The Parrot is Dead: Observing Unobservable Network Communications

Amir Houmansadr Chad Brubaker Vitaly Shmatikov
The University of Texas at Austin

***Abstract*—In response to the growing popularity of Tor and other censorship circumvention systems, censors in non-democratic countries have increased their technical capabilities and can now recognize and block network traffic generated by these systems on a nationwide scale. New censorship-resistant communication systems such as SkypeMorph, StegoTorus, and CensorSpoofer aim to evade censors' observations by imitating common protocols like Skype and HTTP.**

We demonstrate that these systems completely fail to achieve unobservability. Even a very weak, local censor can easily distinguish their traffic from the imitated protocols. We show dozens of passive and active methods that recognize even a single imitated session, without any need to correlate multiple network flows or perform sophisticated traffic analysis.

We enumerate the requirements that a censorship-resistant system must satisfy to successfully mimic another protocol and conclude that "unobservability by imitation" is a fundamentally flawed approach. We then present our recommendations for the design of unobservable communication systems.

***Keywords*-Censorship circumvention; unobservable communications; Tor pluggable transports**

I. Introduction

Censorship-resistant communication systems such as the Tor anonymity network[1] are increasingly used by people in non-democratic countries to bypass restrictions on Internet access, share information, browse websites prohibited by the regime, etc. In response, government censors have greatly improved their technical capabilities and are now able to perform real-time deep-packet inspection and traffic analysis on ISP-level volumes of network traffic (see Section IV-C).

This increase in censors' power threatens to make anonymous communication systems unavailable to users who need them the most. Tor, in particular, has faced frequent blocking even after deploying private "bridges" [17] that hide the addresses of Tor relays in order to circumvent IP address filtering. The problem is that Tor traffic remains recognizable by its characteristic patterns and content signatures.

The continuing availability of low-latency, censorship-resistant communications thus critically depends on their *unobservability*. This has motivated an entire class of circumvention systems that aim to achieve unobservability by imitating popular applications such as Web browsers and Skype clients. In the rest of this paper, we refer to them as *parrot circumvention systems*. For example, SkypeMorph [41] hides Tor traffic by mimicking Skype video calls, CensorSpoofer [59] mimics SIP-based Voice-over-IP, and StegoTorus [60] mimics Skype and/or HTTP.

Our contributions. We present the first in-depth study of unobservability in censorship-resistant communication systems. We develop a taxonomy of adversaries and a detailed list of technical requirements that a parrot system must satisfy to successfully mimic another protocol.

We analyze the recently proposed parrot systems, including SkypeMorph, StegoTorus, CensorSpoofer and their variants, and show that they *completely fail to achieve unobservability*. We demonstrate multiple techniques to distinguish their traffic from the protocols they attempt to imitate and prove that all of these techniques work in practice.

Most of our methods assume a much weaker adversary than considered by the designers of these parrot systems. They aim to foil large-scale statistical traffic analysis by ISP-level adversaries, yet even a *single traffic flow* generated by any of their systems can be recognized at a low cost by a *local* network adversary (e.g, a censor in control of a Wi-Fi access point or local router) because of the glaring discrepancies between their crude imitations and the behavior of genuine protocol implementations.

To give just one example, SkypeMorph and StegoTorus take great care to generate datagrams whose size distributions mimic a Skype video chat session, yet forget to mimic the TCP control channel that always accompanies a genuine Skype session. These imitation mistakes are numerous and in many cases unfixable. Even plausible-looking fixes (e.g., "add an imitated TCP channel") do not help in practice because they do not correctly mimic the complex, dynamic dependences exhibited by the genuine protocols.

We argue that the entire approach of **"unobservability by imitation" is fundamentally flawed**. Convincingly mimicking a sophisticated distributed system like Skype, with multiple, inter-dependent sub-protocols and correlations, is an insurmountable challenge. To win, the censor needs only to find a few discrepancies, while the parrot must satisfy a daunting list of imitation requirements. Furthermore, it is not enough to mimic some protocol; the parrot must plausibly mimic a *specific implementation* of the protocol down to every quirk and implementation-specific bug. For example,

[1] https://www.torproject.org/

StegoTorus's imitation of an HTTP server does not look like any known Web server and is thus easily recognizable.

We conclude with the lessons and recommendations for designing unobservable communication systems. A promising alternative to parrots is offered by systems that operate higher in the protocol stack [28, 29]: for example, instead of imitating Skype, they run genuine Skype and transport their traffic in the encrypted voice or video payloads.

II. Unobservability by imitation

Intuitively, unobservability means that a censor can neither recognize the traffic generated by the circumvention system, nor identify the endpoints engaged in circumvention. *Parrot circumvention systems* aim to achieve unobservability by mimicking a widely used, uncensored *target* protocol. Popular imitation targets include HTTP, Skype, and IETF-based VoIP. It is essential that the target be a common protocol which the censor may be unwilling to block for political or economic reasons. Imitating an unpopular protocol is futile because the censor will simply block both the genuine protocol and its imitations.

Skype. Skype is a very popular Voice-over-IP (VoIP) system based on a P2P overlay network of users running Skype software. Skype's proprietary design has been extensively studied and reverse-engineered (see Appendix A).

A Skype *client* is an ordinary user who makes calls and sends messages. Users are authenticated by a central *login server*. A Skype *supernode* is a resource-rich user with a public IP address and sufficient CPU, memory, and network bandwidth [1, 5]. Supernodes relay media and signals between clients that cannot communicate directly due to network address translation (NAT) and firewalls.

IETF-based VoIP. IETF has several standards for VoIP protocols, including *network discovery* to connect to the VoIP network, *session control* to set up and tear down calls, and *media transmission* to communicate voice datagrams.

Session Initiation Protocol (SIP) [51] is a popular session control protocol. SIP is an application-layer protocol and can run over TCP or UDP. A SIP system comprises *user agents*, *location services*, *registrar servers*, and *proxy servers*. User agents have registered SIP IDs and run SIP client software. A location service is the VoIP provider's database listing users, their SIP IDs and network locations, etc. Registrar servers are network machines operated by the VoIP provider that receive SIP registration requests from user agents and update their information in the location service. Proxy servers receive call requests from user agents and forward them either to the requested callees, or to other SIP proxies.

Once a VoIP session is established between two SIP user agents, they use a media transmission protocol to communicate the call traffic, e.g., audio data. Real-time Transport Protocol (RTP) [53] is an IETF standard for media transmission. Real-time Transport Control Protocol (RTCP) is a sister protocol that controls an established RTP connection by exchanging out-of-band statistics and control information. Both RTP and RTCP run over UDP and have encrypted versions, called SRTP and SRTCP, respectively. If SRTP/SRTCP is used, an additional protocol is needed to establish a shared key, e.g., MIKEY [3], or else user agents may use pre-established keys.

Session Traversal Utilities for NAT (STUN) [50] is a set of methods that allows a VoIP client behind NAT to discover this fact and connect to a VoIP network.

III. Parrot circumvention systems

A. SkypeMorph

SkypeMorph [41] is a pluggable transport [46] for Tor intended to make the traffic between a Tor client and a Tor bridge [17] look like a Skype video call.

The client obtains the bridge's Skype ID in advance, e.g., through Tor's BridgeDB [13]. The bridge logs into Skype and picks a high UDP port. The client logs into Skype, picks a high UDP port, and waits until the bridge's ID is online, then sends a Skype text message to the bridge containing the client's IP address, UDP port, and public key. The bridge replies with a Skype message containing its own IP address, UDP port, and public key. The exchanged public keys are used to derive a shared secret key.

The client simulates the start of a video call by sending a Skype ring signal to the server and then dropping the call. The bridge does not "answer" this call. Instead, it listens on its own UDP port for incoming SkypeMorph messages and responds to the client's UDP port. These messages are encrypted with the shared secret key. Once the encrypted UDP connection starts, both the client and the bridge terminate their Skype runtime.

B. StegoTorus

StegoTorus [59] is a pluggable Tor transport derived from Obfsproxy [44]. It adds *chopping* and *steganography* to Tor clients and bridges. The chopper aims to foil statistical analysis by changing packet sizes and timings. It carries Tor traffic over *links* comprised of multiple connections. Each connection is a sequence of blocks, padded and delivered out of order. The steganography module aims to hide traffic contents by mimicking HTTP, Skype, and Ventrilo.

Embed steganography. StegoTorus-Embed aims to mimic a P2P connection such as Skype or Ventrilo VoIP. The StegoTorus prototype uses a database of genuine, previously collected Skype and Ventrilo packet traces to shape its traffic, but users can supply their own traces. To ensure that traffic patterns match, a StegoTorus client sends packet timings and sizes to the StegoTorus server at the beginning of the connection. In addition, StegoTorus emulates application headers to match the traffic payload. The current prototype generates application headers "by hand" because neither Skype, nor Ventrilo are open-source.

HTTP steganography. StegoTorus-HTTP aims to mimic unencrypted HTTP traffic by using a client-side *request generator* and a server-side *response generator*. Both rely on a pre-recorded trace of HTTP requests and responses. Unlike StegoTorus-Embed, clients and servers use independent HTTP traces; neither trace is temporally arranged.

The request generator picks a random HTTP GET request from the trace and hides the payload produced by the chopper in the `<uri>` and `<cookie>` fields of the request by encoding the payload into a modified `base64` alphabet and inserting special characters at random positions to make it look like a legitimate URI or cookie header.

