Marionette: A Programmable Network Traffic Obfuscation System

Kevin P. Dyer, *Portland State University;* Scott E. Coull, *RedJack LLC.;*
Thomas Shrimpton, *Portland State University*

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/dyer

This paper is included in the Proceedings of the
24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-939133-11-3

Marionette: A Programmable Network-Traffic Obfuscation System

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

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

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

Abstract

Recently, a number of obfuscation systems have been developed to aid in censorship circumvention scenarios where encrypted network traffic is filtered. In this paper, we present *Marionette*, the first programmable network traffic obfuscation system capable of simultaneously controlling encrypted traffic features at a variety of levels, including ciphertext formats, stateful protocol semantics, and statistical properties. The behavior of the system is directed by a powerful type of probabilistic automata and specified in a user-friendly domain-specific language, which allows the user to easily adjust their obfuscation strategy to meet the unique needs of their network environment. In fact, the Marionette system is capable of emulating many existing obfuscation systems, and enables developers to explore a breadth of protocols and depth of traffic features that have, so far, been unattainable. We evaluate Marionette through a series of case studies inspired by censor capabilities demonstrated in the real-world and research literature, including passive network monitors, stateful proxies, and active probing. The results of our experiments not only show that Marionette provides outstanding flexibility and control over traffic features, but it is also capable of achieving throughput of up to 6.7Mbps when generating RFC-compliant cover traffic.

1 Introduction

Many countries have begun to view encrypted network services as a threat to the enforcement of information control and security policies. China [41] and Iran [7] are well-known for their efforts to block encrypted services like Tor [14], while other countries, such as the United Kingdom [18], have begun to express interest in blocking VPNs and anonymity systems. These discriminatory routing policies are empowered by analyzing traffic at both the network layer (e.g. TCP/IP headers) and, more recently, the application layer. The latter looks for specific features of packet payloads that act as a signature for the application-layer protocol being transported.

To combat application-layer filtering, several systems have been proposed to obfuscate packet payloads, and generally hide the true protocol being transported. Broadly speaking, these methods fall into one of three categories: those that use encryption to fully randomize the messages sent (e.g., obfs4 [34], ScrambleSuit [42], Dust [40]); those that tunnel traffic using existing software artifacts (e.g., FreeWave [21], Facet [24]); and those that use encryption in combination with some lightweight ciphertext formatting to make the traffic mimic an allowed protocol (e.g., FTE [15], StegoTorus [38]). A few of these systems have been deployed and are currently used by more comprehensive circumvention systems, such as Lantern [1], uProxy [5], and Tor [14].

Despite the progress these obfuscation systems represent, each of them suffers from one or more shortcomings that severely limit their ability to adapt to new network environments or censorship strategies. Lightweight obfuscation methods based on randomization fail in situations where protocol whitelisting is applied, as in the recent Iranian elections [13]. Tunneling systems are intimately tied to a specific protocol that may not always be permitted within the restrictive network environment, such as the case of Skype in Ethiopia [27]. Protocol-mimicry systems really only aim to mimic *individual* protocol messages, and therefore fail to traverse proxies that enforce stateful protocol semantics (e.g., Squid [39]) . Moreover, these systems can be quite brittle in the face of proxies that alter protocol messages in transit (e.g., altering message headers can render FTE [15] inoperable). In any case, all of the systems are incapable of changing their target protocol or traffic features without heavy system re-engineering and redeployment of code. This is a huge undertaking in censored networks.

Case Study	Message Content	Stateful Behavior	Multi-layer Control	Traffic Statistics	Active Probing	Protocol(s)	Goodput (Down/Up)
Regex-Based DPI	✓	-	-	-	-	HTTP, SSH, SMB	68.2 / 68.2 Mbps
Proxy Traversal	✓	✓	-	-	-	HTTP	5.8 / 0.41 Mbps
Protocol Compliance	✓	✓	✓	-	-	FTP, POP3	6.6 / 6.7 Mbps
Traffic Analysis	✓	✓	✓	✓	-	HTTP	0.45 / 0.32 Mbps
Active Probing	✓	✓	✓	-	✓	HTTP, FTP, SSH	6.6 / 6.7 Mbps

Figure 1: Summary of Marionette case studies illustrating breadth of protocols and depth of feature control.

The Marionette System. To address these shortcomings, we develop the Marionette system. Marionette is a network-traffic obfuscation system that empowers users to rapidly explore a rich design space, without the need to deploy new code or re-design the underlying system.

The conceptual foundation of Marionette is a powerful kind of probabilistic automaton, loosely inspired by probabilistic input/output automata [45]. We use these to enforce (probabilistic) sequencing of individual ciphertext message types. Each transition between automata states has an associated block of actions to perform, such as encrypting and formatting a message, sampling from a distribution, or spawning other automata to support hierarchical composition. By composing automata, we achieve even more comprehensive control over mimicked protocol behaviors (e.g., multiple dependent channels) and statistical features of traffic. In addition, the automata admit distinguished error states, thereby providing an explicit mechanism for handling active attacks, such as censor-initiated "probing attacks."

At the level of individual ciphertext formats, we introduce another novel abstraction that supports fine-grained control. These *template grammars* are probabilistic context-free grammars (CFG) that compactly describes a language of templates for ciphertexts. Templates are strings that contain placeholder tokens, marking the positions where information (e.g., encrypted data bytes, dates, content-length values) may be embedded by user-specified routines. Adopting a CFG to describe templates has several benefits, including ease of deployment due to their compact representation, ability to directly translate grammars from available RFCs, and use of the grammar in receiver-side parsing tasks.

Everything is specified in a user-friendly domain-specific language (DSL), which enables rapid development and testing of new obfuscation strategies that are robust and responsive to future network monitoring tools. To encourage adoption and use of Marionette it has been made available as free and open source software[1].

Case studies. To display what is possible with Marionette, we provide several case studies that are inspired by recent research literature and real-world censor capabilities. These are summarized in Figure 1. For one example, we show that Marionette can implement passive-mode FTP by spawning multiple models that control interdependent TCP connections. For another, we use Marionette to mimic HTTP with enforced protocol semantics and resilience to message alteration, thereby successfully traversing HTTP proxies.

Our studies show that Marionette is capable of implementing a range of application-layer protocols, from HTTP to POP3, while also providing great depth in the range of traffic features it controls. Most importantly, it maintains this flexibility without unduly sacrificing performance – achieving up to 6.7Mbps while still maintaining fully RFC-compliant protocol semantics. We also show that the system performance is network-bound, and directly related to the constraints of the Marionette format being used.

Security Considerations. While our case studies are motivated by well-known types of adversaries, we avoid a formal security analysis of our framework for two reasons. First, the security of the system is intimately tied to the automata and template grammars specified by the user, as well as how the chosen protocols and features interact with the adversary. Second, any principled security analysis requires a generally accepted adversarial model. At the moment, the capabilities of adversaries in this space are poorly understood, and there are no formalized security goals to target. With that said, we believe our case studies represent a diverse sample of adversaries known to exist in practice, and hope that the flexibility of our system allows it to adapt to new adversaries faced in deployment. More fully understanding the limits of our system, and the adversaries it may face, is left for future work.

2 Related Work

In this section, we discuss previous work in the area of obfuscation and mimicry of application-layer protocols, as well as their common ancestry with network traffic generation research. The majority of systems aiming to avoid application-layer filtering are *non-programmable*, in the sense that they adopt one strategy at design-time

[1] `https://github.com/kpdyer/marionette/`

System		Blacklist DPI	Whitelist DPI	Statistical-test DPI	Protocol-enforcing Proxy	Multi-layer Control	High Throughput
Randomization	obfs2/3 [34]	✓	-	-	-	-	✓
	ScrambleSuit [42]	✓	-	✓	-	-	✓
	obfs4 [34]	✓	-	✓	-	-	✓
	Dust [40]	✓	-	✓	-	-	✓
Mimicry	SkypeMorph [26]	✓	✓	✓	-	-	-
	StegoTorus [38]	✓	✓	-	-	-	-
Tunneling	Freewave [21]	✓	✓	-	-	-	-
	Facet [24]	✓	✓	✓	-	-	-
	SWEET [47]	✓	✓	-	-	-	-
	JumpBox [25]	✓	✓	-	-	-	✓
	CensorSpoofer [36]	✓	✓	-	-	-	-
	CloudTransport [8]	✓	✓	-	✓	-	✓
Programmable	FTE [15]	✓	✓	-	-	-	✓
	Marionette	✓	✓	✓	✓	✓	✓

Figure 2: A comparison of features across randomization, mimicry, tunneling, and programmable obfuscation systems. A "✓" in the first four columns mean the system is appropriate for the indicated type of monitoring device; in the last two, it means that the system has the listed property. Multi-layer control is the ability to control features beyond single, independent connections. High-throughput systems are defined as any system capable of > 1Mbps throughput. Both FTE and Marionette can trade throughput for control over ciphertext traffic features.

and it cannot be changed without a major overhaul of the system and subsequent re-deployment. The non-programmable systems can be further subdivided into three categories based on their strategy: randomization, mimicry, or tunneling. A *programmable* system, however, allows for a variety of dynamically applied strategies, both randomization and mimicry-based, without the need for changes to the underlying software. Figure 2 presents a comparison of the available systems in each category, and we discuss each of them below. For those interested in a broader survey of circumvention and obfuscation technologies, we suggest recent work by Khattak et al. that discusses the space in greater detail [23].

