Protocol Misidentification Made Easy with Format-Transforming Encryption

Kevin P. Dyer
Portland State University
kdyer@cs.pdx.edu

Scott E. Coull
RedJack, LLC.
scott.coull@redjack.com

Thomas Ristenpart
University of Wisconsin
rist@cs.wisc.edu

Thomas Shrimpton
Portland State University
teshrim@cs.pdx.edu

ABSTRACT

Deep packet inspection (DPI) technologies provide much-needed visibility and control of network traffic using port-independent protocol identification, where a network flow is labeled with its application-layer protocol based on packet contents. In this paper, we provide the first comprehensive evaluation of a large set of DPI systems from the point of view of protocol misidentification attacks, in which adversaries on the network attempt to force the DPI to mislabel connections. Our approach uses a new cryptographic primitive called format-transforming encryption (FTE), which, intuitively, extends conventional symmetric encryption with the ability to transform the ciphertext into a format of our choosing. We design an FTE-based record layer that can encrypt arbitrary application-layer traffic, and we experimentally show that this forces misidentification for all of the evaluated DPI systems. This set includes a proprietary, enterprise-class DPI system used by large corporations and nation-states. We also show that using FTE as a proxy system incurs no latency overhead and only 16% more bandwidth than standard SSH tunnels. Finally, we integrate our FTE proxy into Tor and demonstrate that it evades real-world censorship by the Great Firewall of China.

1. INTRODUCTION

Network operators increasingly deploy deep packet inspection (DPI) to improve visibility into network activities and control those activities based on identified application-layer protocols and content (HTTP, TLS, BitTorrent, Tor, etc.). DPI systems work by inspecting the contents of packets to find indicators of the protocol being used, such as fixed header values. The most advanced in-use DPI systems use regular expressions, encoding fingerprints for the protocols of interest, to inform port-independent protocol identification [3, 10, 12, 24, 35]. This may be followed by other (potentially) more sophisticated traffic and content analysis methods, or may lead to filtering or other discriminatory behavior. For example, corporations use DPI to block file-sharing protocols that often cause accidental data leakage, while ISPs often use DPI to throttle bandwidth-heavy BitTorrent traffic [22, 53]. More controversially, nation-states use DPI to censor their citizens' use of the Internet, in part by blocking privacy tools like Tor [14].

Despite their growing roles in many security-critical settings, we are unaware of any work that specifically studies the robustness of state-of-the-art DPI protocol-identification tools in the face of dedicated attacks. Thus, we address the following question: *are there practical attacks that will force* any *regular-expression-based DPI into misclassifying connections as protocols of the attacker's choosing?* And, if such attacks do exist, can these misclassified connections be used to carry practically useful amounts of information?

The results of our work conclude that, indeed, misclassification attacks exist against enterprise-grade DPI and that these attacks can be mounted while carrying sufficient information to surf the web, transfer files, and use Tor. It seems that using regular-expression-based DPI to inform protocol identification decisions is a fundamentally flawed approach.

Rather than building an array of ad-hoc schemes to trick this-or-that specific DPI system, we instead target the implicit premise underlying modern DPI that regular expressions (regexes) are sufficient for identifying network protocols. To that end, we develop a generic approach for controlling the format of encrypted data, so that it will match whatever regex we desire to specify. With this ability, we can force protocol misidentification across a broad range of DPI systems.

The development of a generic approach to DPI-evasion implies that current DPI systems will have to move to new, more expensive techniques based on machine learning [26, 33, 34, 55], active probing [28], or something else entirely. At the same time, the approach also suggests a promising way forward for tools aiming to circumvent network censors.

Format-Transforming Encryption. The foundation of our approach is a new cryptographic primitive called format-transforming encryption (FTE). It allows the user to input a regex of their choosing and output ciphertexts that are guaranteed to match it. Therefore, FTE has, built in, a natural mechanism for forcing misidentification by regex-based DPI. It will additionally achieve more traditional privacy and authenticity goals.

We consider a variety of methods to specify the regular expressions that are input to the FTE scheme. In those scenarios where we know which DPI systems are being used, the simplest method is lifting them directly from systems themselves, or manually creating them using knowledge of RFCs and the DPI code. When we do not have information about the DPI system, we provide a simple procedure for learning regexs from network traces of the application-protocols that we wish our traffic to match.

Under the hood, our FTE scheme relies heavily on well-known algorithms for ranking strings in a given regular language [17]. The algorithms were previously suggested for use in the related context of format-preserving encryption [5], however, as far as we are aware, our work gives the first implementation and performance analysis of the algorithms. To realize a working FTE proxy system capable of tunneling arbitrary network traffic, we specify and implement a full, FTE-powered record layer. By this we mean that we build, around the FTE core, logic to manage buffering and fragmentation of incoming plaintext streams on the sender's side, and input ciphertext stream buffering, parsing and fragment reassembly on the receiver's side.

We use this FTE proxy system to explore the resistance of six state-of-the-art DPI systems to protocol misclassification

	DPI	Multi-stage	Classifier Complexity		
System	Type	Pipeline	HTTP	SSH	SMB
			DFA States		
appid	regex-only	✗	15	8	104
l7-filter	regex-only	✗	55	8	6
YAF	regex-only	✓	29	10	5
			Lines of C/C++ Code		
bro	hybrid	✓	1593	30	1188
nProbe	hybrid	✓	807	89	24
DPI-X	?	?	?	?	?

Figure 1: Summary of evaluated DPI systems. *Type* indicates the kind of DPI engine used. *Multi-stage pipelines* chain together several passes over packet contents. *Classifier complexity* is the number of DFA states used for regular expressions or total lines non-whitespace/non-comment C/C++ code.

attacks. We show that even expensive, proprietary systems can be forced to mistakenly identify FTE-protected traffic as any of a number of target protocols chosen by the user, including HTTP, SMB, and SSH. We stress that our approach is shown to work no matter what the FTE-encapsulated application-layer protocol actually is. To the best of our knowledge, this is the first comprehensive analysis exposing how ineffectual modern DPI systems can be rendered.

One immediate implication is that our proxy system provides a ready-made mechanism for circumventing actual, deployed DPI tools. When used to surf the web, FTE imposes as little as 16% bandwidth overhead and no latency overhead compared to conventional encryption of traffic. By comparison, FTE is both more flexible and orders of magnitude faster than existing circumvention tools [20, 30, 52] (see Section 7 for further discussion). Moreover, as a practical matter, our FTE system works as a drop-in pluggable transport [42] for Tor, and we are currently working with the Tor Project to test FTE in countries that block Tor using DPI, such as Iran and China. During initial testing from servers within the Great Firewall of China (GFC), we were able to successfully browse a variety of censored websites and run Tor within an FTE-protected tunnel.

2. MODERN DPI SYSTEMS

In our evaluation we focus on port-independent protocol identification as used by six modern DPI systems. These systems span a wide range of complexity, cost, and expected deployment environments. They also employ a variety of types of DPI engines. A summary of the systems we evaluate and their (known) properties appears in Figure 1, and we explain more below. Unless otherwise mentioned, our discussion and later evaluations will use default configurations.

appid. The appid [3] library uses port-based pre-filtering to determine a set of protocol-identifying regexes, against which each TCP stream should be evaluated. It applies each regular expression in the set against the stream, and returns the first match as the protocol label. A match is attempted for bi-directional (i.e., client-server and server-client) streams. In our evaluation we used the latest available version of appid (as of April 2013), and instantiated it via its included Python module.

L7-filter. The l7-filter [10] software also performs regex-only matching. However, it differs from appid in that it does not pre-filter based on port numbers. Moreover, l7-filter specifies only regexes to identify uni-directional server-to-client streams. In our evaluation we used version 2009-05-28 of the l7-filter userspace classification engine, and invoked it with its included test suite.

Yet Another Flowmeter. YAF [24] is a network monitor that performs application labeling. The YAF protocol-classification engine is predominantly regex-only, although in a few cases (e.g., the classification of TLS) it employs C-based logic. We classify YAF as regex-only because the majority of protocols are classified using a regex-only strategy. For some protocols, YAF performs its regex analysis in two stages; HTTP is one example. A first-pass match for HTTP (caused, say, by matching the string `HTTP/`) triggers a second-pass to extract message contents, such as the HTTP request User-Agent field. In our evaluation we used the latest stable version of YAF (2.3.3, January 2013) built with application identification enabled.