The response generator picks a random HTTP response consistent with the request and hides the data in the files carried by this response. The StegoTorus prototype uses PDF, SWF, and JavaScript files for this purpose.

C. CensorSpoofer

Unlike StegoTorus and SkypeMorph, which are pluggable Tor transports, CensorSpoofer [59] is a standalone system that (1) uses IP spoofing to obfuscate the server's identity, and (2) mimics VoIP traffic to obfuscate traffic patterns.

CensorSpoofer is mainly designed for censorship-resistant Web browsing, where the upstream flow (requested URLs) requires much less bandwidth than the downstream flow (potentially large HTTP responses). Therefore, CensorSpoofer decouples upstream and downstream connections. A CensorSpoofer client uses a low-capacity channel such as email or instant messaging to send requests to the CensorSpoofer server. The server hides HTTP responses by mimicking P2P traffic from an oblivious *dummy host*. The CensorSpoofer prototype focuses on mimicking UDP-based VoIP traffic, thus dummy hosts are chosen by port-scanning random IPs and picking the ones whose SIP ports are open.

A CensorSpoofer client initiates a SIP connection with the CensorSpoofer server by sending a SIP INVITE to the appropriate SIP ID. The CensorSpoofer spoofer replies with a SIP OK message spoofed to look as if its origin is the dummy host. Once the client receives this message, it starts sending encrypted RTP/RTCP packets with random content to the dummy host. At the same time, the spoofer starts sending spoofed, encrypted RTP/RTCP packets to the client ostensibly from the dummy host's address.

To browse a URL, the client sends it through the upstream channel. The spoofer fetches the contents and embeds them in the spoofed RTP packets to the client. To terminate, the client sends a termination signal upstream. The spoofer replies with a spoofed SIP BYE message, the client sends a SIP OK message and closes the call.

IV. Adversary models

A. Capability classification

Passive attacks involve observing and analyzing network traffic and the behavior of Internet entities. Typical techniques are statistical traffic analysis, deep-packet inspection, and behavioral analysis.

Active attacks involve manipulation of network traffic. Typical techniques are delaying, dropping, or injecting packets into existing connections, modifying packet contents, throttling bandwidth, and terminating connections.

Proactive attacks aim to identify network entities involved in circumvention by sending probes that are crafted to elicit recognizable responses. For example, a censor may try to discover Tor bridges by initiating connections to random or suspected IP addresses [40]. By contrast, active attacks perturb already existing connections.

B. Knowledge classification

Local adversary (LO) controls at most a few network devices and can only observe a small number of connections. Examples include compromised home routers or Wi-Fi access points, corporations monitoring employees, etc.

By contrast, a state-level adversary observes large volumes of network traffic. Examples include malicious ISPs and government censors. We further subdivide state-level adversaries into two categories based on their resources.

State-level oblivious adversary (OB) has limited computational and/or storage resources. He can neither keep network traces for a long time, nor perform heavyweight traffic analysis. An OB censor may possess capabilities like deep-packet inspection (DPI), but can only apply them at close to line speeds to short observations of network traffic: for example, to individual packets but not across packets.

State-level omniscient adversary (OM) has ample processing and storage resources. He can aggregate data collected at different network locations and store all intercepted traffic for offline, computationally expensive analysis.

C. Real-world censors

Repressive states like China, Iran, Cuba, Syria, and North Korea have deployed the most aggressive Internet censorship [16, 31, 38, 49], but censorship is practiced even by Australia [4] and Italy [33], as well as enterprise networks and search engines [35]. Some government censors are "passive OB" in our classification, but the number of active and proactive OM censors is growing [31].

The "Great Firewall of China" employs both active and proactive censorship. Chinese censors proactively scan for Tor bridges [61], even resorting to IP spoofing on occasion [63]. In 2011, they were able to identify new Tor bridges in less than 10 minutes [12] by actively probing SSL traffic [56, Slide 41]. Once a Chinese Tor user connects to a bridge for the first time, several probes requesting connection are sent from different IP addresses inside China to verify that this is indeed a bridge [63]. Chinese censors actively enumerated all bridges offered on Tor's website through human interaction over several weeks [55, 56]. They also enumerated and blocked all bridge IP addresses provided via Gmail, leaving Tor with only the social network distribution strategy and private bridges [56, Slide 24].

Iranian censors perform sophisticated deep-packet inspection. In 2011, they managed to detect and block all Tor traffic for several weeks by noticing the difference between the Diffie-Hellman moduli in "genuine" SSL and Tor's SSL. Later, they used the lifetime of Tor's SSL certificates to recognize Tor traffic [56, Slide 38]. Furthermore, Iran repeatedly blocks all encrypted traffic [32].

Censors can even unplug an entire country from the Internet, as in Egypt and Libya [56, Slides 29 and 31].

D. Adversary models of parrot circumvention systems

To infer the adversary models of the existing parrot circumvention systems, we use the statements made in the papers that describe their respective designs.

SkypeMorph. SkypeMorph acknowledges "probes performed by hosts located in China, aimed quite directly at Tor bridges" [41, §1, ¶4] and claims unobservability against "a state-level ISP or authority," able to "capture, block or alter the user's communications based on pre-defined rules and heuristics" [41, §3, ¶1]. We infer that the SkypeMorph censor can perform passive, active, and proactive attacks.

SkypeMorph also claims unobservability against powerful censors who can perform statistical analysis and deep-packet inspection. For example, the designers state that "the censorship arms race is shifting toward the characteristics of connections" [41, §1, ¶5], acknowledging the feasibility of resource-intensive analysis. They also consider "behavioral heuristics" to "detect a user's attempt to circumvent censorship" [41, §3, ¶2], including detection of proxy connections by port knocking: "a TCP SYN packet following a UDP packet to the same host" [41, §3]. We infer that the SkypeMorph censor is OM in our classification.

SkypeMorph assumes that the censor's activities are limited so as not to interfere with the normal use of the Internet by "benign" users (similar to CensorSpoofer), and that the censor does not have prior information about the IP addresses and Skype IDs of SkypeMorph servers.

StegoTorus. Censors can perform IP, content, and statistical filtering but only "in real time on a tremendous traffic volume" [60, §2.2.2, ¶1]. StegoTorus "is not expected to resist sophisticated, targeted attacks that might be launched by a nation-state adversary." The StegoTorus censor is thus OB in our classification.

StegoTorus considers their threat model to be "similar to previous research like Telex" [60, §2.2, ¶1]. The Telex censor can perform passive, active, and proactive attacks, although the following statement implies that the StegoTorus censor is *not* the Telex censor: "...potential application-level attack that involves serving malicious content and then observing a distinctive traffic pattern; although relevant, we are more interested in passive attacks that could be carried out on a large scale" [60, §7].

CensorSpoofer. CensorSpoofer considers a "state-level adversary" who has "sophisticated censorship capabilities of IP filtering, deep packet inspection, and DNS hijacking, and can potentially monitor, block, alter, and inject traffic anywhere within or on the border of its network," [59, §3.1, ¶1] "can rent hosts outside of its own network, but otherwise has no power to monitor or control traffic outside its borders," and "has sufficient resources to launch successful insider attacks, and thus is aware of the same details of the circumvention system as are known to ordinary users" [59, §3.1, ¶3]. We infer that the CensorSpoofer censor is OM and capable of passive, active, and proactive attacks.

V. Requirements for parrot circumvention

Parrot circumvention systems aim to make their communications indistinguishable from another protocol. This requires mimicking *every observable aspect* of the target protocol. Not every requirement applies to a given circumvention system, and the ability to detect discrepancies between the parrot and the genuine article may vary from censor to censor. Nevertheless, in Sections VII, VIII, and IX, we demonstrate that all recently proposed parrot circumvention systems fail so many requirements that their sessions are recognizable at a low cost even by a very weak censor.

A. Mimicking the protocol in its entirety

Correct. The most basic requirement is to mimic the target protocol correctly. The parrot's observable behavior must be *consistent with the protocol specification*.

SideProtocols. Many modern network protocols include multiple "side" protocols and control channels that run alongside the main session. For example, a typical VoIP session involves three protocols: SIP for signaling the session, RTP for streaming the media, and RTCP for controlling the media stream. Another example is the STUN traffic generated by VoIP clients residing behind a firewall.

The parrot must mimic *all channels and side protocols* of its target. For example, even a perfect imitation of an RTP flow is trivial to recognize if, unlike genuine RTP flows, it is not accompanied by a concurrent RTCP connection.

IntraDepend. Multiple connections comprising a single protocol session have complex dependences and correlations. In particular, changes in the main channel often cause observable activity in the control channel and vice versa.

For example, a typical VoIP session starts with an exchange of characteristic messages between the caller and a SIP server, followed by the initialization of RTP and RTCP connections between the caller and the callee. The SIP connection is kept alive while the RTP/RTCP connections are active. The session ends with characteristic SIP messages.

Dropping RTP packets may cause distinct RTCP activity as the encoding of the media stream is being adjusted.

The parrot must faithfully mimic *all dynamic dependences and correlations* between sub-protocols.

InterDepend. A session of a given protocol may trigger other protocols. For example, an HTTP request often triggers multiple DNS queries.

The parrot must (1) trigger other protocols whenever the target protocol would have triggered them, and (2) mimic the target protocol's response when triggered by other protocols.

