Format-Transforming Encryption: More than Meets the DPI

Kevin P. Dyer*, Scott E. Coull†, Thomas Ristenpart‡, and Thomas Shrimpton*

*Department of Computer Science, Portland State University, Portland, USA. Email: {kdyer, teshrim}@cs.pdx.edu
† RedJack, LLC., Silver Spring, MD, USA Email: scott.coull@redjack.com
‡Department of Computer Sciences, University of Wisconsin-Madison, USA. Email: rist@cs.wisc.edu

***Abstract*—Nation-states and other organizations are increasingly deploying deep-packet inspection (DPI) technologies to censor Internet traffic based on application-layer content. We introduce a new DPI circumvention approach, format-transforming encryption (FTE), that cryptographically transforms the format of arbitrary plaintext data (e.g. packet contents) into specified formats that are designed to bypass DPI tests. We show how to build a general-purpose FTE system, in which these formats are defined compactly by families of regular expressions. Moreover, we specify and implement a full FTE record-layer protocol.**

We exhibit formats that are guaranteed to avoid known filters, and give a framework for learning formats from non-censored HTTP traffic. These formats are put to use in our FTE record layer, to explore trade-offs between performance and steganographic capabilities. As one example, we visit the top 100 Alexa webpages through an FTE tunnel, incurring an average overhead of roughly 5%.

I. Introduction

Network operators increasingly deploy deep-packet inspection (DPI) tools to analyze the application-layer contents of Internet packets. This supplements traditional IP-layer analyses (e.g., based on IP number and port) to effect more effective classification of packets, using keyword search and simple regular expression matching. One lamentable use of DPI is for Internet censorship, where countries or other organizations block connections whose traffic contains blacklisted keywords or protocols [18], [26], [42], [43].

While proxy systems such as Tor [10] help to circumvent censorship based on IP-layer filtering, DPI poses new challenges. This is because circumvention tools themselves can be identified by their (plaintext) headers and formatting, leading to blocking. Countries such as China, Iran, Kazakhstan, and Ethiopia have all employed DPI to detect strings that appear as a Tor client and server negotiate which encryption algorithms to use [27].

One option for dealing with this is to encrypt *all* bits sent on the wire, using a pre-shared secret key [13], [40]. This does, indeed, hide plaintext fingerprints. But the resulting lack of any recognizable protocol headers is a potential fingerprint, too, In particular, if a DPI system uses a whitelist to pass only allowed protocols (e.g. HTTP), then such encrypt-everything systems will fail.

We introduce a new cryptographic primitive, format-transforming encryption (FTE), which combines the plaintext-hiding power of the encrypt-everything approach, with the ability to precisely control the format of ciphertexts. For example, one might wish to format ciphertexts so that, on the wire, they simultaneously conform to a whitelisted protocol, and will not match filters looking for (known) blacklisted keywords. Or, perhaps one wants ciphertexts formatted to accurately mimic traffic considered benign by the network operator. This, and more, is possible with FTE.

We design and implement a general-purpose FTE-proxy system that transforms streaming application-layer data into a sequence of formatted ciphertexts. The ciphertext formats are compactly specified by regular expressions. What's more, our FTE system design is modular, so that that changing formats (i.e. how ciphertexts "look" on the wire) requires only specifying a new regular expression. This supports use of a single design across a variety of network-filtering environments, even ones that change over time. And although we focus on building a standalone system to facilitate evaluation, its components could be incorporated into existing systems, such as Tor or SSH.

To make things concrete, we look at browsing the Web through an FTE tunnel. We consider several formats, including: (1) formats that produce ciphertexts guaranteed to pass whitelist filters employed by DPI tools such as l7-filter [24]; (2) formats yielding ciphertexts that mimic HTTP requests and responses semantics, but that would likely look odd to a human observer; and (3) formats that mimic HTTP requests and responses observed in real network data, collected while visiting the top 5,000 Alexa webpages. The last of these formats are, in fact, learned from the collected data, via a simple learning framework that we design. These three classes of formats represent different system performance/security trade-offs, but even the formats in class (3), which offer the least performance, can be used for surfing the web with only a 5.1% average overhead.

We evaluate the security of our FTE design in several ways. First, we use the l7-filter HTTP regular expression to format ciphertexts, and find that 100% of generated ciphertexts are classified by the l7-filter engine as HTTP. (Keep in mind that the underlying plaintext is arbitrary application layer data.) Next, we consider the scenario in which the network operator possesses all of the regular expressions that define our formats. In this case, a DPI

device could be configured to use these to detect and drop FTE ciphertexts. We show experimentally that doing so would lead to dropping up to 55% of legitimate traffic.

We conclude this initial evaluation of our FTE system by discussing various forward-looking extensions. For example, how to handle active DPI-based adversaries, or those that employ stateful tracking of protocol flows. We also suggest methods for dealing with network proxies, and for carrying out session-key exchange. Finally, we note that although this work focuses on HTTP-based ciphertext formats, we can learn formats to mimic (say) JavaScript, CSS, HTML, or even other network protocols.

II. Setting and DPI Threat Model

The adversary in our setting is a network provider that wants to enforce discriminatory routing policies. Simple policies might be that any TCP connection made by a particular communication tool is reset, that only whitelisted protocols are allowed, or that packets including any keyword on a blacklist be dropped. We focus on the case of binary policies. These seek to separate traffic into two classes: targets of discrimination and non-targets. (The more general case follows.)

To apply such a policy, the attacker must first identify target packets. Having full control over the network, the adversary has numerous identification methods at its disposal. A standard approach is to identify traffic by IP-layer information, i.e. IP addresses and port numbers. Numerous prior works [22], [31], [32] have investigated this, and various countermeasures have been explored. We do not consider identification at this layer. Instead, we focus on adversaries that *identify target traffic via DPI analysis of application-layer content*.

DPI adversaries, especially those operating on high-bandwidth links, cannot afford to carry out arbitrary analytical tests on all the traffic they observe. Thus, DPI systems might employ chained analysis, meaning that they apply a sequence of tests in order to build up a decisions. One way to implement this, surfaced by analysis of the Great Firewall of China (GFC) [43], is to have one or more computationally efficient tests on a fast path, i.e. applied to all traffic. The fast path partitions traffic into 'non-target' and 'suspect', and only traffic in the latter category is forwarded to a slow path, where it is subjected to more expensive tests. In cases of chained analysis, a useful tact for DPI-circumvention tools is to cause the computationally efficient tests, to tag its traffic as 'non-target', even if the more expensive tests would decide otherwise. Of course, this assumes reliable knowledge of these efficient tests at the time the circumvention tool is deployed. As the suite of fast-path tests may change over timer, the ability to adapt the tool in situ is desirable.

It is also important to keep in mind that, in most situations, network operators cannot afford to inflict discriminatory policies upon true non-target traffic. That is, they cannot abide by DPI tests having significant false-positive rates. Thus, another useful tact for circumvention tools is to cause computationally efficient tests that would correctly identify its traffic as 'target', to likewise label non-target traffic as 'target'. Again, since circumvention tool designers cannot hope to know all computationally efficient tests in advance, a flexible system that can be easily updated (i.e. without an entire redesign) is desirable.

DPI systems analyzing high-volume traffic are also likely to be restricted in the amount of inter-packet state it maintains. Keeping state from one packet to the next for every connection can be prohibitively expensive. For this reason, many DPI systems perform analysis that only treats a single network packet (or even single packet fragment) at a time [43].

Example 2.1: [Censorship in China.] Perhaps the most well known example of contemporary, large-scale DPI systems is the Great Firewall of China. This has been studied extensively [1], [7], [26], [42], [43] and is generally considered the most advanced large-scale DPI deployment in the world. Among other things, it inspects application-layer content (headers and bodies) for blacklisted keywords. As a concrete example, an HTTP GET message containing the string `Host: torproject.org` results in the associated TCP connection being forcibly reset. Prior work observes that keyword checks do *not* perform IP fragment reconstruction, likely for reasons of efficiency [43].

Additionally, any connection containing traffic that matches against a blacklist results in the source and destination IPs being blocked at the IP layer for an extended period of time.

Some keyword matches result in secondary identification checks. If traffic is found containing a particular TLS ciphersuite string that is particular to Tor (and appears unencrypted), this triggers a follow-up, active probe of the destination [42]. Here the probe actually negotiates a Tor-compliant TLS handshake with the destination machine, in order to positively identify it as a Tor bridge. This is an excellent example of a fast-path/slow-path implementation of chained analysis DPI. □

Generally, we seek countermeasures that resist DPI systems more power than the GFC, but in a similar high-traffic-volume setting. In particular, we will assume adversaries can analyze *individual, fully reconstructed messages*. We will not, however, consider adversaries that can add significant latencies to the channel, that analyze full network transcripts post facto to inform future decisions, or that use human operators to perform inspection of packets or transcripts.

III. Existing Approaches

Before describing our system, we first consider how existing approaches might fare in our setting of interest.

Fragmentation. A simple approach to avoiding some blacklisted-pattern checks is to use IP fragments to split

censored pattern across two or more IP packets. This works against some current DPI systems such as the GFC [43], but: fragmentation prohibitively degrades performance if it requires sending only short fragments and, more importantly, it fails against DPI that reconstructs (portions of) packets.

Steganographic systems. Steganographic systems seek to hide the existence of messages from all observers by way of embedding the message in real cover traffic such as TCP/IP connections [34], HTTP connections [15], or social media [5], [21]. While these systems seem to offer a high level of undetectability, they are too slow to use for most web applications.

Tunneling methods. One might tunnel plaintext data inside an existing, allowed protocol, e.g. encapsulating the plaintext within an HTTP message [33] or using a system like iodine that tunnels messages over DNS [23]. But since our DPI adversary can analyze full messages, such plaintext tunnels will still admit standard keyword searches, regular expression matching, and other forms of plaintext analysis.

To prevent our DPI adversary from "looking deeper" into the message for keywords and the like, we could use encrypted tunnels. For example, sending messages over a TLS or SSH connection. But, crucially, not all of the content of these connections are encrypted, and in fact the plaintext control information sent during a handshake, as well as the structure of the record layer ciphertexts (a type code, version number, and length field followed by random-looking ciphertext bits), are still directly identifiable by DPI. In some settings the policy will be to block such traffic, for example Iran in the past used such techniques to drop a significant fraction of TLS connections [36]. One suggestion for blocking Skype is to search in packets for the pattern "0x1703010000" [37], [39].

Several censorship avoidance systems strive to look to DPI as if they were other, more commonly used encryption protocols. Tor [10], Telex [46], and Cerripede [20] seek to mimic TLS and SkypeMorph [30] mimics Skype. Perfect mimicking would make the systems' output indistinguishable from the target encryption protocol. However, mimicking is often imperfect, and simple DPI keyword matching can distinguish Tor from other TLS connections based on non-standard plaintext control information sent during the Tor TLS handshake. This is currently used by the GFC and has been reported to be used by DPI systems in Iran [8], Kazakhstan [41], and Ethiopia [41]. Even in the case of perfect mimicking, this strategy fails should the mimicked protocol itself be a target of DPI's filtering [18], [36].

Completely encrypted connections. To remove recognizable headers and other structure that serves as a reliable signature for DPI to exploit, suggestions have been made to encrypt all the bits sent on the wire, including headers and other control information. The Dust [13] protocol and Tor's recent obfsproxy [40] systems use a pre-shared secret key to encrypt all bits sent on the wire, making them effectively indistinguishable from random bit strings.

