the behavior of SIP fingerprinting in the context of a prototype SPIT prevention firewall. We describe the evaluation metrics, experimental setup and evaluation results next.

6.2.1 Evaluation Metrics

For firewall performance evaluation, we are interested in two key metrics: call throughput and call establishment time.

Throughput: Throughput is measured as the number of calls the firewall is able to handle per second. Fingerprinting requires the firewall to handle additional probing and fingerprint collecting processing for each call. If the throughput with fingerprinting is reasonably high enough, then it is a good indication that the firewall may be deployable in realistic environments such as an enterprise network.

Response Time: Call setup response time is measured as the amount of time it takes from when an INVITE is sent from the caller to the time when the call is answered by the receiver. This response time includes the time it takes for the firewall to fingerprint the caller and allow the call to go through. Low call response times are desirable, particularly low enough to not hurt the interactive nature of the application.

6.2.2 Experimental Setup

Throughput Experiments

We run the throughput experiments on Emulab [19]. Figure 6

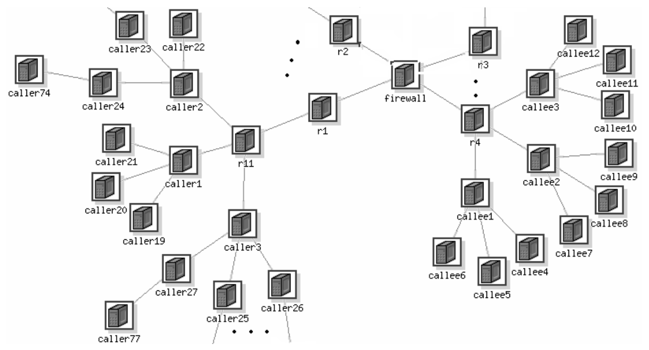


Figure 6: Emulab network topology for call throughput experiment.

shows our emulab testbed setup. We use 100 emulab nodes to simulate callers and another 100 as callees[1]. The callers are connected to the SIP firewall to form a tree topology on one side of the firewall, and the callees also form a tree on the other side of it. All links have 100 Mbps bandwidth and the nodes have the following properties.

- SIP firewall: 2.0 GHz Pentium 4, 512 MB, Linux 2.6.14
- Callers/callees: 850 MHz Pentium III, 256 MB, Linux 2.6.14

We run LinPhone on the callers and callees. We modify LinPhone so that callers repeatedly make calls at specified rates and callees automatically answer and terminate calls. We change the callers' call rates (i.e., offered load on the firewall), and measure the number of calls that successfully complete. We also use the Linux *top* utility to check the firewall's CPU utilization.

Response Time Experiments

To evaluate the response time of the firewall under realistic network delays and conditions, we use 200 machines[2] from the Planetlab [9] testbed as callers. Our Planetlab callers represent a wide variety of processor and network conditions. In the experiment, each Planetlab caller calls the callee repeatedly ten times. The callee is configured to automatically answer the call as soon as it receives it. The firewall is configured the same way as in the throughput experiments.

6.2.3 Call Throughput

Figure 7 shows the call throughput on the left y-axis versus the offered load on the x-axis. The SIP firewall is able to keep up with the increasing load until it reaches 1000 calls per second. However, beyond that rate, *sipfw* starts to drop call requests and the system is not stable. To understand what is causing this bottleneck, we look at the bandwidth and CPU resources consumed by the firewall. We find that the bandwidth is below 10% of the link capacity. However, the CPU on the firewall PC is used intensively. The right y-axis in Figure 7 shows the peak, average, and minimum CPU utilization of the firewall vs. the offered load. After the offered call rate increases to above 1000 calls per second, the peak and average CPU usage surpasses 80%. As a result SIP messages start to get dropped from netfilter_queue because the dequeue speed of the processing thread can not catch up with the arrival rate of the SIP messages.

In this experiment, we find that the throughput of *sipfw* is mainly CPU bound. When installed on a moderate speed PC, its throughput is about 1000 calls per second which is reasonably high enough

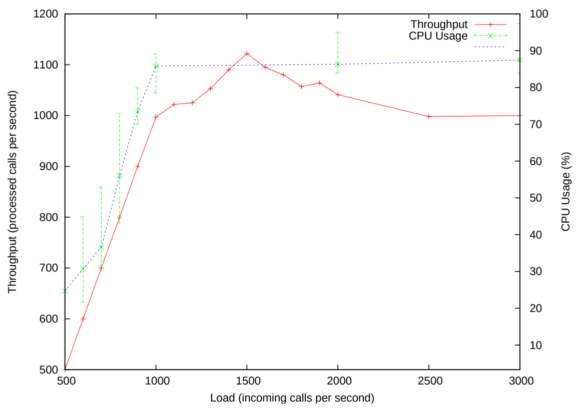


Figure 7: Call throughput and CPU utilization under increasing request load.

to deploy in call centers or enterprise environments.

