Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

The Collateral Damage Trap: Understanding UPGen and Proteus for Censorship Evasion

Lakshit Singh Bisht

lakshitsinghbishttm@gmail.com

Central University of Andhra Pradesh https://orcid.org/0009-0004-0035-5714

Research Article

Keywords: Censorship Evasion, Pluggable Transport, Tor Network, Protocol Obfuscation, Programmable Protocols

Posted Date: September 29th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11180502/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

The Collateral Damage Trap: Understanding UPGen and Proteus for Censorship Evasion

Lakshit Singh Bisht
Central University of Andhra Pradesh

E-mail: lakshitsinghbishttm@gmail.com

Abstract

Nation-state censors maintain blocklists of prohibited services and use Deep Packet Inspection (DPI) to block parts of the Internet, which limits the freedom of people and is contradictory to the principle of free access to the Internet. In this paper, we analyze, evaluate, and test two complementary systems — UPGen, a novel protocol generator which produces distinct structured encrypted protocols, and Proteus, a programmable protocol engine for censorship circumvention. The core idea behind UPGen is the collateral damage trap: a censor trying to block it cannot do so without blocking a large portion of Internet traffic, making such blocking impractical. We study the design and operation of UPGen and Proteus through analysis and a series of small-scale experiments. Our analysis of the public UPGen implementation derives an independent estimate of the protocol configuration space of approximately 4.1×10^{25} possible configurations. We generated and validated 3111 Protocol Specification Files (PSFs), analyzed their structural characteristics, and investigated the protocol-generation process implemented in the public release. Our classifier reproduction experiments on the public artifact dataset found that Random Forest, 1D-CNN, and Deep Fingerprinting classifiers reliably distinguished UPGen traffic from benign protocols, in contrast to the resistance reported in the original evaluation. These results suggest that the publicly released artifacts do not reproduce the collateral-damage properties described in the original work. Our study therefore examines the potential advantages and limitations of the UPGen and Proteus approach in light of this reproducibility gap.

Keywords: Censorship Evasion, Pluggable Transport, Tor Network, Protocol Obfuscation, Programmable Protocols

1 Introduction

Access to information and privacy are important aspects of an open Internet, which has become one of the largest sources of knowledge and communication. However, many countries and regions restrict access to online services and information through various forms of Internet censorship. As a result, access to blocked services becomes difficult or impossible for affected users. As censorship technologies evolve, nation-states deploy increasingly sophisticated systems to restrict access to selected parts of the Internet. As a result, various tools have been developed to combat nation-state censorship. In this paper, we study two of the tools made for this purpose, namely Proteus[1] and UPGen[2].

Before explaining UPGen and Proteus, we define the threat model considered in this study. The adversary we consider is equivalent to a nation-state. We assume the adversary can —

1. Monitor and analyze data entering and leaving the network.
2. Block parts of Internet services legally and completely.
3. Maintain a blocklist of prohibited services and deploy censors.
4. Possess adequate resources and technology.
5. Perform passive monitoring and probing.

However, the adversary is assumed to not be capable of —

1. Compromising devices running UPGen and Proteus.
2. Blocking the Internet as a whole.
3. Maintaining an allowlist instead of a blocklist.
4. Modifying the generated protocols.

For this purpose, censors face challenges such as excessively large traffic flow and large volumes of heterogeneous network traffic, including encrypted and unencrypted protocols, well-known services, previously unseen traffic patterns, etc. As a result, real-world censors prefer classifiers which can easily stop the prohibited traffic while minimizing disruption to benign protocols, which limits the collateral damage.

UPGen and Proteus are designed to exploit this gap. In the original work, the authors explore the question of *can we create and use multiple network protocols to increase the accidental censorship of benign flows (i.e., false positives) while limiting censorship of circumventing flows (i.e., true positives)?* Based on the original work, in this paper we discuss the architecture and working of UPGen and Proteus and analyze how they will be useful against the real-world censors.

There are several proposed systems for censorship circumvention which can be divided into many types based on circumvention technique, e.g., Tun-

neling, Protocol Mimicry, etc. However, some of them such as TLS[3] with ESNI and tunneling systems have been exploited by censors deployed by China[4] and Iran[5]. Since these are not adaptable and lack a contingency plan in case they are blocked, they cannot be used for censorship evasion against the adversaries who identify and block them. This makes us consider a censorship evasion strategy in which new, plausible and unlinkable protocols can be generated to evade censorship. Since this is a new type of traffic, a censor's classifier will not be able to correctly identify it and will allow it through. Even if a censor somehow knows about UPGen and Proteus, it still cannot be trained to find all of the protocols generated by UPGen, due to the very large space of possible protocols. If a censor tries to identify and censor a single protocol, other protocols remain unaffected and it makes no meaningful reduction in the overall protocol space.

UPGen generates plausible yet unlinkable protocols. UPGen protocols do not copy any other existing protocols, so they cannot be blocked simply by blocking any other protocol. However, the generated protocols are structurally similar to existing encrypted protocols and lack any distinguishing feature, so it becomes very difficult to classify them. As a result, if a censor tries to block UPGen and Proteus, it will lead to the blocking of a large number of benign protocols, while several new UPGen protocols can still be generated and used, leading to collateral damage.

Additionally, Proteus allows the protocols to be programmable and it can be adapted to changing censorship environments. It allows the protocols to be quickly implemented and deployed. Proteus is designed to support multiple protocols simultaneously by a single Proteus proxy. As special cases of this, protocol versioning can be used to respond to new censorship rules while still supporting existing clients, and proxies can easily support clients in different locations according to their local censorship environment. However, support for multiple PSFs or protocol versions is not implemented in the Proteus prototype as of v0.2.0 which was analysed in this study.

Proteus ensures safety by specifying a limited runtime system that prevents the protocol specification files from being maliciously used to exploit proxies or clients. For its protocol specification files, Proteus also provides a language specification designed to be usable by programmers, along with a message formatting component that decides the format of each protocol message. This requires very little programming background, so it becomes easy for users to configure it in censored regions.

In the original work, the authors test and find that UPGen and Proteus protocols meet or exceed the latency, throughput, and scalability of other censorship evasion tools such as obfs4 and TLS[3]. In addition, they are

highly maintainable. This suggests that it could become one of the more favourable pluggable transports in the future if further development and deployment are successful.
In this paper, we summarize our primary contributions as follows:

1. Study the design and architecture of UPGen and Proteus for the generation and deployment of plausible unidentified protocols for censorship evasion.
2. Derive and analyze the protocol configuration space of the public UPGen implementation, including an independent estimate of protocol diversity.
3. Evaluate the public implementation through PSF validation, protocol diversity analysis, and reproduction experiments.
4. Compare UPGen and Proteus with established Tor pluggable transports such as obfs4 and Snowflake, including observations from a long-term Snowflake proxy deployment.

2 Background

This section provides the necessary background on Internet censorship, pluggable transports, and existing censorship circumvention systems. We briefly discuss obfs4 and Snowflake, which serve as useful reference points for understanding the design goals of UPGen and Proteus.

2.1 Evolution of Censorship and Circumvention

Censorship has been prevalent for a long time, even before the dawn of the Internet. Rulers have always tried to control the flow of information, which became a lot harder with the invention of the printing press. John Wycliffe translated the Bible into English in $1380 - 81$, and started the Lollard movement, but it was suppressed along with the Peasants' Revolt. However, when William Tyndale tried it in $1524 - 25$, printing helped him spread it to so many people that the princes and bishops could not suppress it. They later burned him at the stake, but by the time they burned him, more than $50,000$ copies of the New Testament had already been printed and the Reformation was under way. Due to this, printers later were closely licensed and they were controlled by the state till the eighteenth century[6].

With the invention of the Internet, this control has shifted to the Internet. Every year, nation-states try to control the Internet. According to the annual report of Freedom on the Net 2025[7], which covered 72 countries, global Internet freedom declined for the 15th consecutive year. In

addition, one key insight is that control over online information has become an essential tool for authoritarian leaders. The report indicates that China and Myanmar remained the world's worst environments for Internet freedom, while Iceland held its place as the freest online environment[8]. The authors report that governments in the countries that suffered the most extreme declines over the 15 years of global deterioration in Internet freedom — Egypt, Pakistan, Russia, Turkey, and Venezuela — have intensified their control over the online environment. Authorities have expanded restrictions on content, escalated surveillance of electronic communications, and imposed more severe penalties on those who expressed dissent online, particularly during protests and elections.

In response to such forms of censorship, several tools have been developed throughout history. One of the most widely used tools for online anonymity and privacy is Tor. In the early 2000s, Roger Dingledine, a recent Massachusetts Institute of Technology (MIT) graduate, named the project Tor, which stood for The Onion Routing.

In 1998, the U.S. Naval Research Laboratory designed Tor, the onion router for private communication[9]. It was called onion routing because of the layered routing of connections. The Tor infrastructure is run by servers known as relays, and they are of three types: Guard, Middle, and Exit. For a user to connect to a site, Tor first connects the user to a guard relay, which is in turn connected to the middle relay, and the middle relay itself is connected to an exit relay. The exit relay then connects to the service and exchanges data. This ensures that even if one of the relays gets compromised or becomes malicious, the anonymity of the user is not compromised. In addition, all connections are fully encrypted by Tor at the network level and no data remains unencrypted. The U.S. Navy opened Tor to the general public in 2003 for a larger pool of users. This is due to the fact that if only US officials used Tor, then everyone using Tor could be identified as a target.

Tor is now maintained by The Tor Project[10], which became a non-profit organization in 2006 and has thousands of volunteers — for relay operations, development, security, outreach, research, etc. This makes it possible for the general public to use Tor and connect to services privately. However, many countries block Tor entirely, so beginning in 2005, development of tools beyond just the Tor proxy began. In 2007, the Tor project began developing pluggable transports for the Tor Network to address censorship[11].

The concept of pluggable transports is based on the fact that while a censor tries to block a given service, other services remain open and unblocked. Thus, an encrypted protocol tries to hide the use of a blocked service and it appears to the censor that the traffic is not directed towards the blocked

service. The two most widely used pluggable transports in the Tor Project are obfs4 and Snowflake, which are explained in the next section.

2.2 Existing approaches: obfs4 and Snowflake

In the Tor Network, obfs4 and Snowflake are widely used pluggable transports for censorship circumvention. This sub-section briefly describes the approach and working of these transports.

2.2.1 obfs4

obfs4[12] is a protocol obfuscation layer used in the Tor Network. Its primary purpose is to provide obfuscation for authenticated protocols such as SSH or TLS and it can also provide authentication and data integrity. It is based on the ScrambleSuit protocol[13] developed by Philipp Winter. obfs4 is much closer to ScrambleSuit than obfs2/obfs3 in terms of protocol ancestry.

obfs4 offers protection against passive Deep Packet Inspection machines that expect the obfs4 protocol. In addition, it offers protection against active attackers who are attempting to probe for obfs4 servers. It offers protection against active attackers that have obtained the server's Node ID and identity public key. Even after obtaining this, such attackers would not be able to impersonate the server without the private key. It also offers protection against some non-content protocol fingerprints like packet size, packet timing, etc.

The major differences between obfs4 and ScrambleSuit are as follows:

1. It uses an authenticated key exchange mechanism based around the ntor handshake of the Tor Project.
2. The handshake does a full key exchange.
3. The link layer encryption uses NaCl secret boxes (Poly1305/XSalsa20).

The detailed working of obfs4 is described in the following figure.

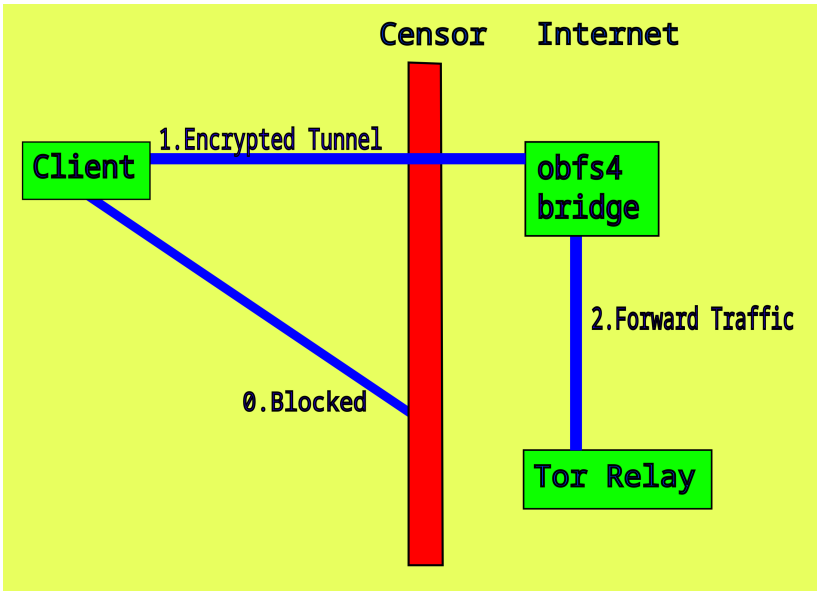


Figure 1: High-level architecture and connection process of obfs4 bridge.

obfs4 has two phases:

1. **Phase 1:** In this phase, keys are established between both parties.
2. **Phase 2:** In this phase, super-enciphered traffic is exchanged between the parties.

The following steps show exactly how it works.