Done carefully, this can erase all traditional fingerprints, but still there are two limitations. First, the secret key may be known to the attacker, for example obfsproxy hardcodes the key in the software, and here the DPI system may be able to cheaply decrypt data in order to reveal underlying fingerprints. Likewise Dust assumes keys are unknown to the attacker.

Second, the lack of any structured plaintext data is, itself, a fingerprint. Since all other protocols exhibit structure, DPI regimes can use whitelists to green-light certain protocols (e.g., HTTP, DNS, etc.), and drop all unidentified traffic or mark it for further, more expensive analysis. The latter could enable more refined detection mechanisms, such as entropy calculations to detect encrypted communications [11], [12], [29] or traffic analysis [6], [14], [45].

IV. System Overview

In this section we give a high-level overview of our FTE system, depicted in Figure 1, by giving a simple example use. Namely, that of browsing the web for content that would be censored by a DPI-powered firewall. In Section VII we report on an implementation of such a system.

Browsing the Web via an FTE Proxy. Say a user whose traffic must cross a DPI device wishes to retrieve `index.html` from `www.banned-site.com`. Concretely, the DPI device whitelists HTTP using the l7-filter [24] regular expression for HTTP; we'll denote this R_{l7}. Any traffic not passing the whitelist is dropped. Traffic that passes the whitelist is then inspected for blacklisted keywords $w_1, w_2, \ldots, w_n$, one of which is `banned-site`. Traffic found containing a blacklisted keyword is dropped.

To circumvent this DPI device, the user makes a connection to a proxy on the other side of the DPI, and both the user and proxy employ FTE. The user's HTTP GET message forms the plaintext data passed to its local FTE system, which in turn passes the plaintext to the FTE encryption unit. FTE encryption uses a secret key K, state st, and one or more *format packages*. For this example, we assume that the key K is pre-shared via out-of-band mechanisms. (In Section VIII we discuss how this can be shared in-band.) The state st is used for remembering various things, such as counter values or a history of recently used format packages.

An individual format package includes a compact description of a desired format for the strings that FTE encryption will send to the network, and a format description is a family of one or more regular expressions. Loosely, to create an FTE ciphertext, FTE encryption uses the plaintext and the key to sample one or more strings uniformly from the languages associated to these regular

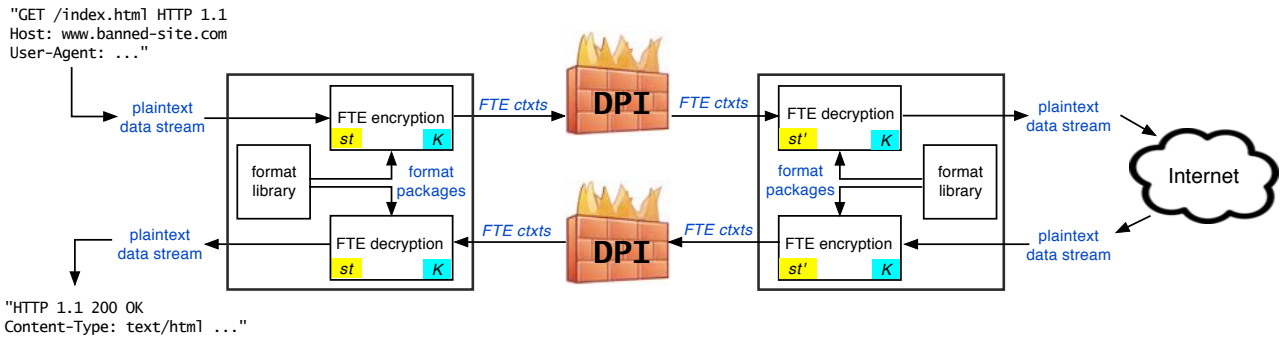

Figure 1: FTE systems in use. Units that share a key K are marked. Local states st and st′ may be synchronized. Network transport of the FTE ciphertexts is handled outside of the FTE system boundary.

expressions. These samples become the ciphertexts. Note that the particular sampling method will allow anyone knowing the format description *and* the key to recover the plaintext from the language samples.

Returning to the example, let

$$B = (.^*)(w_1|w_2|\cdots|w_n)(.^*)$$

be the regular expression for all strings containing at least one of the blacklisted keywords. (Parentheses are for visual grouping, and are not actual symbols.) Say that the user's system employs a format description that is the single regular expression G whose language $L(G) = L(R_{l7})\backslash L(B)$. That is, all strings that will pass the l7-filter HTTP whitelist *and* contain none of the blacklisted keywords as substrings. Thus, ciphertexts formatted according to G will pass the DPI.

The proxy's FTE system takes in the FTE ciphertexts and, using the ciphertexts and the state st′, determines the correct format package. As we said, the correct format description and the key allow FTE decryption to extract the plaintext data —the original HTTP request for `www.banned-site.com`— from the ciphertext. The proxy then retrieves the user's requested web content. Now this content must be sent back to the user, but cannot be sent in the clear as the DPI would drop it. So the proxy, too, employs FTE encryption. Here the downstream direction may use the same language package or a different one.

Instantiating the Components. At a high level, the production of FTE ciphertexts requires two main components: the FTE encryption scheme, and the format packages. Building the FTE encryption and decryption schemes is addressed in Section V, and building format packages in Section VI.

Very briefly, FTE encryption leverages classical algorithms from the language-theoretic literature when it needs to efficiently sample a specified regular language. These algorithms are adapted to our setting, so that, given a fixed regular expression, the plaintext and the secret key determine the sample. Similarly the sample and the key suffice to recover the plaintext for FTE decryption.

As we said earlier, format packages will consist of format descriptions, which are families of regular expressions. We will consider three ways of building these regular expression families. The first is to simply borrow regular expressions used by of-the-shelf network filters, like the l7-filter HTTP regular expression. These are typically used only to identify protocol type, for example, matching any message containing an HTTP status code (e.g., `HTTP/1.1 200 OK`) or POST message (e.g. `POST /form.php HTTP/1.1`.) But while samples from such languages are likely sufficient to pass whitelists (certainly whitelists defined by the regular expressions) they nevertheless lack "realism", since the regular expressions impose relatively little structure. Randomly sampling the l7-filter HTTP languages yields strings like `post Q]Iy7 http/0.0U?|;>es$GDZ=ns.QLlCh?T|.g?7t`...

At the other end of the realism spectrum, we will detail a framework for learning relatively compact regular expressions directly from real network messages. But the corresponding languages sometimes have drawbacks, like being too "small" to support useful system throughput. (Roughly speaking, the logarithm of the size of the language determines the number of plaintext bits that can be transmitted per FTE ciphertext.) So we consider ways to generalize the regular expressions learned from real data, thereby increasing the size of the corresponding languages. Intuitively, aggressive generalization will result in languages whose elements diverge from observed network messages, and conservative generalization will result in language that still have high fidelity to observed messages.

Format packages also contain rules that aid in handling some final, small details in FTE ciphertext production. These will be things that cannot be efficiently represented by regular expressions alone. For example, insertion of non-information carrying symbols (like whitespace), content-lengths, or nonces.

V. FTE Encryption and Decryption

In this section we give the conceptual details of FTE encryption and decryption. For the conceptual flow for encryption, see Figure 2.

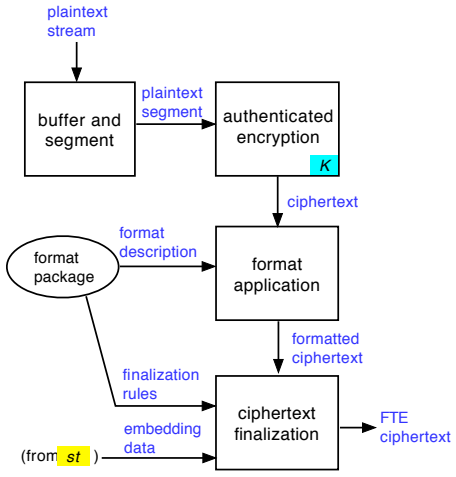

Figure 2: Conceptual flow of FTE encryption.

A. FTE Encryption

We here discuss encryption of individual plaintext segments, and discuss below how we fragment longer plaintext streams into segments. A particular segment is first encrypted using a conventional authenticated encryption (AE) scheme, and the result is an intermediate ciphertext. At this point, the underlying plaintext request enjoys the privacy guarantees provided by the encryption scheme.

The intermediate ciphertext, which is a string of (computationally) random bits, is then turned into a formatted ciphertext by application of one or more *format packages*. These format packages, and how they are applied, form the conceptual core of our FTE system, so let us now describe them in more detail.

Format Packages. A format package $\mathcal{F} = (\mathcal{D}, F)$ consists of a *format description* $\mathcal{D}$ and associated *finalization rules* F. A format description $\mathcal{D}$ is a finite family of one or more regular expressions $\{R_1, \ldots, R_n\}$ for some $n \geq 1$. Each $R_i \in \mathcal{D}$ has an associated regular language $L(R_i)$.

To process the intermediate AE ciphertext, a regular expression R_i is selected from the family; the method of selecting R_i may be randomized, stateful or deterministic. Once R_i is fixed, the AE ciphertext bits are used to sample a string X from $L(R_i)$. (We will describe the sampling algorithms we use in a moment.) The string X then has applied to it finalization rules F. These rules allow post-sampling formatting of X that might be impossible or inefficient to encode as a regular expression. For example, removing certain implementation-specific markers from X, inserting whitespace, and inserting content-sensitive quantities, such as content length. When strings from the languages described by $\mathcal{D}$ require no such finalization, the finalization rules will be empty.

Sampling from Regular Langauges. To support our use of regular expressions, we must be able to sample an element of a regular language. A classic result due to Goldberg and Sipser [17], recently used in the context of format-preserving encryption [3], gives an efficient algorithm for *ranking* members of an arbitrary regular language L. (In our system, this will be $L = L(R_i)$ for some regular expression R_i.) That is, for any L they give an algorithm rank that takes as input a string $X \in L$, and returns a number $x \in \{0, \ldots, |L| - 1\}$. Moreover, for each X the value $\mathsf{rank}(x)$ is unique, so there exists an inverse algorithm unrank which maps numbers back to strings. Thus by picking a random integer in $\{0, \ldots, |L| - 1\}$, one can obtain a random element of L. This process is reversible, so that the integer is uniquely recoverable from the sampled language element.

Now, fix a particular regular language L. When L is finite, we call $\kappa = \lfloor \log_2(|L|) \rfloor$ the *capacity* of L; when L is infinite, so is its capacity. Given a string W of κ random bits, we interpret W as the binary representation of a number in the range $[0..2^\kappa - 1]$, and then compute $\mathsf{unrank}(W)$ to get a (random) sample $C \in L$.[1] To invert this process, we compute $\mathsf{rank}(C)$ to get a number, and represent this as a κ-bit string to recovers W. In our particular case, the string W is instantiated by the intermediate ciphertext output by the authenticated encryption scheme.

Our implementation of Goldberg and Sipser's ranking and unranking algorithms (Figure 3) employs a time-space tradeoff to support more efficient runtime performance. We precompute tables that allow (un)ranking of all strings $x \in L$ with $|x| \leq n$. The complexity of this precomputation is $\mathcal{O}(n \cdot |\Sigma| \cdot |Q|)$, where Σ is the underlying alphabet and Q is the state set for the DFA implementing the FTE regular expression. Given these tables, the complexity of rank_L and unrank_L are $\Omega(n)$ and $\mathcal{O}(n \cdot |\Sigma|)$ where n is the length of the output of rank_L, or input of unrank_L.

