Slitheen++: Stealth TLS-based Decoy Routing

Benedikt Birtel
CISPA – Helmholtz-Zentrum für Informationssicherheit gGmbH
Christian Rossow
CISPA – Helmholtz-Zentrum für Informationssicherheit gGmbH

Abstract

We present Slitheen++, a decoy routing system that—in contrast to its predecessor Slitheen—is not susceptible to traffic analysis in the upstream channel. Slitheen++ overcomes key challenges such as scheduling for covert connections and technologies to more realistically emulate a real user's behavior, such as crawling or delaying overt communication. We measure Slitheen++ according to metrics that not only show the maximum theoretical throughput of the system, but for the first time, also assess the actual user experience by measuring loading times of websites from ten covert targets. We show that emulating a user increases loading times, yet raises the difficulty for an advanced censor to expose decoy routing as such. For example, crawling raises the median of the loading time for covert setups by 1 second from 7 s to 8 s.

1 Introduction

Censorship-resistant communication systems have been challenged by the increasing censoring capabilities in recent years [25]. In a world where slight deviations from uncensored ("overt") communication lead to effective censorship, careful designs must not allow censors to distinguish overt from censorship-evading ("covert") communication. Decoy routing is one anti-censorship concept that has recently seen significant developments [4, 10, 15, 17, 19, 29, 30]. The basic idea is as simple as it is effective: routers on the path of an overt communication redirect tagged communication originally addressed to a non-censored target to a censored host. Basic decoy routing alters communication patterns and is thus vulnerable to traffic analysis attacks [7, 9, 14, 27]. Yet, advanced decoy routing systems nearly perfectly imitate normal communication patterns by carefully replacing content of the overt traffic. Slitheen [4] is one such decoy routing system that provides strong anti-censorship guarantees with the help of a friendly Internet Service Provider (ISP) that is situated between the censored client and an uncensored target. In the original Slitheen design, the authors left open a few key challenges that we address in this work. Most importantly, Slitheen communication is not immune against traffic analysis attacks, as its design changes the client's upstream communication patterns. In particular, clients simply append the covert request data to their overt requests, significantly changing the upstream traffic and allowing for identification by a censor—even if traffic is encrypted. As several attacks on existing steganographic systems [1, 6, 14] have shown, such traffic analysis constitutes a severe threat in practice. To address this challenge, we augment Slitheen with an upstream component that meets the same security guarantees that Slitheen's downstream channel already offers: By using an Hypertext Transfer Protocol Version 2 (HTTP/2)-like compression function, we create space gains to place covert upstream data such that the attacker must not be able to tell whether Slitheen is in use or not, neither based on timing information, nor based on observable changes to traffic patterns (packet sizes, delays, etc.). In this paper, as a result of both our methodological extensions and our fundamental design improvements, we present Slitheen++, a significantly updated Slitheen version that is not susceptible to traffic analysis in the upstream channel and is free of severe bugs identified in the original Slitheen implementation. Along with these additions, a major contribution of our work is an assessment of Slitheen++ in a realistic context, i.e., testing with actual covert targets and leveraging conservative overt communication. All in all, our novel design and the underlying rigorous experiments help to close the gap between academic decoy routing proposals and tight demands for anti-censorship systems in practice [2, 25]. We summarize our contributions as follows: We provide Slitheen++, a design of a stealthy decoy routing system that meets the requirements of today's censored users by tunneling down- *and* upstream data covertly in overt communication. We evaluate it according to metrics to show the maximum theoretical throughput while we also measure the actual loading time of covert websites. We introduce browsing delays and crawling as means to mimic normal user behavior during overt communication, and quantify the negative consequences of such an overt communication.