6.2.4 Call Establishment Time

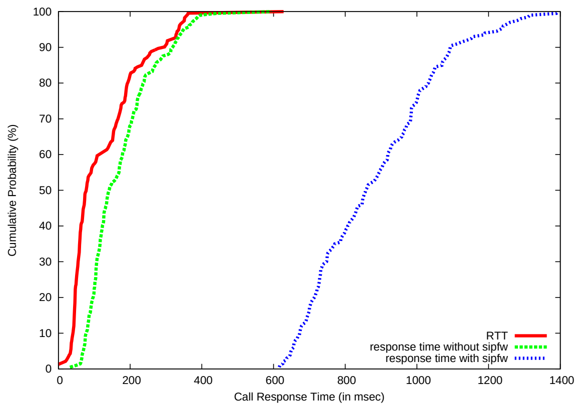


Figure 8: Cumulative distribution of round-trip times and response times with firewall.

In this section, we look at the call establishment time between Planetlab machines and our callee. We first look at the response time without any firewalls as the baseline configuration. We then look at the response time overhead introduced by *sipfw*. Last, we look at the effectiveness of caching fingerprint verdicts.

Figure 8 shows the cumulative distribution of round-trip times, call response times without and with *sipfw*. The firewall increases the response time by 600 to 800 msec. And since *sipfw* uses an adaptive timer, the increase is proportional to the original call establishment time.

This experiment shows that the cost to fingerprint callers adds to the call establishment time by at most 800 msec. This is not likely to cause any observable delay. Further, the response time can be shortened by caching results which we discuss next or additional timer optimizations.

6.2.5 Fingerprinting Result Caching

We also study the effectiveness of caching the fingerprinting results using a 2-month long call trace collected from a moderate deployment of VoIP phones in a small enterprise network. The number of phones is on the order of a few hundreds and the call rate is roughly 5-10 calls/minute.

9[1]Limited by the most we were able to reserve for the experiments.

9[2]Limited by the maximum number of planetlab slices we could get.

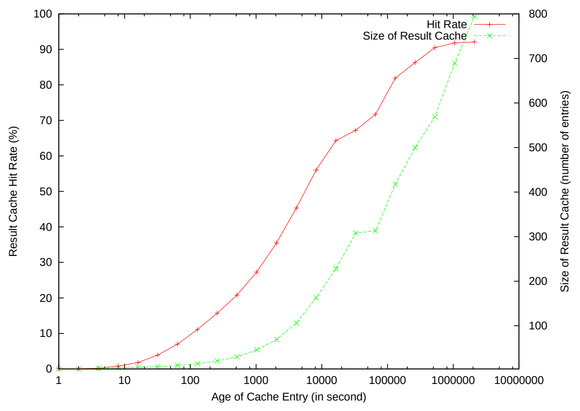


Figure 9: The effect of cache age on hit rate and required cache size.

We look at the following metrics: cache hit rate and caching table size with respect to the "cache age". "Cache age" is defined as the time from when a cache entry is created to the time that it is removed from the cache table. Figure 9 shows the effect of cache age on the performance and overhead of caching. The x-axis is the cache age. The left y-axis is the cache hit rate. After a slow start the cache hit rate reaches 25% when cache age increases to 1000 second. To achieve above 90% hit rate, the cache age needs to be longer than six days. Beyond six days, the hit rate becomes flat, indicating that maintaining cache entries alive for more than six days makes little contribution to improving the hit rate.

The cache table size depicted on the right y-axis in Figure 9 grows with cache age. The growth slows down when the cache age ranges from 10-24 hours. Note that the hit rate also slows down during the same range. We believe that this is related to the characteristic of the data that most calls are made during work hours. Thus, changing cache age from 10 to 24 hours has very small effect on hit rate and cache table size. Note that to achieve a 90% hit rate, we need a cache table size of 700 entries. Since each entry consumes less than 100 bytes (4 B for timestamp, 64 B for request URI, 4 B for IP address and 1 B for result), we need less than 70 KB memory to maintain a 700-entry cache table.

