AutoFlowLeaker: Circumventing Web Censorship through Automation Services

Shengtuo Hu*, Xiaobo Ma†*, Muhui Jiang*, Xiapu Luo*‡, and Man Ho Au*
*Department of Computing, The Hong Kong Polytechnic University
†MOE KLINNS Lab, Xi'an Jiaotong University

***Abstract*—By hiding messages inside existing network protocols, anti-censorship tools could empower censored users to visit blocked websites. However, existing solutions generally suffer from two limitations. First, they usually need the support of ISP or the deployment of many customized hosts to conceal the communication between censored users and blocked websites. Second, their manipulations of normal network traffic may result in detectable features, which could be captured by the censorship system. In this paper, to tackle these limitations, we propose a novel framework that exploits the publicly available automation services and the plenty of web services and contents to circumvent web censorship, and realize it in a practical tool named *AutoFlowLeaker*. Moreover, we conduct extensive experiments to evaluate *AutoFlowLeaker*, and the results show that it has promising performance and can effectively evade real-world web censorship.**

I. Introduction

Internet censorship exists almost everywhere [1], and the range of blocking or filtering can vary from an institutional level to a centralized national level [2]. A recent report shows that 34 out of 65 countries participating in the survey has intensified Internet censorship last year [3]. On the other hand, many anti-censorship tools have been proposed to circumvent Internet censorship [4]–[15]. These tools usually involve a helper outside the censored network to assist the censored users to access blocked resources (e.g., websites).

Existing anti-censorship solutions generally suffer from two limitations. First, they usually need the support of ISP or the deployment of many nodes to conceal the direct communication between users and the blocked websites. For instance, Tor [16], a well-known anonymity network that has been widely used to evade censorship, and Lantern [17] have to deploy many nodes around the world. As another example, Telex [7] and TapDance [8] need the support from friendly ISP routers to the network covert channel through HTTPS connections.

Second, although recent studies propose leveraging publicly available services (e.g., Skype [9]–[11], [15], on-line games [13]–[15], cloud-based storage services [12], [15]) to transfer messages for mitigating the deployment issue, their manipulations of normal network traffic may result in detectable features, which could be captured by the censorship system. Note that such detectable features enable the censor to reasonably confirm that a censored user intentionally accesses blocked information. In other words, these tools cannot provide users enough deniability. For instance, SkypeMorph [9] and FreeWave [10] suffer from eavesdropping attack [11], while the Direct-Sequence Spread Spectrum (DSSS) modulation used by SkypeLine [11] is detectable through a SVM classifier [18]. Exploiting on-line games, Castle [14] encodes secret data into in-game commands (e.g., move or set-rally-point commands) through a combination approach and uses a desktop automation tool to execute the commands. Nonetheless, this approach may send unreasonable commands because it enumerates all possible commands, thus leading to observable anomalies. As another example, CAMOUFLAGE [15] uses the *base64* encoding scheme to convert the binary secret data to text, and then puts it in email or file sharing services. However, pure *base64* contents in email or file sharing services are unusual, which could be detected by the censorship system.

To address these two challenging issues (i.e., deployment and stealthiness), we propose a novel framework and develop a practical tool, named *AutoFlowLeaker*, which exploits the publicly available automation services (e.g., IFTTT [19], Flow [20], Zapier [21], Apiant [22], etc.) and plenty of existing web services and contents to deliver messages and circumvent web censorship. The automation service allows users to define conditions for a web service (a.k.a. trigger channel) and specify certain actions in another web service (a.k.a. action channel). If the conditions are satisfied, the automation service will automatically execute the specified actions. For example, a user can define a condition for twitter like "a new message about SRDS" and specify an action for Gmail like "send an email containing the new twitter message to the Chair". Therefore, by exploiting automation services and customizing the condition and the action, we can not only let the information (e.g., messages) be automatically transferred from one web service to another, but also control when and how the information will be transferred.

Our framework, as shown in Fig.1, tackles the deployment issue by employing the automation services to connect the web services accessible to the censored user and the web services selected by the helper, who will forward the requests from the censored users to the blocked websites and send back the responses. In other words, neither the censored user nor the helper need to deploy any customized nodes. Note that the automation services have gained the popularity due to their capability of automating tedious tasks and connecting numerous web services and IoT devices, thus making the censor's economic and social cost of blocking such services

‡The corresponding author.

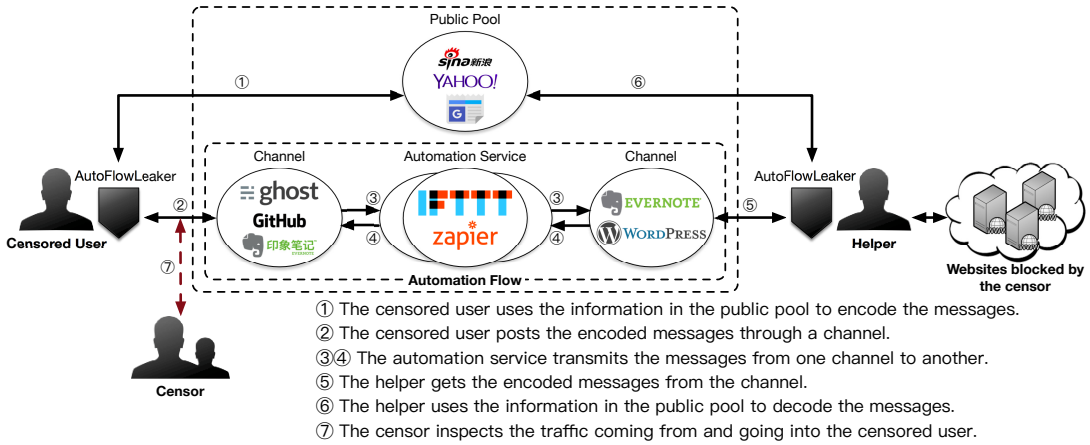


Fig. 1. The censored user first encodes the encrypted messages into web items and puts them on the web services acceptable to the censor. The automation service automatically transmits such items to the helper's web services. After that, the helper recovers the original messages by decoding the items and decrypting the encrypted data. The helper can send the responses from blocked websites to the censored user using the same process.

very high. For example, according to the review of IFTTT in 2016 [23], there are 15 million Applets, which denote the chains of conditions and actions, running on IFTTT. Moreover, more than 326K IoT devices are connected to IFTTT.

To handle the stealthiness issue, our framework adopts two new approaches. First, we propose a novel algorithm (in Section III-C) to convert the encrypted data from censored users or the blocked websites into items acceptable to the censor in the public pool, as shown in Fig.1. For example, the item could be a published research paper or a weather report from a weather website. We use such real items instead of self-generated texts because they can evade NLP-based censorship techniques. Moreover, there are enormous real items with a huge variety, making it difficult, if not impossible, for the censorship system to block all of them. The censored user' *AutoFlowLeaker* will send the items to the web services accessible to the censored user, and the automation services will automatically forward them to the web services selected by the helper. The helper's *AutoFlowLeaker* will recover the encrypted data from the items, and then decrypt it to get the messages from the censored user. Similarly, the helper can first encrypt the response from blocked websites, and then use *AutoFlowLeaker* to encode the encrypted data into items in the public pool. Once the items are delivered to the web services accessible to the censored users through the automation services, the censored users can use *AutoFlowLeaker* to recover the responses. Second, to raise the bar for detecting and tracing the communication between the censor user and the helper, our framework carefully organizes the automation services and adjusts the web services used by the censored user and the helper (in Section III-A) and dynamically changes the sources of items (in Section III-B).