Network Traffic Generation. Before beginning our discussion of obfuscation systems, it is important to point out the connection that they share with the broader area of network traffic generation. Most traffic generation systems focus on simple replay of captured network sessions [33, 19], replay with limited levels of message content synthesis [12, 31], generation of traffic mixes with specific statistical properties and static content [10, 37], or heavyweight emulation of user behavior with applications in virtualized environments [43]. As we will see, many mimicry and tunneling systems share similar strategies with the the key difference that they must also transport useful information to circumvent filtering.

Randomization. For systems implementing the randomization approach, the primary goal is to remove all static fingerprints in the content and statistical characteristics of the connection, effectively making the traffic look like "nothing." The obfs2 and obfs3 [34] protocols were the first to implement this approach by re-encrypting standard Tor traffic with a stream cipher, thereby removing all indications of the underlying protocol from the content. Recently, improvements on this approach were proposed in the ScrambleSuit system [42] and obfs4 protocol [34], which implement similar content randomization, but also randomize the distribution of packet sizes and inter-arrival times to bypass both DPI and traffic analysis strategies implemented by the censor. The Dust system [40] also offers both content and statistical randomization, but does so on a per-packet, rather than per-connection basis. While these approaches provide fast and efficient obfuscation of the traffic, they only work in environments that block specific types of known-bad traffic (i.e., blacklists). In cases where a whitelist strategy is used to allow known-good protocols, these randomization approaches fail to bypass filtering, as was demonstrated during recent elections in Iran [13].

Mimicry. Another popular approach is to mimic certain characteristics of popular protocols, such as HTTP or Skype, so that blocking traffic with those characteristics would result in significant collateral damage. Mimicry-based systems typically perform shallow mimicry of only a protocol's messages or the statistical properties of a single connection. As an example, StegoTorus [38] embeds data into the headers and payloads of a fixed set of previously collected HTTP messages, using various steganographic techniques. However, this provides no mechanism to control statistical properties, beyond what replaying of the filled-in message templates achieves. SkypeMorph [26], on the other hand, relies on the fact that Skype traffic is encrypted and focuses primarily on replicating the statistical features of packet sizes and timing. Ideally, these mimicked protocols would easily blend into the background traffic of the network, however research has shown that mimicked protocols can be distinguished from real versions of the same protocol using protocol semantics, dependencies among connections, and error conditions [20, 17]. In addition, they incur sometimes significant amounts of overhead due to the constraints of the content or statistical mimicry, which makes them much slower than randomization approaches.

Tunneling. Like mimicry-based systems, tunneling approaches rely on potential collateral damage caused by blocking popular protocols to avoid filtering. However, these systems tunnel their data in the payload of real instances of the target protocols. The Freewave [21] system, for example, uses Skype's voice channel to encode data, while Facet [24] uses the Skype video channel, SWEET [47] uses the body of email messages, and JumpBox [25] uses web browsers and live web servers. CensorSpoofer [36] also tunnels data over existing protocols, but uses a low-capacity email channel for upstream messages and a high-capacity VoIP channel for downstream. CloudTransport [8] uses a slightly different approach by tunneling data over critical (and consequently unblockable) cloud storage services, like Amazon S3, rather than a particular protocol. The tunneling-based systems have the advantage of using real implementations of their target protocols that naturally replicate all protocol semantics and other distinctive behaviors, and so they are much harder to distinguish. Even with this advantage, however, there are still cases where the tunneled data causes tell-tale changes to the protocol's behavior [17] or to the overall traffic mix through skewed bandwidth consumption. In general, tunneling approaches incur even more overhead than shallow mimicry systems since they are limited by the (low) capacity of the tunneling protocols.

Programmable Systems. Finally, programmable obfuscation systems combine the benefits of both randomization and mimicry-based systems by allowing the system to be configured to accommodate either strategy. Currently, the only system to implement programmable obfuscation is Format-Transforming Encryption (FTE) [15], which transforms encrypted data into a format dictated by a regular expression provided by the user. The approach has been demonstrated to have both high throughput and the ability to mimic a broad range of application-layer protocols, including randomized content. Unfortunately, FTE only focuses on altering the content of the application-layer messages, and not statistical properties, protocol semantics, or other potentially distinguishing traffic features.

Comparison with Marionette. Overall, each of these systems suffers from a common set of problems that we address with Marionette. For one, these systems, with the exception of FTE, force the user to choose a single target protocol to mimic without regard to the user's throughput needs, network restrictions, and background traffic mix. Moreover, many of the systems focus on only a fixed set of traffic features to control, usually only content and statical features of a single connection. In those cases where tunneling is used, the overhead and latency incurred often renders the channel virtually unusable for many common use cases, such as video streaming. The primary goal of Marionette, therefore, is not to develop a system that implements a single obfuscation method to defeat all possible censor strategies, but instead to provide the user with the ability to choose the obfuscation method that best fits their use case in terms of breadth of target protocols, depth of controlled traffic features, and overall network throughput.

3 Models and Actions

We aim for a system that enables broad control over several traffic properties, not just those of individual application-layer protocol messages. These properties may require that the system maintain some level of state about the interaction to enforce protocols semantics, or allow for non-deterministic behavior to match distributions of message size and timing. A natural approach to efficiently model this sort of stateful and nondeterministic system is a special type of probabilistic state machine, which we find to be well-suited to our needs and flexible enough to support a wide range of design approaches.

Marionette models. A *Marionette model* (or just model, for short) is a tuple $M = (Q, Q_{\text{nrm}}, Q_{\text{err}}, C, \Delta)$. The *state* set $Q = Q_{\text{nrm}} \cup Q_{\text{err}}$, where Q_{nrm} is the set of *normal states*, Q_{err} is the set of *error states*, and $Q_{\text{nrm}} \cap Q_{\text{err}} = \emptyset$. We assume that Q_{nrm} contains a distinguished start state, and that at least one of $Q_{\text{nrm}}, Q_{\text{err}}$ contains a distinguished finish state. The set C is the set of *actions*, which are (potentially) randomized algorithms. A string $\mathcal{B} = f_1 f_2 \cdots f_n \in C^*$ is called an *action-block*, and it defines a sequence of actions. Finally, Δ is a transition relation $\Delta \subseteq Q \times C^* \times (\mathsf{dist}(Q_{\text{nrm}}) \cup \emptyset) \times \mathbb{P}(Q_{\text{err}})$ where $\mathsf{dist}(X)$ the set of distributions over a set X, and $\mathbb{P}(X)$ is the powerset of X. The roles of Q_{nrm} and Q_{err} will be made clear shortly.