Bro Network Security Monitor. bro's [35] Dynamic Protocol Detection (DPD) is implemented as a set of regexes and associated C/C++ based parsers, a parser being triggered in the event that its partnering regular expression(s) match the input stream. The parser is used to extract additional information from the stream, when possible, and to determine when the regular expression match was actually a false positive. In its default configuration, bro parses individual messages and extracts per-message attributes, such as the URI of an HTTP request, or the version number of the SSH protocol being used. In our evaluation, we used the latest, stable version (2.1, Aug. 2012) of bro, and extracted the assigned stream label from the `service` field of the `conn.log` file.

For classification of SMB streams, which is not supported in version 2.1, we used an alpha-release SMB parser available in the bro git repository. We bootstrapped this parser into an SMB-classifier as follows. A stream was labeled as SMB only if the parser did not encounter parse errors, which is consistent with the strategy of other classifiers present in bro.

nProbe Pro. nProbe [12] is an open-source network monitoring utility that costs €299.95 for commercial use. nProbe includes the ability to identify the encapsulated data within a protocol. As an example, it can identify a web request to YouTube or FaceBook. This means that nProbe can reassemble TCP streams, identify the contents of a flow, and then parse individual message within a flow. nProbe uses hard-coded C-based logic, for the sake of efficiency, to identify attributes that could be captured by a regular expression. As an example, for HTTP it searches for a finite list of values (i.e. GET, POST, HTTP, etc.), in order to identify an HTTP stream — when such a match occurs, this triggers C-based second-pass parsing logic. In our evaluation we used version 6.9.5 of nProbe Pro, and the NetFlow output value `%L7_PROTO_NAME` to determine nProbe's stream label.

DPI-X. We obtained access to an enterprise-grade security gateway device sold by a well-known network equipment manufacturer. The DPI capabilities of this device, as advertised, enable classifying over 900 applications and protocols, including nested and tunneled applications (e.g., Facebook over HTTP). We refer to this system as DPI-X, due to ongoing disclosure discussions with the manufacturer.[1] This device belongs to a class of commercial systems, ranging from lower-end systems capable of handling a maximum throughput of 90 Mbps ($600), up to carrier-grade products capable of 100 Gbps (over $25,000). The system used in our testing has a maximum throughput of 1.5 Gbps and a cost of $8,000. We believe DPI-X is representative of the DPI products employed in many enterprise and carrier networks. In fact, the maker of this product has been identified as one of the suppliers of censorship equipment for Iran [36].

Threat model. A primary goal in this work is to ascertain, experimentally, the extent to which these representative DPI systems are vulnerable to *protocol-misidentification attacks.* (We will use misidentification and misclassification interchangeably.) To define this term, consider a setting in which a DPI system monitors all connections traversing a network. Two parties want to communicate via an application-layer protocol that uses connection(s) traversing the DPI-protected network. We will refer to these two parties as the "attacker" to emphasize that the DPI faces adversarially generated traffic. In a *white-box DPI* attack, the attacker knows the DPI classification algorithms that are used. In a *black-box DPI* setting, the attacker does not. The latter may be because the particular DPI being used is not known (even if the set of possible DPIs is known), or simply because the algorithms are proprietary.

We call the application-layer protocol used for communicating data the *source protocol.* In a misidentification attack, the attacker's goal is to have its connections, which use the source protocol, be (mis)identified by the DPI as a *target protocol* of the attacker's choosing. An example would be to have SSH as the source protocol, and HTTP as the target protocol: should the attack be successful, the DPI will incorrectly label the actual (encrypted) SSH connections as (unencrypted) HTTP connections. To declare a practical success, we also require that the attack does not significantly degrade performance of the source protocol.

Despite the clear importance of protocol misidentification attacks, we are unaware of any prior works that evaluate the DPI systems in our test set (or other similar systems). See Section 7 for discussion of analyses of related settings and potential approaches for forcing protocol misidentification that are different from our approach, which we describe next.

3. FORMAT-TRANSFORMING ENCRYPTION

All of the open-source DPI systems in our evaluation set use membership in a regular language — either explicitly with regexes or implicitly by logic coded in languages such as C/C++ — to inform application-layer protocol classification. We therefore target mechanisms which enable an attacker to force protocol misidentification against any DPI that relies on regex checks. Note that this goal is more aggressive than merely defeating the systems in our corpus (but we will do that as well!), and we expect our approach will work against any currently deployed DPI system.

Our main idea is to build into encryption schemes resistance to regex-based DPI. The key technological enabler, which we explain further in the remainder of this section, is augmenting the normal encryption interface to take a regex as an input. The purpose of this regex is to specify the format of ciphertexts: this means that ciphertexts, when taken as a string over the appropriate alphabet, are guaranteed to match against the specified regex. We call the resulting primitive format-transforming encryption (FTE). In turn, we show how to use FTE as a component within a record layer that handles streams of messages from some source protocol. By proper choice of regex, ciphertexts produced by this record layer –which are actually carrying the source protocol– will be classified as messages from another protocol (of our choosing) by the DPI.

Our FTE record layer will be used for two purposes. First, we will build an FTE (single-hop) proxy system by combining the FTE record layer with SOCKS in a straightforward way. We will then use it to support our hypothesis that the regex-based DPI used in practice is fundamentally flawed. We do this by showing how to use the FTE proxy to force protocol misidentification by the entire set of DPI systems and with respect to a variety of target protocols. We also show that doing so has essentially negligible overhead. We believe this suggests that DPI systems must move to more advanced mechanisms (discussed in Section 7) in settings with adversarial users on their network.

Our second purpose, empowered by the ease with which it forces protocol misidentification and the high performance it obtains, will be to incorporate the FTE record layer into Tor for use in circumventing censorship. We discuss this more in Section 6.

3.1 FTE via Encrypt-then-Unrank

An FTE scheme is a triple of algorithms: key generation, encryption and decryption. Key generation works as in conventional encryption, outputting a randomly chosen symmetric key K. Encryption Enc takes inputs a key K, a format $\mathcal{F}$, and a message M. It can be randomized, stateful, or deterministic, and always outputs a ciphertext C or a distinguished error symbol $\perp$. Decryption Dec takes input a key K, a format $\mathcal{F}$, and a ciphertext C. It outputs a message or $\perp$. The format $\mathcal{F}$ specifies a set $L(\mathcal{F})$ called the language of $\mathcal{F}$. The requirement is that any C output by Enc must be a member of $L(\mathcal{F})$.

FTE is a related to format-preserving encryption (FPE), first formalized by Bellare, Ristenpart, Rogaway, and Stegers (BRRS) [5]. FPE is used in the context of in-place encryption of credit-card numbers (or other records) within databases. It likewise uses formats, but requires that both messages and ciphertexts be members of the format-specified language.

We desire FTE schemes that support formats described by regexes. This will allow easy "programming" of formats and endow FTE with the same expressive power as regex-using DPI. To do so, we start by following an approach similar to one used by BRRS. Loosely speaking, to implement $\mathsf{Enc}(K, \mathcal{F}, M)$ for $\mathcal{F}$ a regular expression we: (1) encrypt M using a standard authenticated encryption scheme to obtain an intermediate ciphertext Y; (2) treat Y as an integer in $\mathbb{Z}_{|L(\mathcal{F})|}$ (the set of integers from 0 to the size of the language minus one); and (3) apply an encoding function $\mathsf{unrank}\colon \mathbb{Z}_{|L(\mathcal{F})|} \to L(\mathcal{F})$. To be able to decrypt, we require that unrank is in fact a bijection with efficiently computable inverse $\mathsf{rank}\colon L(\mathcal{F}) \to \mathbb{Z}_{|L(\mathcal{F})|}$.

The key algorithmic challenge is implementing rank and

[1] We hope to provide the actual identity in the camera-ready version of this paper.