Overall, we make the following major contributions:

- To our best knowledge, we propose the *first* framework that exploits automation services to circumvent web censorship. By exploiting plenty of web services and web contents, our framework does *not* need to deploy any customized host. Moreover, it provides strong stealthiness by adopting our new algorithms to convert binary data into texts that consist of legitimate web items, carefully organizing automation services, and dynamically changing the web services and the sources of web items.
- We develop a practical tool, named *AutoFlowLeaker*, to realize the new framework by leveraging existing automation services, web services and items.
- We conduct extensive experiments to evaluate *AutoFlowLeaker*. The experimental result shows that it has promising performance and can effectively evade real-world web censorship.

Roadmap. Section II introduces automation services and the threat model. Section III details our framework and Section IV describes the implementation of *AutoFlowLeaker*. Section V evaluates *AutoFlowLeaker*. After introducing related work in Section VI, we wrap up with discussion in Section VII.

II. Background and Threat Model

A. Automation Flow

An *automation flow* consists of an automation service and a pair of channels, including a trigger channel and an action channel, which can be any web service supported by the automation service. A user can define the condition on the trigger channel and specify the action on the action channel. Once the condition is satisfied, the automation service will execute the corresponding action.

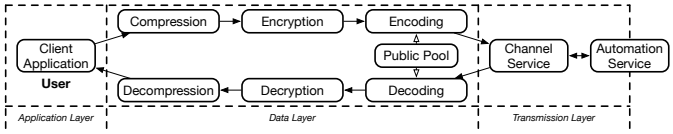


Fig. 2. The procedure of processing outgoing and incoming data.

B. Threat Model

As shown in Fig. 1, the censored user wants to access the websites blocked by the censor, but can only visit websites acceptable to the censor. Although the helper will assist

the censored user to visit the blocked websites, they cannot establish direct communication connection due to the censor. The censor can monitor, block, or delay the censored user's traffic. Moreover, the censor is sophisticated. Specifically, it can analyze all the traffic crossing the border of the censor network to detect and prohibit the communication via anonymity networks such as Tor or encrypted proxies like SSH proxies. Also, it can inspect the traffic to detect covert communication activities [2], [24], [25]. We assume that the censored user and the helper share the secret key and other parameters with minimal out-of-band communication.

III. Design

Fig. 2 shows *AutoFlowLeaker*'s procedure of processing outgoing and incoming data. Outgoing data from the application layer is redirected to *AutoFlowLeaker* that consists of *data layer* (in Section III-B) and *transmission layer* (in Section III-A). For outgoing data, the data layer first compresses and encrypts it, and then converts it into web items (e.g., URL) in the public pool through encoding. These items are extracted from different sources. Then, the transmission layer combines the items to form reasonable texts and sends these texts to the selected web services, which will be forwarded to other web services by the automation services. For incoming data, the transmission layer first collects the texts from the selected web services, recovers them to items, and then forwards items to the data layer. It will convert them into binary data through decoding, and apply decryption and decompression to get the original information before sending it to the upper layer.

A. Transmission Layer

The *transmission* layer uses one or more automation flows to deliver the selected items from the trigger channels to the action channels. We first create a set of automation flows, and then combine them to form multi-level automation flows. We define two basic types of multi-level automation flows: (1) a serial automation flow (Fig. 3a); (2) a parallel automation flow (Fig. 3b). Users can further combine them together to form more complex multi-level automation flows.

The core of a serial automation flow is a chain of automation services (i.e., Zapier, IFTTT in Fig. 3a). For example, the user can connect two automation services with the same channel, Facebook, so that after Zapier transmits the data from Ghost to Facebook, IFTTT forwards the data from Facebook to Wordpress. The serial automation flow can increase the difficulties of tracing the information because it takes much more efforts for the censor to compromise and control all automation services and web services used in the flow.

Besides raising the bar of tracing the information, the parallel automation flow can also increase the data transmission efficiency because it leverages multiple automation services to deliver data. Note that a parallel automation flow is built on the same pair of channels (e.g., Evernote and Github in Fig. 3b) but connects two channels with different automation services. Moreover, the user should set different trigger conditions on each automation service to avoid duplicate transmission.

Algorithm 1: The procedure of `PickServices`.

Input: n, h
Output: $indexes[]$

```
count ← Σ_{i=1}^{n} C(n,i) = 2^n − 1;
r ← h mod count; i ← 1;
while i ≤ n and r >= C(n,i) do
    r ← r − C(n,i);
    i ← i + 1;
end
indexes[] = EncodeCombination(n, i, r);
return indexes[];
```

To enhance the stealthiness by dynamically changing the channel services, we design a mechanism that uses the value h_i from a hash chain to determine the web services used in the ith round of communication.

$$h_i = \mathcal{H}(h_{i-1}), h_0 = key, i \geq 1 \tag{1}$$

Let n be the number of channel services, each of which has an index from 0 to $n-1$. Given h_i, we use `PickServices` (Algorithm 1) to turn h_i into an array $channels$, which represents a combination of channel services' indexes (i.e., $channels[] = \texttt{PickServices}(n, h_i)$). Then, we use the channel services according to the indexes stored in $channels$. For example, assuming $n = 3$ and $h_i = 9$, we have $\sum_{r=1}^{3} \binom{3}{r} = 7$ different combinations. Using `PickServices`, we obtain $channels = \{2\}$ and hence we use the third channel service to send data.

B. Data Layer

The *data* layer protects and hides the original secret messages so that the censor can neither recover the messages nor determine the existence of communication. Fig. 2 shows three essential pairs of components: (1) *compression/decompression*, (2) *encryption/decryption*, and (3) *encoding/decoding*.

Compression/Decompression. It supports more efficient transmission. Any compression/decompression algorithms (e.g., 7z, zip) can be applied in our framework. The compression and decompression are defined as follows:

$$\begin{aligned} D_{compressed} &= \texttt{Compress}(D_{app}), \\ D_{app} &= \texttt{Decompress}(D_{compressed}), \end{aligned} \tag{2}$$

where D_{app} is from or to the application layer.

Encryption/Decryption. It enables confidentiality and integrity by using the key pre-shared between the censored user and the helper to encrypt or decrypt the original secret messages.

$$\begin{aligned} D_{encrypted} &= \texttt{Encrypt}_{key}(D_{compressed}), \\ D_{compressed} &= \texttt{Decrypt}_{key}(D_{encrypted}), \end{aligned} \tag{3}$$

where $D_{encrypted}$ is the encrypted data. Without knowing key, the censor cannot decrypt $D_{encrypted}$ even if the censor collects all the data sent to/from the censored user.