B. Mimicking reaction to errors and network conditions

Err. One of the easiest ways to tell a parrot from a genuine protocol implementation is to observe its reaction to errors, whether natural (e.g., caused by a buggy endpoint) or artificial. Errors include malformed packets, invalid requests, unwanted traffic (e.g., packets from other sessions), etc.

The protocol standard may prescribe how certain errors should be handled, but error handling is often underspecified and left to the discretion of the implementation. Differences in error handling can thus be used to fingerprint implementations of common network protocols such as HTTP [30].

Error handling is extremely difficult to mimic and most parrots fail to do it properly or at all. First, the parrot must produce at least some reaction to *any possible error* that might occur in the target protocol (because any genuine implementation would react in some way). The second requirement is even more challenging. The parrot's reactions to all possible errors must be consistent: they should look as if they were generated by a *particular* genuine implementation. For example, a parrot Web server cannot react to some erroneous requests as if it were a Microsoft IIS and to others as if it were an Apache server.

Network. The Internet is a noisy medium, and network flows may experience packet drops and reorders, repacketization, high latencies caused by dynamic changes in the throughput of certain links, etc. Some protocols prescribe standard reactions to changes in network conditions: for example, TCP uses sequence numbers and a congestion control mechanism, while live-video environments have multiple patented automatic repeat request (ARQ) mechanisms.[2] Streaming media protocols in particular react in very distinct ways to congestion and other network issues. In general, packet losses and congestion cause media applications to lower codec quality and/or adjust transmission rates.

The parrot must mimic the target protocol's responses to *all possible changes in network conditions*, whether natural or artificially induced. Furthermore, if a side protocol is used—for example, to signal codec renegotiation—it must be mimicked, too (see the **InterDep** requirement).

C. Mimicking typical traffic

Content. Many network protocols have specific formats for headers and payloads, all of which must be mimicked by the parrot. For example, HTTP headers contain information about the payload, while port numbers in IP headers reflect higher-level protocols. Encryption does not conceal all of this information. For example, headers of encrypted Skype packets reveal their type and other information [10].

Message payloads generated by the parrot must be indistinguishable, too. In particular, imitated files must be *metadata-compatible* with the genuine files. For example, imitated PDF files must contain correct xref tables and other metadata typically found in real PDF files.

Patterns. Many protocols produce characteristic patterns of packet sizes, counts, inter-packet intervals, and flow rates. These patterns are often stable across the network, observable even when packet contents are encrypted, and can be exploited for traffic analysis [42].

The parrot must produce network flows all of whose *observable characteristics, including packet sizes and timings*, are indistinguishable from the genuine protocol.

Users. User behavior often produces recognizable patterns at the network level. For example, a typical Skype user only makes a few Skype calls at a time. A parrot making hundreds of concurrent Skype calls thus appears very anomalous and can be easily distinguished from a genuine Skype client. Similarly, a typical email user only sends and receives a certain number of messages per day [48]. Users can be fingerprinted based on the frequency of their system usage, number of connection peers, typical volume of traffic associated with each use, etc.

The parrot must faithfully mimic *typical user behavior*.

Geo. Observable behavior of protocol endpoints—including their routing decisions, chosen peers, and even traffic contents—may depend on their geographic location. For example, a Web server may respond differently to the same request depending on its origin; network packets sent by a remote peer enter a given ISP at different points depending on the peer's location; SIP-based VoIP clients always connect to the geographically closest SIP server, etc.

Some implementations of common protocols are country-specific. For example, Skype users in mainland China use a special implementation of Skype called TOM-Skype which has built-in surveillance functionality [34]. Any parrot that mimics a different Skype client is likely to stand out.

The parrot must mimic all *geography-specific aspects* of the genuine protocol and its local implementations.

D. Mimicking implementation-specific artifacts

Soft. A protocol specification can be realized by multiple implementations. For example, there are dozens of Web browsers and Web servers. For inter-operability, each implementation generally complies with its (often idiosyncratic)

[2] http://www.techex.co.uk/other/arq-video-packet-resend

interpretation of the standard, but often with characteristic quirks and tell-tale signs. Sometimes these are explicit—for example, HTTP request headers include information about the browser—but even unintentional discrepancies can be used to fingerprint implementations and different versions of the same implementation [30]. For example, different versions of Apache Web server contain different bugs, which can be triggered by a remote user to identify the version.

It is not enough to mimic or implement the protocol specification. The parrot must mimic a *specific version of a specific popular implementation*, down to every last bug, whether known or unknown! *Any* deviation can be used to distinguish the parrot from the known implementations.

OS. Network protocols are usually designed to be oblivious to the endpoint's operating system (OS), yet the latter can often be revealed by the recognizable characteristics of specific client and server software. For instance, the IETF standard requires that the initial sequence number of a TCP connection be randomly generated. Different OSes, however, use different sequence number generation algorithms, enabling OS identification [43]. This information is also explicitly included in HTTP headers.

The parrot must generate *consistent OS fingerprints*. In particular, when mimicking a network service, OS fingerprints should not change frequently because servers' OSes do not change frequently.

VI. Experimental setup

We obtained the latest implementations of all analyzed parrot systems and their imitation targets (Skype, Ekiga, etc.) from their respective websites and/or authors.

For all Skype and CensorSpoofer experiments, we executed the software in VirtualBox[3] virtual machines (VMs), running on a Funtoo Linux machine with an Intel i5 CPU and 4GB of RAM. Skype clients were executed in two Windows 7 VMs; SkypeMorph and StegoTorus-Embed in separate Ubuntu 12.10 VMs. The VMs were connected through Virtual Distributed Ethernet (VDE) [58], which provides tools for network perturbation. We developed our own plugins for VDE that allow us to drop packets at different rates and modify packet contents on the wire. Each VM is connected to a separate virtual VDE switch, and the switches are connected to a central switch, which provides DHCP connectivity to the Internet.

Experiments with StegoTorus clients and servers in Section VIII were executed on two physical Ubuntu 12.04 machines, using the statistics module of iptables[4] to drop packets at different rates. Our StegoTorus server uses a real Tor bridge to connect to the Tor network. VoIP clients in Section IX were analyzed on a Windows 7 VM, Ubuntu 12.04 VM, and Mac OS X 10.7. The SIP probing test was implemented in Python and performed over a non-firewalled Ubuntu 12.04 server with a public IP address.

VII. Detecting Skype imitators

We demonstrate that parrot circumvention systems that aim to imitate Skype—in particular, SkypeMorph and StegoTorus-Embed[5]—can be easily distinguished from genuine Skype and thus fail to achieve unobservability.

First, we show that their imitation of Skype is incomplete and can thus be recognized even by low-cost, passive attacks. Next, we describe hypothetical improved versions of SkypeMorph and StegoTorus, designed specifically to imitate Skype behaviors that are missing in their current prototypes. We then demonstrate that even these hypothetical improvements can be easily distinguished from genuine Skype by active and proactive attacks.

A. Passive attacks

We present two classes of passive attacks. The first uses the Skype detection tests from Appendix A-B to recognize partial imitations. The second exploits the fact that both SkypeMorph and StegoTorus-Embed rely on recorded Skype traces to mimic packet timings and sizes. All attacks have been empirically confirmed by (1) executing SkypeMorph and StegoTorus prototypes and (2) analyzing their code.

Exploiting deviations from genuine Skype behavior. Skype identification tests (see Appendix A-B) are used by ISPs and enterprise networks and can be easily performed even by a passive, resource-constrained censor. To successfully mimic Skype, a parrot system must pass all or at least the majority of these tests. Unfortunately, Table I demonstrates that both SkypeMorph and StegoTorus fail.

This indicates serious design flaws in both systems. They claim to provide unobservability against sophisticated statistical traffic analysis, yet can be distinguished from Skype even by extremely basic tests which are *less resource-intensive* and *more effective* than the hypothetical tests considered by the designers of these systems.

StegoTorus mimics Skype's traffic statistics, but fails to mimic much more visible aspects of genuine Skype such as HTTP update and login traffic. Neither SkypeMorph, nor StegoTorus mimics Skype's TCP channel, which is an essential component of every genuine Skype session. Furthermore, neither system generates SoM packet headers (see Appendix A-B), which are present in every genuine Skype UDP packet. These tests are (1) passive and (2) can be performed at line speeds, thus **SkypeMorph and StegoTorus fail even against the weakest censor**.

A censor can combine the tests listed in Table I into a hierarchical detection tool. In fact, similar tools have been proposed for real-time detection of Skype traffic [9, 23],

[3] https://www.virtualbox.org/

[4] http://www.netfilter.org/

[5] StegoTorus-Embed also aims to mimic Ventrilo, but we do not consider it in this paper because Ventrilo is not as popular as Skype, and in any case the latest StegoTorus prototype does not fully implement Ventrilo.

Table I
PASSIVE ATTACKS TO DETECT SKYPE PARROTS.

Attack	Imitation requirement	Adversary	SkypeMorph	StegoTorus-Embed
Skype HTTP update traffic (T1)	`SideProtocols`	LO/OB/OM	Satisfied	Failed
Skype login traffic (T2)	`SideProtocols`	LO/OB/OM	Satisfied	Failed
SoM field of Skype UDP packets (T3)	`Content`	LO/OB/OM	Failed	Failed
Traffic statistics (T4, T5)	`Pattern`	LO/OM	Satisfied	Satisfied
Periodic message exchanges (T6, T7)	`SideProtocols`	LO/OB/OM	Failed	Failed
Typical Skype client behavior (T8)	`IntraDepend`	LO/OM	Failed	Failed
TCP control channel (T9)	`SideProtocols`	LO/OB/OM	Failed	Failed

including line-rate detectors by Patacek [47], who used these tests in an NfSen[6] plugin, and by Adami et al. [1]. These tools can be adapted to detect Skype parrots that pass a non-trivial fraction of the tests, but not all of them.