A tuple $(s, \mathcal{B}, (\mu_{\text{nrm}}, S)) \in \Delta$ is interpreted as follows. When M is in state s, the action-block $\mathcal{B}$ may be executed and, upon completion, one samples a state $s'_{\text{nrm}} \in Q_{\text{nrm}}$ (according to distribution $\mu_{\text{nrm}} \in$

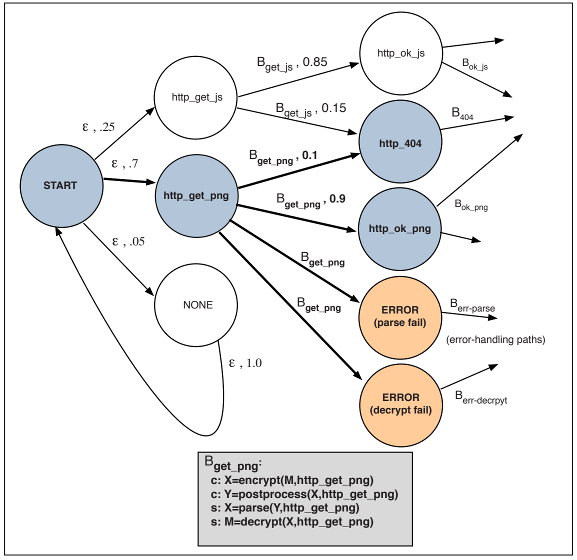
$\mathsf{dist}(Q_{\mathrm{nrm}})$). If the action-block fails, then an error state is chosen non-deterministically from S. Therefore, $\{s'_{\mathrm{nrm}}\} \cup S$ is the set of valid next states, and in this way our models have both proper probabilistic and non-deterministic choice, as in probabilistic input/output automata [45]. When $(s, \mathcal{B}, (\mu_{\mathrm{nrm}}, \emptyset)) \in \Delta$, then only transitions to states in Q_{nrm} are possible, and similarly for $(s, \mathcal{B}, (\emptyset, S))$ with transitions to states in Q_{err}.

In practice, normal states will be states of the model that are reached under normal, correct operation of the system. Error states are reached with the system detects an operational error, which may or may not be caused by an active adversary. For us, it will typically be the case that the results of the action-block $\mathcal{B}$ determine whether or not the system is operating normally or is in error, thus which of the possible next states is correct.

Discussion. Marionette models support a broad variety of uses. One is to capture the intended state of a channel between two communicating parties (i.e., what message the channel should be holding at a given point in time). Such a model serves at least two related purposes. First, it serves to drive the implementation of procedures for either side of the channel. Second, it describes what a passive adversary would see (given implementations that realize the model), and gives the communicating parties some defense against active adversaries. The model tells a receiving party exactly what types of messages may be received next; receiving any other type of message (i.e., observing an invalid next channel state) provides a signal to commence error handling, or defensive measures.

Consider the partial model in Figure 3 for an exchange of ciphertexts that mimic various types of HTTP messages. The states of this model represent effective states of the shared channel (i.e., what message type is to appear next on the channel). Let us refer to the first-sender as the client, and the first-receiver as the server. In the beginning, both client and server are in the start state. The client moves to state http_get_js with probability 0.25, state http_get_png with probability 0.7, and state NONE with probability 0.05. In transitioning to any of these states, the empty action-block is executed (denoted by ε), meaning there are no actions on the transition. Note that, at this point, the server knows only the set {http_get_js, http_get_png, NONE} of valid states and the probabilities with which they are selected.

Say that the client moves to state http_get_png, thus the message that should be placed on the channel is to be of the http_get_png type. The action-block $\mathcal{B}_{\mathsf{get\_png}}$ gives the set of actions to be carried out in order to affect this. We have annotated the actions with "c:" and "s:" to make it clear which meant to be executed by the client and which are meant to be executed by the server, respec-



```
B_get_png:
  c: X=encrypt(M,http_get_png)
  c: Y=postprocess(X,http_get_png)
  s: X=parse(Y,http_get_png)
  s: M=decrypt(X,http_get_png)
```

Figure 3: A partial graphical representation of a Marionette model for an HTTP exchange. (Transitions between http_get_js and error states dropped to avoid clutter.) The text discusses paths marked with bold arrows; normal states on these are blue, error states are orange.

tively. The client is to encrypt a message M using the parameters associated to the handle http_get_png, and then apply any necessary post-processing in order to produce the (ciphertext) message Y for sending. The server, is meant to parse the received Y (e.g. to undo whatever was done by the post-processing), and then to decrypt the result.

If parsing and decrypting succeed at the server, then it knows that the state selected by the client was http_get_png and, hence, that it should enter http_404 with probability 0.1, or http_ok_png with probability 0.9. If parsing fails at the server (i.e. the server action parse(Y,http_get_png) in action block $\mathcal{B}_{\mathsf{get\_png}}$ fails) then the server must enter state ERROR (parse fail). If parsing succeeds but decryption fails (i.e., the server action decrypt(X,http_get_png) in action block $\mathcal{B}_{\mathsf{get\_png}}$ fails) then the server must enter state ERROR (decrypt fail). At this point, it is the client who must keep alive a front of potential next states, namely the four just mentioned (error states are shaded orange in the figure). Whichever state the server chooses, the associated action-block is executed and progress through the model continues until it reaches the specified finish state.

Models provide a useful design abstraction for specifying allowable sequencings of ciphertext messages, as well as the particular actions that the communicating parties should realize in moving from message to message (e.g., encrypt or decrypt according to a particular ciphertext format). In practice, we do not expect sender and receiver instantiations of a given model will be identical. For example, probabilistic or nondeterministic choices made by the sender-side instantiation of a model (i.e., which transition was just followed) will need to be "determinized" by the receiver-side instantiation. This determinization process may need mechanisms to handle ambiguity. In Section 7 we will consider concrete specifications of models.

4 Templates and Template Grammars

In an effort to allow fined-grained control over the format of individual ciphertexts on the wire, we introduce the ideas of ciphertext-format templates, and grammars for creating them. *Templates* are, essentially, partially specified ciphertext strings. The unspecified portions are marked by special placeholders, and each placeholder will ultimately be replaced by an appropriate string, (e.g., a string representing a date, a hexadecimal value representing a color, a URL of a certain depth). To compactly represent a large set of these templates, we will use a probabilistic context-free grammar. Typically, a grammar will create templates sharing a common motif, such as HTTP request messages or CSS files.

Template Grammars. A template grammar $G = (V, \Sigma, R, S, p)$ is a probabilisitic CFG, and we refer to strings $T \in L(G)$ as templates. The set V is the set of non-terminals, and $S \in V$ is the starting non-terminal. The set $\Sigma = \overline{\Sigma} \cup P$ consists of two disjoint sets of symbols: $\overline{\Sigma}$ are the *base terminals*, and P is a set of *placeholder terminals* (or just placeholders). Collectively, we refer to Σ as template terminals. The set of rules R consists of pairs $(v, \beta) \in V \times (V \cup \Sigma)^*$, and we will sometimes adopt the standard notation $v \to \beta$ for these. Finally, the mapping $p\colon R \to (0, 1]$ associates to each rule a probability. We require that the sum of values $p(v, \cdot)$ for a fixed $v \in V$ and any second component is equal to one. For simplicity, we have assumed all probabilities are non-zero. The mapping p supports a method for sampling templates from $L(G)$. Namely, beginning with S, carry out a leftmost derivation and sample among the possible productions for a given rule according to the specified distribution.

Template grammars produce templates, but it is not templates that we place on the wire. Instead, a template T serves to define a set of strings in Σ^*, all of which share the same template-enforced structure. To produce these strings, each placeholder $\gamma \in P$ has associated to it a *handler*. Formally, a handler is a algorithm that takes as inputs a template $T \in \Sigma^*$ and (optionally) a bit string $c \in \{0, 1\}^*$, and outputs a string in Σ^* or the distinguished symbol $\perp$, which denotes error. A handler for γ scans T and, upon reading γ, computes a string in $s \in \Sigma^*$ and replaces γ with s. The handler halts upon reaching the end of T, and returns the new string T' that is T but will all occurrences of γ replaced. If a placeholder γ is to be replaced with a string from a particular set (say a dictionary of fixed strings, or an element of a regular language described by some regular expression), we assume the restrictions are built into the handler.

As an example, consider the following (overly simple) production rules that could be a subset of a context-free grammar for HTTP requests/responses.

$$\begin{aligned}
\langle\text{header}\rangle &\to \langle\text{date\_prop}\rangle\text{: }\langle\text{date\_val}\rangle\texttt{\textbackslash r\textbackslash n} \\
&\quad\ \mid \langle\text{cookie\_prop}\rangle\text{: }\langle\text{cookie\_val}\rangle\texttt{\textbackslash r\textbackslash n} \\
\langle\text{date\_prop}\rangle &\to \texttt{Date} \\
\langle\text{cookie\_prop}\rangle &\to \texttt{Cookie} \\
\langle\text{date\_val}\rangle &\to \gamma_{\text{date}} \\
\langle\text{cookie\_val}\rangle &\to \gamma_{\text{cookie}}
\end{aligned}$$

To handle our placeholders γ_{date} and γ_{cookie}, we might replace the former with the result of FTE["(Jan|Feb|...")], and the latter with the result of running FTE["([a-zA-Z...)"]. In this example our FTE-based handlers are responsible for replacing the placeholder with a ciphertext that is in the language of its input regular expression. To recover the data we parse the string according to the the template grammar rules, processing terminals in the resultant parse tree that correspond to placeholders.