Encoding/Decoding. It enhances the stealthiness and provides the censored user deniability by converting $D_{encrypted}$ to legitimate web items so that the censor cannot determine the existence of communication.

$$\begin{aligned} D_{text}[] &= \texttt{Encode}(D_{encrypted}, pool, h_i), \\ data_{encrypted} &= \texttt{Decode}(D_{text}[], pool, h_i), \end{aligned} \tag{4}$$

(a) A serial automation flow. (b) A parallel automation flow.

Fig. 3. Two kinds of multi-level automation flow.

where *pool* contains web items and h_i denotes a hash value used to dynamically change the sources of web items.

The encoding process (Fig. 4) consists of two steps. In the first step, we split $D_{encrypted}$ into chunks, each of which will be attached a 48-bit header to form a message. The header records the total length of $D_{encrypted}$, the size of the chunk, and the data offset of current chunk in the original data.

In the second step, we use the new *K-property combination* based coding mechanism (in Section III-C) to convert the binary messages into web items acceptable to the censor. For example, users can choose the web pages that contain the list of accepted papers and let each paper be the web item. Each item will have K properties (e.g., the conference of the paper, the publication year of the paper) so that we can divide them into different groups according to the properties.

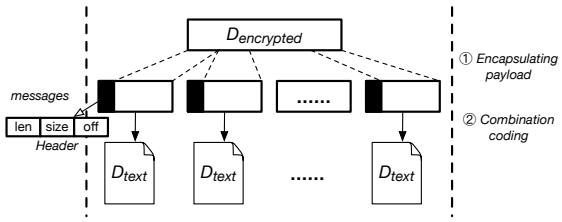


Fig. 4. The encoding process.

Similarly, to enhance the stealthiness, we use `PickServices` (Algorithm 1) to pick different sources of web items. Each source has an index from 0 to $m-1$, where m is the number of sources. That is, we obtain the combination of the indexes of sources: $sources = \texttt{PickServices}(m, h_i)$, and then choose the corresponding sources.

C. K-Property Combination based Coding Scheme

We propose a novel *K-property combination* based coding scheme to construct the mapping between a binary message and the arrangement of web items. For the ease of explanation, let msg be the integer representation of the binary message and A denote the arrangement of the web items selected from the public pool. Moreover, we use *item* to refer to a selected web item and *batch* to indicate a container that accommodates multiple items. For example, the accepted papers of SRDS constitute a batch. An item or batch is regarded as *distinguishable* if it has a unique "ID". For example, the DOI can be used to distinguish a group of papers. Otherwise, the item or the batch is indistinguishable.

K-property combination is based on K counting problems, each of which is defined as grouping I items to B batches ($B \in [1, I]$). For example, given a database of academic papers, grouping three papers among all the papers into two conferences according to one property (e.g., conference name) is one counting problem. We perform such grouping task K times according to K different properties. As each counting problem has different arrangements, we can calculate the total number of arrangements, denoted as C. By designing an algorithm that output each grouping arrangement in a fixed order, we can obtain a mapping between the arrangement and the index that is in the range of $[0, C)$ (i.e., $msg \in [0, C)$). The new algorithms are based on the combinatorial framework [26], [27]. The major difference is that for the new application scenario our new algorithms leverage the multiplication principle to significantly increase the total number of arrangements.

Definition 1 (Cardinality of a set)**.** *Let S be a set. If there are exactly n distinct elements in S where n is a non-negative integer, S is a finite set and n is the cardinality of S. The cardinality of S is denoted by $|S|$. [28]*

Definition 2 (Bijection and cardinality)**.** *Two sets A and B have the same cardinality, $|A| = |B|$, iff there exists a bijection from A to B. [28]*

Lemmas 1-2 introduce two counting problems and the solutions, which are motivated by [26] and [29]. The proofs of Lemmas 1-2 and Theorem 1-2 can be found in the full version of this paper [30].

Lemma 1. *Given I distinguishable items and B ($B \in [1, I]$) distinguishable batches, if the order of items and the order of batches are considered, the number of the arrangements, referred as C_{list}, for grouping I items to B batches is:*

$$C_{list}(I) = \sum_{B=1}^{I} B!I!\binom{I-1}{B-1} \tag{5}$$

Lemma 2. *Given I distinguishable items and B distinguishable batches, if the order of items and the order of batches are not considered, the total number of arrangements, referred as C_{set}, for grouping I items to B batches is:*

$$C_{set}(I) = \sum_{B=1}^{I} B!\mathcal{S}(I, B), \tag{6}$$

where $\mathcal{S}(I, B)$ denotes the number of all partitions of $\{1, 2, ..., I\}$ into B non-empty sets [31].

Based on Lemma 1-2, we propose and prove Theorem 1 that gives the counting equation for the *K-property combination*.

Theorem 1. *Given I distinguishable items, if the order of items and the order of the batches are only considered once,*

the number of the arrangements for grouping the items to distinguishable batches for K times is:

$$T(I,K) = \binom{K}{1} C_{list}(I) C_{set}(I)^{K-1} = K C_{list}(I) C_{set}(I)^{K-1} \quad (7)$$

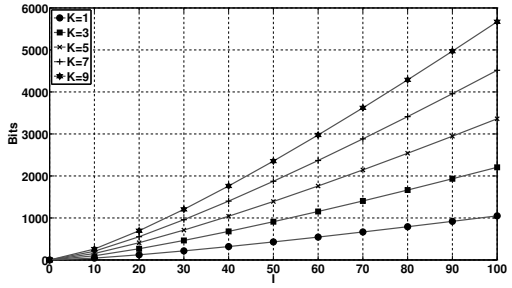


Fig. 5. The capacity of $T(I,K)$ with the change of I and K.

Since Theorem 1 reveals that there are $T(I,K)$ different arrangements for the items in the public pool, they can represent no more than $\lfloor \log_2 T(I,K) \rfloor$ bits of data, which is referred as the capacity (i.e., the number of bits of binary data hidden in one item). The total number of the arrangements (i.e., C) indicates the size of the solution space of the encoding function f (a bijection), and all the possible values of msg constitute the domain of the function. To prove the existence of such a bijection, we propose and prove Theorem 2, which also reveals the range of msg.

Theorem 2. *For any counting problem, let C be the total number of arrangements, there exists a bijection $f : [0, C) \rightarrow \{$all the arrangements$\}$.*

According to Theorem 2, there exist at least one bijection f from X to Y, where X is the integer range $[0, T(I,K))$ (i.e., $msg \in [0, C)$), and Y is the set of all the arrangements. We name this bijection f as the encoding function, while its inverse function f^{-1} as the decoding function. The encoding and decoding algorithms are described in the next subsection.