2 Related Work

Censorship circumvention methodologies enable clients inside a censor's domain to access otherwise-blocked content. For example, Tor has been extended to provide protections against various censorship tactics to block the Tor infrastructure, such as Tor bridges or traffic obfuscation [20, 28]. Yet censors can detect such first-generation evasion attempts [8, 14, 18, 31]. Consequently, researchers designed censorship-resistant communication mechanisms that are not prone to traffic analysis and do not reveal endpoints involved in censorship evasion. For example, Fifield *et al.* presented *domain fronting* [11]. However, network traffic analysis [7, 9, 14, 27] could reveal that fronting services do not belong to the provider in the given domain carried in the Transport Layer Security (TLS) handshake. Furthermore, domain fronting providers could be replaced inside the censor's domain by a similar service under the control of the censor. To tackle this problem, researchers have eliminated the requirement of additional connection endpoints to circumvent a censor. Most prominently, using *decoy routing*, a middle-to-end proxy (also known as a decoy router) is placed outside a censor's domain on the path to a non-censored target [4, 10, 15, 19, 29, 30]. Clients establish encrypted connections to such a target using a special tagging mechanism that is only recognizable by the decoy routers, and enable it to decrypt the traffic addressed to the non-censored ("overt") target. The decoy router extracts the censored target from the messages of the tagged flow and redirects all traffic towards the specified "covert" target. Some of the decoy routing technologies are vulnerable to various attacks like Transmission Control Protocol (TCP) replay attacks. Furthermore, advanced traffic analysis capabilities given by a censor could reveal the usage of decoy routing. Slitheen is a decoy routing system that aims to avoid detection by advanced traffic analysis [4]. Slitheen uses multiplexing to transport data to/from a censored target inside the traffic of a non-censored target. Thereby, the Slitheen authors aim to achieve immunity to a wide range of detection attacks. However, Slitheen is susceptible to traffic analysis attacks, given that upstream covert traffic is just appended to overt upstream, which significantly changes communication patterns. One way to tackle this problem of traditional decoy routing is downstream-only decoy routing as proposed by Nasr *et al.* [21]. The main idea is to transmit covert upload data via the downstream overt data, using overt destinations that offer reflection capabilities. This technology counters the Routing Around Decoys (RAD) attack [24], but requires an out-of-band channel for bootstrapping and synchronization with new clients. While Slitheen++ uses "normal" overt communication, downstream-only routing risks that overt sites recognize and block the excessive abuse of redirect/error messages.

3 Adversary model

We assume that the censor is a state-level omni-scientist adversary [14], that has passive, active and reactive networking capabilities. The classification "omni-scientist" represents rich computation and storage capacities to perform traffic analysis. That means the adversary can store network traces over a longer period of time and analyze its master data set to identify covert connection properties. We leave active attacks against decoy routing, such as RAD attacks or attempts to enforce asymmetric routes, out of the scope of this work. However, in principle, one could integrate proposed additions to decoy routing systems [5] that are compatible to Slitheen++, so it withstands RAD attacks.

4 Slitheen++

Slitheen introduced a novel idea to overcome strong censors that deploy advanced network analysis capabilities [4]. Yet, in their seminal paper, the authors left open some key challenges, such as stealth upload communication, or providing a working prototype of Slitheen that can be evaluated under realistic circumstances. In this section, we provide Slitheen++, an extension of Slitheen that tackles these and further shortcomings of Slitheen. First, we start with the upstream communication. Slitheen's communication is trivial to identify via traffic analysis. By design, only the downstream packet sizes stay equally-large. But Slitheen neglects the fact that simply appending covert upstream data to overt upstream will be detected by censors. In Slitheen++, we augment Slitheen's concept with a stealth data upstream channel by compressing Hypertext Transfer Protocol (HTTP) header fields in overt requests and using the gained space to place covert upload data in overt requests. Technically, we added a HTTP/2-like compression to the Overt User Simulator (OUS) as well as to the relay station such that only the Slitheen components themselves are aware of the compression. We avoided additional GNU ZIP (GZIP) compression to avoid the Compression Ratio Info-leak Made Easy (CRIME) attack [16]. In a next step, we took care about connection scheduling. Covert applications actually require several concurrent connections, which compete for the available covert transmission capacities. Slitheen has no dedicated scheduler and extracts covert upload and download data in the order it was inserted into a single queue shared by all connections—a strategy that does not provide fairness and might cause connection timeouts. In Slitheen++, we support scheduling for up- and downstream data, using a First In First Out (FIFO) and a Sliced Round Robin (SRR) scheduler. Every scheduler maintains a queue per covert connection. The FIFO scheduler uses a FIFO list of covert connections. In contrast, the SRR scheduler consists of a queue of active connections that have to be scheduled. Every connection has a fixed slice that specifies how much data it can transmit before another connection gets scheduled. For our testing, we use