We chose to define format descriptions in terms of regular expressions, rather than the DFAs that rank and unrank natively reference, because regular expressions are easier for implementers to work with; they are also, typically, more compact than their associated minimal DFAs.

While converting a regular expression to a DFA is a well-established process (c.f., [38, chapter 1]), even a relatively simple regular expression can result in DFAs for which $\mathsf{BuildTable}$ becomes impractical to generate and store. For example, consider the language $L = \{0,1\}^{\leq n}$ of all bit strings of length at most n for a large value of n. In language-theoretic notation, the corresponding regular expression is $R = \epsilon \cup (0 \cup 1)^1 \cup \ldots \cup (0 \cup 1)^n$, and the *minimal* DFA for L requires $n + 1$ states. Recall that the tables computed by $\mathsf{BuildTable}$ (Figure 3) are dependent upon the number of states in our DFA; in this case the tables would require $\mathcal{O}(|Q| \cdot |n|) = \mathcal{O}(n^2)$

[1] Note that when $|L|$ is not a power of two, this does not sample uniformly from the whole language but from a subset of it whose size is a power of two.

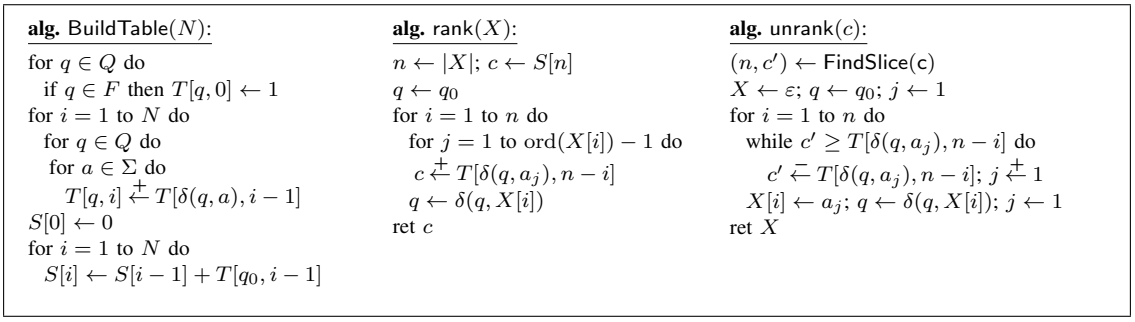

Figure 3: Algorithms for ranking and unranking strings in the regular language L of a DFA $M = (Q, \Sigma, \delta, q_0, F)$. The ordinality of symbol $\alpha \in \Sigma$, written $\mathrm{ord}(\alpha)$, is its position (starting from 1) in the lexicographical ordering of the elements of Σ. $T[q, i]$ is the number of strings of length i that end in an accepting state when starting from state q; thus $T[q_0, i]$ is the number of $X \in L$ such that $|X| = i$. $S[i]$ is the number of strings in L of length at most $i - 1$. Unspecified algorithm FindSlice finds the largest ℓ such that $S[\ell] < c$, and returns $n = \ell + 1$ and $c' = c - S[\ell]$. This can be done in $O(\log_2(|S|))$ time via binary search.

storage. Fortunately, some simple algorithmic tricks help to generate small DFAs.

- Augment a regular expression to generate an infinite language: Replacing $R = \epsilon \cup (0 \cup 1)^1 \ldots \cup (0 \cup 1)^n$ with $R' = (0 \cup 1)^*$ yields a single-state minimal DFA. But calling BuildTable(n) still builds the correct table for $L(R)$.
- Increase the size of the underlying alphabet: This has the effect of decreasing the number of states, in return for an increased number of potential state transitions. This may impact the performance of rank or unrank, but produces smaller tables in BuildTable.
- Use conventional DFA minimization techniques to ensure minimal DFAs.

Our initial FTE implementations make use of these tricks. Further optimizations led to substantial savings of space and running time. For example, in some languages there exist many states $q \in Q$, such that the transition function $\delta(q, a)$ is independent of the symbol a. In this case, we do not need to execute the the innermost for-loop in rank, nor the while-loop in unrank. This is a particularly useful savings when combined with the trick of using a large alphabet.

Finally, for languages that only have fixed length strings, such as $L = \{0, 1\}^n$, we don't need to store all $n \cdot |Q|$ values in table T. For every value $q \in Q$, there exists only one non-zero value i for $T[q, i]$. Hence, we only need to store $|Q| = n + 1$ values for T.

B. Record Layer

The FTE mechanisms we've described are limited by the capacity of the selected language $L(R_i)$. In theory, one can accommodate longer inputs by just increasing the capacity of $L(R_i)$. But this may not always be desirable, due to efficiency concerns, and in general we may want segment the plaintext and a sequence of regular expressions to create ciphertexts. For example, one regular expression describing HTTP headers, followed by one describing HTTP payloads.

Referring to Figure 2, we therefore include a component that maintains a buffer of incoming plaintext data. This buffer is controlled by calling code of FTE-Sender and FTE-Receiver in Figure 4. We assume FTE-Sender is invoked with no more than 2^{16} bytes of data in a single call to prevent unnecessary delay in decoding on the receiver's side. On the receiver's side it is necessary to maintain an additional internal buffer in order to identify and parse complete intermediate ciphertexts (the output of a single invocation of Enc in Figure 5.)

FTE-Sender. On input a key K, a format package $\mathcal{F}$, an initial value IV and a plaintext string M, the sender first encrypts M using a custom AE scheme (Enc, Figure 5), and then encodes the resulting AE ciphertext into one or more samples from languages described in $\mathcal{F}$. To perform AE encryption, we first prepare a record header. This includes the IV and an encoding of the plaintext length. The record header is enciphered by an n-bit blockcipher E into a string W_1 (e.g. $n = 128$ when $E =$ AES). Next, the plaintext is encrypted using counter-mode over E, yielding W_2. (We use a IV_1 to produce W_1, and IV_2 to produce W_2 in order to enforce domain separation between different uses of E.) Finally, we use HMAC to compute a message authentication code (MAC) for $W_1 \parallel W_2$, and produce the AE ciphertext $C = W_1 \parallel W_2 \parallel T$.

The AE ciphertext is passed to Encode (Figure 6), along with the format package $\mathcal{F}$. At a high-level, encoding proceeds by picking a sequence of regular expressions $R_1, R_2, \ldots, R_t$ from $\mathcal{D}$, and unranking κ_i bits of C into $L(R_i)$, for each R_i in the sequence. The length t of the sequence is determined by $|C|$ and the capacities. If the final chunk of C contains fewer than κ_t bits, we pad it with random bits to ensure uniform sampling from $L(R_t)$. The notation $\mathsf{unrank}_{[R]}$ in Figure 6 means to unrank into the language of R.

The unspecified algorithm getRE implements selecting a particular regular expression R from $\mathcal{D}$. It may be desirable for this algorithm to be randomized, for reasons we will discuss shortly. Because getRE may need to

operate slightly differently in Encode than in Decode, its input signals its use.

The unspecified algorithm finalize implements any necessary post-processing of the language sample Z. Post-processing might include things that are not efficiently represented by a regular expression, for example, the computation of the proper content length when Z is an HTTP message with `Content-length` field. It might also handle cosmetic alterations to Z, such as the insertion of whitespace.

FTE-Receiver. The pseudocode in Figure 4 also describes the operation of the receiver in our record layer. The receiver takes as input the shared key K, a format package $\mathcal{F}$, and a ciphertext string Y. We note that Y may not be an entire FTE ciphertext, nor observe underlying ciphertext boundaries. FTE-Receiver maintains an internal buffer B, its persistence marked by making it a superscript in the notation. The input ciphertext string Y is appended to the buffer, and the buffer passed to Decode, along with $\mathcal{F}$ and the record header length n. This first call to Decode is to recover the enciphered record header. The role of the unspecified algorithm parse is to unambiguously parse its input into its constituent, rankable blocks (i.e. language samples) $(X_1, \ldots, X_m)$. When $\mathcal{D} = \{R_1, \ldots, R_n\}$ this is possible (for example) when $L(R_1) \cup \cdots \cup L(R_n)$ is a prefix-free set of strings.

Once the enciphered record header H is recovered, the GetLen algorithm is called, returning the length of the plaintext that is encrypted in this record. With this length, the receiver will be able to Decode and AE decrypt the remainder of the record.

We note that although our pseudocode indicates that we have to make multiple calls to Decode which may fail, a stateful version of Decode can be implemented such that this is not required.

Format Packages, Revisited. Our formalization of an FTE system allows for multiple format packages $\mathcal{F} = (\mathcal{D}, F)$ and, within each associated format description $\mathcal{D}$, multiple regular expressions. Let us take a moment to motivate why we do this.

Conceptually, a format package is meant to capture a particular class of formats. For example, there might be a format package $\mathcal{F}_{\text{rqst}}$ for HTTP request messages, one for HTTP response messages, one for DNS messages, etc. The package $\mathcal{F}_{\text{rqst}}$ would then contain a format description $\mathcal{D}$ whose regular expressions represent strings that are HTTP request messages, and likewise for the other two format packages. Thus, having multiple format packages admits a degree of control over the class of ciphertext formats that FTE will produce. Among other things, this lets the sender and receiver conduct a "consistent" exchange: the sender loads in the format package for HTTP request messages, and the receiver loads in the format package for HTTP response messages.

alg. FTE-Sender$(K, \mathcal{F}, IV, M)$:
$(K_1, K_2) \leftarrow K$
$C \leftarrow \mathsf{Enc}((K_1, K_2), IV, M)$
$X \leftarrow \mathsf{Encode}(\mathcal{F}, C)$
ret X

Global Parameters
n: blocksize of E, in bits
τ: length of HMAC, in bits

alg. FTE-Receiver$^B(K, \mathcal{F}, Y)$:
$(K_1, K_2) \leftarrow K$
$B \leftarrow B \,\|\, Y$
$(H, B') \leftarrow \mathsf{Decode}(\mathcal{F}, B, n)$
if $H = \perp$ then ret ε
$N \leftarrow \mathsf{GetLen}(K_1, H)$
$N \leftarrow N + n + \tau$
$(C, B) \leftarrow \mathsf{Decode}(\mathcal{F}, B, N)$
if $C \neq \perp$
  $M \leftarrow \mathsf{Dec}((K_1, K_2), C)$
ret M

alg. GetLen(K_1, H):
$H \leftarrow \mathrm{msb}(n, H)$
parse $E_{K_1}^{-1}(H)$ as $0 \,\|\, IV \,\|\, \ell$
ret ℓ

Figure 4: Pseudocode implementation of FTE client/server routines. The FTE client listens with the FTE-Sender and responds with the FTE-Receiver, whereas the server listens with the FTE-Receiver and responds with the FTE-Sender. All FTE client (resp. server) algorithms may access and update the implicit FTE client (resp. server) state. FTE-Receiver maintains a persistent variable B (its buffer), its persistence denoted by superscripting. All other variables have local (non-persistent) scope. Global Parameters n, τ are also used in Figure 6 and Figure 5. $\mathrm{msb}(n, H)$ returns the n most-significant bits of H.