D. Encoding and Decoding Algorithms

Last subsection proves that there exists a bijection between the integer set and the set of the arrangements. Although the one-to-one mapping rules can be saved in a codebook for conversion, it is not practical to keep a large codebook due to security concerns. Therefore, we design new encoding and decoding algorithms that can perform the bijective function and its inverse function without the need of a codebook.

Algorithm 2: The procedure of `SeparateValue`.

Input: I, K, msg
Output: $r[]$
$count \leftarrow T(I,K)/K$; $i \leftarrow K-1$;
$r[K] \leftarrow \lfloor msg/count \rfloor$; $msg \leftarrow msg \mod count$;
while $i \geq 0$ **do**
  if $i = r[K]$ **then**
    $count \leftarrow count/A_1(I)$;
  else
    $count \leftarrow count/A_2(I)$;
  end
  $r[i] \leftarrow \lfloor msg/count \rfloor$; $msg \leftarrow msg \mod count$;
  $i \leftarrow i-1$;
end
return $r[]$;

Fig. 6a shows the encoding procedure that takes in I, K and msg ($msg \in [0, 2^{\lfloor \log_2 T(I,K) \rfloor}) \subseteq [0, T(I,K))$), and involves four external functions: `SeparateValue` (Algorithm 2), `Encode1` (Algorithm 3), `Encode2` (Algorithm 4) and `CombineArrangement`.

Given I, K, and msg, we can calculate an non-negative integer array r of size $K+1$. This process is referred to `CombineValue` (i.e., $msg =$ `CombineValue`(I, r)), and its inverse process is `SeperateValue` (Algorithm 2):

$$msg = \sum_{i=0}^{r[K]} r[i] C_{set}(I)^i + C_{list}(I) \sum_{i=r[K]+1}^{K} r[i] C_{set}(I)^{i-1}, \quad (8)$$

where r satisfies the following three conditions: (1) $r[K] \in [0, K)$; (2) for any given integer $i \in [0, r[K]) \cap (r[K], K)$, $r[i] \in [0, C_{set}(I))$; (3) $r[r[K]] \in [0, C_{list}(I))$.

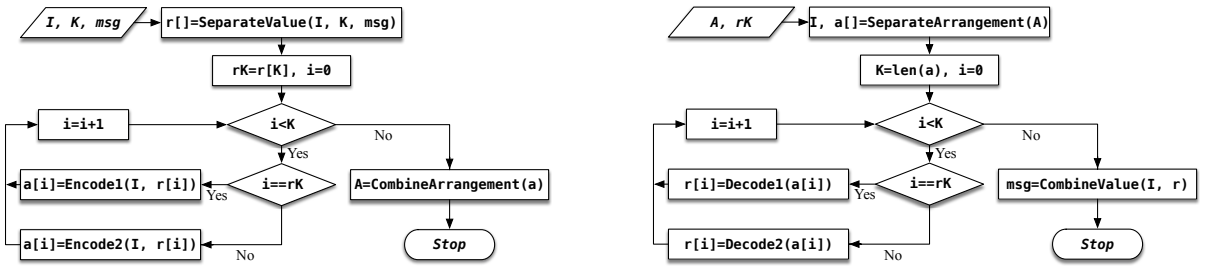
According to Theorem 2, we can derive two bijective functions for the counting problems described in Lemma 1-2, named as `Encode1` and `Encode2`, respectively. Since for any $i \in [0, K)$ $r[i]$ is in the domain of `Encode1` or `Encode2`, we use `Encode1` and `Encode2` to encode each $r[i]$, where $i \in [0, K)$. After that, we get an array A of size K, each element of which is the encoded arrangement of the corresponding $r[i]$: (1) $a[r[K]] =$ `Encode1`$(I, r[r[K]])$; (2) $a[i] =$ `Encode2`$(I, r[i]), i \in [0, r[K]) \cap (r[K], K)$. To obtain the final arrangement A, we use `CombineArrangement` to combine all these arrangements together.

Decoding is the inverse process of encoding (Fig. 6b). It also relies on four external functions: `SeparateArrangement`, `Decode1` (Algorithm 5), `Decode2` (Algorithm 6) and `CombineValue`. They are inverse functions of four external functions in encoding procedure. `SeparateArrangement` first extracts K sub-arrangements (i.e., a) and rK (i.e., $r[K] = rK$) from the composite arrangement A. Then, `Decode1` and `Decode2` convert them to integer numbers stored in the integer array of r: (1) $r[r[K]] =$ `Decode1`$(a[r[K]])$; (2) $r[i] =$ `Decode2`$(a[i]), i \in [0, r[K]) \cap (r[K], K)$. To get the original input msg, we perform function `CombineValue` that uses Eq. (8) to calculate msg.

Algorithms 3-6 involve other external functions since the counting problems in Lemma 1 and Lemma 2 can be divided into basic counting problems (e.g., permutation, combination, and set partition), and we use the existing algorithms for them. Besides, function $len()$ returns the length of the input array.

Permutation. `EncodePermutation`(X, R) and `DecodePermutation`$(X, perm)$ are the encoding and decoding functions for the permutation problem $X!$. $perm = [p_1, ..., p_X]$ is the permutation of X elements. We employ the corresponding algorithms in [32].

Combination. `EncodeCombination` (X, Y, R) and `DecodeCombination`$(X, Y, comb)$ are the encoding and decoding functions for the combination problem $\binom{X}{Y}$, where $comb = [c_1, ..., c_Y]$ is an arrangement of choosing Y elements from X available elements. We adopt the corresponding algorithms (algorithm 2.7 and 2.8) in [31].



(a) Encoding procedure (b) Decoding procedure

Fig. 6. Encoding and decoding procedures of *AutoFlowLeaker*

Algorithm 3: The procedure of `Encode1`.

Input: I, R
Output: arr

calculate the number of batches, denoted as B;
split integer R into three parts: $r[0]$, $r[1]$, $r[2]$;
$items \leftarrow \texttt{EncodePermutation}(I, r[0])$;
$batches \leftarrow \texttt{EncodePermutation}(B, r[1])$;
$comb \leftarrow \texttt{EncodeCombination}(I-1, B-1, r[2])$;
return $(items, batches, comb)$;

Algorithm 4: The procedure of `Encode2`.

Input: I, R
Output: arr

calculate the number of batches, denoted as B;
split integer R into two parts: $r[0]$, $r[1]$;
$batches \leftarrow \texttt{EncodePermutation}(B, r[0])$;
$setp \leftarrow \texttt{EncodeSetPartition}(I, B, r[1])$;
$arr \leftarrow (batches, setp)$
return arr;

Algorithm 5: The procedure of `Decode1`.

Input: arr
Output: R

extract permutation of items, permutation of batches and combination of items and batches from arr;
$I \leftarrow len(items)$; $B \leftarrow len(batches)$;
$r[0] \leftarrow \texttt{DecodePermutation}(I, items)$;
$r[1] \leftarrow \texttt{DecodePermutation}(B, batches)$;
$r[2] \leftarrow \texttt{DecodeCombination}(I-1, B-1, comb)$;
$R \leftarrow r[0] + r[1]I! + r[2]I!B!$;
return R;

Algorithm 6: The procedure of `Decode2`.