an upload slice size of 256 bytes while the download slice has a size of 1024 bytes. Our third improvement is related to the browsing behavior of the OUS overt communication. A fundamental requirement for decoy routing is that clients create sufficient overt traffic. To this end, Slitheen clients constantly re-visit a predefined, overt Web page (e.g., Wikipedia's `/index.html`). This pattern, however, may be recognizable by censors. To avoid a detection and to mimic human-like browsing behavior that mitigates this problem, we added a Web crawler to Slitheen++'s OUS. The crawler extracts and visits links from the current overt website, but stays within the same domain. Another aspect regards the frequency in which Web sites are visited. Slitheen repeatedly reloaded the same overt Web page without any "Thinking Time (TT)" between two page requests. This overly aggressive approach increases the available covert bandwidth massively. In contrast, adding delay ("TT") between two requests to the overt site represents more realistic human behavior. The next challenge present in the original Slitheen implementation was the handling of Out-of-Order traffic. The relay station of Slitheen uses raw sockets to capture, analyze and manipulate traffic. However, such a design requires careful thinking when facing real-world traffic. Most importantly, Slitheen assumes that data of the overt communication is never reordered when being routed to its destination. However, this does not hold in practice, such that Slitheen cannot decrypt reordered TLS records. In fact, the number of decipherable records can decrease to zero. This is based on the fact that the TLS [23] implementation uses an internal state per endpoint with sequence numbers that are increased whenever a new TLS record is sent/received. These sequence numbers are fundamental for TLS's Message Authentication Code (MAC) computation. The relay station needs to keep that state as well in order to decrypt and re-encrypt TLS records as well as to determine the record bounds. Slitheen faces similar problems with regards to fragmented Internet Protocol (IP) packets. When testing Slitheen, we frequently encountered reordered packets that stalled Slitheen's communication. Solving this problem is non-trivial, and we sketch potential solutions in Section 6. For now, we mitigated the problem by augmenting Slitheen with a mechanism to restore the correct order of out-of-order TCP segments, using a proxy named *traffic server* [12] which allows us to tune various connection parameters, including TLS specific configurations like the maximum TLS record size. To this end, we connect the traffic server directly to the relay station to avoid another reordering. Having said that, this adaption cannot withstand an advanced censor, since she could simply measure the reordering rates in downstream traffic originated from a suspicious target. However, if all downstream traffic of a specific connection is in order while others are out of order, the censor would know that this connection is an overt Slitheen connection. Additionally, reordering would require that the relay station temporarily store requests, increasing the packets round trip timings, which leads to another possible detection by the censor. After the conceptual adaptions, Slitheen needed TLS based improvements to be operational.

Invalid Nonces: TLS is fundamental for the functionality of Slitheen and protects against eavesdroppers. Nonces are an important element for TLS in conjunction with block ciphers and Authenticated Encryption with Associated Data (AEAD) schemes and create the Initialization Vector (IV) for the symmetric encryption. Each record will carry its own nonce. Slitheen's implementation replaced the nonce in TLS downstream records with an uninitialized value. This does thus not guarantee that the nonce will change for different records of the same connection, which violates the TLS standard and hence, undermines TLS's security guarantees. Furthermore, its non-unique nonces are easily detectable by censors. We improved this and changed Slitheen such that it maintains the original nonce/IV.

Incoherent TLS/TCP/HTTP Interplay: In the next step, we improved the methodology which Slitheen uses to find replaceable data in TLS records. Slitheen's authors assumed that (a) the HTTP header of a response fits into a single TLS record, and (b) this record is carried in a single TCP segment. If (a) is violated, Slitheen discards the content as not replaceable. If (b) is violated, the TLS record cannot be decrypted by Slitheen, because Slitheen does not buffer any overt download data. To mitigate this problem and to increase the covert bandwidth, we used the traffic server to create TLS records which do not exceed a maximum pre-configured size. The last step to finish Slitheen++ was the correction of several implementation problems of the original Slitheen prototype. So far, we have discussed fundamental adaptations to Slitheen's original concept. However, the original Slitheen implementation was unfortunately inoperable due to various implementation mistakes: (1) Segmentation faults, (2) Use of uninitialized variables, (3) Statically allocated buffer to store messages of arbitrary length—possibly even allowing for remote code execution attacks, (4) OUS SOCKS proxy was unable to handle TCP connection terminations, causing connection timeouts, and (5) HTTP parsing issues causing covert data forwarding to stop due to inappropriate state machine updates. All described issues have been fixed[1]. We will now address a few important inconsistencies in Slitheen's implementation:

Mixed Encryption Modes: In Slitheen, all covert downstream data is encrypted on the relay station and later decrypted by the OUS to mitigate an attack [5] where an additional adversary (not the censor itself) can read traffic from both sites of the relay station. This adversary can manipulate unencrypted covert data multiplexed into the overt traffic when it leaves the relay station towards the censored client. The overt traffic must be encrypted using a symmetric cipher mode that forbids the reuse of nonces when encrypting mes-

[1] More details about those implementation related problems can be found in our technical report [3]

sages. However, if the covert connection is encrypted and protected by integrity checks, the adversary could simply destroy its integrity. Having said this, most websites today are TLS-secured, and the tests in our evaluation also use TLS, making this potential threat obsolete. Consequently, we decided to drop this additional encryption, instead of having aligned the en-/decryption modes.

Inconsistent Type-Length-Value (TLV) Streams: Slitheen uses a TLV encoding for covert messages to indicate how many bytes of user data and garbage were present in the current covert message. Beside the inconsistent management of tagged flows, there was also a problem with the TLV encoding itself. The original was not able to handle split TLV headers, as the authors assumed that they would always have enough space in any TLS record such that they could place an entire header. Furthermore, if a covert message did not replace all overt leaf content in current overt record, the software did interpret the remaining overt bytes as a Slitheen message.

5 Evaluation

Our experimental setup is based on the web, i.e., we assume that the Slitheen++ user wants to browse to censored Web sites and at the same time can use at least one overt web page. Consequently, we use web pages both as overt and covert targets in our evaluation. We used ten scenarios of covert domains for which we evaluate Slitheen++: Twitter (0), Instagram (1), Google Play Store (2), Apple Store (3), Google News (4), BBC (5), Reddit (6), Stack Overflow (7), GitHub (8) and Google Code (9). For later referencing, we will use the scenario ID in parentheses. For each of these scenarios, we use Slitheen++ to load three Uniform Resource Locator (URL)s. In order to focus on the covert target's main content and remove biases, we leverage the browser extension *uMatrix* to block advertisements and statistics scripts from third-party domains. During our evaluation, we define the following parameters that Slitheen++ uses to operate:
(1) **TT**: We vary the TT, described in Section 4, to model a user who loads a website and looks for information, instead of immediately loading the next page, and
(2) **Crawling**: Indicates whether or not the OUS will crawl on the current overt site for new URLs or will stick at the initial overt URL.

We adapt these parameters to reflect different setups and evaluate each scenario ten times per setup to stabilize results. As our overt (starting) URL, we will use the English "Computer Science" article of Wikipedia. For scheduling, we decided to use always the best performing scheduler per scenario. For testing, we used an Ubuntu 17.10 (kernel 4.13) desktop computer with an Intel Core-i5 4690, 32 GB RAM and a 1 Gbps uplink. The computer itself is used as the relay station. Furthermore, two virtual machines were executed on it. One represented the client machine, while the other was used as the traffic server. We used Google Chrome as the covert web browser for all experiments. To evaluate the stealthiness and performance of Slitheen++, we use the following metric:

Overt Forwarding Costs (OFC): To avoid detection by advanced traffic analysis, Slitheen++ has to guarantee that it does not change the fingerprinting characteristics of the overt communication in any detectable way. This includes the latency introduced by Slitheen++ and its covert multiplexing task. Therefore, at the relay station, we measure the forwarding times for overt upload and download IP packets. Technically, we measure the time from when the relay station receives a packet from the kernel until it passes the (possibly modified) packet back to the kernel. In contrast, the original Slitheen evaluation measured the time the PhantomJS needed to load an overt site, not the latency per packet.

Overt and Covert Goodput: We use the term goodput to specify the amount of application data (e.g., HTTP traffic) forwarded for a specific program. That is, overt goodput is the number of bytes used by the application layer of the overt communication, while covert goodput defines the number of bytes used by Slitheen++ to multiplex covert data inside the overt channel (including Slitheen++ headers). To measure the utility of Slitheen++, we monitor the maximum possible covert goodput available, and relate this to the goodput that the resulting covert channel actually generates.

Loading Times for Covert Browser: The time it takes for the covert application to load data greatly influences user experience. The loading times refer to the time needed to load all resources that belong to a URL, determined by observing when the browser's internal state named `document.readyState` changes to "complete". As a baseline, we measure the per-scenario loading time without decoy routing. We then evaluate loading times when running the same scenario using Slitheen++. Each scenario runs exactly ten times in each setup.