alg. Enc$((K_1, K_2), IV, M)$:
$IV_1 \leftarrow 0 \,\|\, IV$; $IV_2 \leftarrow 1 \,\|\, IV$
$W_1 \leftarrow E_{K_1}(IV_1 \,\|\, \langle |M| \rangle)$
$W_2 \leftarrow \mathsf{CTR}[E_{K_1}](IV_2, M)$
$T \leftarrow \mathsf{HMAC}(K_2, W_1 \,\|\, W_2)$
$C \leftarrow W_1 \,\|\, W_2 \,\|\, T$
ret C

alg. Dec$((K_1, K_2), C)$:
parse C as $W_1 \,\|\, W_2 \,\|\, T$
$T' \leftarrow \mathsf{HMAC}(K_2, W_1 \,\|\, W_2)$
if $T \neq T'$ ret $\perp$
parse $E_{K_1}^{-1}(W_1)$ as $0 \,\|\, IV \,\|\, \ell$
$M \leftarrow \mathsf{CTR}[E_{K_1}](1 \,\|\, IV, W_2)$
ret M

Figure 5: Pseudocode implementation of FTE encryption and decryption.

alg. Encode$(\mathcal{F}, C)$:
$(\mathcal{D}, F) \leftarrow \mathcal{F}$
$\ell \leftarrow |C|$; $X \leftarrow \varepsilon$
while $\ell > 0$ do
  $(R, \kappa) \xleftarrow{\$} \mathsf{getRE}(\mathrm{enc}, \mathcal{D})$
  If $\ell > \kappa$ then
    $P \xleftarrow{\$} \{0,1\}^{\ell - \kappa}$
    $C \leftarrow C \,\|\, P$
  $V \leftarrow \mathrm{msb}(\kappa, C)$
  $C \leftarrow \mathrm{lsb}(|C| - \kappa, C)$
  $\ell \leftarrow \ell - \kappa$
  $Z \leftarrow \mathsf{unrank}_{[R]}(V)$
  $X \leftarrow X \,\|\, Z$
ret $\mathsf{finalize}(F, X)$

alg. Decode$(\mathcal{F}, X, L)$:
$(\mathcal{D}, F) \leftarrow \mathcal{F}$
$(X_1, \ldots, X_m) \leftarrow \mathsf{parse}(\mathcal{F}, X)$
$\ell \leftarrow 0$; $i \leftarrow 1$
while $\ell < L$ and $i \leq m$ do
  $(R, \kappa) \xleftarrow{\$} \mathsf{getRE}(\mathrm{dec}, \mathcal{D})$
  $V \leftarrow \mathsf{rank}_{[R]}(X_i)$
  $t \leftarrow \min\{\kappa, n - \ell\}$
  $W \leftarrow W \,\|\, \mathrm{msb}(t, V)$
  $\ell \leftarrow |W|$
  $i \leftarrow i + 1$
if $|W| < L$
  ret $(\perp, X)$
else
  ret $(W, X_{i+1} \,\|\, \ldots \,\|\, X_m)$

Figure 6: Pseudocode implementation of FTE encoding and decoding, using format package $\mathcal{F} = (\mathcal{D}, F)$ where $\mathcal{D}$ is a family of regular expressions, and F is the associated finalization rules. $\mathrm{lsb}(b, C)$ returns the least-significant b bits of C. Unspecified algorithms getRE, finalize, and parse are described in the text.

Now, say that we have fixed $\mathcal{F}_{\text{rqst}}$ as our format package. Why have multiple regular expressions in $\mathcal{D}$, instead of just one? One answer, again, is control. Recall that we select from $\mathcal{D}$ a particular regular expression R, and then sample uniformly from $L(R)$ to produce an FTE ciphertext. Say that $\mathcal{D}$ contains regular expressions $R_1, R_2, \ldots, R_t$, whose languages are disjoint by virtue of the particular `User-agent` substring they encode. Then selecting from $\mathcal{D}$ a particular R_i will give FTE ciphertexts that are HTTP request messages that all have the same `User-agent`. This level of control would not be available if, instead, $\mathcal{D}$ contained a single regular expression whose language was $L(R_1) \cup L(R_2) \cup \cdots \cup L(R_t)$. Uniform samples from this might produce a sequence of HTTP requests with different `User-agent` strings, which might look suspect (to a very careful DPI) when coming from the same IP address.

A second answer relates to usability and performance. Recall that sampling from $L(R)$ actually makes use of an equivalent DFA. (We can assume without loss of generality that it is the minimal DFA.) Having a single regular expression for all of the desired strings in a particular format package will often lead to a DFA with a large number of states; thus, BuildTable will produce a large table T. For performance reasons, one would typically like to have T in local memory, but this may not be feasible if T is large.

A final answer has to do with DPI false-positive rates. Let's assume that the DPI adversary will know all of the algorithms in our record layer, and all of the format packages. Even more, assume that the DPI adversary knows which format package is currently in use. If there were a single regular expression R in $\mathcal{D}$, then there is a immediate test that partitions traffic into two categories: not FTE (doesn't match R), and possibly FTE (does match R). This is exactly the sort of fast classification a chained-analysis DPI needs, passing traffic in the latter category on for slower, more careful analysis. Now if R is designed to match a significant fraction of non-FTE traffic, this test becomes less useful, particularly in high traffic-volume settings. But as we noted above, such an R is likely to lead to large DFAs and tables[2]. Say, instead, that $\mathcal{D} = \{R_1, R_2, \ldots, R_t\}$, where now $L(R_1) \cup L(R_2) \cup \cdots \cup L(R_t) = L(R)$. If the algorithm for selecting a regular expression from $\mathcal{D}$, i.e. getRE, is deterministic, then the DPI adversary will again know exactly which R_i is in use, resulting in the same efficient test. (Worse, the false-positive rate will be lower than before.) But there are ways to avoid this. For example, say that $\mathsf{getRE}(\text{enc}, \mathcal{D})$ uses secret randomness to select one of the R_i uniformly. The DPI adversary can still check traffic against each of $R_1, \ldots, R_t$, and this can still be fast with parallelization. But now the false-positive rate (classifying non-FTE traffic as possibly FTE) is that of the single regular expression R, while the sender and receiver storage requirements are reduced to those for R_i alone.

If the receiver shares the secret randomness with the sender (say via a shared key derived from the FTE key), then it can immediately determine the correct R_i to use. But it may not even be necessary to share the secret randomness, since, like the DPI, the receiver can simply test the traffic against each regular expression in $\mathcal{D}$. The crucial difference is that the receiver *knows* that the traffic is an FTE ciphertext.

As a final note in this discussion, we point out that determination of what level of abstraction should be controlled at the format package level, versus at the regular expression level, is up to the system designer. There are many aspects to consider —the desired granularity of control, what kind of consistency should be maintained, desired DPI false-positive rate, desired language capacity, performance restrictions— just to name a few prominent ones. Our discussion, here, is meant to surface some advantages of allowing for multiple format packages and multiple regular expressions within a package, and to suggest some intuition for their construction.

VI. Building Format Packages

Central to the success of our FTE system are formats that ensure that FTE ciphertexts pass whatever constraints the DPI places on "allowed" traffic. Simultaneously, they must provide sufficient capacity for us to support uses like web browsing. Typically, this two requirements will be at odds.

In this section, we detail our framework for building families of regular expressions. We first discuss how we learn regular expressions from collected network traffic. We then explore methods for generalizing these learned expressions, in order to trade realism for capacity.

Creating Regular Expressions. There are several strategies that one could use to create regular expressions for FTE ciphertexts that bypass DPI filtering rules. These strategies fall into one of two general categories: (1) *constraint matching*, or (2) *generalization*.

The simplest of these categories is constraint matching, where we start with a completely unconstrained RE (e.g., $\{0,1\}^*$, the set of all possible bit strings) and add the DPI's constraints to the RE as they are discovered. Discovering those rules, of course, is a potentially difficult process, but may be achieved if the DPI rules are, say, published as part of an open source project. In fact, our FTE system is uniquely positioned to directly use any RE-based DPI rules to output FTE ciphertexts that exactly match the given rule. For the cases when the rule triggers filtering (i.e., blacklists), we can easily take the complement of the language described by the set of DPI rules. The primary benefit of this constraint matching approach is that it reduces the capacity of the language used to generate

[2]Designing an overly general R solely to have a high FP rate isn't desirable. Such an R is likely to produce "nonsense" FTE ciphertexts that would fail to pass even simple whitelist filters.

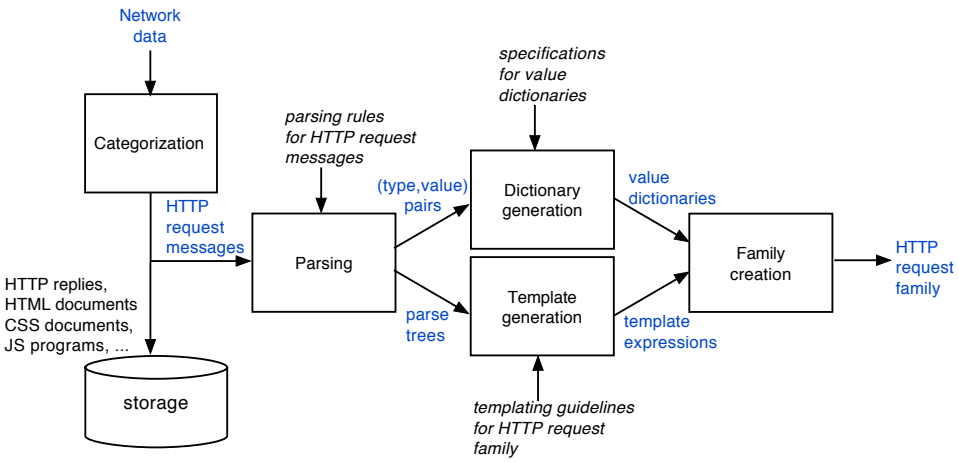

Figure 7: Our process for automated generation of a regular expression family for HTTP-GET messages from captured network data.

FTE ciphertexts only to the extent necessary to meet the DPI constraints. As such, constraint matching is preferable in cases when information carrying capacity of the FTE tunnel is of utmost importance, and the "realism" of the traffic is secondary.

The generalization approach to RE creation, on the other hand, takes the opposite position. Rather than beginning with a completely unconstrained RE, the generalization approach starts with a RE that describes a set of messages (e.g., HTTP headers) that are known to pass the DPI's filtering process. Such a language is likely to have extremely low information carrying capacity, but it exactly describes real messages that would be difficult for the DPI to filter without harming non-FTE traffic due to significant false positives. The process of generalization, therefore, methodically relaxes the constraints of the real messages by allowing new combinations of their components (e.g., HTTP header values, HTML tags, etc.) that effectively generate new messages that share some commonalities with the original messages. By continuing to generalize the observed traffic, we can improve capacity at the cost of producing messages that are less "realistic." Certainly, it is possible to implement a manual generalization procedure with some expert knowledge of the protocols and messages being encoded, however this necessarily restricts the utility of the approach and limits the ability of FTE to quickly adapt to changing DPI environments.