Input: arr
Output: R

extract permutation of batches and set partition of items and batches from arr;
$I \leftarrow$ the number of distinct elements among all the subsets of $setp$;
$B \leftarrow len(batches)$;
$r[0] \leftarrow \texttt{DecodePermutation}(B, batches)$;
$r[1] \leftarrow \texttt{DecodeSetPartition}(I, B, comb)$;
$R \leftarrow r[0] + r[1]B!$;
return R;

Set partition. `EncodeSetPartition`(X, Y, R) and `DecodeSetPartition`(X, Y, $setp$) are the encoding and decoding functions for the set partition problem $\mathcal{S}(X, Y)$, where $setp = [s_1, ..., s_Y]$ is a division of X items into Y batches. We use the `UnrankKSetPartition` and `RankKSetPartition` functions in [33].

E. Example of Encoding and Decoding

We use an example of selecting academic papers to illustrate the encoding and decoding procedures of the *K-property combination*. Let a piece of paper information include title, authors, and abstract. The public pool contains the paper information of all papers accepted by IEEE S&P, ACM CCS, and NDSS from 2014 to 2016. Let $K = 2$ and the two properties selected are "*conference name*" and "*publication year*". The conference name follows the lexicographical order while the publication year is in chronological order. For selected pieces of paper information, they are sorted by their titles in the lexicographical order.

Given $I = 3$ and $K = 2$, we have $T(3, 2) = 1716$ different arrangements, which can encode $\lfloor log_2 1716 \rfloor = 10$ bits of data. To encode a 10-bit message "0011101110" (i.e., 238), we first use `SeparateValue` to divide this message into $K + 1 = 3$ parts, and obtain an integer array $r = [40, 3, 0]$. The meaning of this array is:

$$r[0] + r[1]C_{list}(3) + r[2]C_{list}(3)C_{set}(3) = 40 + 3 \times 66 + 0 = 238$$

Since $rK = r[2] = 0$, when grouping the paper information to conferences, we should consider the order of paper information and conference names according to Lemma 1. Thus, for the property *conference name*, we use `Encode1` to encode $r[0] = 40$. That is, $a[0] = \texttt{Encode1}(3, r[0]) = ((P.0, P.2, P.1), (F.2, F.0, F.1), (0, 1))$. The order of items (i.e., pieces of paper information) and batches (i.e., conference names) are {Paper 0, Paper 2, Paper 1} and {Conference 2, Conference 0, Conference 1}, respectively. The arrangement of pieces of paper information and conferences is $\{(F.2, P.0), (F.0, P.2), (F.1, P.1)\}$ (F is short for conference name and P for paper information). For the property *publication year*, we obtain $a[1] = \texttt{Encode2}(3, r[1]) = ((Y.1, Y.0), ((P.0, P.1), (P.2)))$. The arrangement is $\{(Y.1, P.0), (Y.1, P.1), (Y.0, P.2)\}$

(Y means the publication year). Then, the final arrangement is $A = CombineArrangement(a) = (F.2, Y.1, P.0), (F.0, Y.0, P.2), (F.1, Y.1, P.1)$ (i.e., Paper 0 in Conference 2, Year 1, Paper 2 in Conference 0, Year 0, and Paper 1 in Conference 1, Year 1).

TABLE I
9 CHANNEL SERVICES USED BY *AutoFlowLeaker*.

Service Name	API	Read	Write	Delete
Dropbox	Python SDK	✓	✓	✓
Email	SMTP requests	✓	✓	✓
Evernote	Python SDK	✓	✓	✓
Facebook	Python SDK	✓	✓	✓
Ghost	Web requests	✓	✓	✓
Github	Python SDK	✓	✓	✗
Twitter	Python SDK	✓	✓	✓
Wordpress	REST API	✓	✓	✓
Yinxiang Biji	Python SDK	✓	✓	✓

IV. IMPLEMENTATION

Following the system design, we implement *AutoFlowLeaker* using Python in Ubuntu 14.04, and practically it can work on any systems that support Python.

Transmission layer. Table I lists the channel services integrated in *AutoFlowLeaker*. Since most of these channel services offer SDK, we use them directly. For the services (e.g., Ghost, Wordpress) without SDKs, we create HTTP requests using the library *httplib2* to interact with them. For all nine services, we implement read, write and delete operations, except for Github that supports closing an issue rather than deleting it. Thus, we implement the close operation for Github instead of the delete operation. To enable the transmission layer, we configure automation flows on IFTTT and Zapier.

Data layer. For the compression/decompression component, we use the python package *libarchive*. To encrypt or decrypt data, we use AES in the package *PyCrypto*. Moreover, we implement the combination-based coding scheme by ourselves.

V. EVALUATION

We conduct extensive experiments to evaluate *AutoFlowLeaker* and answer five research questions listed in the subsections below. RQ1∼RQ3 are related to the performance. RQ4 studies the effectiveness of *AutoFlowLeaker* and RQ5 evaluates the stealthiness.

A. *RQ1: What is the performance of the channel services and the automation services?*

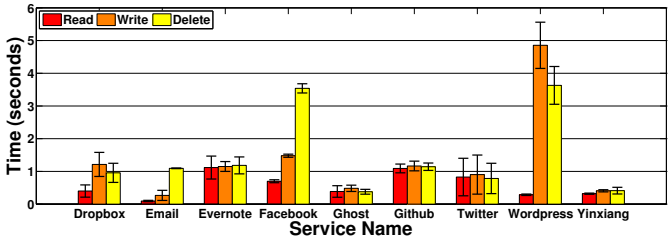


Fig. 7. Average read/write/delete operation time of each channel service.

1) Channel Services: The operation response time is an important criterion to select suitable channel services in the automation flow. We measure the response time of three basic operations of channel services, including read, write, and delete. Each operation response time is defined as the time interval between the time of sending the request for certain operation and the time of receiving the first response packet to the request. Table I enumerates the channel services under investigation, which are used pretty frequently in daily life and have a huge number of users. In each experiment, we use the tool `node-lipsum` to generate a 4096-bytes (4 KB) text file, and then send the content of this file through a trigger channel. *AutoFlowLeaker* will make another request to get that post. Finally, we delete that post. For each service, we repeat the experiment 100 times.

Fig. 7 presents the average values and the standard deviations of the operation response time of each channel service. Apart from Email (we will discuss Email as a trigger service in Section V-B), Yinxiang Biji has the best overall performance. Wordpress and Facebook take lots of time on write and delete operations. Since there is a large time gap between the ACK of the request and the first data packet of the response, the low performance may be due to the overhead of the write and delete operations on the server side.