Scheduler Evaluation The following tests will not reflect the performance of the available scheduler. We decided to only use the best performing scheduler per scenario[2].

Evaluation of TT and Crawling We measure the impact of using Slitheen++ with TT and crawling. Thereby, we will apply our above described metric.

Loading Times: Figure 1 shows the loading times for the individual scenarios. We only considered no (0 s) or low (1 s) TT. The x-axis labels denote the four setups we measured, where "Naive" corresponds to the traffic generation of the original Slitheen, while "Crawl" using crawling. Generally, adding crawling has less negative impact on the covert loading

[2]More details about the scheduler and their evaluation can be found in our technical report [3]

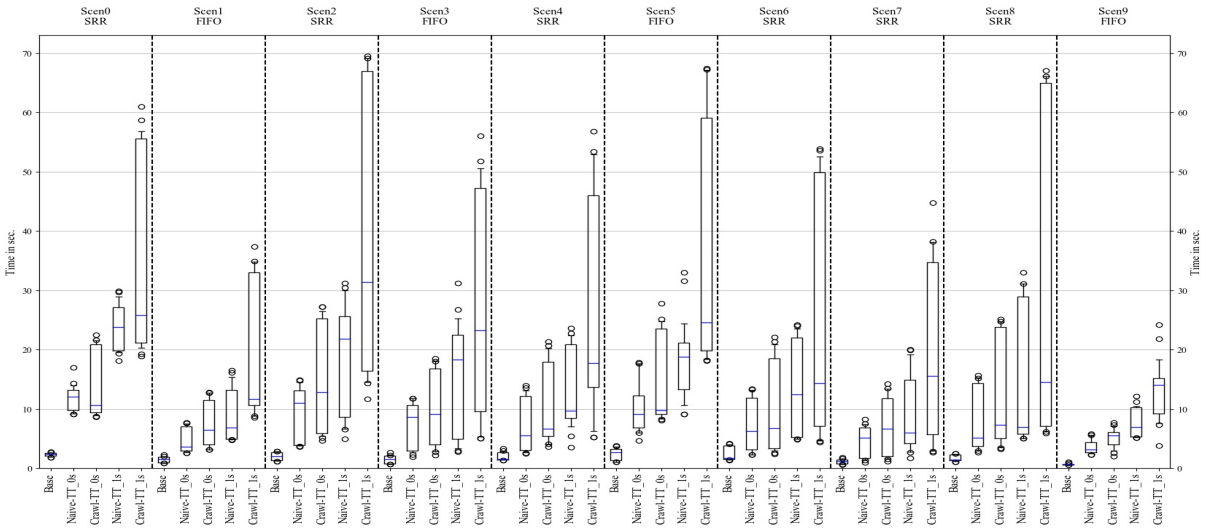

Figure 1: URL resource loading times for phase three evaluation grouped by scenarios

performance than adding 1 second of TT without crawling. Considering scenario 0, crawling can even have a positive effect on the loading time, as demonstrated by the decreased median loading time. This is caused by a random correlation between the available covert data and the offered multiplexing overt capacity. One of the fundamental problems with automated link-extracting crawling (as we use it) is the uncertainty whether the following URL will provide a rich covert goodput. With crawling, the available covert goodput thus fluctuates across overt URLs. On the other site, TT decreases the available covert goodput, which we will explain in the following segment. The average of the medians for the naive variant is 7 seconds, while it becomes 8 seconds when crawling is enabled. The Baseline-to-Covert factor varies from 3.7 to 8.5 without TT. The resulting Baseline-to-Covert factor for crawling with a TT of 1 second varies from 7.6 to 21.4. The average of the medians when TT is used without crawling is 13 seconds, while crawling increases it to 19 seconds. TT in addition to crawling stacks the advantages of both technologies, as well as their disadvantages. In almost all test cases, adding TT or using crawling decreased the user experience. Consequently, using both together leads to a further slowdown in performance the decoy routing system can provide, while in such a setup it gets much harder for a censor to expose the overt communication as decoy routing. **Covert Goodput:** The total amount of goodput (a) fluctuates due to the fact that the crawled URLs varied in their goodput capabilities and (b) is reduced by the TT idle periods where no overt/covert data is transmit. Loading all resources of wikipedia's Computer Science article takes an average of 838 ms on our system. Hence, using a TT of 1000 ms creates a chainsaw pattern of overt goodput, resulting in a covert bandwidth of roughly 50 % compared to a test with no TT used. Figure 2 shows that the available covert goodput decreased when crawling is enabled. Most crawled URLs have a worse covert goodput than the starting URL, and only very few crawled URLs provide