Exploiting re-use of pre-recorded Skype traces. Both StegoTorus and SkypeMorph clients come with pre-recorded traffic traces, which are used to mimic Skype by sending packets with the exact same timings and sizes. Because the censor also has access to the client software, he can match observed flows against these patterns and exploit the fact that genuine Skype traffic is unlikely to match them exactly, while imitated traffic always will.

Such censor must be OM because he needs to allocate resources to match every observed flow against the known trace. This passive attack succeeds because SkypeMorph and StegoTorus fail the `Patterns` requirement.

Exploiting re-use of client-generated Skype traces. To foil the above attack, both StegoTorus and SkypeMorph suggest that a client may generate its own Skype traces and mimic those. This re-use can be detected by a long-term OM censor since multiple genuine flows from the same client are unlikely to ever repeat the exact pattern of timings and sizes.

This passive attack succeeds because SkypeMorph and StegoTorus fail the `Patterns` and `User` requirements.

B. Hypothetical SkypeMorph+ and StegoTorus+

Imagine hypothetical systems called SkypeMorph+ and StegoTorus+ that add the patterns and messages from Table I which are missing from, respectively, SkypeMorph and StegoTorus. StegoTorus+ adds an imitated Skype login (similar to the current SkypeMorph prototype). Both SkypeMorph+ and StegoTorus+ add the missing messages from Appendix A-B and put appropriate SoM fields into imitated Skype packets. To mimic Skype's TCP channel—which is a dead giveaway that the current prototypes of SkypeMorph and StegoTorus are not actually running Skype—they add a fake TCP connection to each Skype call using the TCP port with the same number as the corresponding UDP connection, and send regular "garbage" traffic on this connection to mimic Skype's control traffic.

To foil detection based on trace re-use, StegoTorus and SkypeMorph use a Skype pattern generator instead of pre-recorded or pre-generated traces. This generator produces Skype-like packet timings and sizes on the fly, thus the resulting patterns are unique to each imitated connection. For the sake of the argument, even imagine that this generator cannot be recognized by tools that discover covert communications based on fabricated patterns [24].

C. Active and proactive attacks

Unfortunately, even SkypeMorph+ and StegoTorus+ would not achieve unobservability because they would suffer from the same fundamental flaw as SkypeMorph and StegoTorus: they do not actually run Skype, they only try to mimic it, futilely. Table II summarizes active and proactive attacks that can distinguish a Skype parrot from genuine Skype.

Verifying supernode behavior.
Requirements: `SideProtocols`, `IntraDepend`
Adversary: Proactive, LO/OM
Skype supernodes (SN) relay media traffic and signaling information for ordinary Skype clients [6]. In particular, ordinary clients use nearby supernodes to connect to the Skype network. The following two-stage attack enables a censor to distinguish SkypeMorph+ and StegoTorus+ servers from genuine Skype supernodes.

Phase 1: Supernode identification. We give two ways to find out if a given node is (or pretends to be) a Skype supernode. If it receives Skype calls from nodes behind NAT in the censor's network, then it must be a supernode because ordinary Skype nodes cannot receive calls directly (this supernode is either the callee, or relaying the call for an ordinary node). Second, the censor can use the existing tools for checking whether an IP address is performing NAT [54]. A Skype node that is not behind NAT is a supernode.

This phase filters out all genuine, ordinary Skype nodes, leaving genuine supernodes as well as SkypeMorph+ and StegoTorus+ parrots.

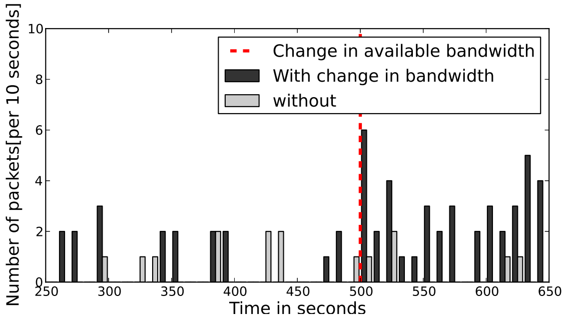
Phase 2: Supernode verification. Consider a target node that looks like a Skype supernode after Phase-1 filtering. The censor can run an ordinary Skype client and flush its *supernode cache*, which is the list of the supernodes discovered by that client, to force the client to use the target node as its supernode [5, 25]. If the target is a genuine supernode, the client will be able to connect to the Skype network and make calls. If the target is a parrot, the

[6] http://nfsen.sourceforge.net/

Table II
ACTIVE AND PROACTIVE ATTACKS TO DETECT IMPROVED SKYPE PARROTS.

Attack	Imitation requirement	Adversary	Skype	SkypeMorph+ and StegoTorus+
Verify supernode behavior by flushing supernode cache	`SideProtocols` `IntraDepend`	Proactive, LO/OM	The target node serves as the adversary's SN, e.g., relays his Skype calls	Rejects all Skype messages
Drop a few UDP packets	`Network`, `Err`	Active, LO/OB/OM	A burst of TCP packets on the control channel (Fig. 1)	No reaction
Close TCP channel	`IntraDepend`, `SideProtocols`	Active, LO/OB/OM	Ends the UDP stream immediately	No reaction
Delay TCP packets	`IntraDepend`, `SideProtocols`, `Network`	Active, LO/OM	Reacts depending on the type of TCP messages	No reaction
Close TCP connection to a SN	`IntraDepend`, `SideProtocols`	Active, LO/OB/OM	Client initiates UDP probes to find other SNs	No reaction
Block the default TCP port for TCP channel	`IntraDepend` `SideProtocols`	Active, LO/OB/OM	Connects to TCP ports 80 or 443 instead	No reaction

Figure 1. Skype TCP activity with and without changes in bandwidth.



connection will fail because StegoTorus and SkypeMorph only mimic the Skype protocol but cannot actually run it.

Manipulating Skype calls.
Requirements: `Network`, `Err`, `IntraDepend`
Adversary: Active, LO/OB/OM
This attack tampers with a Skype connection by dropping, reordering, and delaying packets or modifying their contents, then observes the endpoints' reaction. These changes are fairly mild and can occur naturally, thus they do not drastically affect genuine Skype connections.

When UDP packets are dropped in a genuine Skype call, there is an immediate, very noticeable increase in the activity on the TCP control channel that accompanies the main UDP connection (see Fig. 1). We conjecture that this is caused by Skype endpoints re-negotiating connection parameters due to perceived changes in network conditions.

Neither SkypeMorph, nor StegoTorus implements the TCP control channel. Our hypothetical SkypeMorph+ and StegoTorus+ add a fake TCP connection and may even send dummy packets over it, but this is not enough. The imitated behavior will not be correlated to the network conditions in the same way as in genuine Skype, enabling the censor to distinguish parrots from Skype. We argue that it is *extremely* difficult to convincingly imitate dynamic dependences between network conditions and Skype's control traffic.

This active attack does not adversely affect normal Skype users. Dropping a few packets does not disconnect the call, but only degrades its quality for a short period of time.

Manipulating the TCP control channel.
Requirements: `IntraDepend`, `SideProtocols`, `Network`
Adversary: Active, LO/OB/OM
In the previous test, we showed that perturbing Skype's main UDP connection causes observable changes on the TCP control channel. We now show that perturbing the TCP channel causes observable changes on the UDP connection.

Close the TCP connection. Closing the TCP channel (e.g., by sending an RST packet) causes genuine Skype nodes to immediately end the call. Our hypothetical parrots do not mimic this behavior because their fake TCP channel has no relationship to the actual call. While this attack interferes with genuine Skype users to some extent, spurious call disconnections are fairly common in Skype. A censor may use this attack as a confirmation on a relatively small set of connections suspected to be imitations.

Withhold or drop selected TCP packets. The TCP connection sends packets with frequency varying from 30 to 60 seconds, or when network conditions change. Tampering with these packets causes observable changes on the genuine UDP channel, but not on the imitated one.

Trigger a supernode probe. A Skype client keeps a TCP connection with its supernode. If this connection is closed, a genuine client immediately launches a UDP probe (Appendix A-A) to search for new supernodes. A parrot doesn't.

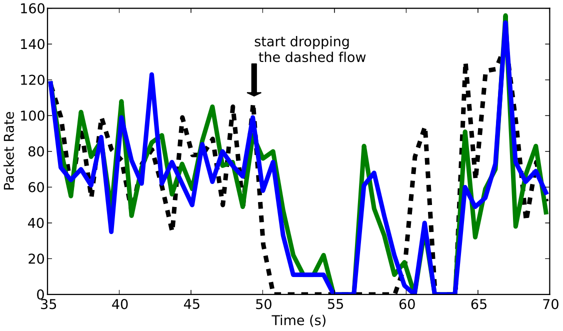
Block a supernode port. After a successful UDP probe, a genuine client establishes a TCP connection with the same port of its supernode. If this port is not available, the client tries connecting to ports 80 or 443 [5]. A parrot doesn't.