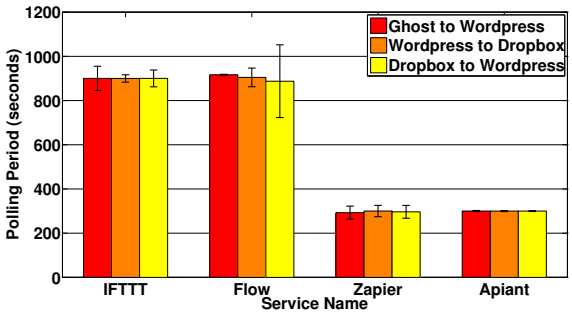


Fig. 8. Average polling period.

2) Automation Service: The automation service checks the data in the trigger channel periodically with a fixed polling period. We evaluate and compare the polling period of four automation services, including IFTTT, Zapier, Flow and Apiant. For the ease of direct comparison, we create the same automation flows for each automation service. Fig. 8 shows the results, which are in consistent with the official documentation of each automation service. For IFTTT, the average polling period is around 15 minutes. Note that the official document of Zapier states that the polling period is either 5 minutes or 15 minutes, depending on the pricing plan of the account. In our experiment, we set the polling period as 5 minutes.

B. *RQ2: How much time delay is introduced by the automation service?*

The automation service's polling mechanism introduces extra delay, defined as the interval between the time of publishing information over the trigger channel service and the time of publishing information over the action channel. We use the same automation flows as in the previous experiment. Fig. 9 shows the time delay. We can see that it is approximately equal

to the polling period. Apiant and Zapier are more stable than IFTTT and Flowd due to their smaller deviation.

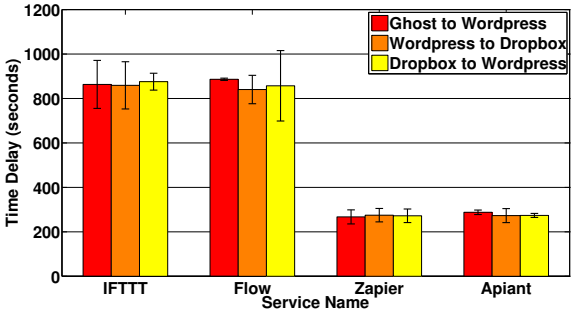


Fig. 9. Average time delay.

For IFTTT, Zapier, and Apiant, users can manually execute a task (i.e., the automation flow) before the automation service polls the task, whereas Flow does not support this feature. Besides, there is a special kind of trigger named instant trigger, which pushes the data to the automation service directly. For example, Email and Short Messaging Service (SMS) can be integrated as the instant trigger services. Therefore, for IFTTT, we measure the time delay of both manually executed automation flows and instant trigger-based automation flows. Fig. 10 presents the results. We see that the average time delay of normal automation flow on IFTTT is about 900 seconds (15 minutes). When applying manual execution (i.e., force check), the time delay is reduced to roughly 300 seconds (5 minutes). Furthermore, using instant service (e.g., Email) as the trigger service largely lowers the time delay to about 5 seconds.

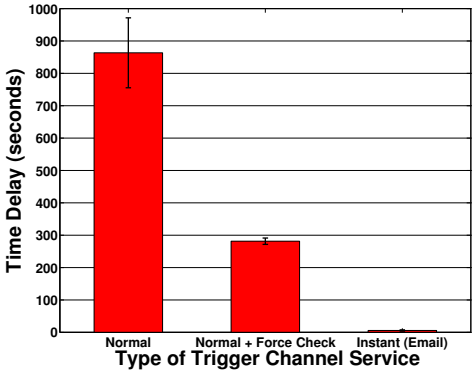


Fig. 10. Average time delay of different automation flows on IFTTT.

C. *RQ3: Does the multi-level automation flow highly increase the time delay?*

Deploying multiple automation services can raise the bar for tracing secret messages, and we design two basic types of multi-level automation flow, as shown in Fig. 3. Here, we measure the time delay of eight multi-level automation flows.

For assessing the serial automation flow illustrated in Fig. 3a, we first connect the trigger channel service (i.e., Ghost or Evernote) and Facebook with Zapier, and then connect Facebook and the action channel service (i.e., Wordpress or Github) with IFTTT. In this case, Zapier forwards the data we publish on the trigger channel to Facebook, and IFTTT transmits the data on Facebook to the action channel. For evaluating the parallel automation flow shown in Fig. 3b), we create different automation flows, each of which is connected with the same trigger channel and action channel. For example, in Fig. 3b, we connect Evernote with Zapier and IFTTT, but set different trigger conditions so that Zapier and IFTTT will not interfere with each other. Fig. 11b demonstrates the results for serial automation flows and parallel automation flows. We observe that the multi-level automation flow does not further increase the time delay.

D. *RQ4: Can AutoFlowLeaker enable the censored users to visit the blocked website?*

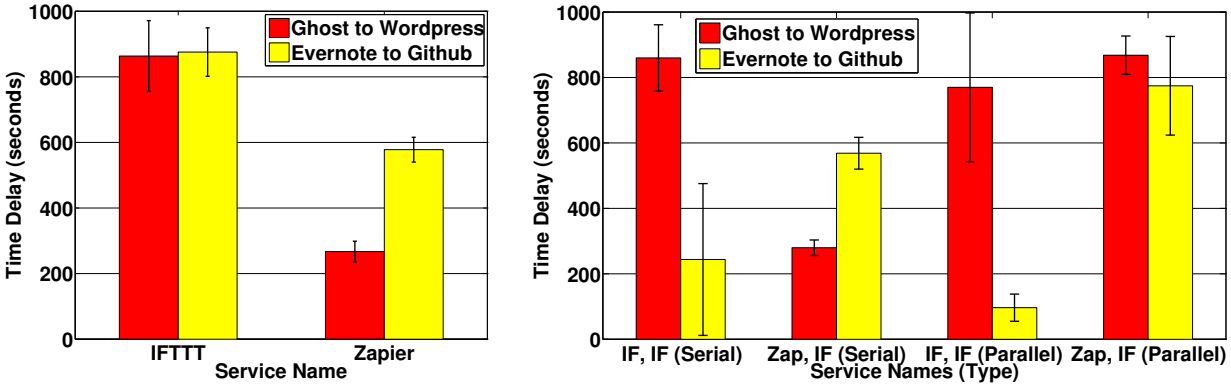
To evaluate the functionality of *AutoFlowLeaker*, in a region where Twitter is blocked, we use it to retrieve the latest tweets. Except for Twitter, *AutoFlowLeaker* could also access other blocked websites. For comparison, we create two groups of automation flows on IFTTT, each of which supports bi-directional communication. The first group builds on the instant trigger channel, Email: (1) from Email (the censored user's web service) to Wordpress (the helper's web service); (2) from Email (the helper's web service) to Github (the censored user's web service). The second group replaces Email of the censored user with Ghost and replaces Email of the helper with Evernote.