1. The client handshake process starts with the generation of an ephemeral Curve25519 key pair and an Elligator 2 representation of the public key. The client sends a handshake request to the server and receives the serverResponse from the server. The client derives several values from the serverResponse and matches them against the server. If values do not match, the client must drop the connection. The client derives a value from the value provided by the server via the Elligator 2 map in the reverse direction. The client completes the client side of the ntor handshake, deriving the 256-bit shared secret (KEY_SEED), and the authentication tag (AUTH) and verifies the received AUTH value.
2. The server handshake process starts when the server receives the clientRequest from the client. The server verifies the authentication values contained in the client request, generates an ephemeral Curve25519 key pair and an Elligator 2 representation of its public key, and derives the required values to complete the handshake. The server completes

the server side of the ntor handshake, deriving the 256-bit shared secret (KEY_SEED), and the authentication tag (AUTH).

3. The server sends a handshake response to the client and at the point that each side finishes the handshake, they have a 256-bit shared secret KEY_SEED that is then extracted or expanded via the ntor KDF to produce the 144 bytes of keying material used to encrypt or authenticate the data.
4. After the above steps, phase 1 is completed and Data Transfer phase starts. Once both sides have completed the handshake, they transfer application data broken up into packets. The packets are then encrypted and authenticated using NaCl crypto_secretbox_xsalsa20poly1305 frames.
5. To avoid transmitting identifiable length fields in the data stream, the frame length is obfuscated by XORing a mask derived from SipHash-2-4 in OFB mode. As the receiver has the SipHash-2-4 key and IV, decoding the length is done via deriving the mask used to obfuscate the length and XORing the truncated digest to obtain the length of the secretbox. The counter is initialized to 1, and is incremented on each frame. The nonce is not transmitted over the wire as both sides of the conversation know the prefix and the initial counter value. The counter does not wrap, and sessions terminate before 2^{64} frames are sent.

This creates the connection between proxy and client and the process is completed.

2.2.2 Snowflake

Snowflake[14] is a pluggable transport which can be used for Internet access in blocked regions. It is used for censorship circumvention in the Tor Network. Snowflake is easy to set up as a browser extension and it allows volunteers to help just by keeping their browser tab open. As a result, many people without significant technical knowledge can also volunteer to run Snowflake. In addition, it can also be configured to run as a standalone service for experienced volunteers. The goal of Snowflake is to outpace the censor's ability to block proxy IP addresses and to provide a reliable method of circumventing censorship which cannot be blocked easily.
The detailed working of Snowflake is described in the following figure.

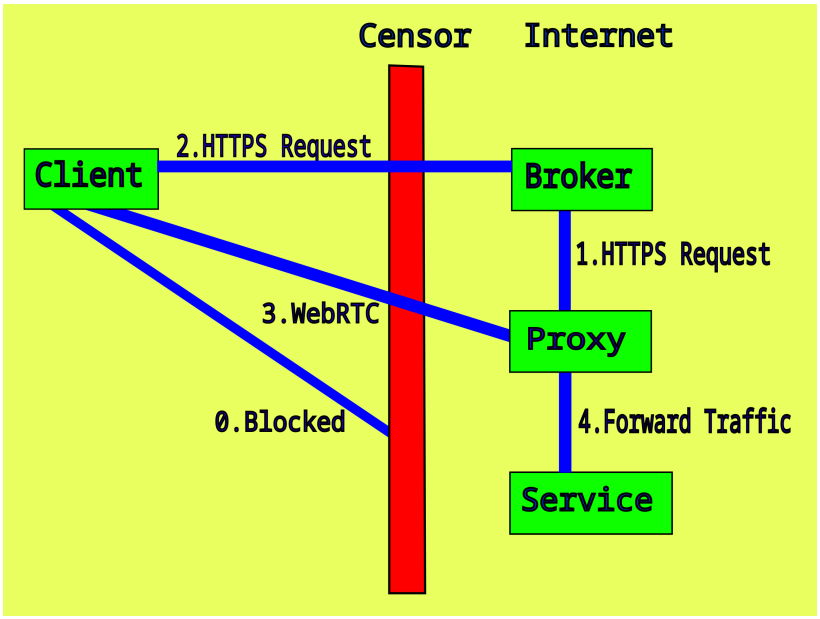


Figure 2: High-level architecture and connection process of Snowflake.

It consists of three main components — client, proxy, and broker.

1. The Snowflake client: It is a client transport plugin and Tor utilizes this just like any other pluggable transport. It conforms to the pluggable transport specification.

2. The Snowflake proxy: It is an in-browser WebRTC proxy. The proxy forwards traffic between the Snowflake client and a Tor relay. Also, it conveys data between Snowflake clients and some destinations.

3. The broker: It is responsible for rendezvous and exclusively uses Domain Fronting.

By relying on a large pool of volunteer proxies and WebRTC-based peer-to-peer connections, Snowflake aims to make large-scale IP blocking more difficult for censors. The following describes the working of Snowflake:

1. A Snowflake proxy sends a POST request to the broker for serving.
2. A Snowflake client sends a POST request with SDP Offer to the broker.
3. The broker forwards the SDP Offer as the reply to one of the Snowflake proxy's subscribe polls.
4. The Snowflake proxy receives the SDP Offer, composes an SDP Answer, and sends a POST request to the broker with the answer.
5. The broker forwards the SDP Answer as the reply to the original Snowflake client.

6. Both the Snowflake proxy and Snowflake client now have each other's SDP messages.

This creates the WebRTC connection between proxy and client and the process is completed.

3 UPGen: Unidentified Protocol Generation

This section explains the heart of the system — the Unidentified Protocol Generation (UPGen), which generates protocols for censorship circumvention.

3.1 Overview

UPGen generates novel protocols which can be used for censorship circumvention. It generates protocols that are not linkable to other encrypted protocols or to other UPGen protocols. This makes the fingerprinting of UPGen protocols very difficult and impractical. The repository structure of UPGen[15] is provided in Appendix B. The source code of UPGen is written in the Python programming language, and it generates PSFs which can be used via Proteus. UPGen protocols run over Transmission Control Protocol (TCP) and operate similarly to Transport Layer Security (TLS), i.e., between the transport and application layers.

What UPGen intends to provide:

1. Generate unique and novel protocols
2. Resist fingerprinting of generated protocols
3. Usable protocols for Proteus and other Programmable Protocol Systems (PPS) such as Marionette[16], WATER[17], etc.
4. Predefined configuration for creating PSFs

Non-Goals of UPGen:

1. A platform or service for using protocols on its own
2. Providing security guarantees outside its threat model
3. Generating protocols exactly the same as others like TLS, SSH, etc.
4. Creating identical protocols again and again

3.2 Configuration Space

The configuration of protocols generated by UPGen is specified in the JSON file in the source. It takes the following parameters with probabilities, which are equally distributed as default but can be tweaked accordingly for different output. Changes to these parameters influence the generated PSF

structure and contents. The following tables show the values and parameter ranges in brief for UPGen.

Table 1: UPGen configuration parameters from the default `config.json`

Parameter	**Values**
256_bit_cipher	AES256GCM, CHACHA20POLY1305
extra_field_nbytes_handshake	0, 1
extra_field_nbytes_data	0, 1
greeting_reversed	false, true
has_greeting	false, true
has_nonce	false, true
key_pattern	pattern1–pattern8
length_field_nbytes	2, 4
length_field_covered_by_mac	false, true
major_version	1, 2
minor_version	1, 2
normal_types_first	false, true
overall_structure	0, 1
padding_length	2, 4
reserved_field_length_nbytes	0, 1
sec_param	128, 256
send_encrypted_cert	false, true
separate_length_field	false, true
subprotocol_nrounds	0, 1, 2
type_field_encrypted	false, true
type_num_start_val	0, 1, 10, 20
version_field_encrypted	false, true
version_field_in_handshake	false, true
version_field_nbytes	1, 2

Table 2: UPGen range parameters from the default `config.json`

Parameter	**Range**
encrypted_cert_nbytes	1–1000
subprotocol_round_sizes	1–1000
ntype_nums	1–10

These fields, based on their probabilities, help generate the PSF. In the original work, the authors provide the following table for calculating the total number of protocols

Table 3: Original Quantitative Analysis of UPGen Generator

Protocol Parameter	**# Options**	**Entropy (bits)**
Security parameter	2	1.0
Encryption cipher	3	0.5
Type field	25	2.7
Length field	2	0.81
Version field	36	4.1
Nonce field	3	1.0
Padding field	2	0.25
Extra field	9	2.4
Reserved field	5	1.1
Certificate	2048	6.5
Key encoding	3	1.6
Greeting string	3	1.1
Handshake pattern	8	3.0
Subprotocol pattern	4.0×10^9	8.3
Field order	12	3.6
Length alone	2	0.48
Total	4.2×10^{22}	**38.4**

It should be noted that the quantitative analysis presented in Table 3 is reproduced from the original UPGen paper. During our analysis of the public UPGen implementation and configuration files, we observed that the parameterization used in the original evaluation is not fully included in the public release. The authors have indicated that some protocol-generation parameters used in their study were intentionally withheld and are therefore unavailable for independent verification. Consequently, the value of 4.2×10^{22} should be interpreted as the diversity estimate reported by the original authors rather than a result independently reproduced in this study. We attempted to estimate it based on the implementation along with Shannon and Hartley entropy, presented in the following table:

Table 4: Implementation-derived quantitative analysis of UPGen generator

Protocol Parameter	**Options**	**Shannon Entropy**	**Hartley entropy**
Security parameter	2	1.0000	1.0000
Encryption cipher	3	1.5000	1.5850
Type field	144	7.1699	7.1700
Length field	4	2.0000	2.0000
Version field	24	4.5000	4.5850
Nonce field	2	1.0000	1.0000
Padding field	3	1.5613	1.5850
Extra field	4	2.0000	2.0000
Reserved field	2	1.0000	1.0000
Certificate	1000	5.9822	9.9658
Key encoding	3	1.5850	1.5850
Greeting string	3	1.5000	1.5850
Handshake pattern	8	3.0000	3.0000
Subprotocol pattern	9.96×10^{11}	21.7128	39.8574
Field order	72	N/A	6.1699
Length alone	2	1.0000	1.0000
Total	4.1×10^{25}	56.5112	85.0878

Derivation:
We provide a short idea of how we arrived at these numbers.

1. The security parameter is derived from `sec_param`, which contains the values 128 and 256, resulting in 2 possible configurations.
2. The encryption cipher is derived from `choose_cipher()`, which may produce AES128GCM, AES256GCM, or CHACHA20-POLY1305, resulting in 3 possible configurations.
3. The type field is derived from `type_field_encrypted`, `normal_types_first`, `type_num_start_val`, and `ntype_nums`, giving $2 \times 2 \times 4 \times 9 = 144$ possible configurations.
4. The length field is derived from `length_field_nbytes` and `length_field_covered_by_mac`, each having two values, resulting in $2 \times 2 = 4$ configurations.
5. The version field is derived from `version_field_nbytes`, `version_field_in_handshake`, `version_field_encrypted`, `major_version`, and `minor_version`, giving $8 + 16 = 24$ configurations.
6. The nonce field is derived from `has_nonce`, which may be either true or false, resulting in 2 possible configurations.
7. The padding field is derived from the absence of padding, together

with `padding_length` values of 2 and 4, resulting in 3 configurations.

8. The extra field is derived from `extra_field_nbytes_handshake` and `extra_field_nbytes_data`, each taking values from $\{0, 1\}$, resulting in 4 configurations.
9. The reserved field is derived from `reserved_field_length_nbytes`, which may take the values 0 or 1, resulting in 2 configurations.
10. The certificate field is derived from `send_encrypted_cert` and `encrypted_cert_nbytes`. This gives one configuration without a certificate and 999 possible certificate sizes, resulting in 1000 configurations.
11. The key encoding field is derived from the three supported encodings, RAW, DER, and PEM, resulting in 3 possible configurations.
12. The greeting string field is derived from `has_greeting` and `greeting_reversed`, resulting in the states of no greeting, normal greeting, and reversed greeting, for a total of 3 configurations.
13. The handshake pattern field is derived from the eight predefined patterns implemented in `choose_handshake_pattern()`, resulting in 8 configurations.
14. The subprotocol pattern field is derived from `subprotocol_nrounds` with values 0, 1, and 2, corresponding to 1, 999^2, and 999^4 possibilities respectively, giving 996,006,994,003 configurations.
15. The field order is derived by independently permuting encrypted and unencrypted fields and selecting one of two overall structures. In the maximum-field configuration implemented by `choose_field_order()`, there are three reorderable unencrypted fields (type, length, version) and three reorderable encrypted fields (padding_length, extra, reserved), giving $2 \times 3! \times 3! = 72$ configurations.
16. The length alone field is derived from `separate_length_field`, which may be either true or false, resulting in 2 configurations.

In the table, the total came to 41,113,418,544,890,089,832,448,000, which suggests that the implementation exposes an extremely large protocol configuration space. In the original work, the authors claim to use Shannon entropy, whereas in this table we calculate Hartley entropy along with Shannon entropy. The total Hartley entropy computed from the product of all parameter configurations was verified independently by summing the individual parameter Hartley entropies, yielding the same value of 85.08781170790766 bits. The Shannon entropy total excludes the field-order parameter, since the implementation does not define an explicit probability distribution over field-order configurations and only an upper bound on the number of possible orderings could be derived.

Calculation of Entropies:
To calculate Shannon Entropy, we used the formula

$$H(X) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

where:
- $H(X)$ is the Shannon entropy of the random variable X
- p_i is the probability of the i-th outcome
- n is the total number of possible outcomes

To calculate Hartley Entropy, we used the formula

$$H_0(X) = \log_2(n)$$

where:
- $H_0(X)$ is the Hartley entropy of the random variable X,
- n is the total number of possible outcomes

3.3 Protocol Generation

The working of UPGen can be summarized briefly in the following process:

3.3.1 Training Model

In the first step, the neural network is trained to generate the greeting messages for UPGen protocols. This is done by providing an input file containing a minimum of 10,000 entries, which is used to train the model to generate greeting strings that resemble repository names and help disguise the protocols. In this step, the model is provided with the input file and it generates the parameters for the model, which are then used to train the model based on those parameters. For parameter generation, it takes 5 samples in each iteration and repeats this process 50 times. For training the model, the training process runs from 1% to 100% progress unless an early-stopping condition is triggered. The trained model is used solely to generate greeting strings and does not influence the remaining protocol structure. After this step, the model is ready to generate the greeting strings, required for the PSFs that include a greeting field as part of a structured message.