5 System Architecture

In Section 3 we described how a Marionette model can be used to capture stateful and probabilistic communications between two parties. The notion of abstract actions (and action-blocks) gives us a way to use models generatively, too. In this section, we give a high-level description of an architecture that supports this use, so that we may transport arbitrary datastreams via ciphertexts that adhere to our models. We will discuss certain aspects of our design in detail in subsequent sections. Figure **??** provides a diagram of this client-server proxy architecture. In addition to models, this architecture consists of the following components:

- The client-side *driver* runs the main event loop, instantiates models (from a model specification file, see Section 6.3), and destructs them when they have reached the end of their execution. The complimentary receiver-side *broker* is responsible for listening to incoming connections and constructing and destructing models.

- *Plugins* are the mechanism that allow user-specified actions to be invoked in action-blocks. We discuss plugins in greater detail in Section 6.2.

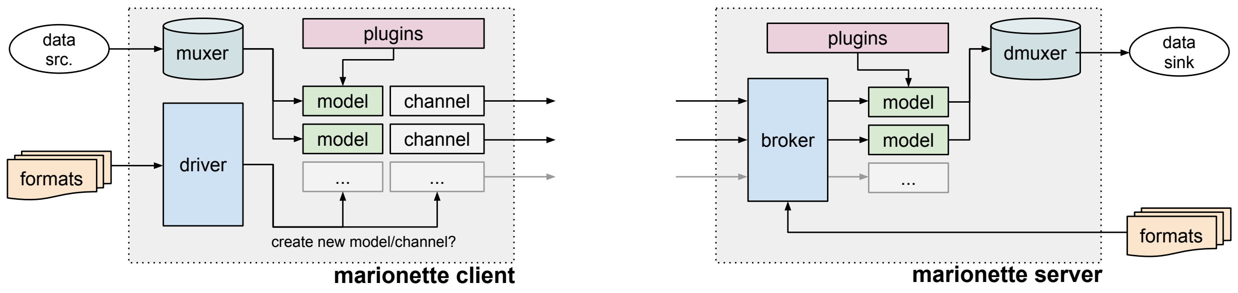


Figure 4: A high-level diagram of the Marionette client-server architecture and its major components for the client-server stream of communications in the Marionette system.

- The client-side *multiplexer* is an interface that allows plugins to serialize incoming datastreams into bitstrings of precise lengths, to be encoded into messages via plugins. The receiver-side *demultiplexer* parses and deserializes streams of cells to recover the underlying datastream. We discuss the implementation details of our (de)multiplexer in Section 6.1.
- A *channel* is a logical construct that connects Marionette models to real-world (e.g., TCP) data connections, and represents the communications between a specific pair of Marionette models. We note that, over the course of a channel's lifetime, it may be associated with multiple real-world connections.

Let's start by discussing how data traverses the components of a Marionette system. A datastream's first point of contact with the system is the incoming multiplexer, where it enters a FIFO buffer. Then a driver invokes a model that, in turn, invokes a plugin that wants to encode n bits of data into a message. Note that if the FIFO buffer is empty, the multiplexer returns a string that contains no payload data and is padded to n bits. The resultant message produced by the plugin is then relayed to the server. Server-side, the broker attempts to dispatch the received message to a model. There are three possible outcomes when the broker dispatches the message: (1) an active model is able to process it, (2) a new model needs to be spawned, or (3) an error has occurred and the message cannot be processed. In case 1 or 2, the cell is forwarded to the demultiplexer, and onward to its ultimate destination. In case 3, the server enters an error state for that message, where it can respond to a non-Marionette connection. We also note that the Marionette system can, in fact, operate with some of its components disabled. As an example, by disabling the multiplexer/demultiplexer we have a traffic generation system that doesn't carry actual data payloads, but generates traffic that abides by our model(s). This shows that there's a clear decoupling of our two main system features: control over cover traffic and relaying datastreams.

6 Implementation

Our implementation of Marionette consists of two command line applications, a client and server, which share a common codebase, and differ only in how they interpret a model. (e.g., initiate connection vs. receive connection) Given a model and its current state, each party determines the set of valid transitions and selects one according to the model's transition probabilities. In cases where normal transitions and error transitions are both valid, the normal transitions are preferred.

Our prototype of Marionette is written in roughly three thousand lines of Python code. All source code and engineering details are available as free and open-source software[2]. In this section, we will provide an overview of some of the major engineering obstacles we overcame to realize Marionette.

6.1 Record Layer

First, we will briefly describe the Marionette record layer and its objectives and design. Our record layer aims to achieve three goals: (1) enable multiplexing and reliability of multiple, simultaneous datastreams, (2) aid Marionette in negotiating and initializing models, and (3) provide privacy and authenticity of payload data. We implement the record layer using variable-length *cells*, as depicted in Figure 5, that are relayed between the client and server. In this section, we will walk through each of our goals and discuss how our record layer achieves them.

Multiplexing of datastreams. Our goal is to enable reliability and in-order delivery of datastreams that we tunnel through the Marionette system. If multiple streams are multiplexed over a single marionette channel, it must be capable of segmenting these streams. We achieve this by including a *datastream ID* and *datastream sequence number* in each cell, as depicted in Figure 5. Sender side, these values are populated at the time of the cell

[2]`https://github.com/kpdyer/marionette`

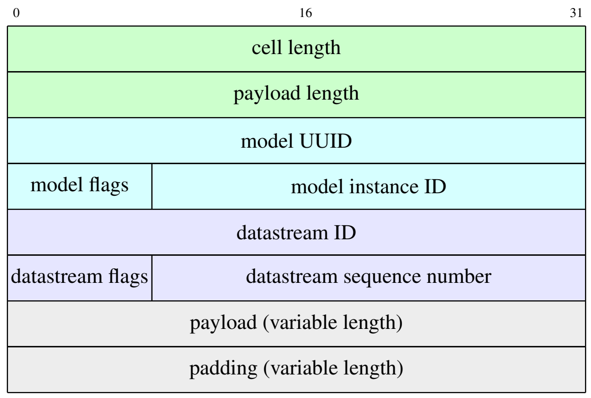


Figure 5: Format of the plaintext Marionette record layer cell.

creation. Receiver side, these values used to reassemble streams and delegate them to the appropriate data sink. The *datastream flags* field may have the value of OPEN, RELAY or CLOSE, to indicate the state of the datastream.

Negotiation and initialization of Marionette models. Upon accepting an incoming message, a Marionette receiver iterates through all transitions from the given model's start state. If one of the action blocks for a transition is successful, the underlying record layer (Figure 5) is recovered and then processed. The *model flags* field, in Figure 5, may have three values: START, RUNNING, or END. A START value is set when this is the first cell transmitted by this model, otherwise the value is set to RELAY until the final transmission of the model where an END is sent. The *model UUID* field is a global identifier that uniquely identifies the model that transmitted the message. The *model instance ID* is used to uniquely identify the instance of the model that relayed the cell from amongst all currently running instances of the model.

For practical purposes, in our proof of concept, we assume that a Marionette instance ID is created by either the client or server, but not both. By convention, the party that sends the first information-carrying message (i.e., first-sender) initiates the instance ID. Once established, the model instance ID has two potential uses. In settings where we have a proxy between the Marionette client and server, the instance ID can be used to determine the model that originated a message despite multiplexing performed by the proxy. In other settings, the instance ID can be used to enhance performance and seed a random number generator for shared randomness between the client and server.

Encryption of the cell. We encrypt each record-layer cell M using a slightly modified encrypt-then-MAC authenticated encryption scheme, namely $C = \mathsf{AES}_{K1}(\mathrm{IV}_1 \| \langle |M| \rangle) \| \mathsf{CTR}[\mathsf{AES}]^{\mathrm{IV}_2}_{K_1}(M) \| T$, where $\mathrm{IV}_1 = 0 \| R$ and $\mathrm{IV}_2 = 1 \| R$ for per-message random R. The first component of the encrypted record is a header. Here we use AES with key $K1$ to encrypt IV_1 along with an encoding of the length of M[3]. The second component is the record body, which is the counter-mode encryption of M under IV_2 and key $K1$, using AES as the underlying blockcipher[4]. Note that CTR can be length-preserving, not sending IV_2 as part of its output, because IV_2 is recoverable from IV_1. The third and component is an authentication tag T resulting from running $\mathsf{HMAC\text{-}SHA256}_{K2}$ over the entire record header and record body. One decrypts in the standard manner for encrypt-then-MAC.

