Technical Report CISPA-TR-2020-03-09-BBCR
on
Slitheen++: Stealth TLS-based Decoy Routing

Benedikt Birtel, CISPA – Helmholtz Center for Information Security
E-mail: benedikt.birtel@cispa.saarland

Christian Rossow, CISPA – Helmholtz Center for Information Security
E-mail: rossow@cispa.saarland

09.03.2020

Contents

List of Figures

List of Tables

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 decoy systems, 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 [1]. Various academic proposals as well as deployed anti-censorship systems aim to survive in an ongoing arms race against blacklisting censors that ban any communication patterns they deem suspicious. 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 such anti-censorship concept that has recently seen significant developments [2, 3, 4, 5, 6, 7, 8]. 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 [9, 10, 11, 12]. Yet, advanced decoy routing systems nearly perfectly imitate normal communication patterns by carefully replacing content of the overt traffic. Slitheen [3] 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 overt target (e.g., a website that is not blocked by the censor). Technically, Slitheen places covert data streams inside overt communication, which provides much stronger resistance to traffic analysis attacks. By deliberately weakening the TLS handshake between the client and the overt website, Slitheen routers extract covert information from the overt communication in this TLS channel. From this hidden data, Slitheen routers can reliably mediate a hidden communication channel between the client and a covert destination.

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 [10, 13, 14] have shown, such traffic analysis constitutes a severe threat in practice. In the special case of decoy routing, a successful attack allows a censor to block evasive communication.

To address this challenge, we augment Slitheen with an upstream component that meets the same security guarantees that Slitheen's downstream channel already offers: An 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.). To solve this, we compress upstream communication in the overt channel to fit upload data in the space gained by compression. We achieve this by deploying an Hypertext Transfer Protocol Version 2 (HTTP/2)-like compression function for Web-based overt traffic, where clients replace Hypertext Transfer Protocol (HTTP) header fields with numbered indexes before sending them to the overt target.

When revisiting design choices of the original Slitheen setup, we identify several inconsistencies between our expectations of the system design and its actual implementation. In fact, we reveal several fundamental bugs at the core of Slitheen's implementation that significantly impact its performance. For example, Slitheen neglects out-or-order packets and silently fails when encountering just a single out-of-order TLS record. Second, Slitheen is based on assumptions on the operating system's TCP stack behavior that do not hold in practice. For example, Slitheen cannot operate on large HTTP request headers. Third, the content in Slitheen's covert channel was unusable, as en-/decryption modes of operation did not match. While these findings do not entirely invalidate the measurements presented in the original Slitheen paper, they demonstrate that Slitheen requires further improvements to be usable in practice.

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 im-

plementation. 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. This is a significant improvement over the basic measurements in the original paper that assumed aggressive communication with a single static overt site. This way, we are the first not only to assess the theoretical throughput of the covert channel, but also to measure its utilization using real covert 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 [1, 15].

We summarize our contributions as follows:

1. 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.

2. We measure Slitheen(++) according to metrics that not only show the maximum theoretical throughput of the decoy systems, but for the first time, also assess the actual user experience by measuring loading times of websites from ten covert targets.

3. 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, although this is not its primary goal, Tor [16] is commonly used to bypass censors. To this end, Tor has been extended to provide protections against various censorship tactics to block the Tor infrastructure, such as Tor bridges (special entry points to the Tor network that are not publicly available to everybody) or traffic obfuscation [17, 18] by imitating non-Tor protocols in so-called pluggable transports. Yet censors can detect such first-generation evasion attempts [19, 20, 21, 10]. In particular, Houmansadr *et al.* showed that censors can use traffic analysis and probing mechanisms to identify even seemingly strongly obfuscated Tor communication.

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* [22]. The basic idea is hiding a service inside a cloud provider such that the establishment of TLS connections to that service look exactly like any other normal connection to that provider. This is achieved by using a provider that offers fronting servers to enter a domain. These frontier servers are contacted by using a regular URL in the TLS handshake, while the real target of the connection stays hidden inside the TLS-secured messages. The fronting server decrypts these message and forwards the covert content to the covert target(s). However, network traffic analysis [9, 10, 11, 12] could reveal that fronting services do not belong to the provider in the given domain carried in the 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 [2, 3, 4, 5, 6, 7]. 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 [3]. 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. A closer look at Slitheen is provided in Section 4. 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 systems is downstream-only decoy routing as proposed by Nasr *et al.* [23]. The main idea is to transmit covert upload data via the downstream overt data. To this end, overt destinations have to offer reflection capabilities and insert (parts of) the upstream messages into downstream messages. For instance, HTTP offers downstream control data as part of 404 error messages that mirror

the requested URL of the original upstream message. The downstream-only decoy concept uses this reflection, leveraging overt targets that place covert upstream messages in the resulting, reflected answer (e.g., as part of the error message). The decoy router will recognize the reflected message and extract the upstream data. This way, downstream-only decoy routing counters attackers that redirect traffic around decoy routers [24]. On the downside, this concept requires an out-of-band channel for bootstrapping and synchronization with new clients. Furthermore, while Slitheen++ uses "normal" overt communication, downstream-only routing risks that overt sites recognize and block the excessive abuse of redirect/error messages.

Our proposed system Slitheen++ provides an alternative to downstream-only decoy routing that requires less effort for synchronization and cannot be easily detected by overt websites. Our idea extends the general concept of Slitheen by stealth upload communication and traffic schedulers. Moreover, we address several problems in the original Slitheen implementation, which for the first time allows a realistic estimate of the practicality of decoy routing systems like Slitheen.

3 Adversary model

We assume that the censor is a state-level omniscientist adversary [10], that has passive, active and reactive networking capabilities. The classification "omni-scientist" represents a censor with rich computation and storage capacities to perform traffic analysis. That means that the adversary can store network traces over a longer period of time, e.g., to perform network traffic analysis or to identify certain characteristics that tell the adversary which endpoints/connections should be blocked.

We leave active attacks against decoy routing, such as Routing Around Decoys (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 [25] that are compatible to Slitheen++, and would allow our system to handle asymmetric routing and to withstand RAD attacks.

4 Background: Slitheen

Slitheen is a traditional decoy routing system proposed by Bocovich and Goldberg in 2016 [3]. In general, decoy routing uses special tagged flows routed towards non-censored *overt* targets. Special forwarding devices, so-called decoy routers, are placed between the client and the overt target. Decoy routers recognize tags and forward traffic to a *covert* target, which is specified using the resulting covert channel. However, this redirection changes the traffic characteristics of the overt communication, which enables detection by advanced censors.

To solve this problem, Slitheen adapts decoy routers, such that the overt traffic carries the data to the censored host, while keeping its patterns. Technically, Slitheen clients multiplex overt and covert traffic, and Slitheen decoy routers demultiplex the two individual data flows and send them to their respective targets. Slitheen places traffic inside encrypted Hypertext Transfer Protocol Secure (HTTPS) connections. This way, the fingerprint of the overt website does not change, and only client and decoy routers can identify that the content has changed. Covert upload data is placed in overt requests sent by the overt application towards the overt destination. Decoy routers will recognize such tagged flows, and consequently, extract and forward the covert data to the covert destination. The main ingredient that distinguishes Slitheen from previous approaches is that the overt communication will be performed as usual, and covert downstream data will be placed inside this overt downstream data. To this end, Slitheen replaces static leaf content of Web sites, i.e., leaf elements in the Document Object Model (DOM) tree (e.g., images), with covert data. Leaf content can typically be replaced without causing any changes in the communication interaction. For example, a picture is leaf content that can be replaced. This way, the overt communication does not change its communication pattern, which makes Slitheen's downstream communication resistant to traffic analysis attacks. To ensure that covert data stays in order and can be assigned to individual covert communications, all covert traffic is prefixed with a Type-Length-Value (TLV)-encoded Slitheen header. The Slitheen entities add those headers when they multiplex covert data and remove the headers when they demultiplex covert data.