3.3.2 Generation of PSFs

After training, the model is used by the generation pipeline to produce new PSFs. The protocol structure is represented using the dataclasses

defined in `psf.py`. Configuration parameters such as security settings, payload formats, handshake patterns, and protocol fields are selected by `upgen_v1.py` according to the ranges specified in the configuration file. Finally, `generate.py` combines these components and generates complete Protocol Specification Files (PSFs).

3.4 Protocol Specification Format

A PSF consists of packet structures, field semantics, protocol message sequences, cryptographic semantics, etc. The generated PSF is the final output of UPGen and can be directly compiled and deployed by Proteus. An example of PSF generated during the study is provided in Appendix D. A brief description of a PSF is given below:

1. SEGMENT.FORMATS: This section defines packet formats and their data types. For example, the `handshake1` format contains fields such as `type`, `nonce`, `ephemeral_key`, `msg_mac`, etc. The exact composition and ordering of these fields are determined by UPGen configuration.

2. SEGMENT.SEMANTICS: This section assigns behavior to each field, such as fixed values and random values. These semantics determine how packet contents are generated during protocol execution.

3. SEGMENT.SEQUENCES: This section defines the protocol state machine by specifying the order of messages exchanged between the client and server. For example, the protocol can begin with a handshake between the client and server and then the data exchange phase. UPGen generates different protocol behaviors by varying the number and ordering of handshake messages according to the configured handshake pattern.

4. SEGMENT.CRYPTO: This section assigns cryptographic parameters and encryption rules. This includes the password, cipher, handshakes, and data itself. It can generate a high-entropy password and cipher from `AES128`, `AES256` or `CHACHA20`.

5. SEGMENT.OPTIONS: This section defines the separate length field which can be either a true or a false boolean value.

However, the structure of PSFs varies and it is possible to generate PSFs with different structures and values. Therefore, this should be regarded only as a reference example of a PSF.

4 Proteus: Programmable Protocol Engine

In this section, we describe the soul of the system — Proteus, a Programmable Protocol System (PPS) used for running the protocols generated by UPGen.

4.1 Overview

Proteus is a programmable tool for censorship circumvention which establishes communication tunnels between clients and proxy servers using programmable network protocols. It has been developed to adapt to a changing censorship environment rapidly and stealthily. It uses Protocol Specification Files (PSFs) generated by UPGen as its input. The repository structure of Proteus[18] is provided in Appendix A. It is implemented in the Rust programming language and provides programmable protocols for its users.

What Proteus intends to provide:

1. Adaptive and programmable framework for protocols
2. Protection against malicious or erroneous protocol updates via a restricted execution environment
3. Compatibility with the Tor Network
4. Graceful, backward-compatible protocol version upgrades
5. A standard library to provide common and required functionalities for communication protocols, including cryptographic primitives

Non-Goals of Proteus:

1. Providing security guarantees outside its threat model
2. Generating PSFs automatically
3. Using a protocol outside its specific format
4. Replacing TCP in the transport layer

The detailed working of Proteus using a UPGen protocol is described in the following figure.

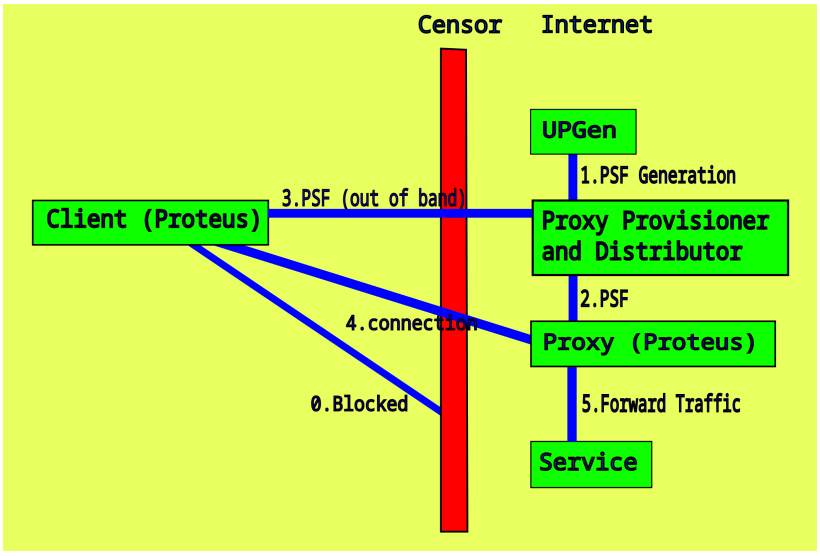


Figure 3: High-level architecture and connection process of Proteus.

4.2 Design Goals

Proteus was designed to enable fast reaction to a changing censorship environment by making protocols themselves programmable rather than fixed. The original design identifies four key goals:

1. **Pairwise communication:** Proteus provides bidirectional communication between a client and a proxy server. It focuses particularly on secure protocols that use cryptography for message confidentiality and integrity; however, unencrypted protocols can also be implemented.
2. **Programmability:** A Proteus protocol can be reconfigured easily, and UPGen can be used to generate PSFs easily. This allows users in authoritarian regimes to adapt to new censorship rules more quickly, and lets users focus on changing PSFs instead of programming a new protocol to evade censorship. In addition, the language is designed to be familiar to programmers, which makes it easier to work with during censorship.
3. **Safety:** Specification files are supposed to be written and distributed by third parties. In the original work, the authors expect the domain experts like Tor Project, Shadowsocks developers, etc. will develop and distribute the PSF files to their communities. As a result, Proteus restricts what a specification can do. The runtime exposes only a restricted set of operations and places limits on the resources consumed

during protocol execution. In addition, the protocol specifications are expressed in a high-level language that even a non-expert can inspect before installing it.

4. **Graceful updates:** Proteus is designed to support multiple simultaneously active protocol versions at a single proxy, so that a server can adopt a new protocol without breaking existing clients, and can simultaneously serve clients in different censorship regions using different specifications. It provides a huge advantage over other systems which do not support this.

4.3 Proteus Language

A Proteus protocol is expressed in a Protocol Specification File (PSF), written in a custom language designed for the purpose. The language's syntax is intentionally close to the Rust programming language, so that programmers without specialized programming languages or security backgrounds can easily read and write PSFs without learning an unfamiliar notation.

Layout: A PSF is organized into three parts — protocol message definitions, global state variables, and event handling functions. The message definitions contain the layout of every message the protocol will send or receive, the global variables contain state like cryptographic keys and the event handlers contain the logic that is invoked when network or application data becomes available.

Language constructs: The original Proteus paper describes the language as simple, minimal, easily edited and interpreted at runtime. It is statically typed and statically allocated, with function-level lexical scoping (global variables excepted, which have static lifetime). It supports variable declaration and assignment, basic logical and arithmetic operations, branching via `if` and `match`, type casting, standard library calls, and statically-bounded `for` loops.

The Proteus language is designed to be limited and it does not support many functionalities such as function or class declaration, infinite loops, floating-point arithmetic, template and macro metaprogramming, other than a few trusted standard library functions.

However, the current implementation exposes a considerably more limited declarative PSF language centered primarily on message formats, semantic annotations, protocol sequences, cryptographic directives, and runtime options. While the original paper describes a richer protocol programming language, the implemented system largely operates through Protocol Specification Files that describe packet structure and protocol behavior

rather than general-purpose program logic.
Standard library: Since the language itself is deliberately minimal, the functionalities of the standard library are also minimal and have only the compulsory functionalities a protocol needs beyond pure field manipulation: I/O functions to inspect and manipulate the network and application buffers, utility functions for environment variables, logging and timers, message formatting and parsing functions, field accessors, and cryptographic primitives from the RustCrypto packages.

4.4 Architecture

This subsection explains the core architecture of Proteus. At a high level, the Proteus architecture separates the system into three layers, namely Intermediate Representation (`src/lang/ir`), Compiler (`src/lang/compiler`), and Interpreter (`src/lang/interpreter`). This mirrors the repository's own module structure provided in Appendix A. A brief description is provided below:

4.4.1 Intermediate Representation (IR)

The IR serves as an abstraction layer between the protocol specification and the runtime system. Proteus first translates the specification into a sequence of intermediate instructions. This design separates the description of a protocol from its execution.

The IR consists of a set of instructions that describe packet processing operations, including reading data from the application or network, constructing messages, computing length fields, encrypting and decrypting fields, and writing data to the network or application. Each instruction performs a single operation and collectively these instructions describe the behavior of a protocol. A protocol task consists of an ordered sequence of instructions together with an associated task identifier. The execution of a protocol therefore becomes the execution of a series of tasks, where each task contains a sequence of packet-processing instructions. Each Proteus connection runs a pair of protocol instances, one for the client and one for the server. Both directions of data flow — application-to-network and network-to-application — execute concurrently using an asynchronous Rust runtime and share only the minimal cryptographic state needed to keep both directions consistent.

4.4.2 Compiler

The Proteus compiler transforms a PSF into executable protocol tasks. The compiler first parses the protocol specification using a grammar-based parser, producing an internal representation containing message formats, semantics, cryptographic directives, and message sequences. The sequence specification is then converted into a task graph that represents the progression of the protocol. Handshake messages create transitions between states. For each protocol message, the compiler generates a sequence of instructions describing the operations required to send or receive that message. As a result, the compiler translates a high-level protocol specification into a low-level sequence of instructions that can be executed by the runtime. This compilation process allows the same execution engine to support multiple protocols without modifying the underlying implementation.

4.4.3 Interpreter

The interpreter serves as the execution engine of Proteus. It receives the instruction sequences generated by the compiler and executes them using a virtual machine architecture. The virtual machine maintains three primary components: a heap for temporary storage, an I/O subsystem for communication with the application and network, and a cryptographic subsystem for encryption and decryption operations. Instructions operate on these components to manipulate messages and protocol state. Protocol execution proceeds by first loading a task, executing each instruction sequentially, and then advancing to the next task. Instructions such as reading data, computing lengths, setting field values, encrypting payloads, writing packets, etc. are executed one at a time. Two execution contexts operate concurrently to support bidirectional communication. One direction processes data flowing from the application to the network, while the other processes data received from the network and delivered to the application. Shared cryptographic state allows both directions to maintain synchronized encryption and decryption contexts.

From an implementation perspective, Proteus behaves similarly to a virtual machine for network protocols. A PSF acts as a high-level program, the compiler translates the specification into `InstructionV1` operations, and the interpreter executes these instructions against the network, memory, and cryptographic subsystems. This separation allows multiple protocol specifications to share the same execution engine.

4.5 Protocol Specification Processing

Before a protocol can be executed, a PSF undergoes a sequence of transformations that convert the textual specification into executable protocol tasks. The implementation follows four stages: parsing, validation, intermediate representation generation, and task construction. We describe them in this subsection below.

1. **Parsing:** The PSF is first parsed using a parsing expression grammar implemented in `proteus_lite.pest` through the `pest` parser generator. The grammar defines the syntax of message formats, field types, semantic annotations, protocol sequences, cryptographic directives, and configuration options. Parsing produces a `Psf` structure containing the complete protocol description, including format definitions, semantic bindings, message ordering, cryptographic parameters, and protocol options.
2. **Validation:** After parsing, the implementation verifies the consistency of the specification. Message formats referenced in the sequence specification must be defined in the format section, and protocol fields must satisfy several structural assumptions used by the compiler. The implementation assumes that payload-carrying messages contain a corresponding length field and that fixed-size length fields appear within the static portion of the message. Invalid or inconsistent specifications are rejected before compilation.
3. **Intermediate representation generation:** The validated protocol description is transformed into a directed task graph. Handshake messages advance the protocol state by creating transitions between graph nodes, while data-phase messages produce self-loops that permit repeated data exchange after the handshake has completed. Each graph edge is subsequently compiled into a sequence of `InstructionV1` operations that describe the actions required to process the message. These operations include reading application or network data, computing length fields, performing encryption/decryption, extracting field values, transmitting packets, etc.
4. **Task construction:** The generated instruction sequences are organized into tasks associated with individual protocol states. These tasks are exposed through the `TaskProvider` interface, which supplies initialization tasks and subsequent protocol tasks to the runtime system. Proteus does not generate native code or external bytecode. Instead, the instruction sequences themselves serve as an intermediate representation that is executed directly by the interpreter.

As a result, the protocol processing pipeline can be summarized as:
PSF → Parser → Task Graph → Instructions → Interpreter

4.6 Proteus Workflow

This section explains the Proteus workflow. A Proteus deployment proceeds through four steps which are explained below:

1. **PSF creation:** A PSF is created directly in the Proteus language, or generated by UPGen and copied into the Proteus working directory. Appendix D shows an example PSF used in our experiments.
2. **Compilation:** The PSF is compiled by invoking the `proteus` binary. In our Experiment 1, we used the `check` subcommand directly, which parses the PSF in both the `Client` and `Server` roles, and reports a compilation error if parsing or validation fails. In a live deployment, the same compilation step is performed internally by Proteus before the proxy or client begins listening for connections.
   The PSF file can also be manually checked using the following command:

   ```
   proteus check file.psf
   ```

3. **Deployment:** Proteus is designed to run as a Tor pluggable transport, communicating with the Tor process via the standard PT v1 environment-variable protocol (`src/cli/pt/config`). A client is configured with the path to the PSF passed as a `psf=` option. Each side loads its PSF once at startup and uses it for the duration of the process.
4. **Execution:** Once both sides are running with compatible PSFs, the interpreter drives the connection. It requests the next task from the compiled task graph, executes the resulting instruction sequence against the live network and application buffers, and repeats until the connection's data phase is reached, after which data-phase instructions repeat for the life of the connection.