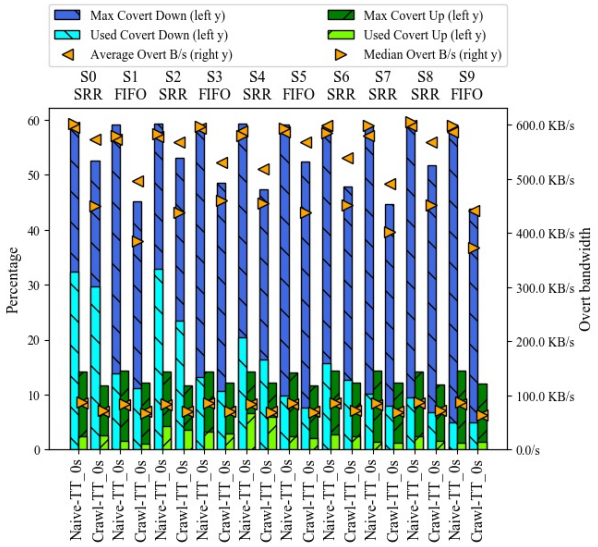

Figure 2: Maximum covert goodput and used covert goodput

an increased covert goodput. Inspecting all crawling-enabled examples, tests with a higher runtime (such as scenarios 0 and 2) show a greater covert transport capability compared to setups with a relatively low runtime (e.g. scenarios 1 and 9). The high average of tests where crawling is used shows that we have some rich URLs which raise the average, but the lower median shows the majority of URLs are less suitable. Furthermore, there is no guarantee that rich covert transport windows can be used, since the generation of the overt carrier is independent from the covert transport needs. Consequently, the percentage of bytes used for covert communications decreased in almost all setups, especially in downstream. Furthermore, covert web servers prematurely closed connections if the upload capabilities were temporarily low such that Distributed Denial-of-Service (DDoS) protection mechanisms close the connection before (a) any covert goodput gets transmitted or (b) the connection can be reused for further data

transmissions. This increased the total number of covert connections needed, including the overhead for establishing the TLS-secured covert communication. In this case, the usage of the FIFO scheduler helps reducing the number of connection closes. Finally, we encountered several connections that have to transmit large amounts of covert downstream data, which are further delayed due to temporal drops in the maximum covert goodput.

OFC: We considered the percentage of packets forwarded in a specific direction that exceed a certain delay (1 ms to 50 ms). 95% of the setups encountered downstream delays between 1 ms and a maximum of 4 ms, but not above this. On average, 0.0029% of downstream packets were involved; the highest value ever measured in a single scenario was 0.006%. Table 1 shows the fraction of delayed upstream packets. The second column named "Avg(D)" provides the average percentage of all packets in all tests that encountered the given delay D. Column number three named "Max(D)" show the maximum percentage of packets delayed by D that occurred in the execution of a single scenario run. The last column indicates how many setups that we have tested encountered the current delay D. One upload outlier was around 60 ms; all others stayed under 60 ms. An investigation showed that the usage of many threads to handle covert connections on the relay station can cause higher delays, especially if new threads must be spawned. Due to crawling and the resulting goodput fluctuations, the relay station's load behavior varies and it encounters more situations where threads compete for resources. This behavior can increase the delays as well.

Delay D	↑ Avg(D)%	↑ Max(D)%	Setups involved %
> 1 ms	0.0149	0.0248	100
> 5 ms	0.0035	0.0090	90
> 10 ms	0.0033	0.0090	90
> 30 ms	0.0018	0.0036	80
> 50 ms	0.0007	0.0007	15