Figure 1 shows the basic Slitheen setup. The client side consists of a *covert application* and the *Overt User Simulator (OUS)*. The covert application (e.g., a browser) wants to exchange data with a censored resource. To realize this, the OUS provides a multiplexing functionality, which will place and extract covert data inside overt streams. The base component of the OUS is the headless PhantomJS browser. Finally, the *relay station* acts as decoy router and extracts/inserts covert data from/into the overt upload/download. Furthermore, it establishes a connection to the specified covert target. The extracted upload data is forwarded to the covert target and the received download data gets queued such that whenever replaceable, overt content is replaced by the covert downstream data. If no such data is present, Slitheen inserts random padding bytes.

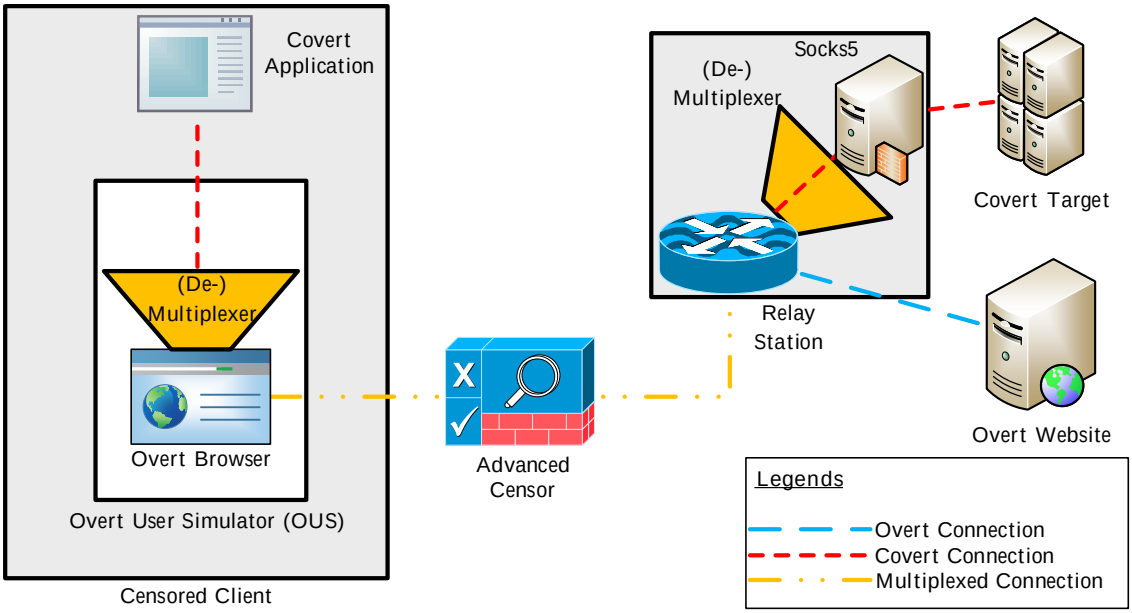

Figure 1: Slitheen Setup: Basic overview with all its components

Relay Station Implementation: Slitheen decoy routers use three different types of threads. The first one, a singleton entity named "Overt Upstream Thread (OUT)", demultiplexes covert messages from tagged flows and assigns them to the corresponding covert connection when forwarding overt upstream. The second type, a singleton named "Overt Downstream Thread (ODT)", multiplexes covert downstream data on leaf contents in tagged flows when forwarding overt downstream. Both need to maintain state, e.g., session keys for overt TLS streams computed from the TLS handshake messages. Third, a "Covert Thread (CT)" performs all networking communication with a specified covert target. Every new covert connection causes the creation of a new covert thread.

5 Slitheen++

Slitheen introduced a novel idea to overcome strong censors that deploy advanced network analysis capabilities [3]. 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.

5.1 Upstream Communication

Slitheen's communication is trivial to identify via traffic analysis. On the one hand, Slitheen leaves censors no chance to reveal deviations in the *downstream* communication, as leaf content of overt communication is replaced with equally-large covert messages. On the other hand, Slitheen neglects the fact that censors could also inspect the *upstream* communication. In fact, Slitheen clients simply append covert upstream data to overt upstream, which significantly increases the upstream volume. Such a drastic change in the communication profile forms an easy target for traffic analysis.

In Slitheen++, we augment Slitheen's concept with a stealth data upstream channel. We transmit covert upload data by compressing HTTP header fields in overt requests and using the gained space to place covert upload data in overt requests. While a similar idea was mentioned by the Slitheen authors, its drawbacks and technical difficulties have never been tested. Technically, we added a HTTP/2-like compression to the OUS. The basic concept of HTTP/2 header compression proposed the usage of a TLV encoding for headers as well as the usage of tables to store already seen header key/value pairs. If such a pair with the same value is transmitted again, a reference to the table entry is inserted such that no retransmission of the whole key/value pair is needed. Thereby, the header size is reduced and covert upload data can be inserted. We avoided additional GNU ZIP (GZIP) compression since research on HTTP/2 revealed that its us-

age makes TLS connections vulnerable to the Compression Ratio Info-leak Made Easy (CRIME) attack [26].

5.2 Scheduling

So far we have abstracted from the technical detail that covert communication actually spans several concurrent connections, as browsers request resources in parallel via several TCP streams. In practice, all covert connections compete for transmission capacities. Slitheen has no dedicated scheduler and extracts covert upload and download data in the order it was inserted into a 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. We introduce two schedulers, a First In First Out (FIFO) strategy and a SRR scheduler. Every scheduler maintains a queue per covert connection. The FIFO scheduler uses a FIFO list of covert connections. The connection at the head of the list is allowed to transmit data until all its data has been sent or until the connection gets closed. If one of the conditions is met, it is removed from the FIFO list and the following connection is allowed to transmit data. New connections are added to the end of the list.

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. The connection at the first place can transmit data, and when its slice is exhausted, the connection is set to the end of the list and its slice is refreshed. After that, the new head of the queue is allowed to transmit data.

5.3 Crawling And TT

A fundamental requirement for decoy routing is that clients create sufficient overt traffic. To this end, Slitheen clients pick a domain and constantly re-visit a predefined Web page of this domain as an overt site (e.g., Wikipedia's `/index.html`). This pattern, however, may be recognizable by censors. 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. This results in more diverse web content that is less susceptible to traffic analysis.

Another aspect regards the frequency in which Web sites are visited. Slitheen repeatedly reloaded the overt website without any "TT" between two Web page requests, i.e., showed an overly aggressive request behavior. This approach increases the available bandwidth of the covert channel massively. In contrast, adding delay ("TT") between two requests to the overt site represents more realistic human behavior. In its current implementation, Slitheen++ supports a fixed TT. This concept can easily be expanded to models that perfectly reflect user browsing behavior (e.g., by replaying browsing sessions).