4.7 Relationship with UPGen

UPGen and Proteus have complementary roles in the same pipeline: UPGen is a generator, and Proteus is an executor. UPGen outputs a PSF, and that PSF is Proteus's input. Proteus has no mechanism of its own for producing new protocol specifications; this division of labor is explicit in Proteus's own stated non-goals, which clearly state that Proteus does not generate PSFs automatically. Conversely, UPGen has no networking or execution

component: a PSF generated by UPGen is not itself a runnable program, but rather a declarative artifact that Proteus's compiler and interpreter must parse, validate, and execute before any bytes are exchanged with a censor.

This separation allows the two systems to be evaluated independently. In the next section, we perform a series of experiments involving UPGen and Proteus. UPGen itself cannot be directly used, and Proteus itself cannot generate a large number of protocols, so both components are combined to form a censorship circumvention tool. This makes UPGen the heart and Proteus the soul of the system.

5 Experiments

In this section, we discuss some small-scale experiments with UPGen and Proteus that we performed to evaluate their usability and the characteristics of the generated protocols. All experiments were conducted locally using the configuration provided in Appendix C, and no special hardware was used.

5.1 Experiment 1

Objective:
In the first experiment, our primary objective was to generate PSFs using UPGen and to check the generated PSFs using Proteus. This experiment was conducted in two phases, and we managed to generate 111 PSFs in Phase I with generation of 1, 10, and 100 PSFs in three runs and 3000 PSFs in Phase II with generation of 1000 PSFs each in three runs.

Prerequisites:

1. UPGen downloaded and set up
2. Proteus downloaded and set up
3. Torch, torchvision, and tqdm installed in Python
4. An input file with a minimum of 10000 entries with each entry in the form of "*name1/name2*"

Input:
In this experiment, only one input file, described in the prerequisites, is required. We performed this experiment using the input file provided by one of the authors of the original work — Ryan Wails, but this file can also be generated independently, with no special requirements other than the number and format of entries.

The generation of the input file can be done in various ways. Ryan Wails generated this file from data collected by GH Archive. For Linux, the input file can be generated in the CLI using the following command:

```
curl -L https://data.gharchive.org/2015-01-01-15.json.gz \
| gunzip -c \
| jq -r '.repo.name' \
| head -n 10000 > file.txt
```

Procedure:
In this experiment, we first trained the model for greeting message generation in 2 steps:
Step 1. Determining model hyperparameters

```
python train.py -y file.txt --output_filepath best_params.pkl
```

After Step 1, `best_params.pkl` gets created, and we need to create a directory named `trained_model` to move to Step 2.
Step 2. Training the model

```
python train.py -b best_params.pkl -d trained_model/ file.txt
```

These two steps are done in the `upgen/src/greetings/` directory. After training the model, the generation of PSFs is done using UPGen, and the working directory for generating PSFs is `upgen/src`. The generation of a PSF can be done using the following command:

```
python generate.py \
    config.json \
    greeting/best_params.pkl \
    greeting/trained_model/encoder.pkl \
    greeting/trained_model/model.torch \
    -n 1 \
    -o file.psf
```

This generates a PSF (Protocol Specification File) in the current directory. Using this process, we generated 1, 10, and 100 PSFs in the first phase and then 1000, 1000, and 1000 PSFs in the second phase. Note: For generating a large number of PSFs, bash scripts can be used to automate this process rather than generating PSFs manually one by one.

Table 5: Generated PSFs

Dataset	**Number of PSFs**
psf_first	1
psfs_10	10
psfs_100	100
psfs_1000	1000
psfs_run2	1000
psfs_run3	1000

After creating the PSFs, we need to take the generated PSF files to the root of the Proteus repository, and then PSFs are checked using the following command.

```
proteus check file.psf
```

Result:
We observed the following results from Experiment 1:

Table 6: Validation results for Phase I

Dataset	Total	Passed	Failed	Success Rate
psf_first	1	1	0	100.0%
psfs_10	10	8	2	80.0%
psfs_100	100	70	30	70.0%
Overall	111	79	32	71.12%

Table 7: Validation results for Phase II

Dataset	Total	Passed	Failed	Success Rate
psfs_1000	1000	779	221	77.9%
psfs_run2	1000	785	215	78.5%
psfs_run3	1000	768	232	76.8%
Overall	3000	2332	668	77.73%

A total of 79 out of 111 generated PSFs successfully passed validation in Phase I. Across the three runs in Phase II, 2332 out of 3000 generated PSFs successfully passed validation.

Failure Analysis:
The causes of failures of the Proteus check are shown in the following tables:

Table 8: Failure categories for Phase I (1+10+100)

Failure Category	Count
PEM parsing failure (Malformed Framing)	14
DER/public-key decoding failure (Explicit Panic)	10
Protocol execution failure (Server received 0/1024 bytes)	6
Payload mismatch	2
Total	32

Table 9: Failure Categories for Phase II (1000 + 1000 + 1000)

Failure Category	**psfs_1000**	**psfs_run2**	**psfs_run3**
PEM parsing failure (Malformed-Framing)	84	83	84
DER/public-key decoding failure (explicit panic)	86	86	86
Protocol execution failure (Server received 0/1024 bytes)	40	35	44
Payload mismatch	11	10	17
Partial payload transfer (750/1024 bytes)	0	1	0
Partial payload transfer (556/1024 bytes)	0	0	1
Total	221	215	232

The validation rate remained relatively stable across large-scale runs, suggesting that the observed failure modes were reproducible and not limited to a small sample.

Limitations:

We noticed the following limitations in Experiment 1:

1. The detailed analysis of why those PSFs failed was not done
2. The original codebase of UPGen was temporarily modified to use a GPU instead of running on a CPU for training the model
3. No analysis was performed to identify patterns among passing and failing PSFs

5.2 Experiment 2

Objective:

In the second experiment, our primary objective was to evaluate the effect of training corpus size on the greeting strings generated by UPGen. This experiment was conducted with 4 input files containing 10, 100, 1000, and 10000 entries respectively.

Prerequisites:

1. UPGen downloaded and set up
2. Input files for the model
3. Python libraries specified in the prerequisites of Experiment 1

Input:

For the file containing 10 entries (File I), the file was curated manually with the following custom entries:

```
lakshitsinghbishttm/xlade, torvalds/linux, official-stockfish/stockfish,
torproject/tor, robgjansen/cellshift, ajay/coreai-protocol-suite,
3b1b/manim, deducteam/dedukti, sanity/freenet-core, leanprover/lean4
```

For the file containing 100 entries (File II), the entries were taken with the script

```
wget -qO- \
"https://api.github.com/search/repositories\
?q=topic:cryptography&per_page=100" \
| jq -r '.items[] | "\(.owner.login)/\(.name)"' \
> curated_repo_100.txt
```

For the file containing 1000 entries (File III), we took the initial 10 entries from our first file, and then took the remaining 990 entries from the file provided by Ryan, starting from line 6969 of the file.
For the file containing 10000 entries (File IV), we used the complete file provided by Ryan.

Procedure:

In this experiment, we first edited the source code in `train.py` and replaced the constant value `10000` with the corresponding file size, and then followed the same procedure as in Experiment 1. After generating the PSFs, we opened the PSF and extracted the greeting from the following PSF entry:

```
@SEGMENT.SEMANTICS
{ FORMAT:       greeting_server ; FIELD:        greeting ;
    SEMANTIC:    FIXED_STRING ("<greeting>") } ;
```

and then recorded the greeting generated for each PSF.

Results:

We observed the following results from Experiment 2:

Table 10: Generated greeting strings across corpus sizes

PSF	File I	File II	File III	File IV
1	-oieeoithloit	p	plateoncript	trambdle-rubacker
2	elstf	eweuptp	tostLxb_devigwe r	LibreHDT
3	xr	ptithnftisoapsp dynciepya	kernel_msm	Hime-Rourer.githu b.io
4	aletlismte	u-	kisn-qnutipytho n	vernodice-gos
5	ttoenestfio	oiyUsru	redrenoce	servo
6	draxafrsd	nlEuchnmotsip eoojue	pouttines-Plso-0 0zhint	android_device_sa msung_samml
7	oepio	AAuiyRvpip	jssUals-Domigo	rails_fronters
8	rtkunaen	Jhry	ppagrefs	Decpachine
9	ttesto-oaultsfs eckorkeffrn	mseuSssPbwpS teroop	Proj080robol.git hub.io	Smallern-Reethes
10	xr	dfeptowcionojI an	teckflow-xbswof p	nodelibs-tomples

Table 11: Quantitative summary of generated greetings

Metric	File I	File II	File III	File IV
Total Greetings	10	10	10	10
Unique Greetings	9	10	10	10
Longest Greeting	26	24	22	28
Shortest Greeting	2	1	8	5
Unique Characters	19	30	35	29
Symbols	2	1	9	11

Table 12: Mathematical analysis of greeting string lengths (characters)

Metric	File I	File II	File III	File IV
Total Length	91	108	145	148
Average Length	9.1	10.8	14.5	14.8
Median Length	8.5	8.5	15.0	15.0
Mode Length	2, 5	7	16	16

Table 13: Human readability assessment of generated greetings

Corpus	Readability Assessment
File I	Predominantly nonsensical
File II	Nonsensical
File III	Repository-like
File IV	Strongly repository-like

The results suggest that larger training corpora produce greetings that more closely resemble repository names. Greetings generated from Files III and IV were generally longer and exhibited repository-like structures, whereas greetings generated from Files I and II were often short and nonsensical.

Limitations:

We noticed the following limitations in Experiment 2:

1. Only 10 PSFs were generated per run (Total 40 PSFs) because of manually extracting and analyzing greetings
2. The original codebase of UPGen was temporarily modified because the original authors had hardcoded a minimum corpus size of 10000
3. Human readability and perceived quality of greetings are subjective

5.3 Experiment 3

Objective:

In the third experiment, our primary objective was to analyze the PSFs generated during Experiment 1 and analyze protocol diversity. However, the results of this experiment led us to find some implementation bugs in UPGen. These are discussed in the results.

Prerequisites:

1. UPGen downloaded and set up
2. PSFs from Experiment 1

Input:

A Python script was developed to analyze the 3111 generated PSFs and is provided as an artifact.

Procedure:

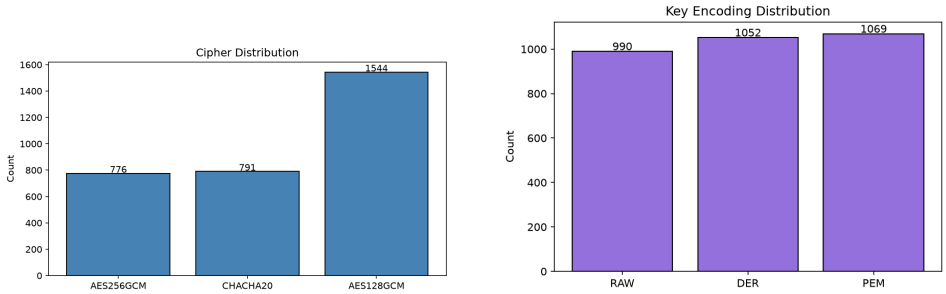
We measured some parameters of each PSF file such as cipher selection, greeting presence, handshake rounds, subprotocol rounds, key encoding, version field, nonce field presence, padding field presence, type field presence, reserved field presence, extra field presence, static key presence, header MAC presence, certificate presence, separate length field setting, payload fake size, etc. The script was then executed on the complete PSF corpus.

Results:
We observed the following results from Experiment 3:

Table 14: Uniqueness analysis of generated PSFs

Metric	**Count**	**Percentage**
Total PSFs	3111	100.0%
Exact unique	3111	100.0%
Structurally unique	3111	100.0%

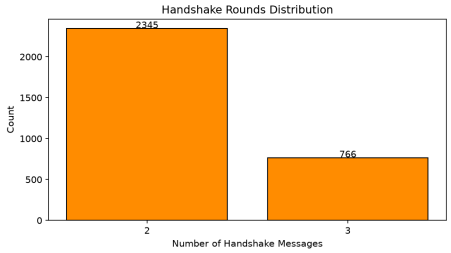
All generated PSFs were unique both at the file level and after structural normalization.



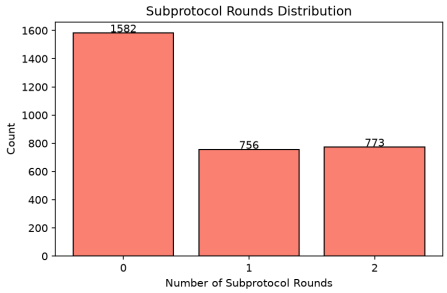
(a) Cipher distribution.

(b) Key encoding distribution.