Creating RE Families from Examples. To assist in producing generalizable REs without tedious manual effort, we developed a framework for automating the process of learning REs from real network data and then generalizing those expressions to increase language capacity. We note that learning REs is, in the general case, an NP-complete problem [16] and that several approximation algorithms have been proposed [4], such as Lang's Exbar algorithm [25]. In our experiments with these algorithms, however, we found that they were unable to cope with the sheer volume of network data necessary to produce robust regular expressions, which often span into thousands of states in the underlying DFA. Instead, we take advantage of the fact that there are many easily accessible protocol grammars (i.e., parsers) for almost all network protocols and content formats for which we would like to generate FTE ciphertexts. We use the protocol grammars as a way of decomposing real network data into parse trees where terminals are atomic data types, which are then used to generalize the examples of real traffic and combine them back into REs.

The RE learning approach is broken into four phases, as shown in Figure 7. First, we capture live network traffic, split the data into its constituent protocols and content, and group the data into categories reflecting those protocol and content types.

Each example in a given group is parsed to create a parse tree. The non-leaf nodes in the tree are labeled by non-terminal symbols, and leaf nodes are labeled by strings of terminal symbols. Each leaf is associated with an atomic type, which is often (but not always) the symbol labeling its parent node. After parsing all of the examples, the terminal symbols are aggregated into sets for their respective types, which we refer to as *type dictionaries*. Each type dictionary may have a corresponding type-RE, whose language is exactly the elements of the dictionary. For example, if the dictionary is $\{\texttt{a}, \texttt{b}, \texttt{c}\}$, the corresponding type-RE is $\texttt{a}|\texttt{b}|\texttt{c}$.

The parse trees are also used to create *template expressions*. A template expression is a string of terminal symbols, non-terminal symbols, and RE meta-symbols. In our framework, each non-terminal symbol in a template will be replaced with the designated type RE. Thus, each template expression gives rise to one *fully qualified* RE. A template expression is valid if and only if the language of the fully qualified RE it generates is parseable, with respect to the parsing rules that created the parse trees. All templates in our work will be valid.

A family is then defined as the set of fully qualified REs produced from the templates. These families will form the format descriptions in the format packages of our FTE system.

Let us look at an example of building a regular expression family from HTTP request messages.

Example 6.1: [HTTP Request] Below is a simple HTTP request that gets a web page from `example.com` with a single query variable.

```
GET /bb/cgi-bin/index.php?f=bar HTTP/1.1\r\n
Host: example.com\r\n
Connection: Keep-Alive\r\n
User-Agent: Mozilla/5.0\r\n
\r\n
```

The (simplified) parse tree, relative to an implicit set of parsing rules, is shown in Figure 8. Here, the atomic types Dir, File, Query, Ver, Host-Header and Host are shown. Dictionaries for each of these atomic types are derived by aggregating their associated values across all HTTP requests in our collected network traffic. In this example, for instance, would contribute `bb` and `cgi-bin` to the type dictionary for Dir.

This example HTTP message suggests one natural template expression

```
GET /Dir/Dir/File?Query HTTP/Ver\r\n
Host: Host\r\n
Connection: Keep-Alive\r\n
User-Agent: AgentString\r\n
\r\n
```

which is created by simply replacing atomic type instances with their atomic type non-terminal, e.g. `bb` with Dir. A fully qualified RE would then be obtained by replacing all non-terminals —Dir, File, Query, Ver, Host and AgentString— with their corresponding type regular expressions, derived from the type dictionaries as described earlier. For example, the RE (`bb` | `cgi-bin` | $\cdots$) would replace every occurrence of Dir.

We note that the template we just gave is only one possibility. One can imagine a wide variety of related template expressions, say, by removing the '`?`Query', or fixing the User-Agent value to a particular string. If there is a valid non-terminal Accept-type, one could add a line '`Accept:` Accept-type', and so on. One could even do "strange" things, such as having '`User-Agent:` Host, Host', so long as the resulting fully qualified RE is parseable. Choices in the design of template expressions are typically driven by concerns for capacity and realism; we'll say more in a moment. In general, though, if the goal is to make REs whose languages mimic realistic traffic, a good approach to template design is to follow the structure and patterns of real traffic. □

Controlling Capacity and Realism. In the HTTP example we just gave, one valid template expression would simply give the specific HTTP request, verbatim. However the language of this RE contains exactly one string, and so has capacity zero. Our framework provides many ways to increase capacity, through *generalization*, and we mention a few here.

One simple way is to replace the learned type-dictionaries with larger ones. Say, making $D_{\mathsf{Dir}} = \Sigma^*$ in the previous example, for some alphabet of terminals Σ. This may result in nonsense strings (relative to what would be seen in practice), but it allows for languages with infinite capacity. For integer fields, such as `Age` or `Content-Length`, we can generalize to include any value in the range $[0, 1, \ldots, 10^{10}-1]$. Being a bit less aggressive, thus sacrificing less realism in the name of capacity, one might generalize time/date fields (e.g. `Expires`) to include any in the range seen in the data set.

The generalizations just given are focused on the type dictionaries. A complementary technique is to increase the number of non-terminals that appear in the templates. For example, allowing for an arbitrary URL depth by repeating `Dir` in our example.

It should be clear that the designer controls what generalization techniques are applied, and to what degree. In this way, one can customize a tradeoff between capacity and realism.

We note that not all generalizations are intended to increase the capacity of a language, but perhaps to allow for simpler parsing/templating rules for easier implementation.

Going beyond HTTP. In the next section, we report on the security and performance of HTTP request and HTTP response formats learned using the procedure outlined above. But one can learn regular expression families for other things typically seen on a network, too. For example, typical HTTP payload contents. We have already learned families for JavaScript, CSS and HTML, and initial experiments using them to form FTE ciphertexts (mimicking HTTP payloads) are promising. We will report on these in a future version of this work.

VII. Experimental Evaluation

Here we give an experimental evaluation of the main pieces of our FTE system. We begin by describing the implementation of our FTE prototype, and how it is used for our target application of browsing the web through an FTE-powered tunnel. Next, we revisit the whitelisting DPI-filter example from Section IV, and discuss how to build formats from the regular expressions used in the publicly available l7-filter DPI engine. We follow this with a concrete example of learning regular expressions, using the framework in Section VI, from collected network traffic. Specifically, we learn regular expressions for HTTP request headers, and HTTP response headers. We explore the potential false-positive rate that a DPI-adversary might

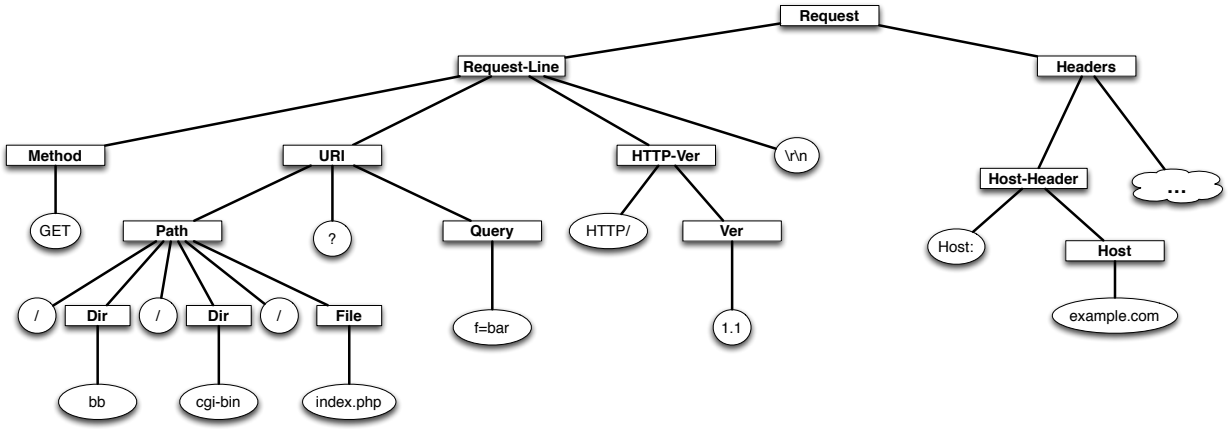

Figure 8: A portion of the simplified parse tree showing the decomposition of an example HTTP request.

incur if using these regular expressions to test for FTE ciphertexts in real traffic. Finally, we report on the performance of our prototype using these various formats.

A. Prototype Implementation

Our prototype implements rank, unrank, BuildTable (Figure 3) in C/C++. We use a customized version of the re2 library[3] for regular expression to DFA conversion, and OpenFST[4] for DFA minimization. Cryptographic algorithms are implemented using PyCrypto[5]. Logic for the record layer and networking is done in Python.

To evaluate our prototype we setup a testbed configured as shown in Figure 9. The client machine was an Intel Core 2 Quad Processor Q6600 running Ubuntu 12.04 and the server was an Intel Core-i7-2600 3.4GHz running RHEL 6.3. The client was connected to the Internet on a residential-grade ADSL connection and the server was hosted on a commercial-grade connection at a university. Round-trip latency between the client and server was 77 milliseconds on average, average server-to-client throughput was 8.0 Mbps, and average client-to-server throughput was 1.8 Mbps.

For our HTTP client we used a command-line version of the `webkit` rendering engine, by utilizing the official `QtWebKit`[6] version 2.2.0 library. The `webkit` rendering engine is a multi-threaded layout engine used by Apple Safari[7] and Google Chrome[8], and its performance is representative of a user's web browsing experience. For our SOCKS proxy we used `Dante`[9] version 1.4.0-pre1. Our web browser redirected HTTP requests through our FTE client implementation, which then encrypted these requests according to the language specified. The FTE server received these FTE ciphertexts, decrypted them, and then passed the resulting plaintexts to the SOCKS proxy (which then fetched the requested webpage from the Internet). After fetching, the process was reversed, with the FTE server encrypting the returned web content and sending it back to the HTTP client through the FTE client.

[3] `http://code.google.com/p/re2/`
[4] `http://www.openfst.org/`
[5] `https://www.dlitz.net/software/pycrypto/`
[6] `http://trac.webkit.org/wiki/QtWebKit`
[7] `http://www.apple.com/safari/`
[8] `http://www.google.com/chrome/`
[9] `http://www.inet.no/dante/`

B. Data Collection

For learning our regular expressions, as well for testing them, we built a corpus of HTTP traffic. This was done by visiting popular webpages, as determined by the Alexa rankings[10]. Alexa establishes rank by a combination of page views and diversity of visitors. Hence, the data collected from these sites should be a good representation of traffic commonly seen by network operators.

We collected two data sets for our evaluation. For $N = 5,000$ and $N = 50,000$ we downloaded the homepage, and dependencies required to render the homepage using `wget`[11], for the top N Alexa webpages — we call these two data sets ALEXA5K[12] and ALEXA50K[13], respectively. For each data set we recorded all payload data from network packets using `tcpdump`[14], and subsequently extracted all HTTP request and HTTP response headers. The ALEXA5K data set yielded 209,970 HTTP request/response pairs. The ALEXA50K data set yielded 2,043,964 HTTP request/response pairs.