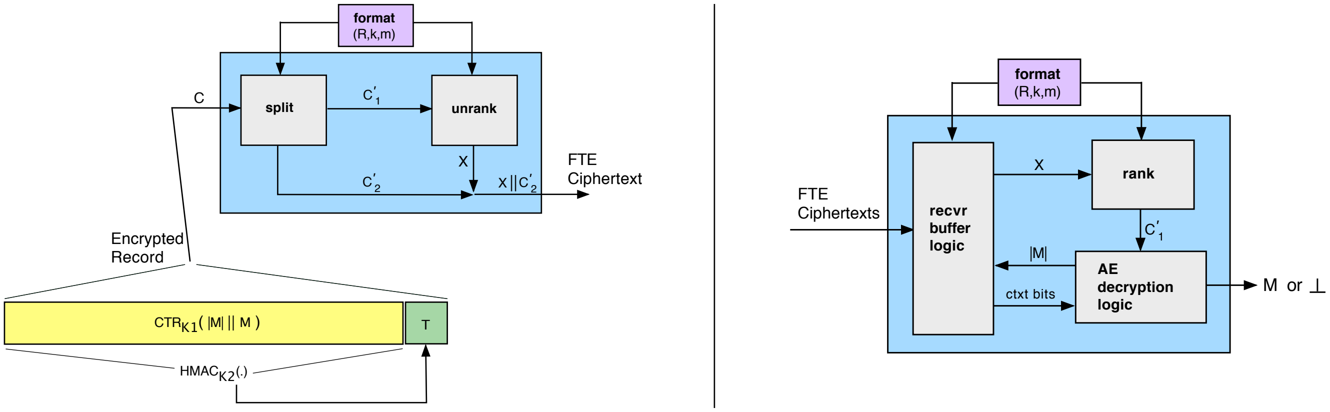

Figure 2: Sender-side (left) and receiver-side (right) record-layer flow. We discuss the various modules in the text.

unrank efficiently. These associate to each string in the language its rank, i.e., its position in a total ordering of the language. The notion of ranking was first explored by Goldberg and Sipser [17] in the context of language compression. They also there gave an efficient way to rank a regular language, when the language is represented by a deterministic finite automaton (DFA). BRRS used this for an (unimplemented) FPE scheme for arbitrary regular languages encoded as DFAs, but they also emphasize that, asymptotically speaking, there is provably no way to give efficient rank and unrank functions starting just from a regex. Standard tools exist for converting from a regex to an nondeterministic finite automaton (NFA) and from there to a DFA (see Section 5), but the second step potentially incurs an exponential blow-up in state size. We observe that this worst-case behavior is not an issue for FTE, in part because the kinds of regexes used by DPIs are themselves designed explicitly to avoid the worst-case blowup.

Our implementation uses the time-space tradeoff of Goldberg and Sipser to support more efficient runtime performance, precomputing tables that allow (un)ranking of all strings $x \in L$ with $|x| \leq n$. Appendix A details the algorithms. 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 found several implementation optimizations that greatly speedup ranking/unranking in practice, for example short-cutting the inner loop of (un)ranking in cases where DFA states only have one outgoing transition. One can also formalize all of the above and prove that the Encrypt-then-Unrank approach preserves the message confidentiality and authenticity security of the underlying authenticated encryption scheme. Due to space constraints, we omit details of the optimizations and formal treatment, and instead move on to discuss how we build a full FTE record layer.

3.2 FTE Record Layer

In order to transform arbitrary TCP streams, we need additional "record layer" machinery to buffer, encode, unambiguously parse, and decode streams of FTE messages. In Figure 2 we give a diagrammatic overview of the processes by which plaintexts are transformed into FTE ciphertexts, and vice versa. We assume that sender and receiver share a pre-established set of keys, possible derived from a single shared master key. Our record layer also assumes an underlying, reliable network transport protocol, e.g. TCP.

Implementing FTE formats. We will find it useful to specify in our formats more than just a regular expression. Thus a format $\mathcal{F}$ for the record layer is as a tuple (R, k, m): a regular expression R, a number $k > 0$ that will specifies a length of strings to use from the language $L(R)$ of R, and an integer $m \geq 0$. The reason to use a specific length k is that it is an expedient way of rendering FTE ciphertexts easy to parse. The value m will be used to cheaply enable more capacity in cases where the desired language is, in fact, any string with a prefix in $L(R)$. All told, for $\mathcal{F} = (R, k, m)$ the language becomes $L(\mathcal{F}) = (L(R) \cap \Sigma^k) \parallel \Sigma^{\leq m}$ where Σ is the alphabet underlying R. For simplicity we assume $\Sigma = \{0, 1\}$ in what follows, but in implementations one will typically use larger alphabets. In our implementation we use values $m = 2^{15}$ and $k = 2^7$, unless otherwise specified.

FTE Sender. See the left diagram in Figure 2 and consider a format $\mathcal{F} = (R, k, M)$. Let $L_k(R) = L(R) \cap \{0,1\}^k$ be the k-bit slice of $L(R)$ we will use and $t = \lfloor \log_2(|L_k(R)|) \rfloor$ be that slice's capacity (the number of bits one can encode using the slice). The first action on the sender's side is to prepare an encrypted record using a secret key K. From a plaintext message buffer, grab a plaintext message M of length at most $|M| < 2^{15}$. Then form a plaintext record, containing an encoding of $|M|$ followed by M. The record is then encrypted using a standard, stateful authenticated-encryption (AE) scheme with K to produce a ciphertext C. We assume that $|C|$ is defined solely by $|M|$, which is true for AE schemes used in practice. We also pad M, if required, to ensure that $|C| \geq t$. Our implementation uses CTR mode over AES and then authenticates the resulting ciphertext with HMAC-SHA256

The encrypted record C is passed to the split module, and appended to an internal buffer maintained by split. The job of split is to produce two strings: one to be passed to unrank for formatting, and one that will be passed along as-is. In particular, split takes up to $t + m$ bits (and at least t bits) from the front of the buffer. Call the string of bits taken C'. Note that C' may be a full AE ciphertext, part of one, or include bits from multiple ciphertexts. Then split partitions C' into C'_1 and C'_2 with $|C'_1| = t$ and $|C'_2| \leq m$.

The first portion C_1' is forwarded to unrank, which produces a formatted string $X \in L_k(R)$. Finally, the concatenation $X \,\|\, C_2'$ becomes the sender's FTE ciphertext which can then be transmitted.

FTE Receiver. Referring now to Figure 2, the receiver buffer logic is responsible for managing the stream of incoming FTE ciphertexts. An issue here is that we do not include in the transmitted bits from the sender any plaintext headers, which in normal record layers would be used to demarcate ciphertext boundaries. We therefore must take care to ensure that the receiver can correctly decrypt given that its buffer may contain multiple contiguous ciphertexts. To do so, the receiver takes advantage of the fact that the sender used a fixed[2] slice $L_k(R)$. As soon as it has the first k FTE ciphertext bits in its buffer, it treats these as a string $X \in L_k(R)$ and applies $\mathsf{rank}(X)$ to recover C_1'. (Should $X \notin L_k(R)$ decryption should abort.) It then feeds C_1' to the AE decryption algorithm in order to retrieve $\ell = |M|$ and possibly some initial bits of a message M. (Those latter message bits should not yet be released to higher layers.) Note that this means that the AE scheme must be able to perform incremental decryption and that it should not be vulnerable to attacks (c.f., [1]) that abuse use of the length field before ensuring integrity. CTR mode plus HMAC provides these properties.

Given ℓ the receiver now knows how many more bits of AE ciphertext are expected. From here it can remove the next up to m bits of ciphertext from the input buffer, and then go back into a state where it treats the subsequent k bits in the buffer as a string in $L_k(R)$, applying rank as above, and so on. When it retrieves a full AE ciphertext, it finishes decryption, verifies integrity of the ciphertext, and only now releases the buffered message bits up to the application.