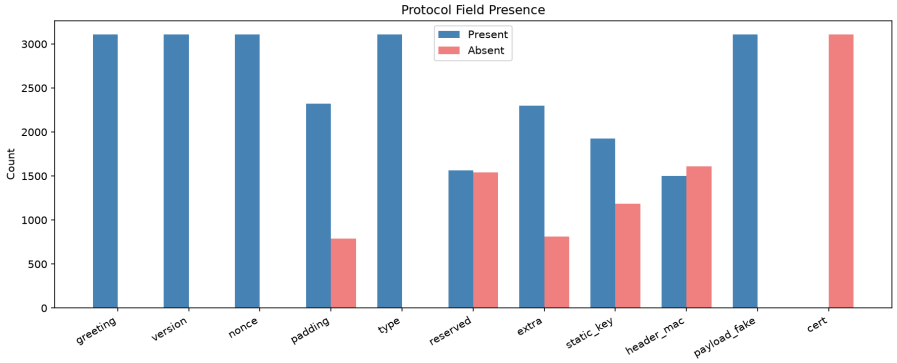
Figure 4: Distribution of cryptographic protocol parameters across generated PSFs.



(a) Handshake rounds.

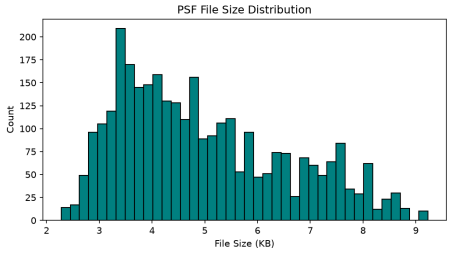


(b) Subprotocol rounds.

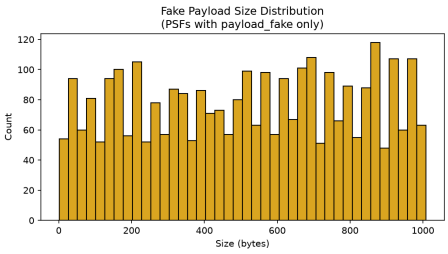


(c) Field presence across generated protocols.

Figure 5: Structural characteristics of generated PSFs.



(a) PSF file size distribution.



(b) Fake payload size distribution.

Figure 6: Size characteristics of generated protocols.

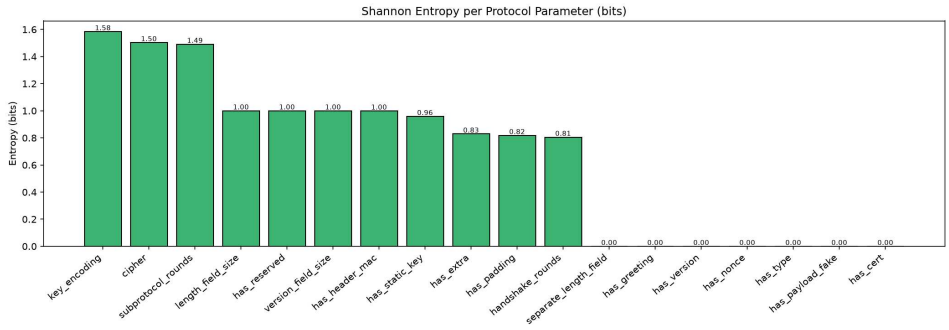


Figure 7: Shannon entropy of protocol parameters estimated from the generated corpus. Parameters with zero entropy were constant across all generated protocols.

Despite the large number of generated PSFs, no duplicate protocols were observed under either exact or structural comparison. This suggests substantial diversity in the generated protocol corpus.

5.3.1 Bug Discovery

During this experiment, we noticed some parameters exhibiting zero entropy such as — separate length field, greeting presence, version presence, nonce presence, type field presence, fake payload presence, and certificate presence. Since some of these parameters were configured with probabilities other than 0 or 1 in the configuration file, we decided to investigate the issue manually.

This led to the finding of 5 bugs in the implementation, out of which four were reproduced experimentally, while the remaining defect was verified through static code analysis. All confirmed defects originated from the same root cause — string-valued boolean parameters being used in Python truthiness checks.

In the config, parameters such as `has_nonce` were assigned a probability of 0.5 for both True and False. When generating a PSF, if False was chosen as the value, then in the code, the intended value was False, which is a Boolean value, but the actual value used in the code was “False”, which was represented as a string due to the quotation marks. Because non-empty strings evaluate to `True` in Python, both cases were interpreted as `True`. “True” (string) evaluated as `True` (boolean) and “False” (string) also evaluated as `True` in Boolean contexts. This led to some of the parameters showing only True and generating the same result in all of the 3111 PSFs. During the investigation, a bug reproduction script was developed to vali-

date the identified defects. The script and detailed reproduction results are provided as artifacts. Still, even with the bug present in some parameters, all 3111 PSFs were found to be both structurally unique and distinct.

These results suggest that UPGen can still generate unique PSFs even when some configuration parameters fail to randomize as intended.

Limitations:

We noticed the following limitations in Experiment 3:

1. The sample size was small relative to the total protocol space
2. All samples were generated with the same trained model
3. Entropy analysis identified anomalies but did not directly identify the underlying bug

5.4 Experiment 4

Objective:

In the fourth experiment, our primary objective was to reproduce the experiment performed by the authors in the original work to determine the distinguishability of UPGen-generated protocols from real-world encrypted protocols. We used Random Forest, 1D Convolutional Neural Network (1D-CNN), and Deep Fingerprinting (DF). However, due to the absence of the complete PCAP dataset in the publicly released artifacts, the full reproduction of the original work could not be done. In addition, we found a couple of anomalies resulting from differences in sample size and characteristics of the available samples which led to the opposite result of the original work due to these constraints, discussed in the results.

Prerequisites:

1. PCAPs from research artifacts of the original work (16500 available)
2. Scapy, Scikit-learn, Pandas, Numpy, Matplotlib, Seaborn, Tqdm, and Torch

Input:

In this experiment, we used a total of 16500 samples of PCAPs, provided by the original authors as artifacts on Zenodo. Out of 16500 samples, 10000 samples were of UPGen, 1000 each of CurveZMQ, secio, snow, SSH, TLS, and TLS 1.3, and 500 of obfs4. We divided the samples into three subsets — training, testing, and OOD. Out of the total 10000 UPGen samples, we used a total of 5593 samples for training and testing, of which 3995 (799×5) were used to train and 1598 (799×2) were used for in-distribution testing, while the remaining 4407 samples were reserved for OOD evaluation. For all benign protocol samples other than obfs4, we used 500 for training, and 500 for testing, and for obfs4 we used 250 for training, and 250 for testing. For

each experiment run, we excluded one of the benign protocols from training and testing, and that protocol was used for OOD along with the samples of UPGen not used for training and testing.

Procedure:

In this experiment, we first extracted the features of PCAP samples using a Python script, and in the second step we trained and tested the classifiers and measured classifier performance using a second Python script, both of which are provided in the research artifacts. First, we used Random Forest and then using CNN and Deep Fingerprinting, we ran the same experiment 7 times each, a total of 21 times.

Results:

We observed the following results from Experiment 4:

Table 15: In-distribution and out-of-distribution TPR/FPR across three classifiers and seven held-out (OOD) benign protocols

		In-distribution		OOD	
Classifier	**OOD Protocol**	**TPR**	**FPR**	**TPR**	**FPR**
Random Forest	CurveZMQ	1.00	0.00	1.00	0.00
	obfs4	1.00	0.00	1.00	0.00
	secio	1.00	0.00	1.00	0.25
	snow	1.00	0.00	1.00	0.00
	SSH	1.00	0.00	1.00	0.00
	TLS	1.00	0.00	1.00	0.00
	TLS 1.3	1.00	0.00	1.00	0.00
1D-CNN	CurveZMQ	1.00	0.00	1.00	0.00
	obfs4	1.00	0.00	1.00	0.00
	secio	1.00	0.00	1.00	0.25
	snow	1.00	0.00	1.00	0.00
	SSH	1.00	0.00	1.00	0.00
	TLS	1.00	0.00	1.00	0.00
	TLS 1.3	1.00	0.00	1.00	0.00
Deep Fingerprinting	CurveZMQ	1.00	0.00	1.00	0.00
	obfs4	1.00	0.00	1.00	0.00
	secio	1.00	0.00	1.00	0.00
	snow	1.00	0.00	1.00	0.00
	SSH	1.00	0.00	1.00	0.00
	TLS	1.00	0.00	1.00	0.00
	TLS 1.3	1.00	0.00	1.00	0.00

Contrary to the findings reported in the original study, all three classifiers achieved near-perfect classification performance on the publicly available dataset.

5.4.1 Observed Discrepancy

In the original study, the authors performed this experiment and reported near-perfect resistance to classification with classifiers unable to identify UPGen from other protocols. This led to the collateral damage argument — classifiers being unable to block UPGen protocols while blocking the benign protocols. However, in our experiment, all three classifiers — Random Forest, 1D-Convolutional Neural Network, and Deep Fingerprinting were able to successfully classify UPGen protocols, which appears to be the opposite of the original result.

However, this was not an exact reproduction as the samples were taken from the public artifact and are not all samples from the original work. As a result, this discrepancy does not contradict the original research.

The experiment results were the same across all three classifiers, suggesting that implementation errors in a particular classifier were unlikely to explain the results. We also found several packets with similar sequences. However, the sha256sum of the samples was different, so the samples were not exactly identical.

```
tls_150.pcap -> [188, 1448, 92, 93, 226, 1029, 1448, 1448,
    1448, 1448]
tls_450.pcap -> [188, 1448, 92, 93, 226, 1029, 1448, 1448,
    1448, 1448]
tls_141.pcap -> [188, 1448, 92, 93, 226, 1029, 1448, 1448,
    1448, 1448]

ssh_407.pcap -> [23, 815, 936, 48, 872, 16, 16, 44, 484, 44]
ssh_39.pcap -> [23, 815, 936, 48, 872, 16, 16, 44, 484, 44]
ssh_586.pcap -> [23, 815, 936, 48, 872, 16, 16, 44, 484, 44]

curvezmq_886.pcap -> [10, 11, 54, 53, 43, 245, 170, 298, 63,
    1042]
curvezmq_275.pcap -> [10, 11, 54, 53, 43, 245, 170, 298, 63,
    1042]
curvezmq_739.pcap -> [10, 11, 54, 53, 43, 245, 170, 298, 63,
    1042]
```

In addition, UPGen exhibited substantially higher timing values than the other protocols although this parameter was not used by classifiers to identify UPGen protocols.

Table 16: Timing characteristics of the first 30 TCP packets across protocol families. Values are reported as mean ± standard deviation over 50 traces per protocol.

Protocol	**Duration (s)**	**Mean IAT (s)**
UPGen	0.6882 ± 0.0879	0.023730 ± 0.003032
CurveZMQ	0.0046 ± 0.0003	0.000159 ± 0.000012
obfs4	0.0087 ± 0.0013	0.000300 ± 0.000043
secio	0.1475 ± 0.0408	0.005086 ± 0.001406
Snowflake	0.1684 ± 0.0044	0.005808 ± 0.000152
SSH	0.1573 ± 0.0091	0.005422 ± 0.000314
TLS	0.0072 ± 0.0005	0.000247 ± 0.000017
TLS 1.3	0.0065 ± 0.0007	0.000224 ± 0.000025

We also measured the diversity of packet traces under the exact feature representation used for classification, namely the first 30 direction-signed packet lengths. For each protocol, we counted the number of unique feature sequences present in the dataset.

Table 17: Sequence diversity under the 30-packet direction-signed length representation.

Protocol	Total Samples	Unique Sequences	Ratio
TLS	1000	3	0.003
TLS 1.3	1000	3	0.003
CurveZMQ	1000	4	0.004
SSH	1000	5	0.005
snow	1000	5	0.005
secio	1000	83	0.083
obfs4	500	500	1.000
UPGen	10000	3468	0.347

Since the number of unique sequences was very small for several benign protocols, the classifiers may have memorized these patterns. This may have allowed the classifiers to distinguish UPGen traffic from benign protocols whose feature representations exhibited very low diversity. However, this explanation remains a hypothesis and may be clarified in future work. Since we did not have access to the complete experimental environment and metadata, the observed differences from the original work are not surprising. Therefore, this result should be interpreted as a discrepancy rather than a

verified contradiction of the original study. This experiment is limited to the publicly available artifacts.

Limitations:

We noticed the following limitations in Experiment 4:

1. The available sample size was smaller than that used in the original study.
2. All samples were generated by the original authors and no control was available over their generation process or experimental environment
3. We were unable to reproduce the complete original results

6 Deployment

As part of the project, we deployed Proteus in a small VPS setup and tested its working. It is explained in detail in this section. This section also describes the steps to run the Proteus bridge both as a proxy and as a client, which may be helpful for future users.

6.1 Environment

We used two devices for the deployment. A VPS was purchased for the role of proxy. The author's laptop performed the role of the client. The detailed specifications and configuration of the proxy are presented below, while those of the client are provided in Appendix C.

(A) Hardware Configuration

1. CPU: 1 vCPU (AMD EPYC 9J14 96-Core Processor)
2. RAM: 2 GB
3. Storage: 50 GB
4. GPU: Red Hat Virtio 1.0 GPU

(B) Device Configuration

1. OS: Ubuntu 24.04.4 LTS
2. Kernel: Linux 6.8.0-124-generic
3. Network Interface: VirtIO virtual Ethernet
4. IP address: 66.116.244.138

The versions of the software used on the proxy are as follows:

1. Proteus: 0.2.0
2. Tor: 0.4.8.10
3. Rust: 1.87.0
4. Cargo: 1.87.0

The versions of the software used on the client are as follows:

1. Proteus: 0.2.0

2. Tor: 0.4.9.9-1
3. Rust: 1.87.0
4. Cargo: 1.87.0
5. Tor Browser: 15.0.17

The public IP address, bridge fingerprint, configuration files, terminal outputs, and log excerpts presented in this section correspond to the actual deployment used during our experiments. The deployment was performed on a publicly accessible VPS for the purpose of evaluating Proteus under realistic conditions. The bridge was an experimental deployment created solely for this study and was not operated as a public Tor bridge. At the time of publication, the deployment has been discontinued and the associated VPS is no longer in use. These identifiers are retained to preserve the fidelity of the deployment record and to document the exact configuration under which the reported results were obtained.