The censored user first sends a request to the helper (outside the censored network). On receipt of a request, the helper retrieves the latest three tweets through Twitter API. Then, the helper encapsulates the tweet information, including the content, user's name, and the publish time of a tweet, to a JSON object, serializes the object to a string, and sends the string back to the censored user through *AutoFlowLeaker*. After that, the censored user uses *AutoFlowLeaker* to recover the JSON object. In this experiment, we can successfully obtain the latest tweets. For the first group using Email, the average time cost (i.e., the interval between the time of sending the request and the time of receiving the JSON object) is around 22.7 seconds; for the second group, it is 393 seconds.

E. *RQ5: How is the stealthiness of AutoFlowLeaker?*

To evaluate the stealthiness of *AutoFlowLeaker*, we first build a one-class Support Vector Machine (SVM) classifier to determine whether a sentence is in natural language (NL) or not. The training set consists of 200K NL sentences crawled from Wikipedia [34] and the Brown corpora [35]. Similar to [36], for each sentence in the training set, we use the lexicalized Stanford Parser to infer its score (the goodness of parsed object) and its Part-of-Speech (POS) tags. Based on the POS tags, we count the number of nouns, verbs, adjectives, and adverbs (each normalized by the total number of words) and the total number of words. We then form three testing sets: (1) another 10K NL sentences from Wikipedia; (2) 10K sentences from the generated paper abstract digests using *K-property combination*; (3) 10K randomly generated sentences.

Fig. 12 presents the percentage of the sentences classified as NL (i.e., NL accuracy) in each testing set. As excepted,



(a) The average time delay of basic automation flows. (b) The average time delay of multi-level automation flows (IF: IFTTT, Zap: Zapier).

Fig. 11. The left figure shows the time delay of basic automation flows; the right figure presents the time delay of multi-level automation flows that has the same trigger and action channel as the left ones. As we can see, the multi-level automation flow does not further downgrade the performance.

the NL accuracy of the first testing set is very high (around 97%) and that of the third testing set is very low (around 5%). The NL accuracy of the second testing set is similar to that of the first testing set. In other words, the sentences generated by *AutoFlowLeaker* are similar to the real NL sentences and therefore the users of *AutoFlowLeaker* could deny that they are using *AutoFlowLeaker* for conveying stealth messages.

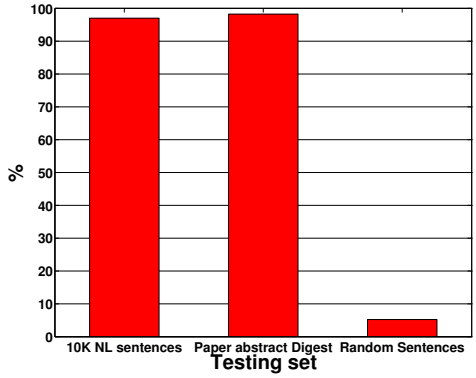


Fig. 12. The percentage of sentences that classified as natural language.

VI. Related work

Existing anti-censorship solutions generally suffer from two limitations (i.e., deployment and stealthiness). First, they usually need the support of ISP (e.g., Telex [7] and TapDance [8]) or the deployment of many nodes (e.g., Tor [4] and Lantern [17]). In addition, solutions like Telex and TapDance may fail to hide the flow features (e.g., packet sizes, timing), and thus vulnerable to large-scale traffic analysis attacks [37]. Infranet [5] and its successor (i.e., Facade [6]) send a special sequence of HTTP requests and receive images carrying hidden messages. They require the participation of web servers and may be vulnerable to stenographic stripping attacks [38].

Although recent studies leverage publicly available services to mitigate the deployment issue, they suffer from the stealthiness problem because the detectable features will negatively affect the censored user's deniability. For example, a few solutions [9], [10] exploit Skype to circumvent the censorship. Particularly, SkypeMorph [9] mimics Skype call messages by sending encrypted data through a traffic shaper to make packet sizes and sending times similar to traffic from normal Skype video calls. However, hiding data by imitation could be easily detected [39]. Instead of mimicking an actual protocol, FreeWave [10] runs the actual protocol. It modulates the IP traffic into acoustic signals via a software modem and then sends it through a VoIP session (e.g., Skype). However, Geddes et al. [40] show that the audio signals of FreeWave cause noticeable differences in the traffic pattern [40], [41]. Moreover, both SkypeMorph and FreeWave are vulnerable to eavesdropping attacks [11] because, except the noise, no actual human voice messages are exchanged. Thus, the censor can easily spot SkypeMorph or FreeWave. Unlike SkypeMorph and FreeWave, SkypeLine [11] embeds secret data in a reasonable VoIP session using Direct-Sequence Spread Spectrum (DSSS) modulation. However, DSSS can be detected with an accuracy of 97.7% by Takahashi and Lee [18].

Several recent studies exploits online game platform to evade the censorship. Rook [13] embeds secret data within the mutable fields of video game network packets. It relies on the Rook server operating on the game server, which may cause deployment issue. Moreover, to identify the mutable fields, Rook relies on the manual analysis of the game network protocol and the correct implementation of at least part of the protocol. Instead, Castle [14] encodes secret data into in-game commands (e.g., move or set-rally-point commands). However, it may send unreasonable commands because it enumerates all possible commands, thus leading to observable anomalies.

Zarras [15] proposes CAMOUFLAGES, a plug-and-play architecture that exploits the genuine services (e.g., email, VoIP, game, file sharing) to evade the censorship. CAMOUFLAGES assembles the secret messages with suitable headers, encrypts them, and then forwards them to the helper through different plugins (e.g., email plugin, VoIP plugin, game plugin). Unfortunately, it has already been shown that some services are vulnerable to detection. Moreover, it uses the base64 encoding scheme to convert the binary secret data to text and then puts it in email or file sharing services. However, pure base64 contents in these services are unusual and thus suspicious.

CloudTransport [12] exploits the cloud-storage services [42]

as a rendezvous point to transfer secret data. However, if the cloud-storage accounts are compromised, CloudTransport cannot keep deniability because the censored user and the helper use the same accounts. In *AutoFlowLeaker*, the censored user and the helper use different accounts "connected" by the automation service. Even though all the services are compromised, *AutoFlowLeaker* can still deny the involvement in the covert communications due to our encoding scheme. Moreover, a censor can employ website fingerprinting technologies to learn traffic patterns that emerge when the user leverages CloudTransport to browse blocked destinations [43].

VII. Conclusion and Future Work

To address current anti-censorship solutions' limitations in deployability and stealthiness, we proposed the *first* framework that exploits automation services to circumvent web censorship. This framework could be easily and widely deployed given the popularity of automation services in daily life. More importantly, it guarantees the stealthiness of message transmissions by converting the encrypted (censored) data into public web content acceptable to the censor. We implemented the framework in *AutoFlowLeaker* and carefully evaluate it. The results showed that, assisted by *AutoFlowLeaker*, one can effectively evade real web censorship with promising performance and plausible deniability for the communication indistinguishable from natural languages. In future work, we will design new mechanisms to handle the (rare) cases where automation services and web services may not be reliable.