Regex negotiation. One feature of FTE is the ability to quickly change regexes on the fly. We would like to be able to provide in-band negotiation of formats. But since all data sent on the wire must be formatted (to pass DPI checks), it is not possible to negotiate regexes in the clear. We alleviate (but do not totally remove) this limitation, by allowing the client/server to agree upon regexes for the duration of a TCP connection assuming they support some initial large set of possible regexes. In more detail, we assume the client and server have shared (ordered) list of FTE formats $(\mathcal{F}_1, \ldots, \mathcal{F}_n)$. We still assume the client and server have negotiated cryptographic keys out-of-band. For each TCP connection the client determines the FTE format $\mathcal{F}_i$ it wishes to use for client-server messages, and the $\mathcal{F}_j$ it wishes to use for server-client message. Then, the client constructs the message $M \leftarrow \langle i \rangle \,\|\, \langle j \rangle$, encrypts M as a distinguished `negotiate` message-type, designated by the value of a first (reserved) byte of the plaintext, encodes it with $\mathcal{F}_i$, then sends it to the server.

When the server receives an initial message from the client it iterates through its list of formats, attempting to decode the message with each of the FTE formats. Once it encounters a successful decryption, it evaluates the message and then uses for the session $\mathcal{F}_i$ for client-server messages and $\mathcal{F}_j$ for server-client messages. The server finalizes the negotiation by responding to the client with a distinguished `negotiate_acknowledge` message-type.

We can improve on the naive receiver-side implementation of this procedure by having an implementation of rank that contains special checks which short-circuits evaluation to terminate early in cases when the string being ranked is not accepted by the DFA. This enables the server to quickly exclude certain formats, thereby making it possible to quickly support dozens of formats.

4. PROTOCOL MISCLASSIFICATION

In this section we experiment with three strategies for providing regexes for use with our FTE record layer: we extract regexes from open-source DPI systems (Section 4.1); program regexes manually (4.2); and automatically learn regexes from samples of target protocol traffic (4.3). We will always use a pair of formats: one for upstream (client-to-server) and one for downstream (server-to-client) communications. We will then use our FTE record layer to asses the vulnerability of the DPI systems in our evaluation to protocol-misidentification attacks. Looking ahead, any of the three strategies we explore is able to succeed in forcing misclassification by the DPI systems in our evaluation set.

Recall that a source protocol is the application layer protocol whose connections the attacker wants to have misclassified. We explored HTTP [16], HTTPS [13], secure copy (SCP) [54], and Tor [14] as source protocols: the choice of source protocol does not influence our results. We therefore focus our discussion primarily on HTTP(S).

For each regex generation strategy we explore three target protocols: HTTP [16], SSH [54], and SMB [29]. The latter is a proprietary plaintext protocol designed by Microsoft that supports sharing of resources such as files and printers over networks. We also explored other target protocols, including SIP [37] and RTSP [38], but limit our discussion here to the prior three as the results for the others were the same. Indeed, we suspect FTE can be successfully used for almost any target protocol.

Experimental testbed. We implemented our FTE record layer as a library that can be used to relay arbitrary data streams. More implementation details appear in Section 5. In our evaluation we use it in the following configuration. The FTE client listens for incoming TCP connections on a local client-side port. Upon receipt of an incoming connection the FTE record layer encrypts the messages using the upstream format and relays them to a remote FTE server. The FTE server receives these FTE ciphertexts, decrypts them, and then relays the recovered plaintexts to a preconfigured destination TCP port. Downstream, the process is reversed, with the FTE server encrypting returned messages using the downstream format.

We use Mozilla Firefox version 17.0.3, controlled by version 2.28.0 of the Selenium browser automation framework to request the top 50 Alexa (US) websites over the FTE record layer; this is repeated five times for each upstream-downstream format pair tested. It generates a mixture of HTTP, HTTPS and DNS source traffic and roughly 12,000 TCP connections on average. On the server-side we relayed all HTTP(S) messages via a SOCKS proxy. We used the default Firefox configuration, with the following exceptions: we tunneled all DNS requests through the SOCKS connection, specified our start page as blank, and disabled caching to disk.

4.1 DPI-Extracted Regular Expressions

[2] One could instead rely on $L(R)$ being prefix-free, but we found using fixed slices simpler and sufficient.

Format	Misclassification Rate					
	appid	l7-filter	YAF	bro	nProbe	DPI-X
appid-http	1.0	0.0	1.0	0.0	0.0	1.0
l7-http	0.0	1.0	0.16	0.0	0.0	1.0
yaf-http1	0.0	0.0	1.0	0.0	0.0	1.0
yaf-http2	0.0	0.0	1.0	0.57	0.0	1.0
appid-ssh	1.0	0.32	1.0	1.0	0.0	1.0
l7-ssh	0.16	1.0	0.16	0.13	0.0	1.0
yaf-ssh1	1.0	0.31	1.0	1.0	0.0	1.0
yaf-ssh2	1.0	0.21	1.0	1.0	0.0	1.0
appid-smb	1.0	1.0	1.0	0.08	0.0	1.0
l7-smb	0.0	1.0	0.38	0.0	0.0	1.0
yaf-smb1	0.0	0.04	1.0	0.0	0.0	1.0
yaf-smb2	0.0	0.04	1.0	0.0	0.0	1.0

Figure 3: Misclassification rates for the twelve DPI-Extracted FTE formats against the six classifiers in our evaluation testbed.

As our first method of programming our FTE record layer, we build FTE formats by lifting them directly from the source code of the open-source regex-only DPI systems. This models a white-box DPI attack setting for the DPI systems from which we extract regexes. For other DPI systems (e.g., DPI-X) it models a black-box DPI attacks.

Extracting regular expressions. Using regexes from DPI systems within our FTE record layer is straightforward. Given a regular expression R from the DPI, we specify a format $\mathcal{F} = (R', k, m)$ where R' is exactly R except without a ".*" (match any) prefix, should there be one. This makes $L(R') \subseteq L(R)$, while ensuring that FTE ciphertexts will include the formatting information found in R after the prefix. We set $k = 128$ and $m = 2^{15}$ for good performance (the misclassification results do not change with other reasonable settings).

We use the naming convention of ⟨DPI system⟩-⟨target protocol⟩ to reference a pair of upstream-downstream regexes extracted from the given DPI system. (Some systems only have a regex for a single direction, in which case we use ".*" for the other direction.) YAF contains at least two regexes for each target protocol, and we indicate that using a 1 or 2 in the format name. The result is 12 different formats.

Misclassification evaluation. The misclassification rates for all 12 formats against the 6 classifiers appears in Figure 3. Here (and throughout this section) the rate is calculated as the number of TCP connections labeled as the target protocol by the classifier, divided by the total number of TCP connections generated when using that FTE format. Thus a rate of 1.0 means complete misclassification success, and 0.0 is complete failure. Throughout our evaluations, a connection that failed to be misclassified as the target protocol by a DPI system was always marked as an unknown protocol, regardless of the DPI system.

What Figure 3 reveals is that DPI-extracted regexes always succeed against the DPI system from which they were extracted, and can even force misclassification by different DPI systems. For an example of the latter, we need only look at DPI-X, which is by far the easiest system to force misclassification against despite it being black-box[3]. While we still do not know the exact DPI strategy used, our best guess is that DPI-X performs minimalistic analyses, favoring performance and optimistic labeling of protocols. These regexes were not effective against nProbe, and had varied success against bro. This can be attributed to the latter DPI systems requiring slightly more stringent protocol conformance.

Intersection formats. We also consider formats whose regex R is the explicit intersection of multiple regexes. FTE based on such an intersection format will produce ciphertexts matching all of the individual regexs used to form R. Using the intersection of the four DPI-extracted formats (for each target protocol) resulted in formats with perfect 1.0 misclassification rates for appid, l7-filter, YAF, and DPI-X. The rates against bro and nProbe are comparable to those for the individual regexes.

4.2 Manually-Generated Regular Expressions

As our next strategy for producing regexes, we will code them manually. Our FTE record layer makes this easy: indeed most developers are already familiar with regexes due to their use in other programming contexts. Moreover, it turns out to be very simple to build fast, simple regexes that achieve perfect misclassification for *all* classifier/target protocol combinations in our evaluation set.