6.2 Prerequisites and Assumptions

In this subsection, we describe the prerequisites and assumptions for the proxy and client setups.

For the proxy setup, we assume that the proxy has Internet access and a publicly reachable port. The proxy needs to download Tor and Rust for setting up Proteus. We also assume that basic utilities such as `curl` and `git` are already installed. If starting from scratch, we recommend installing — `git`, `curl`, `tor`, and `rust`, assuming that basic utilities such as `sudo` and `apt` are already available. We used Ubuntu 24.04 for the proxy, so the commands given are for a Debian-based OS. Equivalent packages can be installed on other operating systems.

1. To download Tor

```
sudo apt install tor
```

2. To download Rust

```
sudo apt install -y build-essential pkg-config libssl-dev
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source "$HOME/.cargo/env"
```

Installing Rust using the curl version is preferred over the `apt` package manager because it may install a different version. Therefore, the above commands are preferred over the following commands.

```
sudo apt install rustup
rustup default stable
```

For the client setup, we assume that the client also has Internet access but does not need to be publicly reachable. The client also needs to install Tor and Rust. In addition, the client also needs to download Tor Browser, which we recommend downloading from the official Tor Project website, and we recommend verifying the checksum to ensure that there was no tampering whatsoever. We used Endeavour OS (Arch Linux) for the client-side setup, so the following commands refer to Arch-based OSes.

1. To download Tor

```
sudo pacman -S tor
```

2. To download Rust

```
sudo pacman -S base-devel pkgconf openssl rustup
rustup toolchain install 1.87
```

Only two additional prerequisites are required for setting up Proteus, and they are the same for both the proxy and the client. After downloading the prerequisites, the next step is to download Proteus, which will be explained in later parts.

We also assume that the firewall is configured correctly and that the required ports are available. If these assumptions do not hold, the deployment will fail, and they must be addressed first. In addition, we recommend creating a directory for the setup. Throughout this section, we assume the working directory is `/home/ajay/bridge` for the proxy.

The final prerequisite is a PSF that must be identical on both the proxy and the client. Ideally, the PSF should be distributed out of band, and the original authors expect Tor to distribute PSFs, but this mechanism was not available during our deployment. Therefore, our deployment differs slightly from the intended real-world deployment model. Our main objective is to test Proteus instead of distributing PSFs, so we focused on Proteus instead of PSFs. Consequently, we assume the PSF is available on both devices already.

In our deployment, the PSF is located at `/home/ajay/bridge/proteus/tests/fixtures/tls_mimic.psf` in the proxy and `/home/ajay/data/tls_mimic.psf` in the client.

6.3 Proxy setup

To run Proteus as a proxy, we first need to download Proteus. We assume the current directory is `/home/ajay/bridge`. The source code of Proteus is available in the GitHub repository[18] and can be downloaded easily. We then build Proteus using the following commands:

```
git clone https://github.com/unblockable/proteus.git
cd proteus
cargo build --release
```

This creates the `proteus` executable in `/home/ajay/bridge/proteus/target/release/` as proteus. The expected output should include `Finished‘release‘profile[optimized]target(s)in27.47s` in the end if it is successful. Verify the installation using

```
./target/release/proteus --help
```

If the help message is displayed, the installation was successful. If the repository was cloned successfully but the build fails, then it may be due to the Rust toolchain. To install Proteus properly, ensure that the prerequisites are done properly.
After Proteus has been installed, we need to add permissions to Tor and Proteus to use port 443 as it is less than 1024. First, determine the location of the Tor executable. We already know that Proteus is installed in `/home/ajay/bridge/proteus/target/release/`, so we need to find the location of Tor only. Find the location of Tor using the following command.

```
which tor
```

This command prints the location of the Tor executable, such as `/usr/bin/tor`. Then assign the required capabilities to Tor and Proteus.

```
sudo setcap cap_net_bind_service=+eip /usr/bin/tor
sudo setcap cap_net_bind_service=+eip
    /home/ajay/bridge/proteus/target/release/proteus
```

However, if the location appears to be something else like `/usr/sbin/tor` and there is an error of `Invalidfile '/usr/sbin/tor' forcapabilityoperation` while giving permission to Tor, then check if it is a symlink using the command.

```
ls -ld /usr/sbin/tor
```

If this produces the following type of result, then it verifies that it was a symlink

```
lrwxrwxrwx 1 root root 10 Apr  1  2024 /usr/sbin/tor ->
    ../bin/tor
```

Then apply the capability to the original Tor executable at `/usr/bin/tor`. The command should then succeed. To verify, use the following commands:

```
getcap /usr/bin/tor
getcap /home/ajay/bridge/proteus/target/release/proteus
```

If it is successful, the output may look like this:

```
/usr/bin/tor cap_net_bind_service=eip
/home/ajay/bridge/proteus/target/release/proteus
    cap_net_bind_service=eip
```

After doing this, it is a good idea to create a directory and probably name it `data/`, available at `/home/ajay/bridge/data`, although any name can be used technically. It will be used for putting the logs and other data afterward. We just need to create the directory but do not need to create files inside it.

```
mkdir -p /home/ajay/bridge/data
chmod 700 /home/ajay/bridge/data
```

After the folder is created, next we need to create torrc at `/home/ajay/bridge` and then write torrc, for which we give a sample as follows:
To create torrc —

```
nano /home/ajay/bridge/torrc
```

Then add the details in the following format

```
ContactInfo Ajay
Nickname Proteus

Address 66.116.244.138
OutboundBindAddress 66.116.244.138
ORPort 66.116.244.138:9001
AssumeReachable 1

ExitRelay 0
ExitPolicy reject *:*
SocksPort 0

BridgeRelay 1
PublishServerDescriptor 0
ExtORPort 0
ServerTransportPlugin proteus exec /home/ajay/bridge/proteus/
    target/release/proteus pt
ServerTransportListenAddr proteus 66.116.244.138:443
ServerTransportOptions proteus psf=/home/ajay/bridge/proteus/
    tests/fixtures/tls_mimic.psf

MaxMemInQueues 1 GB
NumCPUs 1
LogTimeGranularity 1 msec
ShutdownWaitLength 1 seconds

DataDirectory /home/ajay/bridge/data
PidFile /home/ajay/bridge/data/tor.pid
Log notice file /home/ajay/bridge/data/notice.log
```

The details are representational, and we expect proxy operators to edit the locations, IP address, and other details according to their setup, but this corresponds to what we actually put in our own torrc while deploying. After the torrc is written, we just need to start the proxy using the command `/usr/bin/tor -f /home/ajay/bridge/torrc`, and if it is successful, the terminal will print the result as follows:

```
Jun 30 05:26:39.354 [notice] Tor 0.4.8.10 running on Linux
    with Libevent 2.1.12-stable, OpenSSL 3.0.13, Zlib 1.3,
    Liblzma 5.4.5, Libzstd 1.5.5 and Glibc 2.39 as libc.
Jun 30 05:26:39.355 [notice] Tor can't help you if you use it
    wrong! Learn how to be safe at
    https://support.torproject.org/faq/staying-anonymous/
Jun 30 05:26:39.355 [notice] Read configuration file
    "/home/ajay/bridge/torrc".
Jun 30 05:26:39.355 [warn] BridgeRelay is 1, but ExitRelay is
    1 or an ExitPolicy is configured. Tor will start, but it
    will not function as an exit relay.
Jun 30 05:26:39.356 [notice] Opening OR listener on
    66.116.244.138:9001
Jun 30 05:26:39.356 [notice] Opened OR listener connection
    (ready) on 66.116.244.138:9001
```

Once the bridge is running, we need to get the fingerprint but do not stop this process on the terminal. Instead, on a new tab, write the following command, which will print the fingerprint of the bridge.

```
cat /home/ajay/bridge/data/fingerprint
Proteus 4427A74C234C93A7B83BFD853DAE84BCD1992A8F
```

The client uses this fingerprint to connect to the bridge. It confirms that the proxy is set up correctly and it is ready to be used as the bridge. To stop the proxy, either use `Ctrl+C` or use the command below:

```
pkill -f "tor -f /home/ajay/bridge/torrc"
```

This concludes the proxy setup. In addition, just using the command `/usr/bin/tor -f /home/ajay/bridge/torrc` can start the proxy again without any other step and it can be run and stopped easily. Thus, this is a one-time setup and proxy operators would not need to do anything in the future for deployment. Thereafter, a single command is sufficient to start the bridge and another to stop it. In addition, for changing the PSFs, we just need to edit one line in the torrc file containing the location of the PSF, and there is no need to change anything else. This provides flexibility and speed in changing PSFs.

6.4 Client setup

The client-side setup also starts with downloading Proteus and building it. We first clone the repository and build it in the same manner as on the proxy. We assume that Proteus is available on the client at `/home/ajay/proteus/` and executable at `/home/ajay/proteus/target/release/` available as proteus. We also assume the prerequisites are already completed, and Tor Browser and other essentials are available.
Then we find the torrc file of Tor Browser using the command

```
find ~ -iname "torrc" 2>/dev/null
```

The expected location may look like

```
/home/ajay/.local/opt/tor-browser/app/Browser/TorBrowser/Data/
    Tor/torrc
```

We then add the following line to this `torrc` file.

```
ClientTransportPlugin proteus exec
    /home/ajay/proteus/target/release/proteus pt
```

Note that this torrc file is separate from the `/etc/tor/torrc` file and it corresponds to Tor Browser, while `/etc/tor/torrc` corresponds to system Tor. Thus, identifying and editing the correct torrc file is important for setting up the client.
After this step, we just need to open Tor Browser and in `ConfigureConnection`, we manually add a bridge by specifying the bridge IP address, fingerprint (obtained from the proxy), and the location of the PSF file. In our case, we put the following details into the bridge.

```
proteus 66.116.244.138:443
    4427A74C234C93A7B83BFD853DAE84BCD1992A8F
    psf=/home/ajay/data/tls_mimic.psf
```

To verify that the bridge connection was successful, check the Tor logs in the browser, which may look like the following:

```
2026-07-01 09:45:53.657
[NOTICE] DisableNetwork is set. Tor will not make or accept
    non-control network connections. Shutting down all
    existing connections.
2026-07-01 09:45:53.662
[NOTICE] DisableNetwork is set. Tor will not make or accept
    non-control network connections. Shutting down all
    existing connections.
2026-07-01 09:45:53.666
[NOTICE] DisableNetwork is set. Tor will not make or accept
    non-control network connections. Shutting down all
    existing connections.
```

```
2026-07-01 10:04:45.683
[NOTICE] Opening Socks listener on 127.0.0.1:9150
2026-07-01 10:04:45.683
[NOTICE] Opened Socks listener connection (ready) on
    127.0.0.1:9150
2026-07-01 10:04:46.679
[WARN] Managed proxy
    "/home/ajay/proteus/target/release/proteus" wrote a LOG
    line with an invalid severity level: Notice
2026-07-01 10:04:47.034
[NOTICE] Bootstrapped 1% (conn_pt): Connecting to pluggable
    transport
2026-07-01 10:04:47.037
[WARN] Only one bridge (transport: 'proteus') is configured.
    You should have at least two for conflux, for any
    transport that is not 'snowflake'.
2026-07-01 10:04:47.037
[NOTICE] Bootstrapped 2% (conn_done_pt): Connected to
    pluggable transport
2026-07-01 10:04:47.245
[NOTICE] Bootstrapped 10% (conn_done): Connected to a relay
2026-07-01 10:04:48.062
[NOTICE] Bootstrapped 14% (handshake): Handshaking with a relay
2026-07-01 10:04:48.474
[NOTICE] Bootstrapped 15% (handshake_done): Handshake with a
    relay done
2026-07-01 10:04:48.475
[NOTICE] Bootstrapped 75% (enough_dirinfo): Loaded enough
    directory info to build circuits
2026-07-01 10:04:48.476
[NOTICE] Bootstrapped 95% (circuit_create): Establishing a Tor
    circuit
2026-07-01 10:04:51.989
[NOTICE] Bootstrapped 100% (done): Done
```

After this step, the Tor Browser will connect to the bridge, and the client setup is completed. In case it fails to connect, first check the torrc file carefully, and then the bridge details. In addition, check that the PSF file is identical and the bridge is running for a successful connection. For changing the PSFs, we just need to edit the torrc file and edit the bridge in Tor Browser, so it is very easy to run Proteus and change PSFs. This concludes the client setup.

6.5 Evidence and Observations

After the setup was completed, we used Tor Browser to connect to a variety of websites and services and observed its behaviour. We connected to a

variety of clearnet websites and onion services to evaluate how the Proteus bridge handles connections. However, the observations were related to the normal browsing experience, and we did not try to formally measure the speed and network conditions. We first verified that the Proteus bridge was connected correctly.