5.4 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. In practice, however, TCP segments are not guaranteed to arrive in order. This makes it impossible for Slitheen to decrypt such reordered TLS records. In fact, the number of decipherable records can decrease to zero. This is based on the fact that TLS [27] assigns sequence numbers to TLS records. TLS implementations maintain such numbers for each TLS connection both for reading and writing. After sending/receiving a TLS record, the sequence number on the associated state is incremented. 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. Worse, when receiving an out-of-order TCP segment, we cannot determine if this segment represents a new TLS record or belongs to an existing record (e.g., that did not fit into a single TCP segment). 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 7.2. For now, we mitigated the problem by augmenting Slitheen with a mechanism to handle out-of-order TCP segments using a *traffic server* [28]. The traffic server will order the application data of the overt target to restore the correct order of out-of-order segments. To this end, we place the traffic server between relay station and overt target. The distance to the relay station is to be kept as small as possible such that no reordering occurs.

5.5 TLS Based Improvements

Invalid Nonces: TLS is fundamental for the functionality of Slitheen and protects against eaves-

droppers. 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, however, is dangerous since the explicit nonce was used by Slitheen to re-encrypt messages, and finally, transmitted back to the client. Slitheen does thus not guarantee that the nonce will change for different records of the same connection, which undermines TLS's security guarantees. For example, in Cipher Block Chaining (CBC) mode, the IV must be unpredictable. If a counter mode based encryption is used, a nonce *must not* be reused with the same key. Slitheen may break both requirements, plus 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: Furthermore, 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. As both assumptions are not guaranteed in practice, such imprecise handling causes a significant reduction of covert bandwidth. To mitigate this problem, we used the traffic server to create TLS records which do not exceed a maximum preconfigured size.

5.6 Implementation Errors

So far, we have discussed fundamental adaptations to Slitheen's original concept. We will now address several inconsistencies in Slitheen's implementation:

SOCKS Proxy Connection Termination: Slitheen's covert application uses a SOCKS proxy to establish and maintain a connection with the relay station. In Slitheen, when this SOCKS proxy noticed TCP disconnects from the server, it failed to provide appropriate feedback to the client. This effectively forced the client to time out if the server closed a connection. We improved its capabilities and transmit TCP FIN messages from either site (client or server) correctly.

Multiple TLS Upload Records per TCP Segment: Slitheen was not designed to process multiple TLS upload records that occurred in a single TCP segment. Instead, just the first record was processed, and subsequent records in that segment were ignored. This led to data loss if the remaining records carried covert upstream data. Instead, we loop over the upstream data until the entire payload of a segment has been inspected.

Mixed Encryption Modes: In Slitheen, all covert downstream data is encrypted on the relay station and later decrypted by the OUS. Unfortunately, Slitheen uses two different modes of operation for the symmetric de-/encryption. Whereas the relay station encrypted covert data in CBC mode, the OUS failed to decrypt it using Electronic Code Book (ECB) mode. Obviously, this mangled the covert data in such a way that it is completely unreadable to the covert client. Consequently, any attempt to read covert data fails. To fix this problem, we could have aligned the en-/decryption modes, yet we decided to drop this additional encryption. The purpose of encrypting the covert data was to mitigate an attack [25] where an additional adversary (not the censor itself) can read incoming covert traffic at the relay station. Furthermore, this adversary can manipulate multiplexed traffic when it leaves the relay station towards the censored client. The preconditions include: the covert traffic is unencrypted, and a mode of operation for the symmetric cipher of the overt connection is used which forbids the reuse of nonces when encrypting messages. Now, the relay station performs such a re-encryption when it multiplexes data into an overt downstream channel. Then, the adversary can manipulate covert data inside the multiplexed overt connection without any recognition by the covert client. 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.

Uninitialized Data Usage: We encountered variable definitions in the code where data was left uninitialized. This caused random segmentation faults, breaking some experiments as the relay station or the OUS SOCKS component stopped working. We properly initialized all variables to remove these problems.

Statically-Sized Buffers We identified two issues regarding buffers used in the relay station and in the SOCKS components. First, Slitheen uses fixed-size buffers to temporarily store messages. However, it frequently happens that the buffers do not have the capacity to store larger messages, leading to segmentation faults and a loss of covert data that has not been processed—possibly even allowing for remote code execution attacks. Second, the

SOCKS component could not forward more upload data than a single static buffer provided. If more data was present, the SOCKS component was aborted and upload data was lost. We thus added guards and use dynamically-sized buffers.

Inconsistent TLV Streams: Beside the inconsistent management of tagged flows, there was also a problem with the TLV encoding itself. The original Slitheen header was 16 bytes long and indicated how many bytes of user data and garbage were present in the current message. The authors assumed that they would always have enough space in any TLS record such that they could place a header. If this was not the case, the remaining bytes of resource would be interpreted as a Slitheen header. That broke the TLV stream, resulting in a complete downstream data loss. To solve the problem, we extended the header by a field that indicates what kind of Slitheen message follows. This way, we can also define a No Operation (NOP) message that tells the TLV decoder that no Slitheen message will follow.

HTTP Parsing Issues: The Slitheen prototype uses a simple but error-prone technique to parse HTTP headers at the relay station, i.e., it searches for a given string in the currently received HTTP message. Furthermore, Slitheen keeps an HTTP state that is updated based on the identified header values. Unluckily, the string search was not bound to the header of an HTTP message. If a certain keyword was also present in the body of a message, it was possible that the HTTP state machine was never updated. This resulted in an endless loop, such that the relay station was no longer able to forward and replace any downstream data. To avoid this, we remove such endless loops, but ultimately suggest replacing the custom string search with an HTTP parser.

6 Evaluation

We now evaluate Slitheen++ with regard to resilience against traffic analysis attacks, communication delays it introduces, and its offered covert transport capabilities.

6.1 Experimental Setup

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. Table 1 shows the ten scenarios of covert domains for which we evaluate Slitheen++. For each of these scenarios, we use Slitheen++ to load three URLs. 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 several setups which set the following parameters that Slitheen++ uses to operate:

Scenario-Nr.	Webpage
0	Twitter
1	Instagram
2	Google Play Store
3	Apple Store
4	Google News
5	BBC
6	Reddit
7	Stack Overflow
8	GitHub
9	Google Code

Table 1: Overview of the ten scenarios used to test Slitheen++.

1. **Overt Site**: Defines the starting URL for the OUS to use for its overt communication. Choosing the overt site has a large influence on the performance of Slitheen++, as the site determines up- and downstream characteristics. Unless otherwise stated, we will use the English "Computer Science" article of Wikipedia as our overt URL.

2. **Stealth Upload (SU)**: A boolean that describes whether we use header compression and use the space gains to place covert upload data inside overt upload data, or if we use the original authors' concept of inserting complete covert upload messages in overt data regardless of any header compression. We differentiate between the original implementation's idea of naive upload insertion by labeling such test cases with an "S", and our concept of using header compression by labeling such tests with "S++".

3. **Scheduling**: Defines the scheduling algorithm for covert data. The original Slitheen software places covert data in the order in which it has been stored, potentially leading to connection timeouts. We compare this "None" scheduler with our two schedulers, i.e., the SRR and FIFO schedulers (see Section 5.2).