Coding regexes for FTE. We started by inspecting the open-source DPI systems, in particular for cases from the last section where the extracted regexes failed, in order to educate regex design. For HTTP regular expressions, we observed the following requirements. l7-filter requires that responses have an HTTP version of 0.9, 1.0, or 1.1; an HTTP status code in the range of 100-599; and one of the following property-fields: Connection, Content-Type, Content-Length, or Date. appid requires that responses have a string of length greater than zero following the status code, and that the status line is terminated with `\r\n`. YAF requires that we have a valid HTTP method verb for requests (i.e. GET, POST, etc.). nProbe requires that we terminate HTTP messages with `\r\n\r\n`. Finally, bro require that we have no payload for HTTP requests, or a payload and a valid Content-Length field — we accommodate this requirement by not allowing a payload for requests and specifying an FTE format parameter of $m = 0$. (Section 3.1)

For SSH all classifiers require that the first downstream, and in some cases upstream, messages start with `SSH-`. l7-filter demands that the first two messages in a stream start with `SSH-1.x` or `SSH-1.y`. nProbe requires the first messages in an SSH stream to be less than or equal to 99 bytes long, and we achieve this by setting our FTE format parameter to $k = 99$ for both directions of traffic, which constricts message lengths.

All classifiers in our evaluation require that SMB messages have a valid SMB fingerprint, which is the byte `\xFF`, followed by `SMB` encoded in ASCII. In addition, nProbe requires that message have a valid length that matches the length of the message payload, that the length field is located as the first 32-bit word in the message, and that `SMB` is encoded as ASCII in the second 32-bit word. Here we encounter a check that is not easily encoded as a regular language (or avoided as above for nProbe's checking of HTTP Content-Length fields), at least if one wants to support all 2^{32} possible lengths. But we can simply use a specific value for the

[3] Surprised by this, we double-checked that DPI-X works at all, but it does seem to identify real Facebook traffic as Facebook, and SSH traffic as SSH, etc.

length field and fixed-size payload, and as well we set the FTE format parameter m to zero, meaning that we don't append any raw AE ciphertext bytes.

Misclassification evaluation. We specified regexes that met the above requirements for each of the target protocols. It took less than 30 minutes for one of the authors to specify each regex and debug it by testing it against known DPI engines. The resulting regexes achieved perfect misclassification (Figure 4) for all classifier/target protocol combinations: every TCP connection was tagged as the target protocol of our choosing.

4.3 Automatically-Generated Regular Expressions

When we know the DPI systems and their respective classification mechanisms, the previously described DPI-extracted and manually-generated regexes provide guaranteed evasion and optimal capacity. Unfortunately, there are many cases where it is not possible to know this information, like when the DPI classification strategy abruptly changes or when proprietary systems are used. In these situations, we can use a simple but effective process to automatically generate regexes from network traffic samples using widely-available protocol parsers. This allows us to implicitly learn FTE formats from data that is assumed (or known) to pass DPI scrutiny without raising alerts. Although other, more complex, methods of format discovery are available [7,8,41,50], we focus on well-known network message formats to avoid unnecessary complexity while still providing robust regex generation capabilities. The regex generation process proceeds as follows.

1. Collect packet trace data for target protocol message.
2. Apply a parser to label message fields.
3. Create a set containing observed values for each field, called a *dictionary*.
4. Create a *template* for each message type by replacing the values with placeholders for the associated dictionaries.
5. Convert each dictionary into a regex by concatenating the values with an "or" operator between them.
6. For each template, replace the placeholders with the associated dictionary regexes.
7. Choose one or more of the resultant template regexes as the language(s) used by the FTE system.

There are a number of ways this method can be tuned to adjust the quality and capacity of the resultant language(s). First, we can control the properties of the messages that are collected in the packet trace data, such as their message types or payload lengths. Data containing a single, consistent message type will produce more coherent regexes at the cost of smaller dictionaries, while a mix of message types will produce much larger dictionaries with more capacity but with potentially inconsistent, low-quality regexes. We can also control the regexes through the granularity of the parsers used to break the data into fields. Fine-grained parsing produces a greater number of dictionaries within each template and, consequently, an increase in the number of possible combinations among their values. Conversely, coarse parsers will create templates that are more likely to produce valid outputs, but with less overall capacity.

Regex generation. The packet trace data used to evaluate the security of the generated regexes was produced by agents that randomly logged into and crawled HTTP, SMB, and SSH servers. For HTTP, we used the *wget* utility to download the front page of a random selection of web sites on the Alexa Top 1000. SMB and SSH data was generated by scripts that logged into local Linux and Windows servers, randomly crawled the directory structure, accessed files, and logged out several times over the course of a one-hour period. We partitioned the captured data into groups based on their message types and payload lengths. From these partitions, we extracted client headers for HTTP POST requests, SMB transaction requests, and SSH handshake banner messages. The server messages that we use include HTTP 200 OK responses, SMB transaction responses, and SSH handshake banner messages. Each of these message types was parsed using their respective Wireshark dissectors, and the highest-capacity template regular expression was chosen for the FTE format. Any remaining payload bytes not included in the parsed message header were automatically replaced with a regular expression that produced the appropriate number of random bytes.

Format	Misclassification Rate					
	appid	l7-filter	YAF	bro	nProbe	DPI-X
manual-http	1.0	1.0	1.0	1.0	1.0	1.0
manual-ssh	1.0	1.0	1.0	1.0	1.0	1.0
manual-smb	1.0	1.0	1.0	1.0	1.0	1.0
auto-http	1.0	1.0	1.0	1.0	1.0	1.0
auto-ssh	1.0	1.0	1.0	1.0	0.0	1.0
auto-smb	1.0	1.0	1.0	1.0	1.0	1.0

Figure 4: Misclassification rates for the manually-generated and automatically-generated FTE formats against all six classifiers.

Misclassification evaluation. The generated regexes achieved perfect misclassification rates (Figure 4) except for the nProbe SSH classifier, which had a misclassification rate of zero. The nProbe SSH classifier requires that the first message in each direction be an appropriately formatted banner message with an arbitrary length limitation of 100 bytes. Since our regex generation method is limited to only using what it observes, in this case a single client banner and less than five server banners, the generated regex would have only two bits of capacity. To enhance the capacity, we artificially generated RFC-compliant banner messages with the optional comment field used to carry random bytes up to the specified maximum length of 255 bytes, which provided sufficient capacity and the ability to evade all classifiers but nProbe. This highlights a natural limitation of the simple generation process, though avenues for improvement are possible through more advanced generalization procedures or the use of multiple FTE formats within a single connection (e.g., zero-capacity banner messages followed by high-capacity key exchange messages).

5. PERFORMANCE

Our FTE prototype was developed in C/C++ and Python. Performance-critical algorithms such as rank, unrank, and BuildTable (pseudocode is in Appendix A) are implemented in C/C++. We use a customized version of the re2 library for regular expression to DFA conversion, and OpenFST for DFA minimization. Cryptographic algorithms are implemented using PyCrypto. Multiple precision arithmetic is

FTE format	DFA states	avg. unrank (ms)	avg. rank (ms)
intersection-http-downstream	70	0.52	0.48
intersection-smb-downstream	104	0.55	0.54
intersection-ssh-downstream	11	0.55	0.52
manual-http-downstream	38	0.43	0.43
manual-smb-downstream	130	0.53	0.5
manual-ssh-downstream	9	0.42	0.42
auto-http-downstream	13,815	1.6	1.5
auto-smb-downstream	222	0.82	0.79
auto-ssh-downstream	237	0.52	0.49

Figure 5: Average rank and unrank performance for our downstream FTE formats.

performed using GMP. Logic for the record layer and networking is multi-threaded and implemented in Python. The source code for our prototype is publicly available under the GNU General Public License, and a binary distribution for Linux (inclusive of all FTE formats discussed in this document) is less than 7MB in size.

For performance benchmarks our client machine was an AMD Opteron 8220 SE @ 2.80GHz running CentOS 6.4 and server was an Intel Core i5-2400 @ 3.4GHz running Ubuntu 12.04. Each machine was connected to the Internet at an academic institution.

Rank/Unrank. Recall that for FTE we must perform unranking starting from regexes, which can require exponential time in the worst case (see Section 3.1). In practice, however, the regexes we require for misidentification attacks admit fast (un)ranking. In Figure 5 we present the DFA size and average time (over 100k trials using random integers) to perform ranking and unranking for our intersection, manual, and automated formats. We present performance results for downstream formats only. In all cases the upstream formats perform better (lower (un)ranking cost) than their downstream counterpart in (un)ranking benchmarks and have smaller DFA state-spaces. The FTE format parameters used in these benchmarks are described in Section 3.2. The DPI-extracted intersection formats and manual formats resulted in very compact DFAs (no more than 130 states). We note that the automatically generated regexes resulted in the largest DFAs, but were still very fast even for $>10,000$ state DFAs. This all evidences that the worst-case blow-up in state sizes when converting from an NFA to DFA does not greatly impede performance.