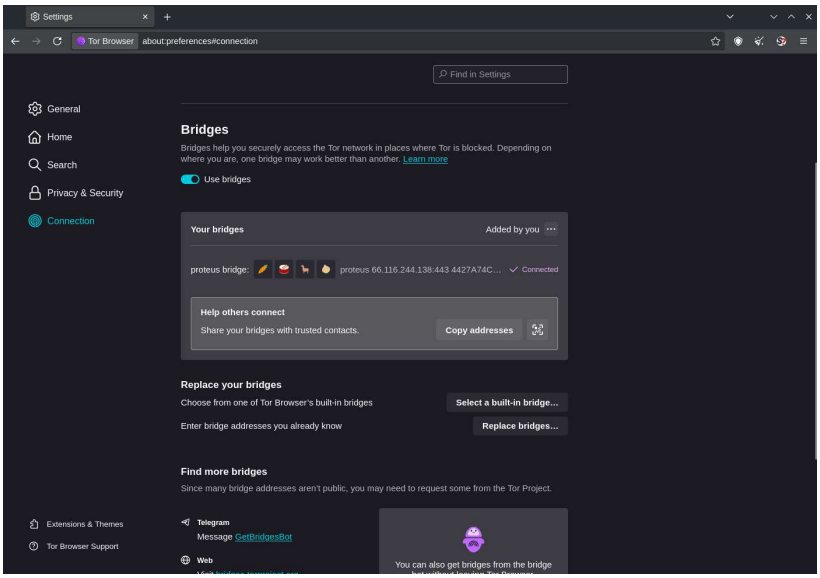


Figure 8: Proteus bridge connected in Tor Browser.

To verify that the bridge is working, go to settings in Tor Browser, and check the bridge section under the connection tab. If the setup is successful, there will be a tick mark along with Connected written on the right side of the bridge entry.

In addition, the Tor circuit should show a Proteus bridge instead of a guard relay while connecting to a service or website. An example of a Tor circuit is shown in the following figure, which is connected to the official website of the Tor Project.



Figure 9: A Tor circuit consisting of a Proteus bridge.

After the Proteus bridge was connected, the first site visited was the author's personal website `http://lakshitnhnghl6j5tjerswfnbsurh5xczmzjsanlvec3tbuifv7nvyad.onion`, which is an onion service. We chose an onion service first to check the basic connectivity and ensure a simple HTML site is working with our bridge.

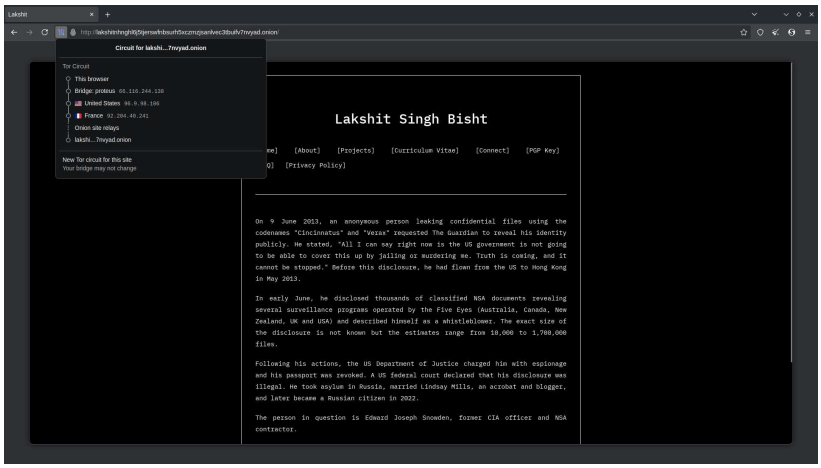


Figure 10: Visiting an onion service with Proteus.

In addition, we next visited several other websites to test the practical usability of Proteus and observed that websites loaded quickly, and we

were able to browse educational and content sites, download content, stream videos and films at 1080p, download large PC games, music, and software, and use all the tested services without any problems. We also played bullet chess (only 60 seconds per player) and were able to flag opponents successfully (winning in time) in real time, which supports the hypothesis that there was no high latency. There were no observable hindrances due to the Proteus bridge, and the connection remained responsive throughout testing. However, we decided to not test the network speed formally because there are a lot of assumptions such as VPS network, other relays' bandwidth, etc., which make the testing complex. As a result, our observations provide evidence that Proteus is deployable but do not support claims about the formal speed and throughput of Proteus.

6.6 Implications

Our deployment was completed successfully. It suggests that Proteus can be used as a Tor bridge easily; however, there is limited documentation available for setting up Proteus at the time of writing, so our main objective is to provide practical documentation for the setup of both proxy and client. However, there may be some deviations from the current version of Proteus in the future, so we may expect future documentation to cover those details in addition to the current documentation.

It should be noted that our deployment is experimental and there are several limitations in it. First, our implementation is OS-based, so the commands written are specific to what we used or followed instead of a general instruction set for all major OSes. In addition, the documentation may become stale if there are some breaking changes or if the project evolves. Thus, this documentation should be considered as helping material rather than an authoritative manual. There is also a risk of compatibility issues with the prerequisites and upstream projects our deployment is dependent upon, so if the required services have any breakage or problems, it may affect Proteus too. Second, our deployment is based on just a single proxy with a single client, and this is very different from real life where there are thousands of clients and proxies. Load balancing and scalability were not tested in our deployment and need to be evaluated in the future. Third, it is available only for PCs and not for mobile devices. Currently the Tor Browser on Android supports Snowflake, obfs4, meek, and custom bridges, but to configure Proteus on mobile devices, it will need to be bundled with Tor instead of standalone like a PC. We found that there is a workaround for using Proteus in the CLI with Termux, and we were able to connect to

our onion site with Proteus inside Termux successfully. However, we advise against relying on such hacks due to the possibility of privacy and security issues, and will suggest waiting for official integration of Proteus with the Tor Browser, and consequently, we would not document that process in the work.

In conclusion, our deployment attempt demonstrates that Proteus is not only a simple deployable tool, but also a highly customizable censorship-circumvention tool. This deployment suggests the practicality of Proteus — it is usable, reproducible, and adaptable. This provides a foundation for future experimental and real-world evaluation.

7 Comparative Analysis with obfs4 and Snowflake

In this section, UPGen and Proteus are compared against the two most widely deployed Tor pluggable transports — obfs4 and Snowflake. Rather than relying only on qualitative labels, we ground each comparison in measured results either from our own experiments or from prior published evaluations of obfs4 and Snowflake.

7.1 Comparative analysis

UPGen and Proteus rely on programmable protocol specifications and protocol generation, which is a fundamentally different strategy than the strategies used by obfs4 and Snowflake. The following table can be used to analyze the major differences between UPGen and Proteus, and obfs4 and Snowflake. Table 18 presents an architectural comparison and Table 19 compares their security and adaptability characteristics.

Table 18: Architectural comparison of censorship circumvention systems

Feature	**obfs4**	**Snowflake**	**UPGen/Proteus**
Approach	Obfuscation	Ephemeral proxies	Protocol generation
Proxy model	Static bridge	Volunteer proxies	Programmable proxies
Discovery	Bridges	Broker	Unblocked proxies
Maturity	High	High	Experimental

obfs4 and Snowflake use bridge and broker systems which are well established and mature with years of real-world deployment. In contrast, the UPGen and Proteus system is still in an experimental stage and has not yet

been established for real-world use, which makes it an interesting case for future work. In addition, the protocol generation and programmable protocol approach makes it highly customizable and usable for users in authoritarian regimes. If a PSF gets blocked, then it can be replaced by an unblocked PSF, which can then be used to connect to the Tor Network.

Table 19: Security and adaptability comparison

Feature	**obfs4**	**Snowflake**	**UPGen/Proteus**
Protocol diversity	Low (1 fixed protocol)	Low (1 fixed protocol)	Very high (large configuration space)
Resistance to heuristic detection	Low	—	Moderate–high
Resistance to classifier detection	—	Low	Low
Adaptability mechanism	None	IP rotation via volunteer pool	PSF substitution

In the original work, the authors report that only 4/1000 (0.4%) of obfs4 configurations survived OpenGFW's Fully Encrypted Traffic heuristic, versus 562/1000 (56.2%) for UPGen. As a result, obfs4 appears to have low resistance to heuristic detection, while UPGen/Proteus appears to have moderate-to-high resistance. In addition, Snowflake's WebRTC handshake has been identified with 100% accuracy against other WebRTC services[19]. Thus, it has low resistance to classifier detection. Our Experiment 4 classifiers achieved near-perfect detection of UPGen, indicating low resistance to classifier detection on the public artifact dataset.

7.1.1 Empirical Observation: Long-Term Snowflake Proxy Operation

As a real-world comparison point, we analyzed data from a Snowflake proxy[20] independently operated by the author for five months (February 2026 to July 2026 and ongoing). Throughout the observation period, the broker continued assigning clients to the proxy, and client connections remained active, resulting in approximately $51,098$ connections and 238.168 GB of relayed traffic by the time this paper was written, with a cumulative service time of

more than 3113 hours.

Table 20: Snowflake proxy deployment statistics

Metric	**Value**
Operating system	Ubuntu Linux, Arch Linux
Observation period	February–July 2026 (ongoing)
Connections served	51,098
Traffic relayed	238.168 GB
Service hours	3,113
Network environment	Consumer broadband Internet

This experiment confirms that Snowflake proxies can be practical for long-term deployment and can serve thousands of connections. Similarly, obfs4 is a reliable and proven pluggable transport. In contrast, UPGen and Proteus are still in an experimental stage. The original authors' deployment suggested that UPGen-generated protocols did not trigger blocking but did not evaluate long-term operational stability or large-scale distribution. Evaluating UPGen and Proteus over a comparable long-term deployment period will be an important next step towards practical adoption.

8 Limitations and Future Work

This section discusses the limitations of both the original UPGen and Proteus systems and the present study. It also outlines potential directions for future work.

8.1 Limitations of UPGen and Proteus

The authors identify several limitations in the original work. For example, traffic shaping, protection against an active attacker, multiplexing proxied connections and existence of unblocked proxies[2]. Traffic shaping is not currently supported because UPGen focuses on protocol generation instead of modifying traffic characteristics. Also, in case of an active attacker, UPGen silently closes the connections and may still fail after successful initial authentication. Additionally, only a single TCP connection is supported per UPGen connection for tunneling.

In addition, the original experiments were done in a controlled laboratory environment with limited experiments against the Great Firewall of China (GFW), so the behaviour of UPGen and Proteus in the real world is

not yet tested sufficiently. Second, the documentation regarding UPGen and Proteus is not extensive and, for real-world use, it needs to be thoroughly written and explained for the general public. Third, some parts in Proteus like `src/lang/ir/mod.rs`, `/src/cli/pt/mod.rs`, etc., [18] are not yet completed and need to be implemented before public use. Fourth, the public release does not include all protocol-generation parameters used in the original evaluation, making independent verification of the full protocol space difficult. Finally, the public UPGen implementation contains implementation defects that reduce the effective protocol configuration space unless corrected.

8.2 Limitations of this study

This study was done with a limited scope and does not attempt to reproduce all of the results of the original work. Only a few small-scale experiments were performed in this study. We were unable to independently reproduce the diversity estimate reported in the original UPGen paper because the parameterization used in the authors' evaluation is not fully included in the public release. As a result, some parameter ranges described in the analytical model differ from those present in the released implementation and configuration files. The intended scope was to analyze and study the architecture and design of UPGen and Proteus and not to test it against real-world censors like GFW. Also, no modifications were made to the original Proteus or UPGen codebase during this study. The lack of infrastructure and resources made rigorous testing of Proteus impractical. Due to its limited scope, the real-world effectiveness of UPGen and Proteus could not be evaluated comprehensively.

Only a very small fraction of the possible protocol space was sampled and analyzed. UPGen and Proteus were not evaluated against real-world censorship systems such as the Great Firewall of China. The classifier evaluation relied entirely on publicly released artifacts and datasets, which may differ from those used in the original evaluation. Finally, due to the limited time for experimentation, the study could not be extended for more than a month, which made a longitudinal evaluation of UPGen and Proteus impractical.

8.3 Future Work

In the original work, the authors have identified many objectives for future implementation like support for hiding distinctive patterns of tunneled

traffic, mapping proxied connections across multiple overt connections, and distribution channels for unblocked proxies[2].

In addition, the future work may focus on improved documentation and long-term testing of Proteus under real-world censors for more conclusive results. Also, the completion of the remaining modules before public adoption and the successful distribution of unblocked proxies remain important objectives. Future work should also investigate and correct the implementation defects identified in the public UPGen release. In addition, the causes of PSF validation failures observed during Experiment 1 should be analyzed in greater detail. The classifier evaluation should also be repeated using more diverse datasets and, where possible, the original experimental environment to better understand the observed discrepancy. Also, if a PSF gets blocked by some censor, then proxies need to find a way to replace the blocked PSF by one or more new unblocked PSFs, while continuing to distribute it to those regions where the censor has not blocked it. It would be interesting to evaluate how such adaptive deployment strategies can be implemented in practice.

9 Conclusion

In this paper, we analyze UPGen and Proteus for censorship evasion against a nation-state censor. Unlike obfs4 and Snowflake, UPGen and Proteus work by generating a large number of distinct encrypted protocols, which makes it difficult for a real-world censor to classify protocols, and increases the risk of collateral damage if it attempts to block all unidentified protocols. Through small-scale experiments, we observed that all generated protocols were unique. In addition, our analysis of the public implementation suggests a protocol space of approximately 4.1×10^{25} possible configurations. This suggests substantial protocol diversity in the generated protocol space.

In our experiments, we found some implementation bugs reducing the effective protocol space, and our reproduction study revealed classifiers successfully identifying UPGen protocols. This should be interpreted as an analysis of the publicly released artifacts rather than a contradiction of the original work.

Taken together, our findings suggest that UPGen and Proteus represent a novel programmable approach to censorship evasion and provide substantial protocol-level diversity. However, our evaluation of the publicly released artifacts does not reproduce the classifier resistance reported in the original work. These findings indicate that the system remains an interesting foun-

dation for future deployment-focused research and evaluation in realistic censorship settings.

Open Science

We have made the following research artifacts publicly available on the Zenodo research repository at the following URL: 10.5281/zenodo.20799019

1. Experimental results and supporting materials.
2. Trained machine learning models generated during this study.
3. Analysis scripts used for the experiments.
4. PSFs generated during the experiments.