6.2 Plugins

User-specified plugins are used to execute actions described in each model's action blocks. A plugin is called by the Marionette system with four parameters: the current channel, global variables shared across all active models, local variables scoped to our specific model, and the input parameters for this specific plugin (e.g., the FTE regex or the template grammar). It is the job of the plugin to attempt its action given the input parameters. By using global and local dictionaries, plugins can maintain long-term state and even enable message passing between models. We place few restrictions on plugins, however we do require that if a plugin fails (e.g., couldn't receive a message) it must return a failure flag and revert any changes it made when attempting to perform the action. Meanwhile, if it encounters a fatal error (e.g., channel is unexpectedly closed) then it must throw an exception.

To enable multi-level models, we provide a *spawn* plugin that can be used to spawn new model instances. In addition, we provide *puts* and *gets* for the purpose of transmitting static strings. As one example, this can be used to transmit a static, non-information carrying banner to emulate an FTP server. In addition, we implemented FTE and template grammars (Section 4) as our primary message-level plugins. Each plugin has a synchronous (i.e., blocking) and asynchronous (i.e., non-blocking) implementation. The FTE plugin is a wrapper around the FTE[5] and regex2dfa[6] libraries used by the Tor Project for FTE [15].

[3]One could also use the cell-length field in place of $\langle |M| \rangle$.

[4]Since $\mathrm{IV}_1 \neq \mathrm{IV}_2$ we enforce domain separation between the uses of AES_{K1}. Without this we would need an extra key.

[5]`https://github.com/kpdyer/libfte`

[6]`https://github.com/kpdyer/regex2dfa`

6.3 The Marionette DSL

Finally, we present a domain-specific language that can be used to compactly describe Marionette models. We refer to the formats that are created using this language as *Marionette model specifications* or *model specifications* for short. Figure 6 shows the Marionette modeling language syntax.

We have two primary, logical blocks in the model specification. The `connection` block is responsible for establishing model states, actions blocks that are executed upon a transition, and transition probabilities. An `error` transition may be specified for each state and is taken if all other potential transitions encounter a fatal error. The `action` block is responsible for defining a set of actions, which is a line for each party (client or server) and the plugin the party should execute. Let's illustrate the Marionette language by considering the following example.

Example: Simple HTTP model specification. Recall the model in Figure 3, which (partially) captures an HTTP connection where the first client-server message is an HTTP get for a JS or PNG file. Translating the diagram into our Marionette language is a straightforward process. First, we establish our connection block and specify `tcp` and port `80` — the server listens on this port and the client connects to it. For each transition we create an entry in our connection block. As an example, we added a transition between the `http_get_png` and `http_404` state with probability 0.1. For this transition we execute the `get_png` action block. We repeat this process for all transitions in the model ensuring that we have the appropriate action block for each transition.

For each action block we use synchronous FTE. One party is sending, one is receiving, and neither party can advance to the next state until the action successfully completes. Marionette transparently handles the opening and closing of the underlying TCP connection.

```
connection([connection_type]):
    start [dst] [block_name] [prob | error]
    [src] [dst] [block_name] [prob | error]
    ...
    [src]  end  [block_name] [prob | error]

action [block_name]:
    [client | server] plugin(arg1, arg2, ...)
    ...
```

```
connection(tcp, 80):
    start         http_get_js   NULL    0.25
    start         http_get_png  NULL    0.7
    http_get_png  http_404      get_png 0.1
    http_get_png  http_ok_png   get_png 0.9
    http_ok_png   ...

action get_png:
    client fte.send("GET /\w+ HTTP/1\.1...")

action ok_png:
    server fte.send("HTTP/1\.1 200 OK...")

...
```

Figure 6: **Top:** The Marionette DSL. The connection block is responsible for establishing the Marionette model, its states and transitions probabilities. Optionally, the `connection_type` parameter specifies the type of channel that will be used for the model. **Bottom:** The partial model specification that implements the model from Figure 3.

7 Case Studies

We evaluate the Marionette implementation described in Section 6 by building model specifications for a breadth of scenarios: protocol misidentification against regex-based DPI, protocol compliance for complex stateful protocols, traversal of proxy systems that actively manipulate Marionette messages, controlling statistical features of traffic, and responding to network scanners. We then conclude this section with a performance analysis of the formats considered.

For each case study, we analyze the performance of Marionette for the given model specification using our testbed. In our testbed, we deployed our Marionette client and server on Amazon Web Services m3.2xlarge instances, in the us-west (Oregon) and us-east (N. Virginia) zones, respectively. These instances include 8 virtual CPUs based on the Xeon E5-2670 v2 (Ivy Bridge) processor at 2.5GHz and 30GB of memory. The average round-trip latency between the client and server was 75ms. Downstream and upstream *goodput* was measured by transmitting a 1MB file, and averaged across 100 trials. Due to space constraints we omit the full model specifications used in our experiments, but note that each of these specifications is available with the Marionette source code[7].

7.1 Regex-Based DPI

As our first case study, we confirm that Marionette is able to generate traffic that is misclassified by regex-based DPI as a target protocol of our choosing. We are reproducing the tests from [15], using the regular expressions referred to as *manual-http*, *manual-ssh* and *manual-smb* in order to provide a baseline for the performance of the Marionette system under the simplest of specifications. Using these regular expressions, we engineered a Mari-

[7] `https://github.com/kpdyer/marionette`

	Misclassification	
Target Protocol	**bro [28]**	**YAF [22]**
HTTP (manual-http from [15])	100%	100%
SSH (manual-ssh from [15])	100%	100%
SMB (manual-smb from [15])	100%	100%

Figure 7: Summary of misclassification using existing FTE formats for HTTP, SSH, and SMB.

onette model that invokes the non-blocking implementation of our FTE plugins.

For each configuration we generated 100 datastreams in our testbed and classified this traffic using bro [28] (version 2.3.2) and YAF [22] (version 2.7.1.) We considered it a success if the classifier reported the *manual-http* datastreams as HTTP, the *manual-ssh* datastreams as SSH, and so on. In all six cases (two classifiers, three protocols) we achieved 100% success. These results are summarized in Figure 7. All three formats exhibited similar performance characteristics, which is consistent with the results from [15]. On average, we achieved 68.2Mbps goodput for both the upstream and downstream directions, which actually exceeds the goodput reported in [15].

7.2 Protocol-Compliance

As our next test, we aim to achieve protocol compliance for scenarios that require a greater degree of inter-message and inter-connection state. In our testing we created model specifications for HTTP, POP3, and FTP that generate protocol-compliant (i.e., correctly classified by bro) network traffic. The FTP format was the most challenging of the three, so we will use it as our illustrative example.

An FTP session in passive mode uses two data connections: a control channel and a data channel. To enter passive mode a client issues the `PASV` command, and the server responds with an address in the form `(a,b,c,d,x,y)`. As defined by the FTP protocol [30], the client then connects to TCP port `a.b.c.d:(256*x+y)` to retrieve the file requested in the `GET` command.

Building our FTP model specification. In building our FTP model we encounter three unique challenges, compared to other protocols, such as HTTP:

1. FTP has a range of message types, including usernames, passwords, and arbitrary files, that could be used to encode data. In order to maximize potential encoding capacity, we must utilize multiple encoding strategies (e.g., FTE, template grammars, etc.)
2. The FTP protocol is stateful (i.e., message order matters) and has many message types (e.g., `USER`, `PASV`, etc,) which do not have the capacity to encode information.
3. Performing either an active or passive FTP file transfer requires establishing a new connection and maintaining appropriate inter-connection state.

To address the first challenge, we utilize Marionette's plugin architecture, including FTE, template grammars, multi-layer models, and the ability to send/receive static strings. To resolve the second, we rely on Marionette's ability to model stateful transitions and block until, say, a specific static string (e.g., the FTP server banner) has been sent/received. For the third, we rely not only on Marionette's ability to spawn a new model, but we also rely on inter-model communications. In fact, we can generate the listening port server-side on the the fly and communicate it in-band to the client via the `227 Entering Passive Mode (a,b,c,d,x,y)` command, which is processed by a client-side template-grammar handler to populate a client-side global variable. This global variable value is then used to inform the spawned model as to which server-side TCP port it should connect.

Our FTP model specification relies upon the upstream password field, and upstream (PUT) and downstream (GET) file transfers to relay data. In our testbed the FTP model achieved 6.6Mbps downstream and 6.7Mbps upstream goodput.