4. **TT**: We vary the TT to model a user who loads a website and looks for information, instead of immediately loading the next page. TT is defined as an idle period where the OUS will wait before calling the next overt URL after the current load of the actual overt URL has finished. Without TT, Slitheen++

requests overt resources using the constant-polling strategy of Slitheen. This naive strategy loads the next overt site as soon as the previous load has finished. Loading all resources of wikipedia's Computer Science article takes an average of 838 ms on our system.

5. **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 setups comparable to those in which Slitheen was evaluated, and evaluate each scenario ten times per setup to stabilize results.

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 web browser for all experiments.

6.2 Evaluation Metric

This subsection defines three metrics to evaluate the stealthiness and performance of Slitheen++.

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. While content-wise the overt channel remains identical—namely exactly the same number of random bytes, due to encryption—we have to test whether Slitheen++ introduces significant delay to the overt communication. Multiple steps happen within Slitheen++ to achieve covert communication besides the normal forwarding, such as compressing contents, replacing them, and so on. Therefore, at the relay station, we measure the forwarding times for overt upload and download data. 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.

While the Slitheen authors aimed for a similar evaluation, their underlying metric did not reflect the censor's capabilities. In Slitheen, they measured the time the PhantomJS headless browser in the OUS requires to load an overt site. However, this measurement not only considers the networking delay, but also includes the time the PhantomJS needs to load and process the page. A real censor could, however, only measure packets' delays when they traverse their domain.

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 by recording the compression and leaf content capacities on the covert communication channel, and relate this to the goodput that the resulting covert channel actually generates on the overt communication.

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. To determine if all resources have been loaded, we observe 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 decoy routing.

6.3 Testing Execution and Results

Our evaluation is divided into three phases. Phase 1 measures the influence of adding scheduling algorithms and stealth upload. Phase 2 evaluates the changes due to TT for the best scheduler identified in the previous phase. Phase 3 evaluates if crawling an overt website changes the system's performance.

Phase One: Scheduling and SU Evaluation We first measure the performance of our schedulers, as well as the impact of using SU.

Loading Times: Figure 2 shows the loading times of the ten scenarios (separated by vertical lines) and four setups and relates them to the baseline (each leftmost measurement per scenario, labeled "Base"). The labels for each covert setup follow a simple naming scheme. "S" represents the original, naive upload insertion, while "S++" denotes tests that use the stealthy variant of covert upload transmission. The scheduler used in each setup is shown by the suffix that follows a minus in the label ("None", "SRR", or "FIFO").

In general, the usage of the decoy routing system causes a slowdown in the loading performance of the overt application. The slowdown can vary overall from a median factor of 2.2 to 6.1, compared to the baseline. The resulting absolute timings stay within acceptable bounds to load a web page's resources, as indicated by the medians, which range from 3 to 14 seconds for covert setups. One core explanation for this slowdown that is common to all

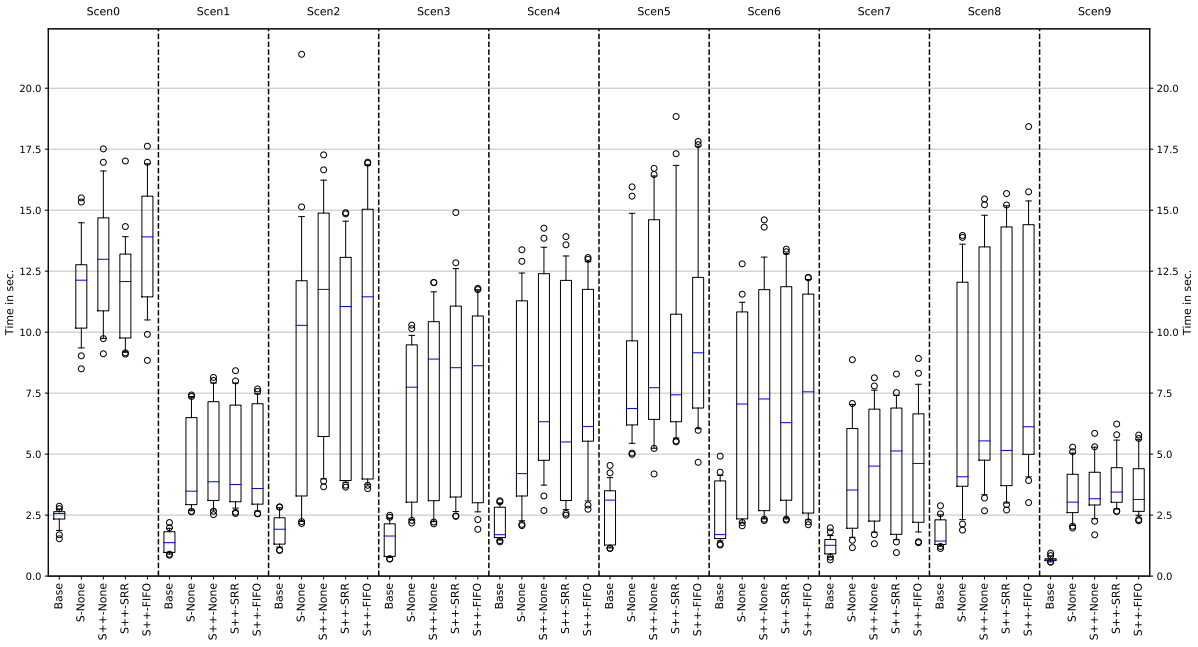

Figure 2: URL resource loading times for phase one evaluation grouped by scenarios

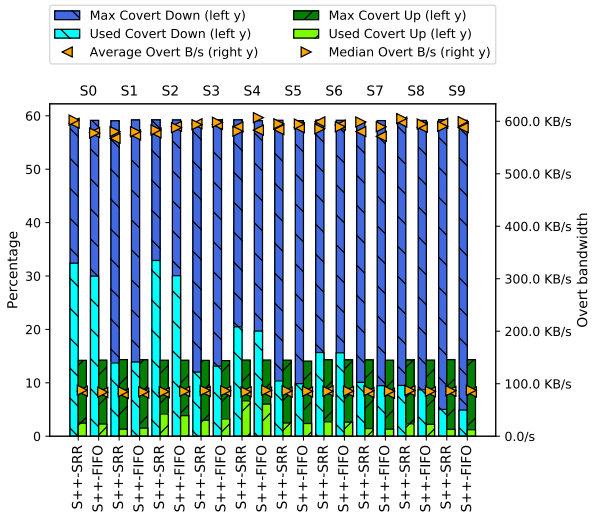

Figure 3: Maximum covert goodput and used covert goodput based on total overt goodput in phase one (left bar downstream, right bar upstream).

(non-baseline) setups is the bottleneck in the covert downstream. Large resources (multiple megabytes) at particular websites take several complete overt resource loads until the covert data is transmitted.

When comparing the two upload concepts, naive and stealthy, the latter one can cause additional bottlenecks in uploads. Thus, loading covert sites using SU slows down slightly by 0.96 seconds on average. The added scheduling strategies partially make up for this slowdown (see scenarios 0-2 and 6). In almost all stealthy upload scenarios, the None scheduler performs worse compared to our schedulers. In some cases, scheduling even helps the stealthy upload concept to perform better than the naive design. Scenario 6 is the best example to demonstrate its positive effect, since the median is truly below the None scheduler's, naive upload concept. In scenario 0, the box plot of the SRR scheduler also shows the positive scheduling effect, indicated by the lower Q1 value, while the median also is slightly lower than that of the original concept. The main reason for this difference is that web servers (or anti-Distributed Denial-of-Service (DDoS) protection services) close idle connections. Schedulers address this problem and feed every connection stepwise, which improves the utility of HTTP keep-alive connections. This reduces the necessity for the covert application to create new connections due to shutdowns of idle connections.