Table 1: Phase Three Upload OFC Evaluation

6 Discussion and Future Work

We now discuss limitations of Slitheen++ and describe our future work directions to address them. When testing other overt domains, we encountered problems with those that embed *Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA)s* that require input from users. We tried to evade CAPTCHAs by randomizing the user agent, but with little success. The consequence is a sudden drop in covert goodput. In our evaluation, we ignored such websites and stuck to Wikipedia, and we believe that censored clients can always find overt domains without CAPTCHAs. Furthermore, we encountered timeouts if covert targets deployed specific DDoS protection mechanisms, especially if web servers close TCP connections if they have not been used for a few (e.g., two) seconds. If TT is used, this causes problems on the client site, as it frequently takes several seconds to transfer a covert request. This can force the client to establish multiple connections to load a single resource or even deny the client to load that resource. For our evaluation, we used the traffic server, which limits the maximum TLS record size and reorders all incoming TCP traffic. However, in a real-world setup, downstream traffic has to pass the relay station in the same order as it comes from the wire, as reordering would be detectable by a censor. Yet as mentioned in Section 4, reordered traffic is potentially unusable for multiplexing of overt and covert data. Hence, unordered traffic would reduce the available maximum covert goodput, resulting in longer loading times for resources. To tackle this challenge, Slitheen++ could be made aware of Out-of-Order (OOO) traffic, including proper handlers for all protocols involved, such as TCP and TLS. For example, a TLS handler would have to wait for the TCP layer to collect all data, such that the TLS handler can count the number of records passed and re-encrypt traffic. Adding such capabilities requires various protocol handlers for the overt connection, such as for IP, TCP, TLS and HTTP. TT and crawling increase loading times. To stay stealthy, users thus face a dilemma. Either they can reach blocked resources slower, or their communication can potentially be revealed by the censor. Therefore we envision additional concepts that further reduce loading times. First, we could add a cache to the relay station to store complete request messages with unique IDs from a covert client. The OUS could replace an already sent request with the according ID and use the rest of the overt message to place covert data there. Second, our observations show that crawling can also *decrease* loading times. We could use a predefined list of URLs which guarantee higher covert goodput or adapt the crawler to generate the overt carrier based on the current covert queue state. On the other site, a fixed crawling could be identified using traffic analysis, such that the usage of the system is still detectable by an advanced censor. Finally, mixing user applications for Slitheen++ could also increase the available bandwidth, such as video/music streaming or online games [26]. Third, a replacement for the PhantomJS as overt browser in the OUS, such as Chrome or Firefox, would also help. Both support a headless running mode, which shows much better performance compared to PhantomJS [13]. Our own measurement (crawling 20 pages belonging to Wikipedia) indicated that a headless Chrome is 51.9% faster on average than PhantomJS and sends 37.9% more requests. In essence, we believe that the usage of Chrome or Firefox in the OUS would improve the performance of Slitheen++. The performance results of our current evaluation show that the CPU-based, multithreading design of the system (a) was limiting the maximal throughput of the system and (b) caused high delays, especially for upstream packets. We think that the usage of a Protocol-Independent Switch Architecture (PISA)-based implementation [22] of the relay station could speed up Slitheen++ massively, as the PISA-based switches can perform both stateful operations as well as custom packet parsing at line rate.

References

[1] Davide Adami, Christian Callegari, Stefano Giordano, Michele Pagano, and Teresa Pepe. Skype-Hunter: A real-time system for the detection and classification of Skype traffic. *International Journal of Communication Systems*, 25(3):386–403, 2012.

[2] Sadia Afroz, David Fifield, Michael C Tschantz, Vern Paxson, and JD Tygar. Censorship Arms Race: Research vs. Practice. In *Workshop on Hot Topics in Privacy Enhancing Technologies*, 2015.

[3] B. Birtel and C. Rossow. Technical report cispa-tr-2020-03-09-bbcr on slitheen++: Stealth tls-based decoy routing. Technical report, CISPA - Helmholtz Center for Information Security, 2020. Available online at https://cispa.saarland/group/rossow/papers/tr-slitheen++.pdf.

[4] Cecylia Bocovich and Ian Goldberg. Slitheen: Perfectly imitated decoy routing through traffic replacement. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1702–1714. ACM, 2016.

[5] Cecylia Bocovich and Ian Goldberg. Secure asymmetry and deployability for decoy routing systems. *Proceedings on Privacy Enhancing Technologies*, 3:1–20, 2018.

[6] Dario Bonfiglio, Marco Mellia, Michela Meo, Dario Rossi, and Paolo Tofanelli. Revealing skype traffic: when randomness plays with you. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 37–48. ACM, 2007.

[7] Sambuddho Chakravarty. *Traffic analysis attacks and defenses in low latency anonymous communication*. PhD thesis, Columbia University, 2014.

[8] Jedidiah R Crandall, Daniel Zinn, Michael Byrd, Earl T Barr, and Rich East. ConceptDoppler: A weather tracker for internet censorship. In *ACM Conference on Computer and Communications Security*, pages 352–365, 2007.