7.3 Proxy Traversal

As our next case study, we evaluate Marionette in a setting where a protocol-enforcing proxy is positioned between the client and server. Given the prevalence of the HTTP protocol and breadth of proxy systems available, we focus our attention on engineering Marionette model specifications that are able to traverse HTTP proxies.

When considering the presence of an HTTP proxy, there are at least five ways it could interfere with our communications. A proxy could: (1) add HTTP headers, (2) remove HTTP headers, (3) modify header or payload contents, (4) re-order/multiplex messages, or (5) drop messages. Marionette is able to handle each of these cases with only slight enhancements to the plugins we have already described.

We first considered using FTE to generate ciphertexts that are valid HTTP messages. However, FTE is sensitive to modifications to its ciphertexts. As an example, changing the case of a single character of an FTE ciphertext would result in FTE decryption failure. Hence, we need a more robust solution.

Fortunately, template grammars (Section 4) give us fine-grained control over ciphertexts and allows us to tolerate ciphertext modification, and our record layer (Section 6.1) provides mechanisms to deal with stream multiplexing, message re-ordering and data loss. This covers all five types of interference mentioned above.

Building our HTTP template grammar. As a proof of concept we developed four HTTP template grammars. Two languages that are HTTP-GET requests, one with a header field of `Connection: keep-alive` and one with `Connection: close`. We then created analogous HTTP-OK languages that have keep-alive and close headers. Our model oscillates between the keep-alive GET and OK states with probability 0.9, until it transitions from the keep-alive OK state to the GET close state, with probability 0.1

In all upstream messages we encode data into the URL and cookie fields using the FTE template grammar handler. Downstream we encode data in the payload body using the FTE handler and follow this with a separate handler to correctly populate the content-length field.

We provide receiver-side HTTP parsers that validate incoming HTTP messages (e.g., ensure content length is correct) and then extract the URL, cookie and payload fields. Then, we take each of these components and reassemble them into a complete message, independent of the order they appeared. That is, the order of the incoming headers does not matter.

Coping with multiplexing and re-ordering. The template grammar plugin resolves the majority of issues that we could encounter. However, it does not allow us to cope with cases where the proxy might re-order or multiplex messages. By multiplex, we mean that a proxy may interleave two or more Marionette TCP channels into a single TCP stream between the proxy and server. In such a case, we can no longer assume that two messages from the same incoming datastream are, in fact, two sequential messages from the same client model. Therefore, in the non-proxy setting there is a one-to-one mapping between channels and server-side Marionette model instances. In the proxied setting, the channel to model instance mapping may be one-to-many.

We are able to cope with this scenario by relying upon the non-determinism of our Marionette models, and our record layer. The server-side broker attempts to execute all action blocks for available transitions across all active models. If no active model was able to successfully process the incoming message, then the broker (Section 5) attempts to instantiate a new model for that message. In our plugins we must rely upon our record layer to determine success for each of these operations. This allows us to deal with cases where a message may successfully decode and decrypt, but the model instance ID field doesn't match the current model.

Testing with Squid HTTP proxy. We validated our HTTP model specification and broker/plugin enhancements against the Squid [39] caching proxy (version 3.4.9). The Squid caching proxy adds headers, removes header, alters headers and payload contents, and re-orders/multiplexes datastreams. We generated 10,000 streams through the Squid proxy and did not encounter any unexpected issues, such as message loss.

In our testbed, our HTTP model specification for use with Squid proxy achieved 5.8Mbps downstream and 0.41Mbps upstream goodput, with the upstream bandwidth limited by the capacity of the HTTP request format.

7.4 Traffic Analysis Resistance

In our next case study, we control statistical features of HTTP traffic. As our baseline, we visited `Amazon.com` with Firefox 35 ten times and captured all resultant network traffic[8]. We then post-processed the packet captures and recorded the following values: the lengths of HTTP response payloads, the number of HTTP request-response pairs per TCP connection, and the number of TCP connections generated as a result of each page visit. Our goal in this section is to utilize Marionette to model the traffic characteristics of these observed traffic patterns to make network sessions that "look like" a visit to `Amazon.com`. We will discuss each traffic characteristic individually, then combine them in a single model to mimic all characteristics simultaneously.

Message lengths. To model message lengths, we started with the HTTP response template grammar described in Section 7.3. We adapted the response body handler such that it takes an additional, integer value as input. This integer dictates the output length of the HTTP response body. On input n, the handler must return an HTTP response payload of exactly length n bytes.

From our packet captures of `Amazon.com` we recorded the message length for each observed HTTP response payload. Each time our new HTTP response template grammar was invoked by Marionette, we sampled from our recorded distribution of message lengths and used this value as input to the HTTP response template grammar. With this, we generate HTTP response payloads with lengths that match the distribution of those observed during our downloads of `Amazon.com`.

[8]Retrieval performed on February 21, 2015.

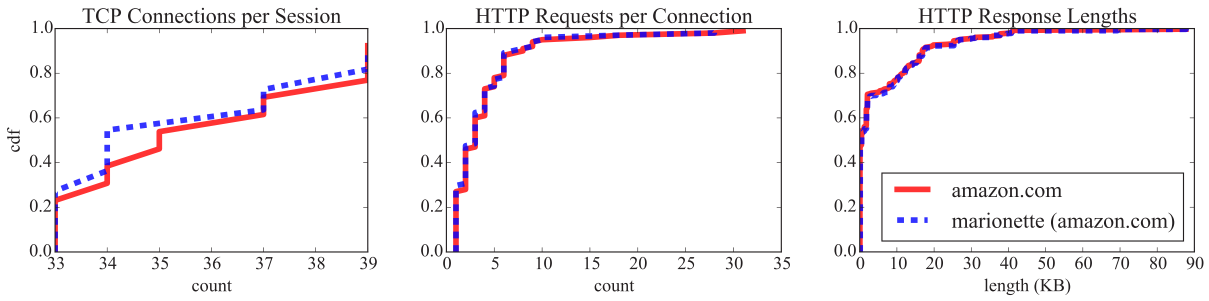


Figure 8: A comparison of the aggregate traffic features for ten downloads of amazon.com using Firefox 35, compared to the traffic generated by ten executions of the Marionette model mimicking amazon.com.

Messages per TCP connection. We model the number of HTTP request-response pairs per TCP connection using the following strategy, which employs hierarchical modeling. Let's start with the case where we want to model a single TCP connection that has n HTTP request-response pairs. We start by creating a set of models which contain exactly n request-response pair with probability 1, for all n values of interest. We can achieve this by creating a model M_n with $n+1$ states, n transitions, and exactly one path. From the start state each transition results in an action block that performs one HTTP request-response. Therefore, M_n models a TCP connection with exactly n HTTP request-response pairs.

Then, we can employ Marionette's hierarchical model structure to have fine-grained control over the number of HTTP request-response pairs per connection. Let's say that we want to have n_1 request-response pairs with probability p_1, n_2 with probability p_2, and so on. For simplicity, we assume that all values n_i are unique, all values p_i are greater than 0, and $\Sigma_{i=0}^{m} p_i = 1$. For each possible value of n_i we create a model M_{n_i}, as described above. Then, we create a single parent model which has a start state with a transition that spawns M_{n_1} with probability p_1, M_{n_2} with probability p_2, and so on. This enables us to create a single, hierarchical model that that controls the number of request-response pairs for arbitrary distributions.

Simultaneously active connections. Finally, we aim to control the total number of connections generated by a model during an HTTP session. That is, we want our model to spawn n_i connections with probability p_i, according to some distribution dictated by our target. We achieve this by using the same hierarchical approach as the request-response pairs model, with the distinction that each child model now spawns n_i connections.

Building the model and its performance. For each statistical traffic feature, we analyzed the distribution of values in the packet captures from our `Amazon.com` visits. We then used the strategies in this section to construct a three-level hierarchical model that controls all of the traffic features simultaneously: message lengths, number of request-response pairs per connection, and the number of simultaneously active TCP connections. With this new model we deployed Marionette in our testbed and captured all network traffic it generated. In Figure 8 we have a comparison of the traffic features of the `Amazon.com` traffic, compared to the traffic generated by our Marionette model.

In our testbed, this model achieved 0.45Mbps downstream and 0.32Mbps upstream goodput. Compared to Section 7.3 this decrease in performance can be explained, in part, by the fact that `Amazon.com` has many connections with only a single HTTP request-response, and very short messages. As one example, the most common payload length in the distribution was 43 bytes. Consequently, the majority of the processing time was spent waiting for setup and teardown of TCP connections.