Similar attacks include tampering with SoM fields in UDP packet headers.

VIII. DETECTING STEGOTORUS

In Section VII, we showed how to detect StegoTorus-Embed's flawed imitation of Skype. In this section, we show how to detect the chopper and StegoTorus-HTTP.

Figure 2. Correlated behavior of StegoTorus connections.



A. Attacks on StegoTorus chopper

Correlating IP addresses between links.
Requirement: `Geo`
Adversary: Passive, LO/OM
A StegoTorus session, called a link, comprises several connections. This enables easy passive detection of StegoTorus clients because normal users do not keep multiple, concurrent HTTP and Skype connections to the same server.[7] A possible countermeasure is to have different links handled by geographically distributed servers, but this will impose an intolerable delay on low-latency traffic (e.g., Tor) because the servers must cooperate to reconstruct chopped packets.

Exploiting connection dependences.
Requirements: `Network`, `User`
Adversary: Passive/Active, LO/OM
Multiple connections created by the StegoTorus chopper carry packets from the same Tor session, thus their reactions to network conditions and perturbations are highly correlated. By contrast, genuine HTTP connections to different servers exhibit no such correlation. The correlations between StegoTorus connections can be observed by a passive censor or exploited for an active attack, as shown in Fig. 2: once packets on one StegoTorus connection are dropped, the other two belonging to the same link immediately slow down.

B. Passive attacks on StegoTorus-HTTP

The StegoTorus paper acknowledges several passive attacks, including (1) discrepancies between the typical patterns of GET requests and the StegoTorus imitation, and (2) abnormal changes in cookies due to the embedding of hidden payloads. The list in the paper is incomplete, illustrating how difficult it is to foresee all the ways in which an imitation may deviate from the genuine protocol.

Exploiting discrepancies in file-format semantics.
Requirement: `Content`
Adversary: Passive, LO/OB/OM
StegoTorus-HTTP embeds hidden traffic inside innocuous-looking documents that appear to have been requested by the client via an HTTP request. The StegoTorus prototype supports PDF, SWF, and JavaScript, but instead of generating documents in these formats, it uses real files and replaces specific fields with hidden content. This preserves the file's syntactic validity, but not its semantics. The StegoTorus paper claims that checking file semantics at line speeds requires a lot of resources from a state-level censor dealing with large volumes of traffic.

This claim is false. We demonstrate that it is possible to detect discrepancies between real files and StegoTorus's imitations at a very low cost and at line speed.

As a proof of concept, we show how to analyze PDF files, but similar techniques work against any other format. The fake-trace generator in the StegoTorus prototype produces templates for PDF files that miss an essential object called *xref table*. In a genuine PDF file, this table follows the `xref` keyword and declares the number and position of various objects. The absence of this table in StegoTorus's imitations is detectable via simple deep-packet inspection at line speed, without any need to reconstruct or parse the file.

Adding a fake xref table to the PDF file will not help. A simple script can verify the table's (in)validity without parsing the file by comparing the positions of PDF objects with their xref entries. StegoTorus may try to adjust xref tables to match the embedded hidden payload, but changing even a single character in a PDF file results in multiple format errors and is detectable by the most basic PDF parser. Instead of replacing binary fields, a sophisticated steganography module might craft PDF files that only show the hidden content when rendered. This would complicate both the client and the server and greatly reduce performance, making the system unusable by Tor and other low-latency transports. Furthermore, a simple script can extract the encoded text from PDF files (e.g., using the `pdftotext` Unix command) and perform linguistic verification, although this test may not be feasible at line speeds.

C. Active and proactive attacks on StegoTorus-HTTP

Fingerprinting HTTP server.
Requirements: `Correct`, `Err`, `Soft`
Adversary: Proactive, LO/OB/OM
The HTTP module in StegoTorus does not actually run an HTTP server. It responds to HTTP requests such as GET, HEAD, OPTIONS, and DELETE simply by replaying responses from its database. A censor may use an HTTP server fingerprinting tool to submit requests to the server and analyze its responses to determine whether (a) the server generates a consistent software fingerprint, and (b) the server's reaction to erroneous and invalid requests is consistent with its fingerprint. This test can be active (the censor manipulates requests sent by the clients) or proactive (the censor generates his own probe requests).

We used the *httprecon* tool [30] to send 9 types of requests: *GET existing* - a GET request for an existing

[7] While the IETF standard [21] prohibits browsers from opening more than two concurrent connections to the same server, some Web servers allow this restriction to be circumvented [11]. Concurrent connections, however, exhibit a characteristic pattern [2] not mimicked by StegoTorus.

resource like the server's front page; *GET long request* - a very long ($>$ 1024 bytes in URI) GET request for an existing resource; *GET non-existing* - a GET request for a non-existing resource, e.g., a randomly generated file name; *GET wrong protocol* - a GET request with a non-existing protocol version, e.g., HTTP/9.8; *HEAD existing* - a HEAD request for an existing resource; *OPTIONS common* - an OPTIONS request, which is used by HTTP clients to determine the options and/or requirements associated with a resource or the server's capabilities; *DELETE existing* - a request to delete an existing resource; *TEST method* - a non-existing HTTP method called TEST; *Attack request* - a GET request that tries to access a URI which includes well-known attack patterns, e.g., SQL injection and cross-site scripting.

Table III summarizes how the HTTP module of the StegoTorus server reacts to these 9 tests. In summary, StegoTorus fails the tests and can be easily distinguished from any real HTTP server. In particular, StegoTorus does not generate appropriate error responses to invalid and malformed requests.

This attack is feasible even for an OB censor who can record at line speed the IP addresses of all packets destined to port 80 and fingerprint these addresses afterwards.

Manipulating HTTP requests.
Requirements: `Err`
Adversary: Active, LO/OB/OM
A censor can tamper with a suspected StegoTorus connection and observe the server's reaction to modified, possibly malformed HTTP requests. Most of the proactive attacks listed above can be carried out in this manner. For instance, the censor can man-in-the-middle a connection and modify the client's request so that it asks for a non-existing URI. If the server returns "404 Not Found," the censor drops the error response and replays the client's original HTTP request. If the server returns "200 OK," it is a tell-tale sign that the server is *not* an HTTP server but a (poor) imitation.

IX. Detecting CensorSpoofer

SIP packets explicitly contain the name and version of the SIP client. Therefore, each CensorSpoofer connection must mimic a specific client. The CensorSpoofer prototype mimics Ekiga.[8] The attacks in this section exploit the discrepancies between CensorSpoofer and genuine Ekiga, but would apply to any other SIP client, too.

Manipulating the `tag` field.
Requirement: `Soft`
Adversary: Active, LO/OB/OM
SIP messages use random-looking tags in their headers to identify a SIP session [51]. CensorSpoofer's spoofer replaces these tags with the hash of the spoofed IP address [59, § 6.4]. If a censor manipulates the spoofed address, the hash will no longer verify and the CensorSpoofer client will close the call, similar to a genuine client's reaction to the change of callee's IP address. Unfortunately, this enables another, much cheaper attack. The censor can simply change the `tag` field containing the hash to a different, valid `tag` value. A CensorSpoofer client will terminate the call because the new `tag` is not the hash of the spoofed IP address, but a genuine SIP client will continue the call.

SIP probing.
Requirements: `SideProtocols`, `Soft`, `Err`
Adversary: Active, LO/OB/OM
The SIP connection between a client and a CensorSpoofer server is relayed through a public Ekiga registrar located outside the censoring ISP. Because the censor cannot verify the callee's IP address, the CensorSpoofer server can put a spoofed address in its SIP messages.

What the censor *can* do, however, is probe the callee by sending SIP messages to this IP address and checking whether a genuine SIP client is listening. **This is the exact attack that CensorSpoofer aimed to prevent**. To choose the IP addresses to be spoofed, the spoofer performs a random nmap scan [59, Algorithm 1] and picks any address that does not return either "closed" or "host seems down" on the SIP, RTP, and RTCP ports. As mentioned in [59], the censor cannot tell for sure whether these addresses are running a SIP client.

Unfortunately, there is an easier way for the censor to verify whether an IP address is running a SIP client. As specified in the SIP standard [51], "more than one user can be registered on a single device at the same time." Typical SIP clients thus respond to *any* SIP request that looks for *any* SIP ID, even if it is not coming from the VoIP provider's registrar. We confirmed this behavior for several popular SIP clients, including Ekiga, PhonerLite,[9] Blink,[10] and Twinkle,[11] on various operating systems.

The main functionality of a SIP registrar is to discover the current IP addresses of dynamic SIP IDs. As specified in the SIP standard, "registration is used for routing incoming SIP requests and has no role in authorizing outgoing requests." If the censor knows the current IP address of a suspected SIP client, he can directly call it instead of going through a registrar. This is the basis of SIP probing.

We describe several SIP probing tests. In our experiments, all of them were effective in distinguishing a CensorSpoofer callee from a genuine Ekiga client—see Table IV. All IP addresses in our tests satisfy the address selection algorithm of [59]. Some of the tests may produce different results depending on the type of the callee's SIP client; however, the censor can always identify the callee's client from the SIP messages and adjust the tests accordingly.

Send a SIP INVITE. The censor can call a fabricated SIP ID at the suspected IP address by sending a SIP INVITE.