[9] Maurizio Dusi, Manuel Crotti, Francesco Gringoli, and Luca Salgarelli. Tunnel Hunter: Detecting application-layer tunnels with statistical fingerprinting. *Computer Networks*, 53(1):81–97, 2009.

[10] Daniel Ellard, Christine Jones, Victoria Manfredi, W Timothy Strayer, Bishal Thapa, Megan Van Welie, and Alden Jackson. Rebound: Decoy routing on asymmetric routes via error messages. In *Local Computer Networks (LCN), 2015 IEEE 40th Conference on*, pages 91–99. IEEE, 2015.

[11] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. Blocking-resistant communication through domain fronting. *Proceedings on Privacy Enhancing Technologies*, 2015(2):46–64, 2015.

[12] Apache Software Foundation. Traffic Server. Available online at http://trafficserver.apache.org/.

[13] hartator. Headless Chrome vs PhantomJS benchmark. Available online at https://hackernoon.com/benchmark-headless-chrome-vs-phantomjs-e7f44c6956c.

[14] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The parrot is dead: Observing unobservable network communications. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 65–79. IEEE, 2013.

[15] Amir Houmansadr, Giang TK Nguyen, Matthew Caesar, and Nikita Borisov. Cirripede: Circumvention infrastructure using router redirection with plausible deniability. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 187–200. ACM, 2011.

[16] Thai Duong Juliano Rizzo. The CRIME attack. Available online at https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit#slide=id.g1d134dff_1_222.

[17] Josh Karlin, Daniel Ellard, Alden W Jackson, Christine E Jones, Greg Lauer, David Mankins, and W Timothy Strayer. Decoy Routing: Toward Unblockable Internet Communication. In *USENIX Workshop on Free and Open Communications on the Internet*. ACM, 2011.

[18] Sheharbano Khattak, Mobin Javed, Philip D Anderson, and Vern Paxson. Towards Illuminating a Censorship Monitor's Model to Facilitate Evasion. In *3rd USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2013.

[19] Donghyun Kim, Glenn R Frye, Sung-Sik Kwon, Hyung Jae Chang, and Alade O Tokuta. On combinatoric approach to circumvent internet censorship using decoy routers. In *Military Communications Conference, MILCOM 2013-2013 IEEE*, pages 593–598. IEEE, 2013.

[20] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. Skypemorph: Protocol obfuscation for Tor bridges. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 97–108. ACM, 2012.

[21] Milad Nasr, Hadi Zolfaghari, and Amir Houmansadr. The Waterfall of Liberty: Decoy Routing Circumvention that Resists Routing Attacks. In *Proceedings of the*

2017 ACM SIGSAC Conference on Computer and Communications Security, pages 2037–2052. ACM, 2017.

[22] Barefoot Networks. The World's Fastest and Most Programmable Networks. Available online at https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/.

[23] The Transport Layer Security (TLS) Protocol Version 1.2. Request for comments, Internet Engineering Task Force, August 2008.

[24] Max Schuchard, John Geddes, Christopher Thompson, and Nicholas Hopper. Routing around decoys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 85–96. ACM, 2012.

[25] Michael Carl Tschantz, Sadia Afroz, Vern Paxson, et al. Sok: Towards grounding censorship circumvention in empiricism. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 914–933. IEEE, 2016.

[26] Paul Vines and Tadayoshi Kohno. Rook: Using video games as a low-bandwidth censorship resistant communication platform. In *Proceedings of the 14th ACM Workshop on Privacy in the Electronic Society*, pages 75–84. ACM, 2015.

[27] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective Attacks and Provable Defenses for Website Fingerprinting. In *USENIX Security Symposium*, pages 143–157, 2014.

[28] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. Stegotorus: A camouflage proxy for the Tor anonymity system. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 109–120. ACM, 2012.

[29] Eric Wustrow, Colleen Swanson, and J Alex Halderman. TapDance: End-to-Middle Anticensorship without Flow Blocking. In *USENIX Security Symposium*, pages 159–174, 2014.

[30] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J Alex Halderman. Telex: Anticensorship in the Network Infrastructure. In *USENIX Security Symposium*, 2011.

[31] Xueyang Xu, Z Morley Mao, and J Alex Halderman. Internet censorship in China: Where does the filtering occur? In *International Conference on Passive and Active Network Measurement*, pages 133–142. Springer, 2011.