Web-browsing Performance. We setup our FTE record layer to proxy HTTP(S) traffic as described in Section 4. As a baseline for comparison, we use a conventional encrypted tunnel. Using OpenSSH's integrated functionality we established a SOCKS proxy which listened client-side, such that all connections were routed through the SSH connection to the remote server, and then ultimately on to the destination IP address. We call this our *socks-over-ssh* configuration.

We download the Alexa top fifty websites five times each using the socks-over-ssh setup. We do the same with the FTE proxy for each of the intersection, manual, and automatically generated regexes. Each of the four separate runs (inclusive of socks-over-ssh) therefore resulted in 250 data points. In our socks-over-ssh configuration, websites in the Alexa top fifty took an average (over the fifty websites and five trials) of 5.5 seconds to render the webpage and all dependencies, and required an average of 1,164KB of data transfer including all TCP/IP overhead. In Figure 6 we show the cumulative distribution of the latency and bandwidth for the four different runs, where we have included in the CDF all 250 points for each run.

The lowest average download time of the FTE formats was the intersection-ssh format, which incurred no increase (5.5s avg.) in latency compared to socks-over-ssh, and 1,348KB (16%) increase in data transfer overhead. The highest average download time of all the formats was the automatic-http format, with an average page-render time of 7.1s (29%) and an average of 3,279KB (181%) transferred per website. The increase in data transferred is due to (1) ciphertext expansion and (2) Firefox generating persistent TCP connections that cause FTE/SOCKS negotiation, but do not result in data transfer. (These latter empty connections do not use much bandwidth with socks-over-ssh.)

In our testing we note that engineering issues often overshadow the overheads of FTE. For example, the socks-over-ssh system performed better on sites with low bandwidth requirements and a small number of TCP connections. Our FTE prototype actually performs *better* than socks-over-ssh for websites that use a large number of TCP connections. This is because our FTE implementation uses multi-threading more aggressively than OpenSSH, and this is better aligned with the Firefox's use of multiple TCP connections in parallel.

Goodput. As a goodput baseline, we performed a direct copy of a 100MB file with SCP from our server to our client and achieved 58Mbps on average over 100 trials. Round-trip latency between the client and server was 70ms on average. In our evaluation we used a connection (university-to-university) with such high capacity because tests on residential-grade connections could not measure the maximum throughput of our FTE prototype.

When using SCP with the FTE record layer, our best performing format was intersection-ssh and it achieved 42Mbps. All other FTE formats with $m > 0$ exhibited similar performance. For our worst performing format, automated-http, we achieved 1.9Mbps and other formats with $m = 0$ had similar goodput. The slower performance for the latter formats stems from their not allowing raw AE ciphertext bytes to follow unrank output.

In our goodput tests our FTE implementations was never CPU bound. Hence, the performance of our (un)ranking algorithms was not the bottleneck. Profiling of our FTE prototype indicates that future performance gains could be had by optimizing the buffer management and networking logic.

6. CENSORSHIP CIRCUMVENTION

In the previous sections, we focused on using FTE to evaluate the efficacy of modern enterprise-grade DPI. We showed that not only can our FTE record layer easily force DPI misclassification, but it can do so while incurring almost negligible performance impact. This suggests that FTE can be a useful tool for settings where one wants to circumvent DPI-enabled censorship. Here we experimentally investigate integrating our FTE record layer into a fuller circumvention mechanism. Specifically, we consider using FTE as a pluggable transport [2] for Tor.

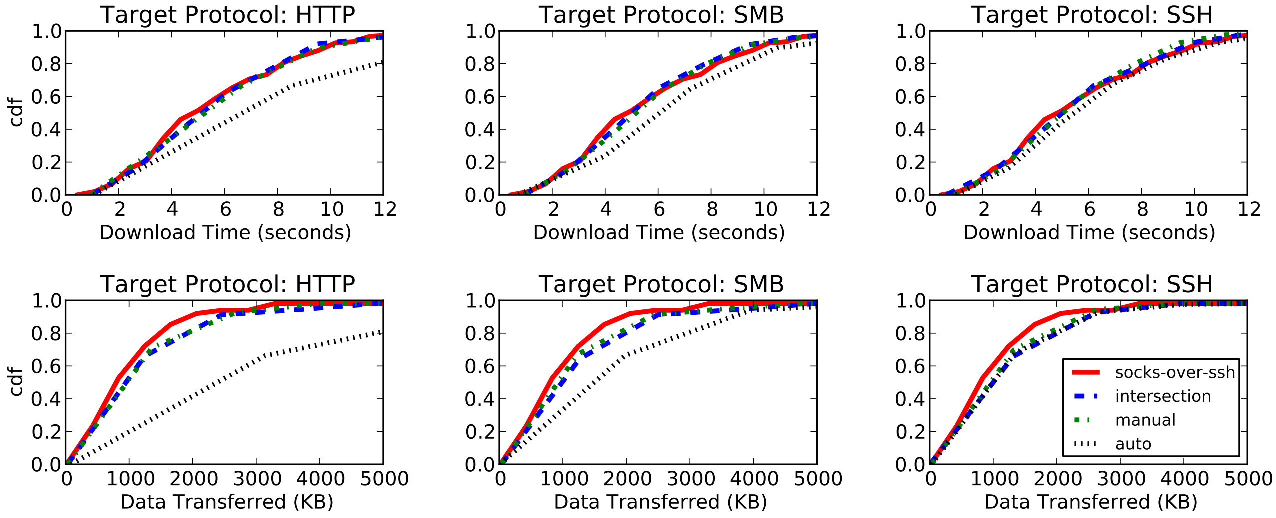

Figure 6: Distribution of webpage (Alexa top fifty) download times (top row) and data transferred (bottom row) for our intersection, manually-generated and automatically-generated FTE formats, compared to using our socks-over-ssh configuration.

Integration. A pluggable transport is a record-layer mechanism that processes Tor messages before being transmitted on the wire. The only currently deployed transport is obfsproxy [42], which applies a stream cipher, using a shared key, to every bit output by Tor. Originally, this shared key was a hard-coded, but a newer version (not yet deployed) replaces this with an in-band, anonymous Diffie-Hellman key exchange [43]. The result of this latter approach is a cryptographic guarantee that all the bitstrings seen by a (passive) DPI are indistinguishable from random strings, meaning, in particular, that the obfuscated Tor messages will not have fixed fingerprints. This does not, however, force protocol misclassification.

We can do better using our FTE record layer as the pluggable transport. Integrating it into Tor with a hard-coded key is immediate, and it is straightforward to add key exchange to our record layer. Specifically, one could just initiate sessions using the existing obfs3 [43] Diffie-Hellman key exchange, but running the key-exchange messages through our unranking mechanisms before being sent on the wire. (Since the messages in this exchange are indistinguishable from uniformly random bit strings, they behave like the AE ciphertext bits in our record layer.) Together with our existing library of regex formats, we arrive at a version of Tor that can easily force misclassification for the DPI systems currently used in practice. Indeed we verified that misclassification rates for all the six systems in our corpus are as seen in Section 4, but now using Tor with FTE as the pluggable transport. We view FTE as a pluggable transport attractive for several reasons:

- *Flexibility*: The FTE record layer (already) supports a variety of target protocols, and adding new ones requires only specifying new regexes. Extending prior steganographic (see Section 7) systems to support many targets would be very labor-intensive.
- *Sufficiency*: The FTE record layer forces misclassification by all evaluated DPIs, even DPI-X whose proprietary classification strategy is unknown to us and is similar to systems used in censorship settings [36].
- *Speed*: The FTE record layer has essentially negligible overhead and, when used with Tor, its overheads are lost in the noise of the Tor network's performance variability.