Covert Goodput: Looking at the covert channel utilization provides further insights into why loading times increased. Figure 3 shows the percentage of the maximum covert goodput (data that could have been replaced) and the used covert goodput (data that actually was replaced) compared to the complete available overt goodput. Since we load all resources from a single overt URL, the maximum covert goodput stays rather stable. More downstream than upstream goodput can be used, as covert upload requires compression of relatively short HTTP headers, while downstream the comparatively large leaf content is replaced. Figure 3 also shows the average/median of the overt bandwidth (right y-axis), which stay roughly con-

stant in this experiment.

Interestingly, neither the full upload nor the full download capacities were used. This observation highlights the need to measure actual utilization instead of solely relying on theoretic goodput (such as in Slitheen's evaluation). A closer inspection revealed that the relay station does not have a continuous overt up- and downstream flow at all times. Due to the request/response pattern in HTTP, the decoy router is *either* sending upload *or* receiving download data, but rarely both at the same time. This leaves replaceable data unused, which is indicated by the reduced covert transport utilization. For the interested reader, Appendix B illustrates and discusses this behavior in detail. Furthermore, the OUS might need to process data, during which no data (neither overt nor covert) is transmitted. This can add a delay to covert transmissions that get passed from the OUS to the relay station and vice versa. The added latency rises up to several hundred milliseconds, which significantly influences loading times.

OFC: Finally, we measured the OFC separately for download and upload. We considered the percentage of packets forwarded in a specific direction that exceed a certain delay (1 ms to 50 ms). Tables 2 and 3 show the fraction of delayed packets downstream and upstream, respectively. For example, considering Table 2, the vast majority of downstream packets (2nd column) have little or no delay. In the worst case (3rd column), a single scenario showed 0.0029% of packets with a delay of more than 1 ms, which we believe is a negligible fraction that does not allow for practical traffic analysis attacks. In fact, only 20% of all tests (4th column) exceed a delay of 50 ms for any packet observed during the entire experiment.

Delay D	↓ Avg(D)%	↓ Max(D)%	Setups involved %
$>$ 1 ms	0.0014	0.0029	37.5
$>$ 5 ms	0.0000	0.0000	0
$>$ 10 ms	0.0000	0.0000	0
$>$ 30 ms	0.0000	0.0000	0
$>$ 50 ms	0.0000	0.0000	0

Table 2: Phase One Download OFC Evaluation

Delay D	↑ Avg(D)%	↑ Max(D)%	Setups involved %
$>$ 1 ms	0.0081	0.0174	95
$>$ 5 ms	0.0058	0.0139	87.5
$>$ 10 ms	0.0056	0.0139	87.5
$>$ 30 ms	0.0030	0.0072	70
$>$ 50 ms	0.0010	0.0021	20

Table 3: Phase One Upload OFC Evaluation

The downstream computations add less overhead than the upstream operations. The upstream thread performs more complex operations than the download thread. It initiates various elements in data structures and creates new CTs, while the ODT takes advantage of the work done by the OUT (e.g., it merely needs to read data structures). Furthermore, we realized that the upstream thread has more conflicts with covert threads when competing for processing resources. In the worst case, several CTs start to perform computations at the same time that the upstream thread is also forwarding overt upload data. The scheduler of the Linux Operating System (OS) tries to assign fair processing times to all threads. Since the upstream thread spends a lot more time performing computations, the covert threads are more likely to be scheduled when they change their state to "running". Under certain conditions, this can cause high latencies. We mainly found these exceptions either when a lot of covert threads had been spawned before or when covert data arrived at them. During this phase, we found some single exceptions up to 72 ms. For the overt downstream thread, the highest peak was 4 ms, indicating that there is no comparable resource conflict in the downstream communication.

Summary: Stealth upload ensures that the censor cannot detect any unusual overt upload message sizes, but it also decreases the amount of available covert upload. Compared to the Slitheen++ None scheduler setup, the schedulers reduced the loading times in almost all cases, reducing (or even closing) the performance gap compared to the non-stealthy Slitheen setup. Loading times of the covert web pages are acceptable. We identified that forwarding covert messages from one Slitheen end to the other has delays, caused by all stages that need to be passed to reach the censored entities, which in turn increase the loading time. This problem is amplified by the fact that overt carrier generation is independent of the covert data. An improvement to this could be a mechanism that generates overt traffic that is tailored to the needs of the currently-queued covert data. The stealth upload also causes a decrease in performance, especially for resource-hungry websites.

Phase Two: TT Evaluation The measurements so far have assumed an aggressive reloading of the overt website. Constantly polling the same overt website might be detectable by a censor. Consequently, we now test the impact of adding TT between two overt requests. For conciseness, we focus on Slitheen++ and choose (per scenario) the overall better-performing scheduler (SRR or FIFO). We compare the results with the baseline as well as the outcomes of phase one for the individual scheduler used.

Loading Times: Figure 4 shows the resulting loading times for each scenario using the best-

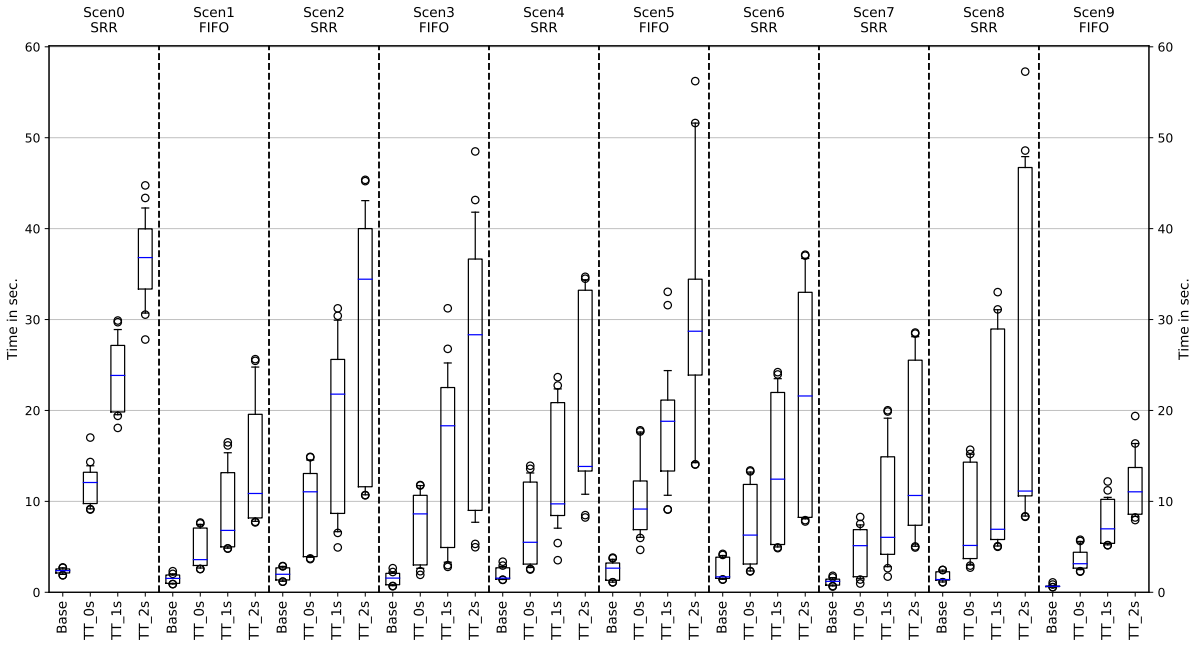

Figure 4: URL resource loading times for phase two evaluation grouped by scenarios

performing scheduler (see label at top). The x-axis varies the TT between zero, one and two seconds, respectively. Not too surprisingly, idling increases the loading times. On average, the median for a TT of 0 is 7 seconds, increasing the idle time to 1 second leads to an average of 13 seconds for the medians, and another additional second of TT raises the average of the medians to 21 seconds.