[8] http://www.ekiga.net

[9] http://phonerlite.de/index_en.htm

[10] http://icanblink.com/

[11] http://www.twinklephone.com/

Table III
RESPONSES TO DIFFERENT *httprecon* REQUESTS BY STEGOTORUS SERVER AND REAL HTTP SERVERS.

HTTP request	Real HTTP server	StegoTorus's HTTP module
GET existing	Returns "200 OK" and sets `Connection` to `keep-alive`	Arbitrarily sets `Connection` to either `keep-alive` or `Close`
GET long request	Returns "404 Not Found" since URI does not exist	No response
GET non-existing	Returns "404 Not Found"	Returns "200 OK"
GET wrong protocol	Most servers produce an error message, e.g., "400 Bad Request"	Returns "200 OK"
HEAD existing	Returns the common HTTP headers	No response
OPTIONS common	Returns the supported methods in the Allow line	No response
DELETE existing	Most servers have this method not activated and produce an error message	No response
TEST method	Returns an error message, e.g., "405 Method Not Allowed" and sets `Connection=Close`	No response
Attack request	Returns an error message, e.g., "404 Not Found"	No response

Table IV
DISTINGUISHING CENSORSPOOFER FROM GENUINE SIP CLIENTS.

Attack	Imitation requirement	Adversary	Typical SIP clients (e.g., Ekiga)	CensorSpoofer
Manipulate `tag` in SIP OK	`Soft`	LO/OB/OM	Nothing	Client closes the call
SIP INVITE to `fakeID@suspiciousIP`	`SideProtocols` `Soft`, `Err`	LO/OB/OM	Respond with "100 Trying" and "180 Ringing", "483 Busy Here", "603 Decline", or "404 Not Found"	Nothing
SIP INVALID	`SideProtocols`,`Err`	LO/OB/OM	Respond "400 BadRequest"	Nothing
SIP BYE with invalid `SIP-ID`	`SideProtocols` `Soft`, `Err`	LO/OB/OM	Respond "481 Call Leg/Transaction Does Not Exist"	Nothing
Drop RTP packets (only for confirmation)	`SideProtocols` `Soft`, `Network`	LO/OB/OM	Terminate the call after a time period depending on the client, may change codec in more advanced clients.	Nothing

A genuine SIP client returns a status message, e.g., "100 Trying" and "180 Ringing", or "483 Busy Here", or "603 Decline", or "404 Not Found". CensorSpoofer returns nothing and, furthermore, cannot *ever* mimic the proper response because, by design, it does not receive the censor's INVITE.

Send an invalid SIP message. In response to any message not defined by the SIP standard, a genuine SIP client returns "400 BadRequest [Malformed Packet]". CensorSpoofer returns nothing. In contrast to the SIP INVITE probe, this test is completely transparent to genuine callees.

Send a message for a non-existing call. Each SIP call has a unique ID, which is negotiated in the call's first packet. If the censor sends a SIP message (e.g., BYE) for a random call ID, a genuine SIP client returns "481 Call Leg/Transaction Does Not Exist". CensorSpoofer returns nothing. This test, too, is transparent to genuine callees.

To prevent these SIP probing attacks, a CensorSpoofer spoofer may change its IP address selection algorithm and use similar probes to find addresses that are running genuine SIP clients. This significantly reduces the set of addresses that can be used for spoofing. The nmap-based selection algorithm of [59], which is less accurate than SIP probing, finds only 12.1% of $10,000$ random IP addresses to be suitable for spoofing. Our SIP probes to $10,000$ random addresses *did not return a single host* running IETF-based VoIP software such as Ekiga. The main reason is that proprietary VoIP services like Skype, Oovoo, and Google Voice are significantly more popular than IETF-based services.

Instead of Ekiga, CensorSpoofer may attempt to mimic a more popular proprietary service. This imitation will be easily detectable due to CensorSpoofer's use of spoofed IP addresses. Genuine clients react in a certain way to probes and manipulated messages, but CensorSpoofer cannot mimic the right reaction because it does not actually receive the probes sent to the spoofed IP address. This is a fundamental design flaw that cannot be fixed.

Manipulating upstream packets.
Requirements: `SideProtocols`, `Soft`, `Network`
Adversary: Active, LO/OB/OM

According to the standard [53, § 6], the primary function of RTCP is "to provide feedback on the quality" of RTP sessions. This feedback may be used for "control of adaptive encodings," so one might expect that changes in network bandwidth during an RTP session would result in RTCP negotiations as clients adjust their VoIP codec. Nevertheless, none of the tested VoIP clients, including Ekiga, Blink, PhonerLite, and Twinkle, appear to react when RTP and RTCP packets are dropped at various rates. Only dropping all RTP packets for 10 seconds to 2 minutes, depending on the client, results in the client terminating the call.

This allows easy detection of imitated sessions. Dropping all RTP packets will cause a genuine RTP session to close, but a CensorSpoofer session will not react. This attack is acknowledged in [59], but described as expensive because it interrupts genuine sessions. Note, however, that the censor can use it only for confirmation, e.g., for calls that failed SIP probing tests. If a more advanced implementation of RTP/RTCP adjusts codecs according to the network conditions, this behavior must be imitated, too.

X. RELATED WORK

Pfitzmann and Hansen [45] proposed definitions for privacy-related concepts including unobservability. Unobservability has been interpreted as anonymity or plausible deniability in various systems [8, 37], none of which hide the fact that a given user is participating in the system. We do not consider such systems in this paper because they are easily blockable and thus not censorship-resistant.

Several proposals for unobservable systems assume that the participants share some secret not known to the censors. An Infranet [20] client sends a special sequence of HTTP requests to a friendly Web server who decodes the requested URL and steganographically hides its content inside images returned to the client. The identities of such servers must be hidden from the censors. In Collage [14], a client and a server secretly agree on websites with user-generated content, e.g., flickr.com, and use steganography to communicate through these sites. To achieve sender unobservability, Nonesuch [26] steganographically hides data inside messages submitted to public Usenet newsgroups and dispatches them through a cascade of mixes that probabilistically detect and remove cover traffic until the hidden message reaches its intended recipient. None of these systems support low-latency communications like Web browsing.

Pluggable Tor transports. To evade IP address filtering, many circumvention technologies rely on proxies such as UltraSurf [57] and Tor bridges [17]. They face the problem of distributing the proxies' addresses to legitimate clients while hiding them from the censors [39, 40].

With the emergence of advanced censorship techniques like deep-packet inspection and active probing [63], hiding the proxy's address is not enough. A circumvention system must also disguise is traffic contents and patterns. Tor recently adopted pluggable transports [46] that aim to remove all content and pattern signatures characteristic of Tor.

Obfsproxy [44] is the first pluggable Tor transport. It tries to remove Tor-related content identifiers, but preserves the patterns such as inter-packet times and packet sizes. Therefore, it fails to achieve unobservability against today's censors [18]. Furthermore, Obfsproxy does not make Tor traffic look like another, "benign" protocol. This is the main motivation for the recently proposed pluggable transports that try to mimic Skype and/or HTTP (see Section III).

To evade proxy blocking, Flashproxy [22] proxies the traffic between a Tor client and a Tor bridge through short-term, frequently changing proxies provided by Internet users who visit volunteer websites helping Flashproxy. Flashproxy does not attempt to mimic another protocol and our initial analysis shows that it fails several requirements from Section V. It fails `Users`: a Flashproxy client receives consecutive *incoming* connections from geographically distributed IP addresses and the lengths of these connections are similar to typical Web browsing sessions. It also fails `Content` and `Pattern` because it does not completely remove the characteristic content and statistical patterns of Tor traffic.

Other pluggable Tor transports include Dust [62], which defines a new format for packet-level (as opposed to connection-level) obfuscation, and FTE [19], a system for imitating arbitrary packet formats. As we show in this paper, packet-level imitation is insufficient for unobservability.

Decoy routing. An alternative approach to unobservable circumvention is decoy routing [27, 64]. In this approach, a client steganographically hides her request to a blocked destination inside traffic sent to non-blocked destinations. A friendly "decoy" router intercepts this traffic, extracts the request, and deflects it to the true destination. While not yet implemented, traffic shaping is essential in decoy routing systems to protect against traffic analysis [27].

In general, decoy routing systems do not mimic other protocols and are outside the scope of our study. Furthermore, a recent study [52] shows that an adversary capable of changing routing decisions can effectively block decoy routing systems if they are deployed by only a few ISPs.

XI. Lessons and recommendations

Unobservability may very well be the most important property of censorship-resistant communication systems. Users of these systems run a very real risk of imprisonment and even death, and extra care must be taken to ensure that censorship circumvention solutions offered to them provide meaningful privacy and anonymity protection.

First, a thorough **understanding of the adversaries** is a must. Systems like SkypeMorph, StegoTorus, and CensorSpoofer deploy ad-hoc defenses against large-scale traffic analysis, yet leave their communications trivially recognizable even by very weak, local censors. Real-world censors are much more likely to look for tell-tale local deviations from genuine protocols (Section IV-C) than run sophisticated statistical algorithms on ISP-wide traffic traces.

Second, **unobservability by imitation is a fundamentally flawed approach**, unlikely to ever succeed due to the daunting list of requirements that an imitator must satisfy (Section V). The failure of all proposed parrot circumvention systems to achieve unobservability confirms this conclusion.