C. Whitelist DPI-defined Regular Expressions

As a first step, we consider building formats from a regular expression used in the publicly available l7-filter DPI tool. The l7-filter HTTP regular expression classifies HTTP traffic by identifying HTTP response status codes (e.g., `HTTP/1.1 200 OK`) or HTTP POST messages (e.g., `POST /form.php HTTP/1.1`.) The actual regular expression, which we denote here as R_{l7}, is given in full in

[10] `http://wwww.alexa.com/`
[11] `http://www.gnu.org/software/wget/`
[12] Collected July 9, 2012.
[13] Collected July 17, 2012.
[14] `http://www.tcpdump.org/`

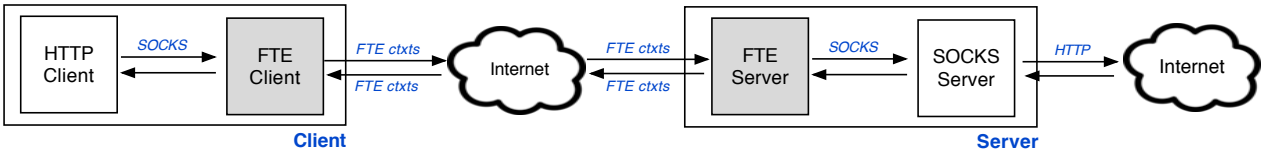

Figure 9: Prototype configuration for web-browsing-over-FTE evaluation.

Appendix A. The l7-filter DPI tool passes (whitelists) any string $U \| V$, where $U \in L(R_{l7})$, and $V \in \Sigma^*$ is arbitrary. To create FTE ciphertexts that pass this whitelist, we format the first 760 AE ciphertext bits using R_{l7} ($\kappa = 760$ for $L(R_{l7})$), and format any remaining bits using the RE $.*$, which is equivalent to passing raw AE ciphertext without formatting.

To make sure that ciphertexts formatted in this way do actually pass the l7-filter whitelist, we visited each of the top 100 Alexa webpages five times, for a total of five hundred homepage visits. We recorded full network payload data between the FTE client and FTE server using `tcpdump`, and identified 12,937 distinct TCP sessions. We extracted the TCP sessions using `tcpflow`[15], and stored each session transcript in a distinct datafile.

The l7-filter software distribution includes a testing framework to evaluate regular expressions against data samples. We utilized the l7-filter testing framework to test the l7-filter HTTP regular expression against the datafiles extracted from our FTE traffic. The l7-filter classification engine classified 100% of the FTE traffic as HTTP.

We note that the l7-filter HTTP regular expression has a high false-positive rate, matching many strings that do not conform to HTTP standards. However, we were unable to find a publicly-available regular expression that describes a more restrictive interpretation of HTTP.

D. Learned Regular Expressions

Now, say we want to produce FTE ciphertexts that not only pass HTTP whitelisting filters, but have semantic structure similar to non-FTE HTTP messages seen by network operators. Following our discussion in Section VI, we set out to learn two families of regular expressions, one for HTTP requests, and one for HTTP responses, from the ALEXA5K data set.

HTTP Request REs, with Conservative Generalization. Remember than an HTTP request is structured as follows:

```
GET /path/to/file?queryString HTTP/X.Y
Label_1 : Value_1
...
Label_n : Value_n
```

To build our HTTP request family, we do the following. We select a random subset of $w = 5000$ HTTP requests from our ALEXA5K data set[16], and parse each HTTP request in this subset. From each parse tree, we extract the particular labels and their order; for the i-th such tree, $\sigma_i = (\texttt{Label}_1, \ldots, \texttt{Label}_{n_i})$ is recorded. For each distinct `Label` that appears in any of the σ, we create a dictionary $D_{\texttt{Label}}$ that contains every `Value` associated to that `Label`, i.e. '`Label: Value`' appeared in some request. Similarly, from each parse tree we extract the component strings from the URL. From right to left: all query strings are collected in a dictionary D_{qs}, all filenames in a dictionary D_{file}, and each string appearing between consecutive '`/`' is collected into a directory dictionary D_{dir}; for example '`/src/code/ver1/foo.bar`' contributes `foo.bar` to D_{file}, and `ver1`, `code`, and `src` to D_{dir}. Finally, from the parse trees we create an HTTP-version dictionary D_{ver} that contains all observed strings following `HTTP/`. From each of the dictionaries D, we create a corresponding type-RE R.

Now, each of the σ gives rise to seven template expressions, one for each URL directory depth $d = 0, 1, \ldots, 6$, where $d = 0$ is just a filename. This is a conservative generalization of the URL depths actually seen in the data set. Recall that each template expression yields one regular expression for the HTTP request family. We then parameterize our regular expression templates on: the header label-order σ, the directory depth d, and the inclusion/exclusion of the languages R_{file} and R_{qs} in the `GET` request line. For example, if $\sigma = (\texttt{Host}, \texttt{User-Agent}, \texttt{Accept})$, $d = 2$ and we want to include R_{file} and R_{qs}, then we have, with some abuse of notation, the regular expression

$$\texttt{GET } /(R_{\text{dir}})/(R_{\text{dir}})/R_{\text{file}}\texttt{?}R_{\text{qs}} \texttt{ HTTP/}(R_{\text{ver}})\backslash\texttt{r}\backslash\texttt{n}$$
$$\texttt{Host: } R_{\texttt{Host}}\backslash\texttt{r}\backslash\texttt{n}$$
$$\texttt{User-Agent: } R_{\texttt{User-Agent}}\backslash\texttt{r}\backslash\texttt{n}$$
$$\texttt{Accept: } R_{\texttt{Accept}}\backslash\texttt{r}\backslash\texttt{n}$$

where parentheses and linebreaks are only for visual grouping. Thus strings in the language of this regular expression contain a URL that is two-levels deep, each directory string can be any of the observed ones (those in D_{dir}); the filename can be any of the observed ones; the query string can be any of those observed. The string

[15] `http://circlemud.org/jelson/software/tcpflow/`

[16] We explored values $w \in \{1000, 2000, \ldots, 10000\}$ and selected $w = 5000$ for performance reasons. It is the value that resulted in regular expressions with equivalent minimal DFAs containing $|Q| \leq 2^{14}$ states, which bounds the time-complexity and storage requirements of the BuildTable algorithm. Implementations requiring higher fidelity languages may use a larger value w.

following `Host:` can be any of those observed, and so on.

HTTP Response REs, with Conservative Generalization. We build our HTTP response family in much the same way as we built the HTTP request family. We parse a random subset of $w = 5000$ HTTP response from our ALEXA5K data set. From each parse tree, we extract the particular labels and their order; for the i-th such tree, $\sigma_i = (\texttt{Label}_1, \ldots, \texttt{Label}_{n_i})$ is recorded. For each distinct `Label` that appears in any of the σ, we create a dictionary $D_{\texttt{Label}}$ that contains every `Value` associated to that `Label`, i.e. '`Label: Value`' appeared in some request. Finally, from the parse trees we create an HTTP status-code dictionary D_{stat} that contains all observed status code strings (e.g. `HTTP/1.1 200 OK`). From each of the dictionaries D we create a corresponding regular expression R whose language is exactly D.

Each of the σ gives rise to a template, and hence to one regular expression for the HTTP response family. For example, if $\sigma = (\texttt{Server}, \texttt{Content-Encoding})$ then we have (again, with a some abuse of notation) the regular expression

$$R_{\text{stat}}\texttt{\textbackslash r\textbackslash n}$$
$$\texttt{Server: } R_{\texttt{Server}}\texttt{\textbackslash r\textbackslash n}$$
$$\texttt{Content-Encoding: } R_{\texttt{Content-Encoding}}\texttt{\textbackslash r\textbackslash n}$$

where linebreaks are only for visual grouping.

HTTP response messages often contain fields that specify dates and times, and the values are often related to our time of data collection. In the data we observed dates in the range January 1, 1970 to December 30, 2030. Hence, we generalize the {`Date`, `Expires`, `Last-Modified`} dictionaries to include all valid dates in this range.

Using Aggressive Generalization. We also considered a set of more aggressive generalizations for these language families. In particular, we expanded many of the type dictionaries to include the full range of values allowed by standards. Generalizing in this way fields that include, say, hashes simultaneously provides significant increase in capacity, and removes potential static fingerprints.

For HTTP requests we generalize directories, file names and query strings by allowing any alphanumeric string up to 128 characters. We allow the `User-Agent` field to contain any alphanumeric string up to 64 characters. The `Host` and `Referrer` fields are generalized to allow alphanumeric domain names and URLs, respectively. Finally, we allow up to five alphanumeric `Cookie` strings, separated by semicolons, up to 128 characters each.

For HTTP responses we identified seven fields that benefit from aggressive generalization of the dictionaries: `Keep-Alive`, `ETag`, `Location`, `Set-Cookie`, `Content-Length`, `Age`, and `Cache-Control`. As an example, we generalized `Content-Length` and `Age` by allowing any integer up to a nine decimal values. For the `ETag` field we identified the five most common formats, such as hex strings of length 32, and manually specified the regular expression. Finally, we generalized custom HTTP headers of the form `X-...` by allowing any alphanumeric string up to 32 characters (following `X-`) for the header label, and any alphanumeric string up to 128 characters for the field value.

	DFA states	capacity (bits)	unrank/rank avg. (ms)	ciphertext expansion
DPI Whitelist				
l7-filter HTTP	57	760	0.2 / 0.2	1.34
Conservative Generalization				
HTTP request	12,336	33	0.1 / 0.1	41.7
HTTP response	4,228	114	0.2 / 0.2	21.6
Aggressive Generalization				
HTTP request	107	2540	0.8 / 0.7	1.6
HTTP response	3,522	1228	0.5 / 0.4	2.9

Figure 10: Statistics for our l7-filter whitelist language, and languages from our HTTP request and response families with the largest FP rate against their learning data set. The unrank/rank benchmarks were taken from an average of 10,000 trials, by selecting an integer/string to unrank/rank uniformly from the domain of each function. Ciphertext expansion is calculated as the average length of a string (in bytes) in the language divided by the capacity (in bytes) of the language.

In Figure 10 we report the statistics for the HTTP request and response language families, under both conservative and aggressive generalization.

We note that not all field values can be generalized in all cases. As an example, in the presence of a header-modifying HTTP proxy, we may be unable to use fields such as `Keep-alive` for language capacity. Alternatively, if a proxy is validating our messages against the HTTP RFC, we may omit the `Content-Length` field from our regular expression and then embed the correct `Content-Length` value prior to sending the FTE ciphertext. However, managing such restrictions is an implementation-specific consideration. In Section VIII we discuss solutions for deploying FTE in the presence of an adversarial proxy.

E. Learned Languages False-Positive Rate Analysis

In Section V-B we discussed that a chained-analysis DPI system could theoretically use the regular expressions employed by the FTE system to partition traffic into two categories: not FTE, and possibly FTE. Here, we consider the cost of this DPI attack on our system, in terms of false positives: what fraction of real, non-FTE traffic might be incorrectly labeled as possibly FTE, and subjected to further analysis?