The division of the median of a current covert setup by the baseline median, named "Baseline-to-Covert factor" in the following, shows a linear growth of the median for the first seven scenarios. Considering the TT, the Baseline-to-Covert factor is approximately (1 + ¡number of seconds TT¿). In other words, every second of TT increased the Baseline-to-Covert factor by 100% compared to the corresponding setup with no TT. For instance, the Baseline-to-Covert factor for scenario 3 is approximately 6. Increasing the TT by 1 second, the Baseline-to-Covert factor rises to 12; adding another second leads to a factor of 18.

Furthermore, the idle times spread the distribution of loading times, in particular in scenarios 2, 4 and 8. TT generates idle transfer periods where no covert data will be transmitted. That increases the delay needed to transmit messages from one Slitheen++ endpoint to the other. Aggressive DDoS protection mechanisms can amplify these problems, as TT increases chances that a covert connection stays idle for several seconds because of (i) more covert connections competing for less available transport capabilities, and (ii) a general worst-case offset added by the TT in combination with the implicit idle times when the overt browser takes its time to perform computations.

Covert Goodput: Figure 5 shows the maximum and the actually used goodput for the longest TT period and the corresponding, previous test without TT. In almost all tests, longer TT increases the fraction of covert goodput that can be used. The main reason is the idle periods in which covert data gets queued for later transfer, which reduces the length of periods where overt data is transmitted without a covert payload. In contrast to phase 1, the computed average bandwidth and the median of bandwidth show an extreme diverse behavior. Even though the covert bandwidth average has significantly decreased, the median is even smaller compared to a run without TT. Using a TT creates a chainsaw pattern of overt goodput. The average is influenced by the high number of outliers, which makes the median the preferred value for a comparison.

OFC: OFC are similar to phase one. We had a single outlier in one setup where the upload delay reached 61 ms, likely due to thread scheduling. For completeness, we list the overviews of OFC in Appendix A.

Summary: As expected, introducing additional delays using TT increased the overall loading times. It takes longer to transport covert data, because the idle times between overt resource loads increase dramatically. A TT of 2 seconds causes an unacceptable user experience. To this point, TT has shown that it comes at a cost, while it also provides a much better user overt loading behavior; in particular, lower TTs of 1 second seem viable.

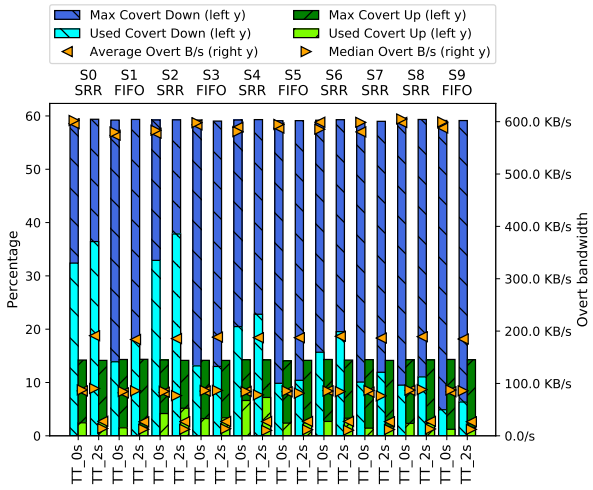

Figure 5: Maximum covert goodput and used covert goodput based on total overt goodput in phase two

Phase Three: Crawling Evaluation The previous two phases used a single static overt website, which might cause traffic patterns suspicious to a censor. We now add crawling to the setups of phase two to gauge its impact. We again evaluate this on both Slitheen++ schedulers and, for brevity, report on the overall better one per setup.

Loading Times: Figure 6 shows the loading times for the individual scenarios. Following our observations in phase two, 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 in phase two, and "Crawl" using crawling.

Generally, adding crawling has less negative impact on the covert loading 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. The Baseline-to-Covert factor varies from 3.7 to 8.5 without TT. The average of the medians for the naive variant is 7 seconds, while it becomes 8 seconds when crawling is enabled. 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. The resulting Baseline-to-Covert factor for crawling with a TT of 1 second varies from 7.6 to 21.4. Known from phase 2 evaluation, the average of the medians when TT is used without crawling is 13 seconds, while the usage of crawling increases it to 19 seconds.

Covert Goodput: The total amount of goodput fluctuates due to the fact that the crawled URLs varied in their goodput capabilities. Figure 7 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 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). A closer look at the overt average and median bandwidth also indicates that the overt bandwidth is reduced compared to earlier phases without crawling. 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 DDoS protection mechanisms close the connection before any covert goodput gets transmitted. 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: The OFC for downstream show a slight increase compared to the two previous phases. Almost all setups encountered delays between 1 ms and 4 ms, but not above this. The average for this delay class reached 0.0029%; the highest value ever measured was 0.006%. During upload, the fraction of packets that were delayed between 1 and 5 ms reached 0.0149%; the maximum value is 0.0248%. All other uploading delay classes showed a slight decrease. 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. In general, we have a slight increase in some of the OFC. Yet the maximum delay added to downstream traffic is 4 ms. One upload outlier was around 60 ms; all others stayed under 60 ms. Full details are provided in Appendix A.