In particular, it is not enough to simply mimic a popular protocol. To achieve unobservability, the parrot must mimic *a concrete implementation* and be compatible with every implementation-specific quirk and bug (a similar observation has been made in other contexts such as HTML filtering [7] and file parsing [36]). For example, StegoTorus's imitated HTTP server is very distinct from any known HTTP server and thus trivially recognizable. Mimicking side protocols is especially difficult due to their complex, dynamic interdependences and correlations. As we demonstrated, the absence of such dependences is a dead giveaway of an imitation. Some imitation flaws are impossible to fix at any cost. For example, in the asymmetric, spoofing-based design of CensorSpoofer, the imitator cannot see the censor's probes and thus cannot mimic appropriate responses.

Third, **partial imitation is worse than no imitation at all**. For example, Tor traffic may be recognizable by certain traffic patterns, but this requires fairly sophisticated analysis of multiple flows. On the other hand, the not-quite-Skype imitation performed by SkypeMorph is easily recognizable given even a short observation of a single flow. Users of SkypeMorph, StegoTorus, and similar systems may be putting themselves at greater risk than the users of plain Tor because these ostensibly "unobservable" Tor transports are more distinct than Tor itself!

One promising alternative is to **not mimic, but run the actual protocol**, i.e., move the hidden content higher in the protocol stack. For example, FreeWave [28] hides data in encrypted voice or video payloads sent over genuine Skype, while SWEET [29] embeds it in email messages. This approach is well-known in steganography [15]: the covert information is always encoded into the features of an *actual* cover medium (e.g., an image), as opposed to synthesizing the medium. Embedding low-latency network services like Tor into another protocol is a challenging task, however. As in steganography, much research is needed to find the right balance between the unobservability of hidden messages and communication efficiency. For example, sizes of datagrams containing hidden messages may appear statistically anomalous in comparison to regular Skype datagrams. On the positive side, detection of such anomalies typically requires large-scale analysis of multiple flows and thus OM capabilities, raising the technical threshold for the censors.

Acknowledgments. We are grateful to Suman Jana for his insights and to Zack Weinberg for helping us understand StegoTorus. This research was supported by the Defense Advanced Research Agency (DARPA) and SPAWAR Systems Center Pacific, Contract No. N66001-11-C-4018, NSF grant CNS-0746888, and the MURI program under AFOSR Grant No. FA9550-08-1-0352.

References

[1] D. Adami, C. Callegari, S. Giordano, M. Pagano, and T. Pepe. Skype-Hunter: A Real-Time System for the Detection and Classification of Skype Traffic. *Int. J. Communication Systems*, 25(3):386–403, 2012.

[2] K. Allen. A Software Developer's Guide to HTTP. http://odetocode.com/articles/743.aspx.

[3] J. Arkko, E. Carrara, F. Lindholm, M. Naslund, and K. Norrman. MIKEY: Multimedia Internet KEYing. RFC 3830.

[4] Joining China and Iran, Australia to Filter Internet. http://www.foxnews.com/scitech/2009/12/15/like-china-iran-australia-filter-internet.

[5] S. Baset and H. Schulzrinne. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. In *INFOCOM*, 2006.

[6] S. Baset and H. Schulzrinne. Skype Relay Calls: Measurements and Experiments. In *INFOCOM*, 2008.

[7] D. Bates, A. Barth, and C. Jackson. Regular Expressions Considered Harmful in Client-side XSS Filters. In *WWW*, 2010.

[8] O. Berthold, H. Federrath, and S. Köpsell. Web MIXes: A System for Anonymous and Unobservable Internet Access. In *Design Issues in Anonymity and Unobservability*, 2000.

[9] D. Bonfiglio and M. Mellia. Tracking Down Skype Traffic. In *INFOCOM*, 2008.

[10] D. Bonfiglio, M. Mellia, and M. Meo. Revealing Skype Traffic: When Randomness Plays With You. In *SIGCOMM*, 2007.

[11] R. Breen. Circumventing Browser Connection Limits for Fun and Profit. http://www.ajaxperformance.com/2006/12/18/circumventing-browser-connection-limits-for-fun-and-profit/, 2006.

[12] Bridge Easily Detected by GFW. https://trac.torproject.org/projects/tor/ticket/4185, 2011.

[13] Tor BridgeDB. https://gitweb.torproject.org/bridgedb.git/tree.

[14] S. Burnett, N. Feamster, and S. Vempala. Chipping Away at Censorship Firewalls with User-Generated Content. In *USENIX Security*, 2010.

[15] I. Cox, J. Kilian, F. T. Leighton, and T. Shamoon. Secure Spread Spectrum Watermarking for Multimedia. *IEEE Transactions on Image Processing*, 6(12), 1997.

[16] Defeat Internet Censorship: Overview of Advanced Technologies and Products. http://www.internetfreedom.org/archive/Defeat_Internet_Censorship_White_Paper.pdf, 2007.

[17] R. Dingledine and N. Mathewson. Design of a Blocking-Resistant Anonymity System. https://svn.torproject.org/svn/projects/design-paper/blocking.html.

[18] M. Dusi, M. Crotti, F. Gringoli, and L. Salgarelli. Tunnel Hunter: Detecting Application-layer Tunnels with Statistical Fingerprinting. *Computer Networks*, 53(1):81–97, 2009.

[19] K. Dyer, S. Coull, T. Ristenpart, and T. Shrimpton. Format-Transforming Encryption: More than Meets the DPI. Cryptology ePrint Archive, Report 2012/494, 2012.

[20] N. Feamster, M. Balazinska, G. Harfst, H. Balakrishnan, and D. Karger. Infranet: Circumventing Web Censorship and Surveillance. In *USENIX Security*, 2002.

[21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616.

[22] D. Fifield, N. Hardison, J. Ellithrope, E. Stark, R. Dingledine, D. Boneh, and P. Porras. Evading Censorship with Browser-Based Proxies. In *PETS*, 2012.

[23] E. Freire, A. Ziviani, and R. Salles. On Metrics to Distinguish Skype Flows from HTTP Traffic. *J. Netw. Sys. Management*, 17(1-2):53–72, 2009.

[24] S. Gianvecchio and H. Wang. Detecting Covert Timing Channels: An Entropy-based Approach. In *CCS*, 2007.

[25] S. Guha, N. Daswani, and R. Jain. An Experimental Study of the Skype Peer-to-Peer VoIP System. In *IPTPS*, 2006.

[26] T. Heydt-Benjamin, A. Serjantov, and B. Defend. Nonesuch: a Mix Network with Sender Unobservability. In *WPES*, 2006.

[27] A. Houmansadr, G. Nguyen, M. Caesar, and N. Borisov. Cirripede : Circumvention Infrastructure Using Router Redirection with Plausible Deniability. In *CCS*, 2011.

[28] A. Houmansadr, T. Riedl, N. Borisov, and A. Singer. I Want My Voice to Be Heard: IP over Voice-over-IP for Unobservable Censorship Circumvention. In *NDSS*, 2013.

[29] A. Houmansadr, W. Zhou, M. Caesar, and N. Borisov. SWEET: Serving the Web by Exploiting Email Tunnels. *CoRR*, abs/1211.3191, 2012.

[30] The httprecon Project. http://www.computec.ch/projekte/httprecon/.

[31] Internet Censorship Listed: How Does Each Country Compare? http://www.guardian.co.uk/technology/datablog/2012/apr/16/internet-censorship-country-list.

[32] Iran Reportedly Blocking Encrypted Internet Traffic. http://arstechnica.com/tech-policy/2012/02/iran-reportedly-blocking-encrypted-internet-traffic.

[33] Italy Censors Proxy That Bypasses BTjunkie and Pirate Bay

Block. http://tiny.cc/fcmksw.

[34] J. Knockel, J. Crandall, and J. Saia. Three Researchers, Five Conjectures: An Empirical Analysis of TOM-Skype Censorship and Surveillance. In *FOCI*, 2011.

[35] J. McNamee. The Slide from Self-regulation to Corporate Censorship. http://www.edri.org/files/EDRI_selfreg_final_20110124.pdf.

[36] S. Jana and V. Shmatikov. Abusing File Processing in Malware Detectors for Fun and Profit. In *S&P*, 2012.

[37] D. Kesdogan, M. Borning, and M. Schmeink. Unobservable Surfing on the World Wide Web: Is Private Information Retrieval an Alternative to the MIX based Approach? In *PET*, 2002.

[38] C. Leberknight, M. Chiang, H. Poor, and F. Wong. A Taxonomy of Internet Censorship and Anti-censorship. http://www.princeton.edu/~chiangm/anticensorship.pdf.

[39] D. McCoy, J. Morales, and K. Levchenko. Proximax: A Measurement Based System for Proxies Dissemination. In *FC*, 2011.

[40] J. McLachlan and N. Hopper. On the Risks of Serving Whenever You Surf: Vulnerabilities in Tor's Blocking Resistance Design. In *WPES*, 2009.

[41] H. Moghaddam, B. Li, M. Derakhshani, and I. Goldberg. SkypeMorph: Protocol Obfuscation for Tor Bridges. In *CCS*, 2012.

[42] S. Murdoch and G. Danezis. Low-Cost Traffic Analysis of Tor. In *S&P*, 2005.

[43] S. Murdoch and S. Lewis. Embedding Covert Channels into TCP/IP. In *Information Hiding*, 2005.

[44] A Simple Obfuscating Proxy. https://www.torproject.org/projects/obfsproxy.html.en.