Fix a regular expression R with language $L(R)$, and let $\mathcal{M} = \{M_1, M_2, \ldots, M_k\}$ be a set of strings seen by the DPI, yet not produced by running our FTE system. We define the false-positive (FP) rate of R to be $\frac{|L(R) \cap \mathcal{M}|}{|\mathcal{M}|}$, or the fraction of non-FTE strings that match R. We extend this

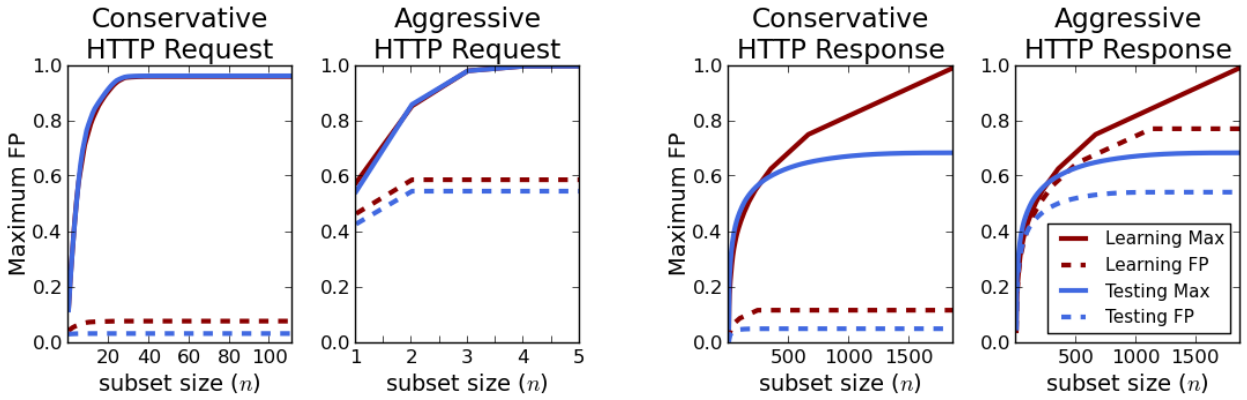

Figure 11: Maximum false-positive (FP) rate for the HTTP request and response languages. The maximum is taken over all subsets of sizes $1, 2, \ldots, n$, where n is the size of the family. A random subset of size $w = 5000$ from the ALEXA5K data set was used for learning the families, and we used the complete ALEXA50K data set for testing. The Learning/Testing Max lines indicates the maximum FP rate, versus the learning/testing data sets, that could be achieved given the templating strategy.

measure to a regular expression family $\mathcal{D} = \{R_1, \ldots, R_n\}$ by defining $L(\mathcal{D}) = L(R_1) \cup L(R_2) \cup \cdots \cup L(R_n)$. Then the FP rate of the family is $\frac{|L(\mathcal{D}) \cap \mathcal{M}|}{|\mathcal{M}|}$, or the fraction of non-FTE strings that match at least one of the regular expressions in $\mathcal{D}$.

Now, say that $\mathcal{D}$ is the language family that will be used for the current FTE ciphertext, and this is known to the DPI adversary. If the DPI adversary can determine exactly which $R_i \in \mathcal{D}$ will be used, then the best strategy for the FTE system (in terms of FP cost to the DPI) is to use the R_i that will maximize the FP rate. If, instead, the FTE system (via getRE) picks an R_i uniformly from $\mathcal{D}$ (and keeps the sampling coins secret from the DPI), then the effective FP rate is that of the family.

We explore both possibilities, fixed R_i and randomly sampled R_i, in Figure 11. Here we have four plots, one for both HTTP request and response languages, under the conservative and aggressive generalization strategies just discussed. These plots consider what are the maximal FP rates, over all subsets of the specified family, if getRE uniformly selects from a subset of size n. Thus the case of $n = 1$ is the maximal FP rate of any individual RE in the family, $n = 2$ is the maximal FP rate for $L(R_i) \cup L(R_j)$, where $i \neq j$, over all R_i, R_j in the family, and so on. The largest value of n is the number of regular expressions in the family. To generate these plots, we learn the specified family from a random subset L of size $w = 5000$ from our ALEXA5K data set, then test our regular expressions against L (learning) and the full ALEXA50K (testing) data set.

In each plot we also give a line labeled "Learning Max" and "Testing Max"; let us explain what it means. Recall how each regular expression is derived from a template expression: the non-terminals in the template expression are replaced by their corresponding type-REs, which are derived from the type dictionaries. Say that these dictionaries were expanded to Σ^*. Then the Learning (resp. Testing) Max is the FP rate of the corresponding fully qualified RE, i.e. it is the FP rate of the maximally generalized RE associated to that template. As above, when $n = 1$ we have the best FP rate over all single such REs, for $n = 2$ the best over all pairs, etc. The Learning (resp. Testing) Max lines serve as a reference, since this is the best FP rate achievable for the given subset of templates.

FP Rate using Conservative Generalization. In testing, for conservative generalization, we achieve only a 3% FP rate for HTTP request and 6% for HTTP response languages. This low rate is due to fields such as `Host` and `ETag`, because the larger testing data set presents many unique values for these that, in particular, do not appear in the learning set.

Our HTTP request language family has $n = 120$ templates reflecting our template construction strategy: we model directory paths six levels deep and have an optional filename and query string. We then consider the five unique HTTP request label-orders in our learning set to get a total of $n = 6 \cdot 2 \cdot 2 \cdot 5 = 120$ templates. Our HTTP response language family has $n = 1930$ templates, which is the number of unique templates identified in learning by selecting a subset of size $w = 5000$ from our ALEXA5K data set.

For our HTTP request family both our Learning/Testing Max quickly plateau around $n = 32$ at 96%, this is due to our templating strategy. We model directories to a depth of at most six directory values deep, so we don't capture any HTTP requests with seven or more directories in the URL, despite maximal generalization of dictionaries.

For our HTTP response family our Learning/Testing Max are roughly equivalent until $n = 256$. This is because a small set of templates capture the majority of header label-orders in the HTTP response dataset. For $n > 512$ we see a divergence between the Learning/Testing Max lines: our templating strategy necessarily restricts the possible FP

rate we could achieve in testing, due to label-order pairs seen in testing that were not seen in learning.

FP Rate using Aggressive Generalization. For aggressive generalization, in testing, we achieve 55% for HTTP request and 56% for HTTP response languages. Our aggressive HTTP request language has at most $n = 5$ templates, one for each of the five distinct label-orders in our learning dataset. Similarly to our conservative HTTP response languages, our aggressive HTTP response language has a maximal family size of $n = 1930$.

In testing for the aggressive HTTP request family we reach a plateau of 55% at $n = 2$. This can be improved in future versions of our framework by identifying further refinements to our manually-developed regular expressions for fields such as `Cookie`.

The aggressive HTTP response language family achieves an FP rate of approximately 40% using a subset of size $n = 64$. For subsets of size $n > 64$ we achieve modest gains until we reach a plateau at $n = 1024$. As with the HTTP request family, we can further improve our FP rate by increasing the breadth and fidelity of generalized fields.

F. FTE Prototype Performance

Now we evaluate the performance of our FTE prototype using the languages in Figure 10. Specifically, we use the command-line `webkit` engine to request, download and render every homepage from the top 100 Alexa webpages, and measure the full time it takes to download the homepage and all dependencies, and to execute any javascript required to render the webpage. The average per-page rendering times listed in Figure 12 are over five trials. We also report the average overhead, relative to the rendering time when we bypass the FTE client and server, and connect directly to the remote SOCKS server to download each webpage.

We first test performance when our record layer is implemented to simply output the AE ciphertext directly, i.e. skipping formatting altogether. This is done both at the client and server, giving us a measure of the overhead introduced by performing just authenticated encryption of the plaintext. Here the average overhead is 2.9%.

Next, we used R_{l7} in our system to create FTE ciphertexts as described in Section VII-C. This is listed in Figure 12 as 'l7-filter HTTP$\|$raw AE', meaning that the initial portion of the FTE ciphertexts result from sampling the language of the l7-filter RE, and the remainder of the FTE ciphertext is raw AE ciphertext bits. Here the average overhead is 4.0%.

When employing one of our learned formats, we configure the prototype to use the (single) regular expression R from that family with the largest FP rate against their learning data set. Then the FTE ciphertexts are prepared as just described: if $L(R)$ has capacity κ, the first κ-bits of the AE ciphertext formatted according to R, and the remaining bits are passed through. We use the shorthand HTTP-request-ag (resp. -cg) to denote that R comes from the HTTP request family with aggressive (resp. conservative) generalization, and likewise for the HTTP response family.

The learned REs with aggressive generalization have an average page rendering time of only 9.07s, with 4.5% overhead. Then, not surprisingly, the learned REs with conservative generalization have the worst performance. After all, these offer the least capacity. But even in this case, the average rendering time is only 9.12s, and the overhead is just 5.1%.

G. Learned Languages DPI-Whitelist Analysis.

As another test of formats we produce, we subject the learned languages in Figure 10 to the same evaluation performed against our whitelist DPI-defined regular expressions. We want to determine the percent of FTE traffic (from our learned languages) identified as HTTP by the l7-filter DPI engine. We collected all traffic produced in our performance evaluation for our learned languages. Using `tcpflow` we identified 13,318 unique TCP connections instantiated with our conservative languages, and 12,791 TCP connections instantiated with our aggressive languages. For both language classes, across all TCP connections, the l7-filter DPI engine classified 100% of FTE traffic as HTTP.

H. False-positive Rate/Performance Trade-off

Informally, we can characterize the trade-off between FP rate and performance as follows. The conservatively generalized languages have a low FP rate and are the worst performers, but would look, even to human observer, very much like normal HTTP traffic. For example, HTTP requests will only contain URLs with directories, filenames, and query strings seen in our data set. On the other hand, aggressively generalized HTTP requests have better performance, but URLs contain random-looking directories, filenames, and query strings. A human analyst would likely detect oddities, but it is not clear to what extent these can be exploited by real-world DPI, especially the fast path of chained-analysis DPI. A future version of this work will present a detailed investigation of such FP/performance trade-offs.

VIII. Extensions

In-band key sharing. Our prototype implementation uses a pre-shared secret key. We recognize that, in many settings of practical interest, the secrecy of the key cannot be assumed. For example, when the only distribution channel for shared keys is the application software itself, and the DPI adversary has access to the software. In such instances, one will need to negotiate session keys in-band, i.e., over a channel potentially observed by the DPI adversary. Traditional approaches to generate a shared secret, like using a public-key infrastructure (PKI), may not work here, since the key-exchange messages themselves may

client → server ctxts	server → client ctxts	average per-page render time (sec)	average overhead
no format (raw AE)	no format (raw AE)	8.93 ± 0.39	2.9%
l7-filter HTTP || raw AE	l7-filter HTTP || raw AE	9.03 ± 0.37	4.0%
HTTP-request-ag || raw AE	HTTP-response-ag || raw AE	9.07 ± 0.40	4.5%
HTTP-request-cg || raw AE	HTTP-response-cg || raw AE	9.12 ± 0.42	5.1%
baseline: SOCKS only, no FTE		8.68 ± 0.39	0%

Figure 12: Performance statistics for web browsing through our prototype FTE-tunnel when downloading the top 100 Alexa webpages, over five trials. Average per-page render time is the average time required to fully download and render each page, across all trials. Average overhead is the average percent increase in per-page render time compared to our SOCKS only (no FTE) configuration.

trigger filtering. Indeed, such key exchange messages for TLS are already targeting by censorship systems [9], [26].

One alternative is to exploit the asymmetry of workload between the parties using FTE, and the DPI system. Say that the FTE client and server share a master key K_m, which may be known by the DPI adversary. A per-session nonce N is picked by the client and encoded into to the first ciphertext sent in the upstream direction. The session key K, used for subsequent FTE encryption, is derived via $K = \mathsf{HMAC}^c(K_m, N \parallel ip_{src} \parallel ip_{dest} \parallel port_{src} \parallel port_{dest})$ where HMAC^c is the HMAC construction iterated c times. Choosing c to be relatively large, for example $c = 10,000$, places a modest (once per session) burden upon the FTE client and server. But the accumulated workload for the DPI system is much larger, even when it knows K_m, since it must carry out the computations for each suspected FTE connection. We note that a similar mechanism is described in the Dust protocol [13]. Also, we point out that variations of so-called client-server puzzles could be used for this purpose.