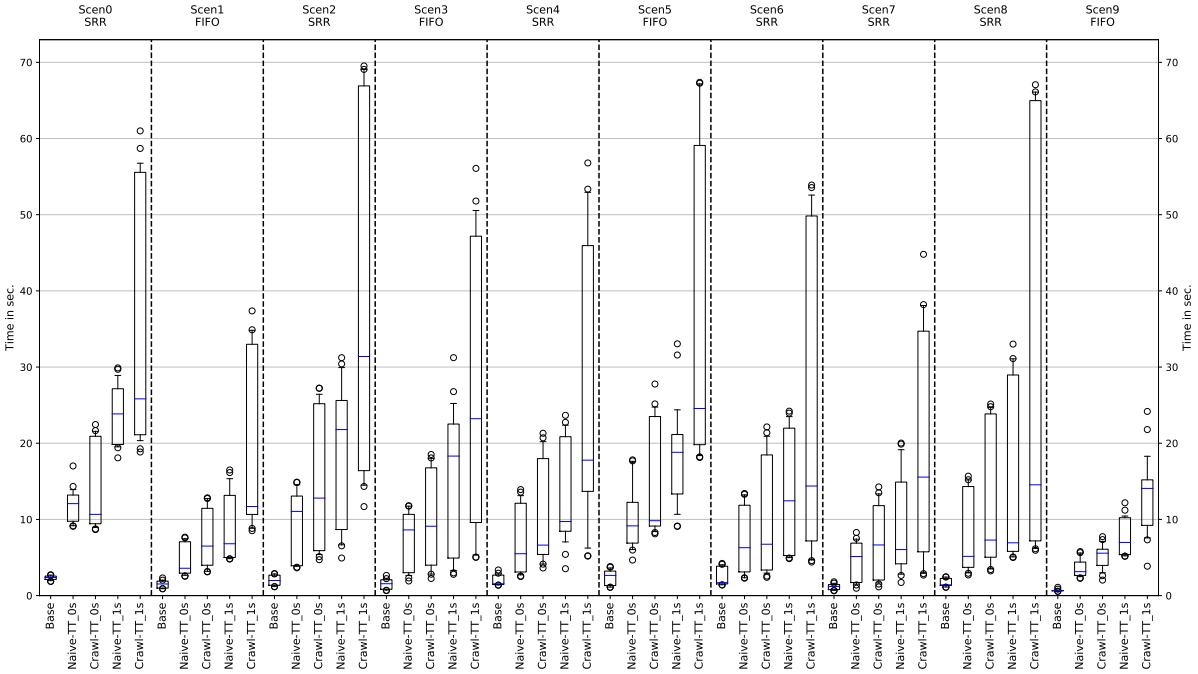

Figure 6: URL resource loading times for phase three evaluation grouped by scenarios

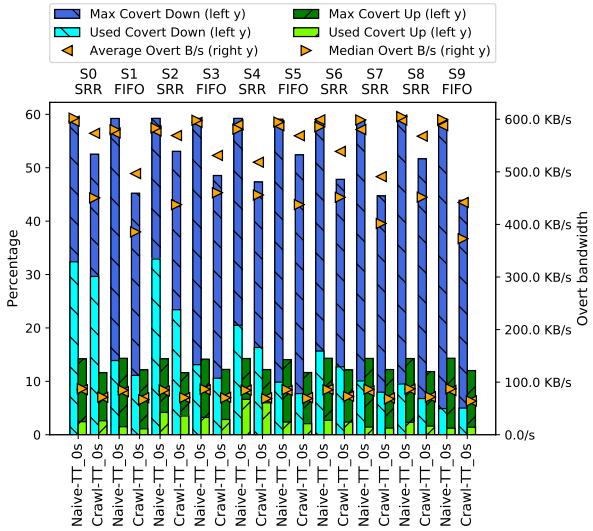

Figure 7: Maximum covert goodput and used covert goodput based on total overt goodput in phase three

Summary: We think that crawling is vital to mitigate the risk of being detected, even if the usability for a censored client is reduced. Crawling generates a more realistic overt behavior, close to real user behavior. The previous two phases used the exact same URL and loaded all its resources repeatedly, which gave us a fixed amount of covert goodput per load. By using crawling, we have fluctuations in the covert goodput, which makes the system's covert goodput unpredictable. The experiments show that the system's usability decreases when we try to stay under the radar of a censor. To mitigate this problem, one could easily build a whitelist of overt URLs (or URL patterns) with particularly useful replacement capabilities. We refrained from doing so in order to provide reproducible measurements.

7 Discussion and Future Work

We now discuss limitations of Slitheen++ and describe our future work directions to address them.

7.1 Overt/Covert Target Requirements

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 forces the client to establish multiple connections to load a single resource. In the worst case, if the delay between opening and actually using a covert connection is too long for the web server, the client is never able to load any resource.

7.2 Replacing the Traffic Server

In 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 5.4, 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. For example, consider the send and receive counters that the normal TLS stack maintains. Slitheen++'s own 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.

7.3 Increasing User Experience

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 due to suspicious overt traffic patterns. 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. For example, crawling can be adapted to use a predefined list of URLs which guarantee higher covert goodput than our Breadth-First Search (BFS). Furthermore, we could adapt the crawler to generate the overt carrier based on the current covert queue state. Finally, mixing user applications for Slitheen++ could also increase the available bandwidth, such as video/music streaming or online games [29].

Third, a replacement for the PhantomJS as overt browser in the OUS would also help. We recognized that a modern browser like Google Chrome or Firefox has many more features implemented than PhantomJS. PhantomJS relies on the outdated QtWebKit and does not use the already available, more advanced QtWebEngine [30]. Furthermore, Chrome and Firefox support a headless running mode, which shows much better performance compared to PhantomJS [31]. We used a simple test where we crawled 20 pages belonging to Wikipedia, starting with the Computer Science article. Our own measurement indicated that a headless Chrome is 51.9% faster on average than PhantomJS. Interestingly, the headless Chrome sent more requests than PhantomJS. Mostly, PhantomJS did not query resources in link tags (e.g. stylesheets), so Chrome sent 37.9% more requests. In essence, we believe that the usage of Chrome or Firefox in the OUS would improve the performance of Slitheen++, while at the same time also making the latest web features available.

8 Conclusion

Using the general idea of Slitheen [3], for the first time, we created a software prototype that can be used to evaluate Slitheen(++) in a representative way. We showed that merely measuring the ratio of replaceable content can only partially capture the actual performance and stealthiness of decoy routing prototypes. Our enhanced metrics showed that—despite decent goodputs—Slitheen clients face higher loading times. A major factor that increases the loading times is the latency of Slitheen's end-to-end transport. Overt non-data transmission periods and limited covert goodput per overt URL in combination with rare covert bidirectional transmission phases cause additional delay.

We furthermore assessed how a stealthier design affects performance. In contrast to Slitheen, Slitheen++ blends covert upload into overt upload, without which censors can trivially detect Slitheen. We measured how Slitheen++ performs in settings where the overt communication mimics real users that do not just repeatedly visit the same URL constantly, but rather (i) have delays between requests (TT) and (ii) follow links on web pages (Crawling). While this represents a state-of-the-art and convincingly strong stealth setting, the loading times to fetch a URL significantly increased compared to naive setups without decoy routing. This calls for follow-up research that reduces the loading times and thus increases user experience of decoy routing clients.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[26] Thai Duong Juliano Rizzo. The CRIME attack. Available online at `https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit#slide=id.g1d134dff_1_222`; visited on 2018-03-07.

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

[28] Apache Software Foundation. Traffic Server. Available online at `http://trafficserver.apache.org/`; visited on 2018-03-07.

[29] 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.

[30] Lars Knoll. Introducing the Qt WebEngine. Available online at `http://blog.qt.io/blog/2013/09/12/introducing-the-qt-webengine/`; visited on 2018-04-17.

[31] hartator. Headless Chrome vs PhantomJS benchmark. Available online at `https://hackernoon.com/benchmark-headless-chrome-vs-phantomjs-e7f44c6956c`; visited on 2018-04-16.