7.5 Resisting Application Fingerprinting

In our final case study, we evaluate Marionette's ability to resist adversaries that wish to identify Marionette servers using active probing or fingerprinting methods. We assume that an adversary is employing off-the-shelf tools to scan a target host and determine which services it is running. An adversary may have an initial goal to identify that a server is running Marionette and not an industry-standard service (e.g., Apache, etc.). Then, they may use this information to perform a secondary inspection or immediately block the server. This problem has been shown to be of great practical importance for services such as Tor [41] that wish to remain undetected in the presence of such active adversaries.

Our goal is to show that Marionette can coerce fingerprinting tools to incorrectly classify a Marionette server

```
connection(tcp, 8080):
  start    upstream       http_get    1.0
  upstream downstream     http_ok     1.0
  upstream downstream_err http_ok_err error
  ...

action http_ok_err:
  server io.puts("HTTP/1.1 200 OK\r\n" \
             + "Server: Apache/2.4.7\r\n..."
             ...
```

Figure 9: Example HTTP model specification including active probing resistance.

as a service of our choosing. As one example, we'll show that with slight embellishments to the formats we describe in Section 7.1 and Section 7.2, we can convince nmap [4] that Marionette is an instance of an Apache server.

7.5.1 Building Fingerprinting-Resistant Formats

In our exploration of fingerprinting attacks we consider three protocols: HTTP [16], SSH [46], and FTP [30]. For HTTP and SSH we started with the formats described in Section 7.1, and for FTP we started the format described in Section 7.2. We augmented these formats by adding an error transition (Section 3) that invokes an action that mimics the behavior of our target service. This error transition is traversed if all other potential transitions encounter fatal errors in their action blocks, which occur if an invalid message is received.

As an example, for our HTTP format we introduce an error transition to the `downstream_err` state. This transition is taken if the `http_ok` action block encounters a fatal error when attempting to invoke an FTE decryption. In this specific format, a fatal error in the `http_ok` action block is identified if an invalid message is detected when attempting to perform FTE decryption (i.e., doesn't match the regex or encounters a MAC failure). In the example found in Figure 9, upon encountering an error, we output the default response produced when requesting the index file from an Apache 2.4.7 server.

7.5.2 Fingerprinting Tools

For our evaluation we used nmap [4], Nessus [3], and metasploit [2], which are three commonly used tools for network reconnaissance and application fingerprinting. Our configuration was as follows.

nmap: We used nmap version 6.4.7 with version detection enabled and all fingerprinting probes enabled. We invoked nmap via the command line to scan our host. Nmap's service and version fields were used to identify its fingerprint of the target.

Protocol	**Fingerprint Target**	**Scanner** nmap	Nessus	metasploit
HTTP	Apache 2.4.7	✓	✓	✓
FTP	Pure-FTPd 1.0.39	✓	✓	✓
SSH	OpenSSH 6.6.1	✓	✓	✓

Figure 10: A ✓ indicates that Marionette was able to successful coerce the fingerprinting tool into reporting that the Marionette server is the fingerprint target.

Nessus: For Nessus we used version 6.3.6 and performed a Basic Network Scan. We invoked Nessus via its REST API to start the scan and then asynchronously retrieved the scan with a second request. The reported fingerprint was determined by the `protocol` and `svc_name` for all plugins that were triggered.

metasploit: We used version 4.11.2 of metasploit. For fingerprinting SSH, FTP, and HTTP we used the `ssh_version` , `ftp_version` and `http_version` modules, respectively. For each module we set the RHOST and RPORT variable to our host and the reported fingerprint was the complete text string returned by the module.

7.5.3 Results

We refer to the *target* or *fingerprint target* as the application that we are attempting to mimic. To establish our fingerprint targets we installed Apache 2.4.7, Pure-FTPd 1.0.39 and OpenSSH 6.6.1 on a virtual machine. We then scanned each of these target applications with each of our three fingerprinting tools and stored the fingerprints.

To create our Marionette formats that mimic these targets, we added error states that respond identically to our target services. As an example, for our Apache 2.4.7, we respond with a success status code (200) if the client requests the `index.html` or `robots.txt` file. Otherwise we respond with a File Not Found (404) error code. Each server response includes a `Server: Apache 2.4.7` header. For our FTP and SSH formats we used a similar strategy. We observed the request initiated by each probe, and ensured that our error transitions triggered actions that are identical to our fingerprinting target.

We then invoked Marionette with our three new formats and scanned each of the listening instances with our fingerprinting tools. In order to claim success, we require two conditions. First, the three fingerprinting tools in our evaluation must report the exact same fingerprint as the target. Second, we require that a Marionette client must be able to connect to the server and relay data, as described in prior sections. We achieved this for all

Section	Protocol	Percent of Time Blocking on Network I/O	
		Client	Server
7.1	HTTP, SSH, etc.	56.9%	50.1%
7.2	FTP, POP3	90.1%	80.5%
7.3	HTTP	84.0%	96.8%
7.4	HTTP	65.5%	98.8%

Figure 11: Summary of case study formats and time spent blocking on network I/O for both client and server.

nine configurations (three protocols, three fingerprinting tools) and we summarize our results in Figure 10.

7.6 Performance

In our experiments, the performance of Marionette was dominated by two variables: (1) the structure of the model specification and (2) the client-server latency in our testbed. To illustrate the issue, consider our FTP format in Section 7.2 where we require nine back-and-forth messages in the FTP command channel before we can invoke a PASV FTP connection. This format requires a total of thirteen round trips (nine for our messages and four to establish the two TCP connections) before we can send our first downstream ciphertext. In our testbed, with a 75ms client-server latency, this means that (at least) 975ms elapse before we send any data. Therefore, a disproportionately large amount of time is spent blocking on network I/O.

In Figure 11 we give the percentage of time that our client and server were blocked due to network I/O, for each of the Marionette formats in our case studies. In the most extreme case, the Marionette server for the HTTP specification in Section 7.4 sits idle 98.8% of the time, waiting for network events. These results suggest that that certain Marionette formats (e.g., HTTP in Section 7.4) that target high-fidelity mimicry of protocol behaviors, network effects can dominate overall system performance. Appropriately balancing efficiency and realism is an important design consideration for Marionette formats.

8 Conclusion

The Marionette system is the first programmable obfuscation system to offer users the ability to control traffic features ranging from the format of individual application-layer messages to statistical features of connections to dependencies among multiple connections. In doing so, the user can choose the strategy that best suits their network environment and usage requirements. More importantly, Marionette achieves this flexibility without sacrificing performance beyond what is required to maintain the constraints of the model. This provides the user with an acceptable trade-off between depth of control over traffic features and network throughput. Our evaluation highlights the power of Marionette through a variety of case studies motivated by censorship techniques found in practice and the research literature. Here, we conclude by putting those experimental results into context by explicitly comparing them to the state of the art in application identification techniques, as well as highlighting the open questions that remain about the limitations of the Marionette system.

DPI. The most widely used method for application identification available to censors is DPI, which can search for content matching specified keywords or regular expressions. DPI technology is now available in a variety of networking products with support for traffic volumes reaching 30Gbps [11], and has been demonstrated in real-world censorship events by China [41] and Iran [7]. The Marionette system uses a novel template grammar system, along with a flexible plugin system, to control the format of the messages produced and how data is embedded into those messages. As such, the system can be programmed to produce messages that meet the requirements for a range of DPI signatures, as demonstrated in Sections 7.1 and 7.2.

Proxies and Application Firewalls. Many large enterprise networks implement more advanced proxy and application-layer firewall devices that are capable of deeper analysis of particular protocols, such as FTP, HTTP, and SMTP [39]. These devices can cache data to improve performance, apply protocol-specific content controls, and examine entire protocol sessions for indications of attacks targeted at the application. In many cases, the proxies and firewalls will rewrite headers to ensure compliance with protocol semantics, multiplex connections for improved efficiency, change protocol versions, and even alter content (e.g., HTTP chunking). Although these devices are not known to be used by nation-states, they are certainly capable of large traffic volumes (e.g., 400TB/day [6]) and could be used to block most current obfuscation and mimicry systems due to the changes they make to communications sessions. Marionette avoids these problems by using template grammars and a resilient record layer to combine several independent data-carrying fields into a message that is robust to reordering, changes to protocol headers, and connection multiplexing. The protocol compliance and proxy traversal capabilities of Marionette were demonstrated in Sections 7.2 and 7.3, respectively.