This experiment demonstrates that caller locality does exist in today's enterprise call patterns and that caching fingerprinting results for about one week can reduce probing by as much as 90%.

7. RELATED WORK

Next, we review related work on SPIT prevention. Rebahi et al. [10] proposed a reputation-based solution that fights spam by building social networks within SIP communities Dantu et al. [6] developed a SPIT classification technique using a combination of "black/white lists, trust and reputation functions and media quarantining"[6]. Shin et al. [14] used an algorithm called Progressive Multi Gray- Leveling to learn the calling patterns of VoIP spammers and block unwanted calls. Our proposed fingerprinting techniques can be incorporated into those SPIT prevention systems to provide more information for classifying VoIP calls.

There are several fingerprinting tools that remotely identify operating systems. *nmap* [8] and *p0f* [7] detect the operating system and its version number by exploiting the variations in different vendor's implementations of the TCP/IP stacks. While those works focus on OS identification by OS layer protocol analysis, we classify SIP applications for traffic filtering purpose. Note that because SIP is an application layer protocol, there are many more implemented versions of SIP than TCP stacks. OS level fingerprinting can be complementary to our application layer fingerprinting. For example, some SIP applications only run on specific platforms and knowing the underlying operating systems helps to confirm the identities of the applications.

The SIP IETF working group has recently published an informational draft [17] of the SIP torture test for the purpose of improving interoperability. The long test message list confirms our observation that existing SIP applications behave differently upon a variety of SIP requests. While efforts are made to eliminate incorrect software behaviors under unusual protocol requests, we argue that due to the diversity of software implementations we can always find fingerprints of applications because there are many ways in which they can be implemented correctly but differently.

8. SUMMARY

In this paper, we present an implementation of a SPIT firewall that leverages fingerprints of remote VoIP applications. We show that the firewall can be tuned to provide reasonable performance and can be deployed in either personal computing or corporate environments. We also fingerprinted twenty different devices including infrastructure components such as hardware-based voice gateways and SIP proxies, and a multitude of hardphones and softphones using our SIP fingerprinting techniques. We find that each device has a unique fingerprint and can be uniquely identified using a small number of probes.

9. REFERENCES

[1] VOIP PBX and Servers. http://www.voip-info.org/wiki-VOIP+PBX+and+Servers.

[2] VOIP Phones. http://www.voip-info.org/wiki/view/VOIP+Phones.

[3] M. Arango, A. Dugan, I. Elliott, C. Huitema, and S. Pickett. *Media Gateway Control Protocol (MGCP) Version 1.0*, Oct. 1999. RFC 2705.

[4] I. E. Consortium. H.323. Web proforum tutorial.

[5] CounterPath. http://www.counterpath.com.

[6] R. Dantu and P. Kolan. Detecting Spam in VoIP Networks. In *Proceedings of USENIX Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI)*, July 2005.

[7] M. A. Davis, K. Kuehl, and K. Currie. p0f. http://lcamtuf.coredump.cx/p0f.shtml, Sept. 2004.

[8] Fyodor. Nmap Version Scanning. http://www.insecure.org/nmap/, Sept. 2003.

[9] PlanetLab. http://www.planet-lab.org.

[10] Y. Rebahi and D. Sisalem. SIP Service Providers and the Spam Problem. In *2nd Workshop on Securing Voice over IP*, June 2005.

[11] J. Rosenberg, C. Jennings, and J. Peterson. The Session Initiation Protocol (SIP) and Spam. Internet-Draft (Work in Progress), Mar. 2006.

[12] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002.

[13] H. Schulzrinne. Listing of Public SIP Servers. http://www.cs.columbia.edu/sip/servers.html.

[14] D. Shin and C. Shim. Voice Spam Control with Gray Leveling. In *2nd Workshop on Securing Voice over IP*, June 2005.

[15] SIPfoundry. http://www.sipfoundry.org/.

[16] Skype. http://www.skype.com.

[17] R. J. Sparks, A. Hawrylyshen, A. Johnston, J. Rosenberg, and H. Schulzrinne. Session Initiation Protocol Torture Test Messages, Nov. 2005.

[18] Vonage. http://www.vonage.com.

[19] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[20] N. Willing. VOIP Subscriber Numbers Soar. http://www.lightreading.com, July 2005.

[21] H. Yan, K. Sripanidkulchai, H. Zhang, Z. Shae, and D. Saha. Information Leak Vulnerabilities in SIP Implementations. *IEEE Network Magazine Special Issue on Securing Voice over IP*, 3Q 2006.