FTE through the GFC. We set up an FTE client on a Virtual Private Server (VPS) located within China, and an FTE server in the United States. The server was configured to accept incoming connections on port 80. To set a censorship baseline, we first attempted to browse several websites that are known to be censored, including YouTube and Facebook, *without* using FTE to tunnel the traffic. Indeed, these sites were blocked. We then attempted to browse the same websites through the FTE tunnel, using our intersection, manual, and automatic HTTP formats. In every case, the FTE tunneled traffic successfully traversed the GFC, and we were able to browse the censored websites.

Next, we considered using FTE to tunnel Tor traffic. Again, to set a baseline, we attempted to connect to a private Tor bridge listening on port 443 of our server, using the default Tor distribution. We observed behavior consistent with the recent analysis of the Great Firewall of China (GFC) by Winter et al. [49]: initial Tor connections to our private bridge were successful, and they were followed up by an active probe from a Chinese IP address after roughly 15 minutes. The probe performed a handshake with the bridge, then blacklisted the (ip,port)-combination used by the bridge. We validated the blacklisting by observing that subsequent attempts to connect to our Tor bridge (ip,port)-combinations resulted in a successful SYN packet from the VPS to our bridge, followed by spoofed TCP RSTs transmitted to the client and bridge to terminate the TCP connection.

Having established that Tor was indeed being censored, we then attempted the same tasks through our FTE tunnel, again using each of our intersection, manual, and automatic HTTP formats. Despite port 443 being blacklisted from our previous Tor tests, using FTE on port 80 was successful, and we were able to circumvent the GFC with this FTE-tunneled Tor circuit.

On detecting FTE. Censors have been aggressive at rolling out new DPI-based mechanisms for detecting and blocking circumvention tools. How will FTE fare in this kind of arms race? The first idea would be for DPI systems to (attempt to) obtain the FTE regex formats and then use it to mark any traffic exactly matching the regex as FTE. For most of the regexes we consider, this would lead to prohibitively high false positive rates (e.g., 100% of HTTP traffic). The one exception is the automatically generated regexes, which may not match against other traffic, because of (for example) time stamps or unique hash values that were learned from collected traffic traces.

A second approach might be for DPI to perform more nonregular checks, such as verifying correctness of length fields for protocols that include them, e.g., the Content-Length field of HTTP responses. Note that actually this example would not work for DPI in practice, as one-third of the response messages generated when downloading the Alexa top 50 did not include a valid Content-Length field (despite it being strongly recommended in the HTTP RFC). Elsewhere this kind of check can be addressed by, for example, developing formats that encode a large number of lengths that are frequently observed in legitimate traffic, and for each length ensuring appropriate length of ciphertexts. In theory one can also use FTE for more powerful language classes (i.e., an algorithm for ranking unambiguous CFGs appears in [17]).

More generally, DPI is faced with finding checks for protocol semantics or formatting with fidelity beyond what is captured by the FTE record layer. Since the latter takes a minimalist approach, there are innumerable ways in which FTE communications differ from real target protocol runs. The recent work of Houmansadr et al. [19], for example, shows how to exploit discrepancies in other circumvention systems that do much more than FTE in terms of attempting to mimic a target protocol [30, 45, 46]. But finding such discrepancies is easy, and the hard open question (not addressed in [19]) is how to make such checks effective—fast, scalable, and with negligible errors—in the messy deployment environments faced by DPI and for all of the essentially arbitrary target protocols FTE supports.

The GFC, as discussed above, also engages in active probing. For example, attempting to connect to destination systems suspected of undesirable behavior. Determining how to resist such active attacks in practice is an ongoing research topic (c.f., [39]). Use of FTE, however, can hope to force active probing for all legitimate connections using the target protocol, vastly increasing the complexity of such censorship techniques.

7. RELATED WORK

Steganography/censorship circumvention. 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 [32], HTTP [15], email [45], VoIP [21], or social media [9, 25]. FTE does not embed data in real cover traffic, but instead ensures that ciphertexts are formatted so that “embedded” within them are the telltale protocol fingerprints that DPI systems look for.

Recent works target censorship circumvention via so-called protocol mimicking [20, 30, 31, 45, 46, 52]. Here one builds systems whose traffic attempts to look, on the wire, like that of a specific (unblocked) target protocol. These systems target forcing DPI misclassification, but (to the best of our knowledge) their efficacy against production DPI systems has not been systematically measured. FTE, when used as a circumvention mechanism, can be thought of as new lightweight protocol mimicking tool that targets a certain class of adversary, those being the regex-based (or similarly powerful) DPI used in practice. We note that FTE is orders of magnitude more efficient than prior protocol mimicing systems, yet already forces misclassification against enterprise-grade DPI.

Tunneling traffic over another protocol (e.g., SSH) does not usually lead to protocol misclassification because the protocol used for the tunnel will be correctly identified by the DPI. Tunneling may suffice for DPI circumvention in some settings, but is more expensive and less flexible than using an FTE record layer: tunneling can have high overheads for certain source protocol/tunnel protocol pairs (e.g., HTTP over DNS [27]) and FTE can easily switch between target protocols in a fine-grained manner.

The Dust [48] protocol and Tor’s recent obfsproxy [42] systems strive to ensure that no application-layer bits are left unencrypted. FTE also encrypt all bits sent, but additionally allows one to selectively add back into ciphertexts the formatting needed to force protocol misclassification.

DPI. Performance and scalability of regex-based traffic classification has been extensively studied [4, 18, 44]. Alternative protocol identification strategies that have been explored include using packet sizes and timings [6, 51], the types and number connections initiated by a host (its “social behavior”) [23, 26], and various machine learning techniques [33, 34, 55]. However, the feasibility of deploying more sophisticated classification strategies for DPI at scale remains unclear [11, 34, 40].

Recent work uses fast small-sample hypothesis tests to identify and discard compressed or encrypted packets [47]. Such tests might flag FTE packets from our simpler formats (which produce some packets that only include AE ciphertext bits) as compressed/encrypted, which would prevent protocol misclassification. However, these tests are easy to defeat by simply changing the format so that the hypothesis test fails on the first few packets in a flow.

8. CONCLUSIONS

In this paper, we experimentally showed just how easy it is to force state-of-the-art DPI systems into misidentifying protocols. Our approach was to attack the underlying assumption implicit in the modern DPI systems used in practice, which is that regex-based checks are sufficient for port-independent protocol identification. To do so, we introduce format-transforming encryption (FTE). FTE and its associated record layer provides a generic and flexible mechanism for turning arbitrary application-layer protocol messages into ciphertexts that DPI will classify as messages from a protocol of our choosing. Not only did we successfully trick every one of our corpus of six DPI tools (including an enterprise-class commercial system), but we did so with very little overhead compared to standard tunneling with SSH. Finally, we performed real-world field tests using

Tor with our FTE record layer integrated as a pluggable transport to successfully evade censorship in China. All this demonstrates that FTE has immediate value for circumvention tools. At the same time, our results indicate that network monitoring tools may have to transition from regex-based checks to more sophisticated techniques.

9. REFERENCES

[1] Martin R Albrecht, Kenneth G Paterson, and Gaven J Watson. Plaintext recovery attacks against ssh. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 16–26. IEEE, 2009.

[2] Jacob Appelbaum and Nick Mathewson. Pluggable transports for circumvention. Available at: `https://gitweb.torproject.org/torspec.git/HEAD:/proposals/180-pluggable-transport.txt`, 2010.

[3] Arbor Networks, Inc. appid. Available at: `https://code.google.com/p/appid/`.

[4] M. Becchi, M. Franklin, and P. Crowley. A workload for evaluating deep packet inspection architectures. In *IEEE International Symposium on Workload Characterization, 2008.*, pages 79–89, 2008.

[5] 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*. 2009.

[6] Laurent Bernaille and Renata Teixeira. Early recognition of encrypted applications. In Steve Uhlig, Konstantina Papagiannaki, and Olivier Bonaventure, editors, *Passive and Active Network Measurement*, volume 4427 of *Lecture Notes in Computer Science*. 2007.

[7] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[8] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Theory and techniques for automatic generation of vulnerability-based signatures. *Dependable and Secure Computing, IEEE Transactions on*, 5(4):224–241, 2008.

[9] 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, 2010.