Active attacks. Consider that a censor actively probes servers it suspects might be running FTE. For example, by sending an HTTP GET request that is formatted similarly to previous ones delivered to the suspected server. Without using *authenticated* encryption, any packet that properly ranks (i.e. any packet in the specified regular language) will also decrypt. This could potentially lead to telling responses by the server. Using AE ensures that only FTE-produced ciphertext packets will properly decrypt. What's more, recognizing that the received FTE ciphertext is invalid allows for an informed response. For example, a DNS query that fails to FTE-decrypt could be handled by following the normal DNS protocol for that query. Care must be taken in the handling of decryption failures, however, in order to avoid side-channel attacks that would give away that FTE is being performed.

Dealing with adversarial proxies. In some settings, FTE ciphertexts may have to cross a proxy controlled by the network operator. HTTP proxies are common, aiding in parsing, analyzing, and potentially blocking connections. Sometimes HTTP proxies will alter the HTTP headers of traffic, and this could cause a problem for FTE ciphertexts, since the header type, order and associated values all may be encoding encrypted bits. (One can view this as another type of active adversarial attack.) Fortunately, there are straightforward methods for handling (at least) header reordering and insertion of headers. For example, by having finalize embed into the FTE ciphertext an encoding of the header order. Then parse can undo proxy header reordering (and insertion). Consistent and known alterations of header values, like changing the HTTP version number, can be accommodated by restricting the regular expressions.

We have already investigated strategies to enable compatibility with the Squid proxy, but will delay discussion of experiments against it (and other proxies) until a future version of this work.

Really stateful DPI. The FTE schemes in this work target the circumvention of DPI adversaries that do not perform cross-message analysis. That is, DPI technology that does not maintain state for each connection. Based on recently published works (e.g. [43]), this seems to be appropriate for national-scale DPI regimes. In this case, it is fine for the client to send a ciphertext that is an HTTP request for a Javascript object, and for the server to respond with a ciphertext that is a PNG object. By introducing higher-level logic into the FTE sender and receiver, such inconsistencies can be managed. We have already tested a version of our FTE system that, for example, has the receiver recognize the type of object requested in the senders HTTP-request ciphertext, and selects the proper format package for the reply. With additional post-processing of FTE ciphertexts (before they are placed on the wire), we can ensure consistent sequence numbers across flows, proper echoing of data in responses, etc.

Traffic analysis. Our initial FTE implementation does not seek to obfuscate the lengths of the underlying plaintext, nor the message timing, both of which have led to traffic analysis attacks in other settings [14], [19], [28], [35]. Future implementations of our system will seek to address this, perhaps using known techniques [44].

Broader Assortment of Languages. For expositional purposes, our discussion has maintained HTTP as a com-

mon thread. However, we note there are limitations in deploying header-only languages. For example, the language "HTTP-response-cg || raw AE" may be identified as encryption by adversaries that perform statistical tests on the HTTP message body. Towards this, we have already learned regular expression families for a variety of other formats, such as Javascript, CSS and HTML. When used in our FTE system, this allows us to use, say, ciphertext HTTP requests in one direction, and ciphertext HTTP responses with Javascript bodies in the other.

Other network protocols may be interesting to "mimic" too, for example DNS. Our learning framework can handle this, and once the regular expressions are prepared, they can be dropped into our FTE system. We will report on these languages and their use in a future version of this work.

REFERENCES

[1] Torproject.org blocked by gfw in china: Sooner or later? Available at: `https://blog.torproject.org/blog/torprojectorg-blocked-gfw-china-sooner-or-later`, 2008.

[2] appid. Available at: `http://code.google.com/p/appid/`.

[3] Mihir Bellare, Thomas Ristenpart, Phillip Rogaway, and Till Stegers. Format-preserving encryption. In Michael Jacobson, Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography*, volume 5867 of *Lecture Notes in Computer Science*, pages 295–312. Springer Berlin / Heidelberg, 2009.

[4] Miguel Bugalho and Arlindo L. Oliveira. Inference of regular languages using state merging algorithms with search. *Pattern Recognition*, 38(9):1457–1467, 2005.

[5] Sam Burnett, Nick Feamster, and Santosh Vempala. Chipping away at censorship firewalls with user-generated content. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 29–29, Berkeley, CA, USA, 2010. USENIX Association.

[6] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 191–206, Washington, DC, USA, 2010. IEEE Computer Society.

[7] Richard Clayton, Steven J. Murdoch, and Robert N. M. Watson. Ignoring the great firewall of china. In *in G. Danezis & P. Golle, eds, Privacy Enhancing Technologies workshop (PET 2006), LNCS*. Springer-Verlag, 2006.

[8] Roger Dingledine. Iran blocks Tor; Tor releases same-day fix. Available at: `https://blog.torproject.org/blog/iran-blocks-tor-tor-releases-same-day-fix`, 2011.

[9] Roger Dingledine. Iran blocks Tor; Tor releases same-day fix. Available at: `https://blog.torproject.org/blog/iran-blocks-tor-tor-releases-same-day-fix`, 2011.

[10] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *In Proceedings of the 13th USENIX Security Symposium*, pages 303–320, 2004.

[11] Peter Dorfinger, Georg Panholzer, and Wolfgang John. Entropy estimation for real-time encrypted traffic identification (short paper). In Jordi Domingo-Pascual, Yuval Shavitt, and Steve Uhlig, editors, *Traffic Monitoring and Analysis*, volume 6613 of *Lecture Notes in Computer Science*, pages 164–171. Springer Berlin Heidelberg, 2011.

[12] Peter Dorfinger, Georg Panholzer, Brian Trammell, and Teresa Pepe. Entropy-based traffic filtering to support real-time skype detection. In *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, IWCMC '10, pages 747–751, New York, NY, USA, 2010. ACM.

[13] Dust. Available at: `blanu.net/Dust.pdf`.

[14] K. Dyer, S. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *Oakland' 12: Proceedings of the 33nd IEEE Symposium on Security and Privacy*, Oakland, CA, 05/2012 In Press. IEEE.

[15] Nick Feamster, Magdalena Balazinska, Greg Harfst, Hari Balakrishnan, and David Karger. Infranet: Circumventing Web Censorship and Surveillance. In *Proceedings of the 11th USENIX Security Symposium*, pages 247–262. USENIX Association, 2002.

[16] Mark E. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[17] A Goldberg and M Sipser. Compression and ranking. In *Proceedings of the 17th Annual ACM symposium on Theory of computing*, STOC '85, pages 440–448, New York, NY, USA, 1985. ACM.

[18] Josh Halliday and Saeed Shah. Pakistan to ban encryption software. Available at: `http://www.guardian.co.uk/world/2011/aug/30/pakistan-bans-encryption-software`, 2011.

[19] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naive-bayes classifier. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, CCSW '09, pages 31–42, 2009.

[20] Amir Houmansadr, Giang T. K. Nguyen, Matthew Caesar, and Nikita Borisov. Cirripede: circumvention infrastructure using router redirection with plausible deniability. In *in Proceedings of CCS*, 2011.

[21] Luca Invernizzi, Christopher Kruegel, and Giovanni Vigna. Message In A Bottle: Sailing Past Censorship. In *Hot Topics in Privacy Enhancing Technologies – PETS*, 2012.

[22] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. Blinc: Multilevel traffic classification in the dark. In *In Proceedings of ACM SIGCOMM*, pages 229–240, 2005.

[23] Kryo. iodine DNS Tunnel. Available at: `http://code.kryo.se/iodine/`, 2010.

[24] l7-filter. Available at: `http://l7-filter.clearfoundation.com/`.

[25] Kevin J. Lang. Faster algorithms for finding minimal consistent dfas. Technical report, 1999.

[26] Andrew Lewman. China block Tor: Round Two. Available at: `https://blog.torproject.org/blog/china-blocking-tor-round-two`, 2010.

[27] Andrew Lewman. Update on Internet Censorship in Iran. Available at: https://blog.torproject.org/blog/update-internet-censorship-iran, 2011.

[28] Marc Liberatore and Brian Neil Levine. Inferring the source of encrypted http connections. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 255–263, New York, NY, USA, 2006. ACM.

[29] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy*, 5(2):40–45, March 2007.

[30] H.M. Moghaddam, B. Li, M. Derakhshani, and I Goldberg. Skypemorph: Protocol obfuscation for tor bridges. In *Technical report, University of Waterloo*, 2012.

[31] Andrew W. Moore and Konstantina Papagiannaki. Toward the accurate identification of network applications. In *In PAM*, pages 41–54, 2005.

[32] Andrew W. Moore and Denis Zuev. Internet traffic classification using bayesian analysis techniques. In *In ACM SIGMETRICS*, pages 50–60, 2005.

[33] Steven J. Murdoch. Proof of concept transport plugin: http headers. Available at: https://trac.torproject.org/projects/tor/ticket/2759, 2011.

[34] Steven J. Murdoch and Stephen Lewis. Embedding covert channels into tcp/ip. In *Proceedings of the 7th International Workshop on Information Hiding*, pages 247–261. Springer, 2005.

[35] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*, WPES '11, pages 103–114, New York, NY, USA, 2011. ACM.

[36] Phobos. Iran partially blocks encrypted network traffic. Available at: https://blog.torproject.org/blog/iran-partially-blocks-encrypted-network-traffic, 2012.

[37] Oscar Santolalla. Why and how to block skype. Helsinki University of Technology, TKK T-110.5190.

[38] Michael Sipser. *Introduction to the Theory of Computation*. Thompson Course Technology, 2nd edition, 2006.

[39] Cisco Systems. Getting started with cisco ios flexible packet matching. http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6586/ps6723/prod_white_paper0900aecd80633b0a.html.

[40] Tor obfsproxy. Available at: https://gitweb.torproject.org/obfsproxy.git, 2012.

[41] torproject. deep packet inspection. Available at: https://blog.torproject.org/category/deep-packet-inspection, 2012.

[42] Tim Wilde. Knock Knock Knockin' on Bridges' Doors. Available at: https://blog.torproject.org/blog/knock-knock-knockin-bridges-doors, 2012.

[43] Philipp Winter and Stefan Lindskog. How the Great Firewall of China is Blocking Tor. In *Free and Open Communications on the Internet*, Bellevue, WA, USA, 2012. USENIX Association.

[44] Charles Wright, Scott Coull, and Fabian Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *Proceedings of the 16th Network and Distributed Security Symposium*, pages 237–250, February 2009.

[45] Charles V. Wright, Lucas Ballard, Scott E. Coull, Fabian Monrose, and Gerald M. Masson. Uncovering spoken phrases in encrypted voice over ip conversations. *ACM Trans. Inf. Syst. Secur.*, 13:1–30, December 2010.

[46] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman. Telex: Anticensorship in the network infrastructure. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

APPENDIX

A. DPI-defined Regular Expressions

L7-filter The following is the regular expression defined by the l7-filter engine to identify HTTP traffic. We use " \" to denote the line break is for formatting and not part of the regular expression.

```
(http/(0.9|1.0|1.1) [1-5][0-9][0-9] \
[\x09-\x0d -~]* \
(connection: \
 |content-type: \
 |content-length: \
 |date:)) \
|post [\x09-\x0d -~]* http/[01]\.[019]}
```