Acknowledgment

We thank the anonymous reviewers for their quality reviews and suggestions. This work is supported in part by the Hong Kong GRF (No. PolyU 5389/13E, 152279/16E), Hong Kong RGC Project (No. CityU C1008-16G), Shenzhen City Science and Technology R&D Fund (No. JCYJ20150630115257892), the National Natural Science Foundation of China (No. 61402080, 61602396, 61602371), Natural Science Basic Research Plan in Shaanxi Province (2016JQ6034), China Postdoctoral Science Foundation (2015M582663).

References

[1] S. Burnett and N. Feamster, "Encore: Lightweight measurement of web censorship with cross-origin requests," in *Proc. ACM SIGCOMM*, 2015.
[2] A. Dainotti, C. Squarcella, E. Aben, K. C. Claffy, M. Chiesa, M. Russo, and A. Pescapé, "Analysis of country-wide Internet outages caused by censorship," in *Proc. ACM IMC*, 2011.
[3] Freedom House, "Freedom on the Net 2016," https://freedomhouse.org/sites/default/files/FOTN_2016_Full_Report.pdf.
[4] "The Tor Project," https://www.torproject.org/.
[5] N. Feamster, M. Balazinska, G. Harfst, H. Balakrishnan, and D. Karger, "Infranet: Circumventing web censorship and surveillance," in *Proc. USENIX Security*, 2002.
[6] B. Jones, S. Burnett, N. Feamster, S. Donovan, S. Grover, S. Gunasekaran, and K. Habak, "Facade: High-throughput, deniable censorship circumvention using web search," in *Proc. FOCI*, 2014.
[7] E. Wustrow, S. Wolchok, I. Goldberg, and J. A. Halderman, "Telex: Anticensorship in the network infrastructure," in *Proc. USENIX Security*, 2011.
[8] E. Wustrow, C. M. Swanson, and J. A. Halderman, "TapDance: End-to-middle anticensorship without flow blocking," in *Proc. USENIX Security*, 2014.
[9] H. M. Moghaddam, B. Li, M. Derakhshani, and I. Goldberg, "SkypeMorph: Protocol obfuscation for Tor bridges," in *Proc. ACM CCS*, 2012.
[10] A. Houmansadr, T. Riedl, N. Borisov, and A. Singer, "I want my voice to be heard: IP over voice-over-IP for unobservable censorship circumvention," in *Proc. NDSS*, 2013.
[11] K. Kohls, T. Holz, D. Kolossa, and C. Pöpper, "SkypeLine: Robust hidden data transmission for VoIP," in *Proc. ACM AsiaCCS*, 2016.
[12] C. Brubaker, A. Houmansadr, and V. Shmatikov, "CloudTransport: Using cloud storage for censorship-resistant networking," in *Proc. PETS*, 2014.
[13] P. Vines and T. Kohno, "Rook: Using video games as a low-bandwidth censorship resistant communication platform," in *Proc. WPES*, 2015.
[14] B. Hahn, R. Nithyanand, P. Gill, and R. Johnson, "Games without frontiers: Investigating video games as a covert channel," in *Proc. IEEE European Symposium on Security and Privacy*, 2016.
[15] A. Zarras, "Leveraging Internet services to evade censorship," in *Proc. ISC*, 2016.
[16] R. Dingledine and N. Mathewson, "Design of a blocking-resistant anonymity system," The Tor Project, Tech. Rep., 2006. [Online]. Available: https://goo.gl/ChgqPS
[17] "Lantern," https://getlantern.org/.
[18] T. Takahashi and W. Lee, "An assessment of VoIP covert channel threats," in *Proc. SecureComm*, 2007.
[19] "IFTTT," https://ifttt.com/.
[20] "Flow," https://flow.microsoft.com/.
[21] "Zapier," https://zapier.com/.
[22] "Apiant," https://apiant.com/.
[23] "2016: Year in review - IFTTT," https://goo.gl/2izCXp.
[24] J. R. Crandall, D. Zinn, M. Byrd, E. Barr, and R. East, "ConceptDoppler: A weather tracker for Internet censorship," in *Proc. ACM CCS*, 2007.
[25] T. Zhu, D. Phipps, A. Pridgen, J. R. Crandall, and D. S. Wallach, "The velocity of censorship: High-fidelity detection of microblog post deletions," in *Proc. USENIX Security*, 2013.
[26] X. Luo, P. Zhou, E. Chan, R. Chang, and W. Lee, "A combinatorial approach to network covert communications with applications in web leaks," in *Proc. DSN*, 2011.
[27] X. Luo, E. Chan, P. Zhou, and R. Chang, "Robust network covert communications based on tcp and enumerative combinatorics," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 6, 2012.
[28] K. H. Rosen, *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education, 2002.
[29] R. Proctor, "Let's expand Rota's twelvefold way for counting partitions," http://www.unc.edu/math/Faculty/rap/30FoldWay.pdf, June 2006.
[30] S. Hu, X. Ma, M. Jiang, X. Luo, and M. H. Au, "Autoflowleaker: Circumventing web censorship through automation services," https://www4.comp.polyu.edu.hk/ csxluo/AutoFlowLeakerFull.pdf, 2017.
[31] D. Kreher and D. Stinson, *Combinatorial Algorithms: Generation, Enumeration and Search*. CRC press, 1998.
[32] W. Myrvold and F. Ruskey, "Ranking and unranking permutations in linear time," *Information Processing Letters*, vol. 79, pp. 281–284, 2001.
[33] S. Pemmaraju and S. Skiena, *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Cambridge University Press, 2003.
[34] "Extracting Text from Wikipedia," https://goo.gl/sFNpOZ.
[35] "Brown Corpus Versions," https://goo.gl/pNTJYd.
[36] I. Safaka, C. Fragouli, and K. Argyraki, "Matryoshka: Hiding secret communication in plain sight," in *Proc. FOCI*, 2016.
[37] Z. Weinberg, J. Wang, V. Yegneswaran, L. Briesemeister, S. Cheung, F. Wang, and D. Boneh, "StegoTorus: A camouflage proxy for the Tor anonymity system," in *Proc. ACM CCS*, 2012.
[38] G. Fisk, M. Fisk, C. Papadopoulos, and J. Neil, "Eliminating steganography in internet traffic with active wardens," in *Proc. IH*, 2002.
[39] A. Houmansadr, C. Brubaker, and V. Shmatikov, "The parrot is dead: Observing unobservable network communications," in *Proc. IEEE Symposium on Security and Privacy*, 2013.
[40] J. Geddes, M. Schuchard, and N. Hopper, "Cover your ACKs: Pitfalls of covert channel censorship circumvention," in *Proc. ACM CCS*, 2013.
[41] S. Li, M. Schliep, and N. Hopper, "Facet: Streaming over videoconferencing for censorship circumvention," in *Proc. ACM WPES*, 2014.
[42] X. Luo, H. Zhou, L. Yu, L. Xue, and Y. Xie, "Characterizing mobile *-box applications," *Computer Networks*, July 2016.
[43] D. Barradas, "Unobservable covert streaming for internet censorship circumvention," Master's thesis, Universidade de Lisboa, 2016.