[10] Clear Foundation. l7-filter. Available at: `http://l7-filter.clearfoundation.com/`.

[11] Alberto Dainotti, Antonio Pescapè, and Kimberly C Claffy. Issues and future directions in traffic classification. *Network, IEEE*, 26(1):35–40, 2012.

[12] Luca Deri. nprobe: an open source netflow probe for gigabit networks. In *In Proc. of Terena TNC 2003*, 2003.

[13] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

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

[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*, 2002.

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

[17] A Goldberg and M Sipser. Compression and ranking. In *Proceedings of the 17th Annual ACM symposium on Theory of computing*, STOC '85, 1985.

[18] Danhua Guo, Guangdeng Liao, Laxmi N. Bhuyan, Bin Liu, and Jianxun Jason Ding. A scalable multithreaded l7-filter design for multi-core servers. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, 2008.

[19] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The Parrot is Dead: Observing Unobservable Network Communications. In *The 34th IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[20] Amir Houmansadr, Giang T.K. Nguyen, Matthew Caesar, and Nikita Borisov. Cirripede: circumvention infrastructure using router redirection with plausible deniability. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, 2011.

[21] Amir Houmansadr, Thomas Riedl, Nikita Borisov, and Andrew Singer. I Want my Voice to be Heard: IP over Voice-over-IP for Unobservable Censorship Circumvention. In *The 20th Annual Network and Distributed System Security Symposium (NDSS)*, 2013.

[22] Marios Iliofotou, Hyun-chul Kim, Michalis Faloutsos, Michael Mitzenmacher, Prashanth Pappu, and George Varghese. Graph-based p2p traffic classification at the internet backbone. In *INFOCOM Workshops 2009, IEEE*, pages 1–6. IEEE, 2009.

[23] Marios Iliofotou, Prashanth Pappu, Michalis Faloutsos, Michael Mitzenmacher, Sumeet Singh, and George Varghese. Network monitoring using traffic dispersion graphs (tdgs). In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, 2007.

[24] Christopher M. Inacio and Brian Trammell. Yaf: yet another flowmeter. In *Proceedings of the 24th international conference on Large installation system administration*, LISA'10, 2010.

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

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

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

[28] Eitan Menahem, Gabi Nakibly, and Yuval Elovici. Network-based intrusion detection systems go active! In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 1004–1006. ACM, 2012.

[29] Microsoft. [MS-SMB2]: Server Message Block (SMB) Protocol Versions 2 and 3. Available at: `http://msdn.microsoft.com/en-us/library/cc246482.aspx`, 2013.

[30] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. Skypemorph: protocol obfuscation for tor bridges. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, 2012.

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

[32] Steven J. Murdoch and Stephen Lewis. Embedding covert channels into TCP/IP. In *Proceedings of the 7th International Workshop on Information Hiding*, 2005.

[33] Thuy T. T. Nguyen, Grenville Armitage, Philip Branch, and Sebastian Zander. Timely and continuous machine-learning-based classification for interactive IP traffic. *IEEE/ACM Trans. Netw.*, 20(6), December 2012.

[34] Thuy TT Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. *Communications Surveys & Tutorials, IEEE*, 10(4):56–76, 2008.

[35] Vern Paxson. Bro: a system for detecting network intruders in real-time. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, SSYM'98, 1998.

[36] Christopher Rhoads and Farnaz Fassihi. Iran vows to unplug internet. Available at: `http://online.wsj.com/article/`

SB10001424052748704889404576277391449002016.html, December 2011.

[37] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002.

[38] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326 (Proposed Standard), April 1998.

[39] Rob Smits, Divam Jain, Sarah Pidcock, Ian Goldberg, and Urs Hengartner. Bridgespa: improving tor bridges with single packet authorization. In *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*. ACM, 2011.

[40] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010.

[41] Géza Szabó, Zoltán Turányi, László Toka, Sándor Molnár, and Alysson Santos. Automatic protocol signature generation framework for deep packet inspection. In *Proceedings of the 5th International ICST Conference on Performance Evaluation Methodologies and Tools*, pages 291–299. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011.

[42] Tor Project. Obfsproxy. Available at: https://www.torproject.org/projects/obfsproxy.html.en, 2013.

[43] Tor Project. Obfsproxy3. Available at: https://gitweb.torproject.org/pluggable-transports/obfsproxy.git/blob/HEAD:/doc/obfs3/obfs3-protocol-spec.txt, 2013.

[44] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. The nids cluster: Scalable, stateful network intrusion detection on commodity hardware. In Christopher Kruegel, Richard Lippmann, and Andrew Clark, editors, *Recent Advances in Intrusion Detection*, volume 4637 of *Lecture Notes in Computer Science*. 2007.

[45] Qiyan Wang, Xun Gong, Giang Nguyen, Amir Houmansadr, and Nikita Borisov. CensorSpoofer: Asymmetric Communication using IP Spoofing for Censorship-Resistant Web Browsing. In *The* 19^{th} *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[46] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. Stegotorus: a camouflage proxy for the tor anonymity system. In *ACM Conference on Computer and Communications Security*, 2012.

[47] Andrew M White, Srinivas Krishnan, Michael Bailey, Fabian Monrose, and Phillip Porras. *Clear and Present Data: Opaque Traffic and its Security Implications for the Future*. NDSS, 2013.

[48] Brandon Wiley. Dust: A blocking-resistant internet transport protocol. Technical report, School of Information, University of Texas at Austin, 2011.

[49] Philipp Winter and Stefan Lindskog. How the Great Firewall of China is Blocking Tor. In *Free and Open Communications on the Internet*, 2012.

[50] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, and Scuola Superiore S Anna. Automatic network protocol analysis. In *15th Symposium on Network and Distributed System Security (NDSS)*, 2008.

[51] Charles V. Wright, Fabian Monrose, and Gerald M. Masson. On inferring application protocol behaviors in encrypted network traffic. *Journal on Machine Learning Research*, 7, December 2006.

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

[53] Ai-min Yang, Sheng-yi Jiang, and He Deng. A p2p network traffic classification method using svm. In *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 398–403. IEEE, 2008.

[54] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard), January 2006.

[55] Sebastian Zander, Thuy Nguyen, and Grenville Armitage. Automated traffic classification and application identification using machine learning. In *Local Computer Networks, 2005. 30th Anniversary. The IEEE Conference on*, pages 250–257. IEEE, 2005.

APPENDIX

A. ALGORITHMS FOR RANKING AND UNRANKING A REGULAR LANGUAGE

In Figure 7 we have the core algorithms BuildTable, rank, and unrank, used in our FTE record layer. The ordinality of symbol $\alpha \in \Sigma$, written $\text{ord}(\alpha)$, 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.

```
alg. BuildTable($N$):
for $q \in Q$ do
  if $q \in F$ then $T[q,0] \leftarrow 1$
for $i = 1$ to $N$ do
  for $q \in Q$ do
    for $a \in \Sigma$ do
      $T[q,i] \xleftarrow{+} T[\delta(q,a), i-1]$
$S[0] \leftarrow 0$
for $i = 1$ to $N$ do
  $S[i] \leftarrow S[i-1] + T[q_0, i-1]$
```

```
alg. rank($X$):
$n \leftarrow |X|$; $c \leftarrow S[n]$
$q \leftarrow q_0$
for $i = 1$ to $n$ do
  for $j = 1$ to $\text{ord}(X[i]) - 1$ do
    $c \xleftarrow{+} T[\delta(q, a_j), n-i]$
  $q \leftarrow \delta(q, X[i])$
ret $c$
```

```
alg. unrank($c$):
$(n, c') \leftarrow$ FindSlice(c)
$X \leftarrow \varepsilon$; $q \leftarrow q_0$; $j \leftarrow 1$
for $i = 1$ to $n$ do
  while $c' \geq T[\delta(q, a_j), n-i]$ do
    $c' \xleftarrow{-} T[\delta(q, a_j), n-i]$; $j \xleftarrow{+} 1$
  $X[i] \leftarrow a_j$; $q \leftarrow \delta(q, X[i])$; $j \leftarrow 1$
ret $X$
```

Figure 7: Algorithms for ranking and unranking strings in the regular language L of a DFA $M = (Q, \Sigma, \delta, q_0, F)$.