Advanced Techniques. Recent papers by Houmansadr et al. [20] and Geddes et al. [17] have presented a number of passive and active tests that a censor could use to identify mimicry systems. The passive tests include examination of dependent communication channels that are not present in many mimicry systems, such as a TCP control channel in the Skype protocol. Active tests include dropping packets or preemptively closing connections to elicit an expected action that the mimicked systems do not perform. Additionally, the networking community have been developing methods to tackle the problem of traffic identification for well over a decade [9], and specific methods have even been developed to target encrypted network traffic [44].

To this point, there has been no evidence that these more advanced methods have been applied in practice. This is likely due to two very difficult challenges. First, many of the traffic analysis techniques proposed in the literature require non-trivial amounts of state to be kept on every connection (e.g., packet size bi-gram distributions), as well as the use of machine learning algorithms that do not scale to the multi-gigabit traffic volumes of enterprise and backbone networks. As a point of comparison, the Bro IDS system [28], which uses DPI technology, has been known to have difficulties scaling to enterprise-level networks [35]. The second issue stems from the challenge of identifying rare events in large volumes of traffic, commonly referred to as the base-rate fallacy. That is, even a tiny false positive rate can generate an overwhelming amount of collateral damage when we consider traffic volumes in the 1 Gbps range. Sommer and Paxson [32] present an analysis of the issue in the context of network intrusion detection and Perry [29] for the case of website fingerprinting attacks.

Regardless of the current state of practice, there may be some cases where technological developments or a carefully controlled network environment enables the censor to apply these techniques. As we have shown in Section 7.4, however, the Marionette system is capable of controlling multiple statistical features not just within a single connection, but also across many simultaneous connections. We also demonstrate how our system can be programmed to spawn interdependent models across multiple connections in Section 7.2. Finally, in Section 7.5, we explored the use of error transitions in our models to respond to active probing and fingerprinting.

Future Work. While the case studies described in the previous section cover a range of potential adversaries, we note that there are still many open questions and potential limitations that have yet to be explored. For one, we do not have a complete understanding of the capabilities of the probabilistic I/O automata to model long-term state. These automata naturally exhibit the Markov property, but can also be spawned in a hierarchical manner with shared global and local variables, essentially providing much deeper conditional dependencies. Another area of exploration lies in the ability of template grammars to produce message content outside of simple message headers, potentially extending to context-sensitive languages found in practice. Similarly, there are many questions surrounding the development of the model specifications themselves since, as we saw in Section 7.6, these not only impact the unobservability of the traffic but also its efficiency and throughput.

References

[1] Lantern. `https://getlantern.org/`.

[2] metasploit. `http://www.metasploit.com/`.

[3] Nessus. `http://www.tenable.com/`.

[4] Nmap. `https://nmap.org/`.

[5] uproxy. `https://uproxy.org/`.

[6] Apache traffic server. `http://trafficserver.apache.org/`.

[7] Simurgh Aryan, Homa Aryan, and J. Alex Halderman. Internet censorship in iran: A first look. In *Presented as part of the 3rd USENIX Workshop on Free and Open Communications on the Internet*, Berkeley, CA, 2013. USENIX.

[8] Chad Brubaker, Amir Houmansadr, and Vitaly Shmatikov. Cloudtransport: Using cloud storage for censorship-resistant networking. In *Proceedings of the 14th Privacy Enhancing Technologies Symposium (PETS 2014)*, July 2014.

[9] A. Callado, C. Kamienski, G. Szabo, B. Gero, J. Kelner, S. Fernandes, and D. Sadok. A survey on internet traffic identification. *Communications Surveys Tutorials, IEEE*, 11(3):37–52, rd 2009.

[10] Jin Cao, William S. Cleveland, Yuan Gao, Kevin Jeffay, F. Donelson Smith, and Michele Weigle. Stochastic models for generating synthetic http source traffic. In *IN PROCEEDINGS OF IEEE INFOCOM*, 2004.

[11] Cisco sce 8000 service control engine. `http://www.cisco.com/c/en/us/products/collateral/service-exchange/sce-8000-series-service-control-engine/data_sheet_c78-492987.html`, June 2015.

[12] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy H. Katz. Protocol-independent adaptive replay of application dialog. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS)*, February 2006.

[13] Holly Dagres. Iran induces internet 'coma' ahead of elections. `http://www.al-monitor.com/pulse/originals/2013/05/iran-internet-censorship-vpn.html`, May 2013.

[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] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, November 2013.

[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 1999.

[17] John Geddes, Max Schuchard, and Nicholas Hopper. Cover your acks: Pitfalls of covert channel censorship circumvention. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, pages 361–372. ACM, 2013.

[18] Andrew Griffin. Whatsapp and imessage could be banned under new surveillance plans. The Independent, January 2015.

[19] Seung-Sun Hong and S. Felix Wu. On interactive internet traffic replay. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID'05, pages 247–264, Berlin, Heidelberg, 2006. Springer-Verlag.

[20] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The Parrot is Dead: Observing Unobservable Network Communications. In *The* 34^{th} *IEEE Symposium on Security and Privacy*, 2013.

[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 *Proceedings of the Network and Distributed System Security Symposium - NDSS'13*. Internet Society, February 2013.

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

[23] Sheharbano Khattak, Mobin Javed, Philip D. Anderson, and Vern Paxson. Towards illuminating a censorship monitor's model to facilitate evasion. In *Presented as part of the 3rd USENIX Workshop on Free and Open Communications on the Internet*, Berkeley, CA, 2013. USENIX.

[24] Shuai Li, Mike Schliep, and Nick Hopper. Facet: Streaming over videoconferencing for censorship circumvention. In *Proceedings of the 12th Workshop on Privacy in the Electronic Society (WPES)*, November 2014.

[25] Jeroen Massar, Ian Mason, Linda Briesemeister, and Vinod Yegneswaran. Jumpbox–a seamless browser proxy for tor pluggable transports. *Security and Privacy in Communication Networks. Springer*, page 116, 2014.

[26] 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*, 2012.

[27] Katia Moskvitch. Ethiopia clamps down on skype and other internet use on tor. BBC News, June 2012.

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

[29] Mike Perry. A critique of website traffic fingerprinting attacks. `https://blog.torproject.org/`, November 2013.

[30] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Standard), October 1985. Updated by RFCs 2228, 2640, 2773, 3659.

[31] Sam Small, Joshua Mason, Fabian Monrose, Niels Provos, and Adam Stubblefield. To catch a predator: A natural language approach for eliciting malicious payloads. In *Proceedings of the 17th Conference on Security Symposium*, 2008.

[32] 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.

[33] Tcpreplay. `http://tcpreplay.synfin.net/`.

[34] Tor Project. Obfsproxy. `https://www.torproject.org/projects/obfsproxy.html.en`, 2015.

[35] 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 *Recent Advances in Intrusion Detection*, pages 107–126. Springer, 2007.

[36] 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*, 2012.

[37] Michele C. Weigle, Prashanth Adurthi, Félix Hernández-Campos, Kevin Jeffay, and F. Donelson Smith. Tmix: A tool for generating realistic tcp application workloads in ns-2. *SIGCOMM Comput. Commun. Rev.*, 36(3):65–76, July 2006.

[38] 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.

[39] D. Wessels and k. claffy. ICP and the Squid web cache. *IEEE Journal on Selected Areas in Communications*, 16(3):345–57, Mar 1998.

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

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

[42] Philipp Winter, Tobias Pulls, and Juergen Fuss. Scramblesuit: a polymorphic network protocol to circumvent censorship. In *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, pages 213–224. ACM, 2013.

[43] Charles V. Wright, Christopher Connelly, Timothy Braje, Jesse C. Rabek, Lee M. Rossey, and Robert K. Cunningham. Generating client workloads and high-fidelity network traffic for controllable, repeatable experiments in computer security. In Somesh Jha, Robin Sommer, and Christian Kreibich, editors, *Recent Advances in Intrusion Detection*, volume 6307 of *Lecture Notes in Computer Science*, pages 218–237. Springer Berlin Heidelberg, 2010.

[44] 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.

[45] Sue-Hwey Wu, Scott A Smolka, and Eugene W Stark. Composition and behaviors of probabilistic i/o automata. *Theoretical Computer Science*, 176(1):1–38, 1997.

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

[47] Wenxuan Zhou, Amir Houmansadr, Matthew Caesar, and Nikita Borisov. Sweet: Serving the web by exploiting email tunnels. *HotPETS. Springer*, 2013.