[45] A. Pfitzmann and M. Hansen. Anonymity, Unobservability, and Pseudonymity: A Consolidated Proposal for Terminology. In *Design Issues in Anonymity and Unobservability*, 2000.

[46] Tor: Pluggable transports. https://www.torproject.org/docs/pluggable-transports.html.en.

[47] L. Ptáček. Analysis and Detection of Skype Network Traffic. Master's thesis, Masaryk University, 2011.

[48] S. Radicati and Q. Hoang. Email Statistics Report, 2011-2015, 2011.

[49] H. Roberts, E. Zuckerman, and J. Palfrey. 2007 Circumvention Landscape Report: Methods, Uses, and Tools. http://cyber.law.harvard.edu/sites/cyber.law.harvard.edu/files/2007_Circumvention_Landscape.pdf.

[50] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389.

[51] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261.

[52] M. Schuchard, J. Geddes, C. Thompson, and N. Hopper. Routing Around Decoys. In *CCS*, 2012.

[53] H. Schulzrinne, S. L. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550.

[54] D. Schwartz and B. Sterman. Method and Apparatus for Server-side NAT Detection. US Patent US 2006/0187912 A1.

[55] Ten ways to discover tor bridges. https://blog.torproject.org/blog/research-problems-ten-ways-discover-tor-bridges.

[56] How Governments Have Tried to Block Tor. https://svn.torproject.org/svn/projects/presentations/slides-28c3.pdf.

[57] Ultrasurf. http://www.ultrareach.com.

[58] Virtual Distributed Ethernet. http://vde.sourceforge.net/.

[59] Q. Wang, X. Gong, G. Nguyen, A. Houmansadr, and N. Borisov. CensorSpoofer: Asymmetric Communication Using IP Spoofing for Censorship-Resistant Web Browsing. In *CCS*, 2012.

[60] Z. Weinberg, J. Wang, V. Yegneswaran, L. Briesemeister, S. Cheung, F. Wang, and D. Boneh. StegoTorus : A Camouflage Proxy for the Tor Anonymity System. In *CCS*, 2012.

[61] T. Wilde. Knock Knock Knockin on Bridges Doors. https://blog.torproject.org/blog/knock-knock-knockin-bridges-doors, 2012.

[62] B. Wiley. Dust: A Blocking-Resistant Internet Transport Protocol. http://blanu.net/Dust.pdf.

[63] P. Winter and S. Lindskog. How the Great Firewall of China Is Blocking Tor. In *FOCI*, 2012.

[64] E. Wustrow, S. Wolchok, I. Goldberg, and J. Halderman. Telex: Anticensorship in the Network Infrastructure. In *USENIX Security*, 2011.

Appendix A. Background on Skype

A. Overview of Skype protocol

Skype client start up. Every time the Skype application launches, the Skype client (SC) checks for a *Skype HTTP update* by connecting to ui.skype.com. The next step is *selecting the neighbor supernode (NSN)*. The client goes through its supernode cache (also called host cache), which is the "shared.xml" file with the list of nearby supernodes (SNs), and sends batches of UDP probes. The first SN who returns a positive response is selected as the NSN. A *Skype UDP probe* is an exchange of messages between SC and SN to discover the Skype network and its characteristics. There are two kinds of UDP probes, long and short, consisting of four- and two-packet exchanges, respectively. These packets have characteristic sizes [1] and contents [47].

SC then connects to the Skype network by opening a TCP connection to the selected NSN, using the same port as UDP probing. If the port is blocked, SC tries ports 80 and 443. If TCP connection establishment fails, SC reruns UDP probes to find another NSN. SC and NSN then perform *Skype TCP handshake*, which involves six messages. Their payloads are encrypted, but the handshake packets have characteristic sizes and the TCP PSH flag set [1, Fig. 2].

The final step is *Skype TCP authentication*. SC connects over TCP to Skype's central login server (LS) to get a certificate that SC uses to authenticate to other Skype nodes. This exchange involves several messages with the PSH flag set. The number of messages varies for different LSes and under different network conditions [47], but they exhibit characteristic sizes and patterns [5].

Making Skype calls. To make a call, a Skype client uses its TCP connection with the neighbor supernode (NSN) to find the callee's IP address and Skype port. The caller then uses UDP probes to check network connectivity—in particular, whether its own UDP port is open and whether its host is behind NAT. If the callee is not behind NAT, then the caller initiates a TCP connection to the callee and sends the ringing signal. Finally, if the callee accepts the call, a UDP connection is established for transferring the call data [47].

If the callee is behind NAT, then the caller, using the SN as a relay, tells the callee to send a UDP packet to the caller's IP address/Skype port in order to add a NAT entry for the call. The call then proceeds as without NAT. If both the caller and the callee are behind NAT, they use the SN to send UDP packets to each other. If Skype cannot bypass NAT/firewall, the call is handled by a relay and all traffic is encapsulated in an encrypted TCP stream.

Unrestricted connection (both SC_A and SC_B have public IP addresses). After probing multiple peers with UDP probes, SC_A establishes a TCP connection to SC_B and sends several signaling messages over it. This TCP connection is kept alive until the end of call. Voice and/or video contents are sent over a UDP connection between SC_A and SC_B.

NAT/firewall connection (SC_A and/or SC_B are located behind a NAT or a firewall). In this case, SC_A sends the signaling information to SC_B through the SNs. If only SC_A is behind NAT, SC_A and SC_B are usually able to establish a direct UDP connection after the signaling [47]. Otherwise, SC_A finds appropriate relay nodes, and SC_A and SC_B directly connect to a relay which exchanges traffic between them. For fault tolerance and backup, several relays are typically used [47]. Most calls use different relays for the caller-to-callee and callee-to-caller flows [6].

If the Skype TCP connection used for signaling is closed, the UDP connection also closes. Furthermore, Skype clients periodically send *Skype UDP pings*, which consist of two keep-alive packets, in order to preserve their "online" status in the Skype network. These packets can be identified by the "0x02" string in their function field.

B. Passive detection of Skype traffic

There are many techniques for detecting Skype traffic [1, 5, 6, 10, 47]. They recognize characteristic strings sent unencrypted during Skype sessions (content analysis) and/or characteristic traffic patterns such as packet sizes (pattern analysis). The tests below work against all versions of Skype.

T1: HTTP update messages. When the Skype client (SC) starts up, it makes an HTTP connection to ui.skype.com to check for updates to the client software [47].

T2: Login messages. In order to authenticate itself, SC needs to obtain a certificate from Skype's login server (which could be a Skype supernode) confirming the client's Skype identity. Unlike software update messages, logins are not handled by a single, known server, thus login messages cannot be easily detected by IP address matching. They can be recognized, however, by characteristic sizes and contents. In particular, the second message in a login TCP connection carries the header `0x170301` in plaintext.

T3: Start of Message (SoM) fields in UDP packets. Skype uses special headers, so called SoM fields, for its UDP packets [10]. The SoM fields are present in both UDP probes and UDP packets carrying the media stream. They are not encrypted and have specific values for different kinds of packets. In particular, ID and Fun fields are easily recognizable in a SoM header [10].

The first two bytes of SoM contain an *ID* that uniquely identifies that message. This value is randomly generated by the sender and copied in the receiver's reply. *Fun* is a 5-bit field obfuscated into the third byte of SoM and revealed by applying the `0x8f` bitmask. Previous research [10, 47] investigated the values of Fun for different messages. For instance, `ox02`, `0x03`, `0x07`, and `0x0f` indicate signaling messages during the login process and connection management, while `0x0d` indicates a data message, which may contain encoded voice or video blocks, chat messages, or data transfer chunks.

T4: Packet sizes. A *UDP probe* consists of four packets [1]. The second packet is 11 bytes long, while the length of the fourth packet reveals the outcome of that UDP probe.

A *Skype TCP handshake* consists of six messages with the PSH flag set and payload sizes of 27 and 4 bytes for the fourth and sixth packets, respectively.

Authentication messages from a Skype client to the login server include four or more packets with the PSH flag set; the first two have 5-byte payloads.

An *HTTP update request* returns a single unencrypted packet from ui.skype.com. This packet has a fixed value in the first 29 bytes for the Linux version of Skype and another fixed value in bytes 95-124 for the Windows version [47].

T5: Packet timings and rate. Skype audio and video traffic exhibits a characteristic packet timing pattern, depending on the codec used. SILK, Skype's audio codec, samples at 8, 12, 16, or 24 KHz, resulting in four ranges of data rates for UDP flows carrying Skype audio [47, Fig. 3.6]. While Skype voice packets are about 150 bytes, video packets are around 1380 bytes. Sample inter-packet gaps and size distribution for a Skype video call can be found in [47, Fig. 3.10].

T6: NAT traversal. Once SC starts up, it performs a sequence of tests to detect whether it is behind NAT or a firewall. SCs use different variants of the STUN [50] protocol for NAT traversal.

T7: Periodic message exchanges. Skype is a P2P system, and SCs frequently exchanges messages with other Skype nodes to detetermine their online/offline status [47]. In particular, each SC establishes about five short TCP connections per hour and performs UDP probes on approximately thirty Skype peers per hour [47, Fig. 3.4].

T8: Typical Skype client behavior. Each typical task performed by a SC, like searching for a contact or placing a call, generates characteristic traffic (Appendix A-A).

T9: TCP control channel. Skype uses various TCP control channels. In particular, each call is accompanied by a TCP signaling connection, described in Appendix A-A, which remains active during the call.