Delay D	↓ Avg(D)%	↓ Max(D)%	Setups involved %
> 1 ms	0.0009	0.0015	30
> 5 ms	0.0000	0.0000	0
> 10 ms	0.0000	0.0000	0
> 30 ms	0.0000	0.0000	0
> 50 ms	0.0000	0.0000	0

Table 4: Phase Two Download OFC Evaluation

Delay D	↑ Avg(D)%	↑ Max(D)%	Setups involved %
> 1 ms	0.0078	0.0174	96.67
> 5 ms	0.0059	0.0139	86.67
> 10 ms	0.0054	0.0139	86.67
> 30 ms	0.0022	0.0070	73.33
> 50 ms	0.0010	0.0012	10

Table 5: Phase Two Upload OFC Evaluation

Delay D	↓ Avg(D)%	↓ Max(D)%	Setups involved %
> 1 ms	0.0029	0.0060	95
> 5 ms	0.0000	0.0000	0
> 10 ms	0.0000	0.0000	0
> 30 ms	0.0000	0.0000	0
> 50 ms	0.0000	0.0000	0

Table 6: Phase Three Download OFC Evaluation

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 7: Phase Three Upload OFC Evaluation

A OFC

OFC are interesting with respect to fingerprinting characteristics that a censor can use to identify unusual traffic within an uncensored, legal context. In the main paper, we already listed the detailed OFC data for phase 1. For our experiments in phase 2, Table 4 and Table 5 represent the detailed evaluation results (cf. Section 6.3). Likewise, Table 6 and Table 7 contain the corresponding results for the phase 2 evaluation in Section 6.3.

B Covert Utilization Examples

To illustrate the utilization of the covert channel, we provide explanatory figures that show the fraction of available covert goodput that was actually used to carry covert data. In the corresponding figures, the x-axes represent the timeline. At the same time, the x-axes separate covert down- and upstream numbers, such that the upper and lower half shows the downstream and upstream utilization, respectively. The vertical gray lines indicate when we started and/or finished loading the three URLs we tested per scenario.

Figure 8 shows an example covert utilization in phase 1 for scenario 0, using the SRR scheduler. At the beginning, the covert browser starts to initiate the first connection to load the corresponding Hypertext Markup Language (HTML) root document, followed by loading several resources requires by this document. The high and at places even continuous downstream utilization indicate a bottleneck in the available covert downstream. In this case, another overt site with more replaceable content would be helpful to reduce the loading time. In contrast, the available covert upstream goodput is sufficient, as it is never fully utilized. Consequently, our stealth upload did not cause any significant bottleneck in this example. After the event "Last page finished", there is a small upload spike, caused by a POST request issued by JavaScript code embedded on the loaded twitter.com URL to track users. Figure 9 shows the covert utilization in phase 3 for the same scenario, again using the SRR scheduler and no thinking time, but this time with crawling enabled. Compared to Figure 8, the load of the first URL's resources take longer due to less replaceable content on the overt carrier (on an absolute scale). Only loading the third URL is faster, as the overt carrier provides rich covert goodput at that point in time. Again, bottlenecks can be identified by contiguous time spans of full covert goodput utilizations, especially in downstream. While in some cases the utilization does not reach 100%, this is primarily caused by Slitheen's message fragmentation overhead (e.g., after 29 s). Interestingly, enabling crawling significantly increased the upload utilization. An inspection revealed that the fluctuating number of resources loaded per crawled URL causes fluctuations in the available covert upload goodput. Furthermore, covert upload insertion do not always take place when rich covert upload goodput capacities are available, which causes the increased utilization.

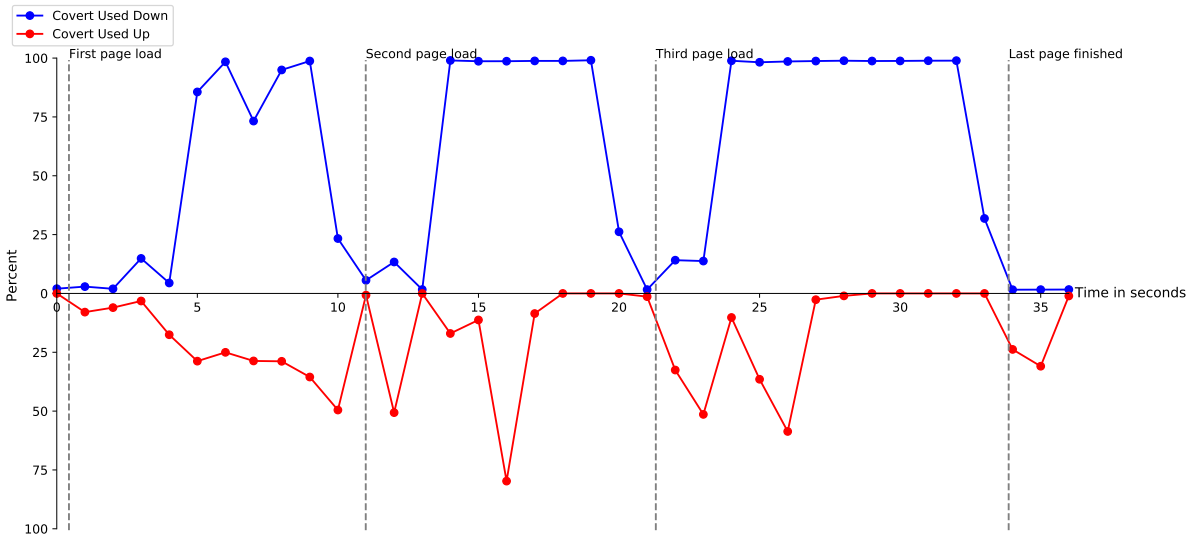

Figure 8: Covert utilization in phase 1 using SRR scheduler for scenario 0 without crawling and no TT

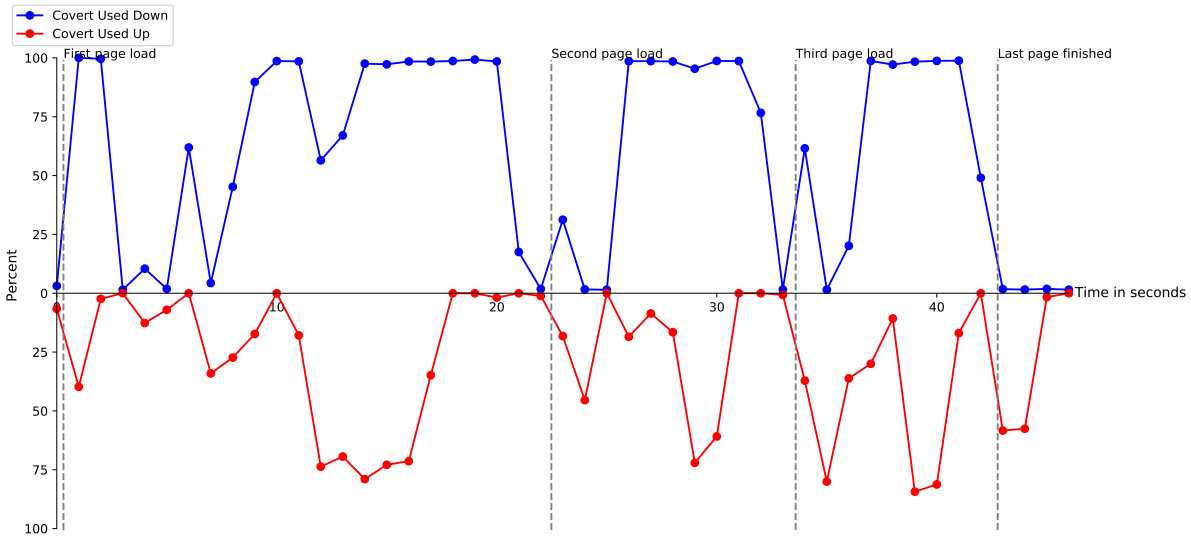

Figure 9: Covert utilization in phase 3 using SRR scheduler for scenario 0 with crawling and no TT