Acknowledgments

The author would like to thank Dr. Rob Jansen (U.S. Naval Research Laboratory) for his guidance and support regarding UPGen and Proteus. The author also thanks Ryan Wails (Georgetown University) for providing the training dataset for UPGen.

References

[1] Ryan Wails, Rob Jansen, Aaron Johnson, and Micah Sherr. *Proteus: Programmable Protocols for Censorship Circumvention*. Workshop on Free and Open Communication on the Internet, 2023.

[2] Ryan Wails, Rob Jansen, Aaron Johnson, and Micah Sherr. *Censorship Evasion with Unidentified Protocol Generation*. Proceedings of the 34th USENIX Security Symposium, 2025.

[3] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3. (8446) in Request for Comments (RFC)*. Accessed: June 2026. Available publicly at: `https://datatracker.ietf.org/doc/html/rfc8446`

[4] Kevin Bock, iyouport, Anonymous, Louis-Henri Merino, David Fifield, Amir Houmansadr, and Dave Levin. *Exposing and Circumventing China's Censorship of ESNI*. Accessed: June 2026. Available publicly at: `https://gfw.report/blog/gfw_esni_blocking/en/`

[5] Diwen Xue, Michalis Kallitsis, Amir Houmansadr, and Roya Ensafi. *Fingerprinting Obfuscated Proxy Traffic with Encapsulated TLS Handshakes*. Proceedings of the 33rd USENIX Security Symposium, 2024.

[6] Ross Anderson. *Security Engineering*. Third Edition, 2020.

[7] Vesteinsson, Baker, Brody, Funk, Grothe, Slipowitz eds. *Freedom on the Net 2025*. Freedom House, 2025. Available publicly at: `https://freedomhouse.org/sites/default/files/2025-12/FOTN%202025_final_digital_120525.pdf`

[8] Freedom House: Countries and Territories. Accessed: June 2026. Available publicly at: `https://freedomhouse.org/country/scores`

[9] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. *Anonymous connections and onion routing*. IEEE Journal on Selected Areas in Communications, 1998.

[10] The Tor Project. Accessed: June 2026. Available publicly at: `https://www.torproject.org/`

[11] History of Tor. Accessed: June 2026. Available publicly at: `https://www.torproject.org/about/history/`

[12] Yawning Angel. *obfs4*. Accessed: June 2026. Available publicly at: `https://github.com/yawning/obfs4`

[13] Philipp Winter. *ScrambleSuit*. Accessed: June 2026. Available publicly at: `https://github.com/NullHypothesis/scramblesuit`

[14] Serene Han, Arlo Breault, and David Fifield. *Snowflake*. Accessed: June 2026. Available publicly at: `https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/Snowflake`

[15] Unblockable. *UPGen*. Accessed: June 2026. Available publicly at: `https://github.com/unblockable/upgen`

[16] Kevin P. Dyer, Scott E. Coull, and Thomas Shrimpton. Marionette. *A Programmable Network Traffic Obfuscation System*. USENIX Security '15. USENIX Assn, 2015.

[17] Erik Chi, Gaukas Wang, J. Alex Halderman, Eric Wustrow, and Jack Wampler. *Just add WATER: WebAssembly-based Circumvention Transports*. FOCI, 2024.

[18] Unblockable. *Proteus (v0.2.0)*. Accessed: June 2026. Available publicly at: `https://github.com/unblockable/proteus`

[19] Kyle MacMillan, Jordan Holland, and Prateek Mittal. *Evaluating Snowflake as an Indistinguishable Censorship Circumvention Tool*. arXiv, 2020.

[20] Lakshit Singh Bisht. *Snowflake*. Accessed: July, 2026. Available publicly at: `https://github.com/LakshitSinghBishtTM/snowflake`

Appendices

A Proteus Repository

The high-level structure of the Proteus repository used for experimentation and analysis can be summarized as follows:

```
proteus/
|-- src/
|   |-- cli/
|   |-- crypto/
|   |-- lang/
|   |   |-- compiler/
|   |   |-- interpreter/
|   |   |-- ir/
|   |-- net/
|   |-- common/
|-- tests/
|   |-- fixtures/
|   |-- integration/
|   |-- shadow/
|-- lib/
|-- assets/
```

B UPGen Repository

The complete structure of the UPGen repository used for experimentation and analysis is shown below:

```
upgen/
```

```
|-- .gitignore
|-- LICENSE
|-- README.md
|-- requirements.txt
|-- src/
    |-- config.json
    |-- generate.py
    |-- greeting/
    |   |-- encode.py
    |   |-- model.py
    |   |-- predict.py
    |   |-- train.py
    |-- psf.py
    |-- upgen_v1.py
    |-- util.py
```

C Hardware Configuration

1. RAM: 24 GB DDR5
2. CPU: Intel i5-13450HX
3. GPU: Nvidia RTX 4050 6GB
4. Storage: 512 GB
5. OS: Endeavour OS (Arch Linux)
6. Kernel Version: 7.0.x

D PSF Example

```
@SEGMENT.FORMATS
DEFINE greeting_server
{ NAME: greeting ; TYPE:        [ u8 ; 3 ] };
DEFINE greeting_client
{ NAME: greeting ; TYPE:        [ u8 ; 3 ] };
DEFINE handshake1
{ NAME: version ; TYPE: [ u8 ; 1 ] },
{ NAME: type ; TYPE:    [ u8 ; 1 ] },
{ NAME: nonce ; TYPE:   [ u8 ; 12 ] },
{ NAME: extra ; TYPE:   [ u8 ; 1 ] },
{ NAME: reserved ; TYPE:        [ u8 ; 1 ] },
{ NAME: ephemeral_key ; TYPE:   [ u8 ; 115 ] },
{ NAME: msg_mac ; TYPE: [ u8 ; 16 ] };
DEFINE handshake2
```

```
{ NAME: version ; TYPE: [ u8 ; 1 ] },
{ NAME: type ; TYPE:    [ u8 ; 1 ] },
{ NAME: nonce ; TYPE:   [ u8 ; 12 ] },
{ NAME: extra ; TYPE:   [ u8 ; 1 ] },
{ NAME: reserved ; TYPE:        [ u8 ; 1 ] },
{ NAME: ephemeral_key ; TYPE:   [ u8 ; 115 ] },
{ NAME: msg_mac ; TYPE: [ u8 ; 16 ] },
{ NAME: payload_fake ; TYPE:    [ u8 ; 400 ] };
DEFINE handshake_subprotocol_client_0
{ NAME: length_864 ; TYPE:      [ u8 ; 2 ] },
{ NAME: type ; TYPE:    [ u8 ; 1 ] },
{ NAME: padding_length_fake ; TYPE:     [ u8 ; 2 ] },
{ NAME: payload_fake ; TYPE:    [ u8 ; 848 ] },
{ NAME: msg_mac ; TYPE: [ u8 ; 16 ] };
DEFINE handshake_subprotocol_server_0
{ NAME: length_608 ; TYPE:      [ u8 ; 2 ] },
{ NAME: type ; TYPE:    [ u8 ; 1 ] },
{ NAME: padding_length_fake ; TYPE:     [ u8 ; 2 ] },
{ NAME: payload_fake ; TYPE:    [ u8 ; 592 ] },
{ NAME: msg_mac ; TYPE: [ u8 ; 16 ] };
DEFINE handshake_subprotocol_client_1
{ NAME: length_800 ; TYPE:      [ u8 ; 2 ] },
{ NAME: type ; TYPE:    [ u8 ; 1 ] },
{ NAME: padding_length_fake ; TYPE:     [ u8 ; 2 ] },
{ NAME: payload_fake ; TYPE:    [ u8 ; 784 ] },
{ NAME: msg_mac ; TYPE: [ u8 ; 16 ] };
DEFINE handshake_subprotocol_server_1
{ NAME: length_432 ; TYPE:      [ u8 ; 2 ] },
{ NAME: type ; TYPE:    [ u8 ; 1 ] },
{ NAME: padding_length_fake ; TYPE:     [ u8 ; 2 ] },
{ NAME: payload_fake ; TYPE:    [ u8 ; 416 ] },
{ NAME: msg_mac ; TYPE: [ u8 ; 16 ] };
DEFINE data
{ NAME: length ; TYPE:  u16 },
{ NAME: type ; TYPE:    [ u8 ; 1 ] },
{ NAME: padding_length ; TYPE:  u16 },
{ NAME: payload ; TYPE: [ u8 ; length.size_of ] },
{ NAME: padding ; TYPE: [ u8 ; padding_length.size_of ] },
{ NAME: msg_mac ; TYPE: [ u8 ; 16 ] };
@SEGMENT.SEMANTICS
{ FORMAT:       greeting_server ; FIELD:        greeting ;
   SEMANTIC:    FIXED_STRING ("ava") } ;
{ FORMAT:       greeting_client ; FIELD:        greeting ;
   SEMANTIC:    FIXED_STRING ("ava") } ;
{ FORMAT:       handshake1 ; FIELD:     version ;
   SEMANTIC:     FIXED_BYTES (0x01) } ;
{ FORMAT:       handshake1 ; FIELD:     type ;
   SEMANTIC:        FIXED_BYTES (0x00) } ;
{ FORMAT:       handshake1 ; FIELD:     nonce ;
```

```
    SEMANTIC:       RANDOM (12) } ;
{ FORMAT:       handshake1 ; FIELD:     extra ;
    SEMANTIC:       RANDOM (1) } ;
{ FORMAT:       handshake1 ; FIELD:     reserved ;
    SEMANTIC:    FIXED_BYTES (0x00) } ;
{ FORMAT:       handshake1 ; FIELD:     ephemeral_key ;
    SEMANTIC:       PUBKEY (PEM) } ;
{ FORMAT:       handshake1 ; FIELD:     msg_mac ;
    SEMANTIC:     RANDOM (16) } ;
{ FORMAT:       handshake2 ; FIELD:     version ;
    SEMANTIC:     FIXED_BYTES (0x01) } ;
{ FORMAT:       handshake2 ; FIELD:     type ;
    SEMANTIC:        FIXED_BYTES (0x00) } ;
{ FORMAT:       handshake2 ; FIELD:     nonce ;
    SEMANTIC:       RANDOM (12) } ;
{ FORMAT:       handshake2 ; FIELD:     extra ;
    SEMANTIC:       RANDOM (1) } ;
{ FORMAT:       handshake2 ; FIELD:     reserved ;
    SEMANTIC:    FIXED_BYTES (0x00) } ;
{ FORMAT:       handshake2 ; FIELD:     ephemeral_key ;
    SEMANTIC:       PUBKEY (PEM) } ;
{ FORMAT:       handshake2 ; FIELD:     msg_mac ;
    SEMANTIC:     RANDOM (16) } ;
{ FORMAT:       handshake2 ; FIELD:     payload_fake ;
    SEMANTIC:        RANDOM (400) } ;
{ FORMAT:       handshake_subprotocol_client_0 ; FIELD:
    length_864 ; SEMANTIC:  FIXED_BYTES (0x0360) } ;
{ FORMAT:       handshake_subprotocol_client_0 ; FIELD: type ;
    SEMANTIC:        FIXED_BYTES (0x00) } ;
{ FORMAT:       handshake_subprotocol_client_0 ; FIELD:
    padding_length_fake ; SEMANTIC: RANDOM (2) } ;
{ FORMAT:       handshake_subprotocol_client_0 ; FIELD:
    payload_fake ; SEMANTIC:        RANDOM (848) } ;
{ FORMAT:       handshake_subprotocol_client_0 ; FIELD:
    msg_mac ; SEMANTIC:     RANDOM (16) } ;
{ FORMAT:       handshake_subprotocol_server_0 ; FIELD:
    length_608 ; SEMANTIC:  FIXED_BYTES (0x0260) } ;
{ FORMAT:       handshake_subprotocol_server_0 ; FIELD: type ;
    SEMANTIC:        FIXED_BYTES (0x00) } ;
{ FORMAT:       handshake_subprotocol_server_0 ; FIELD:
    padding_length_fake ; SEMANTIC: RANDOM (2) } ;
{ FORMAT:       handshake_subprotocol_server_0 ; FIELD:
    payload_fake ; SEMANTIC:        RANDOM (592) } ;
{ FORMAT:       handshake_subprotocol_server_0 ; FIELD:
    msg_mac ; SEMANTIC:     RANDOM (16) } ;
{ FORMAT:       handshake_subprotocol_client_1 ; FIELD:
    length_800 ; SEMANTIC:  FIXED_BYTES (0x0320) } ;
{ FORMAT:       handshake_subprotocol_client_1 ; FIELD: type ;
    SEMANTIC:        FIXED_BYTES (0x00) } ;
```

```
{ FORMAT:       handshake_subprotocol_client_1 ; FIELD:
    padding_length_fake ; SEMANTIC: RANDOM (2) } ;
{ FORMAT:       handshake_subprotocol_client_1 ; FIELD:
    payload_fake ; SEMANTIC:        RANDOM (784) } ;
{ FORMAT:       handshake_subprotocol_client_1 ; FIELD:
    msg_mac ; SEMANTIC:     RANDOM (16) } ;
{ FORMAT:       handshake_subprotocol_server_1 ; FIELD:
    length_432 ; SEMANTIC:  FIXED_BYTES (0x01b0) } ;
{ FORMAT:       handshake_subprotocol_server_1 ; FIELD: type ;
    SEMANTIC:         FIXED_BYTES (0x00) } ;
{ FORMAT:       handshake_subprotocol_server_1 ; FIELD:
    padding_length_fake ; SEMANTIC: RANDOM (2) } ;
{ FORMAT:       handshake_subprotocol_server_1 ; FIELD:
    payload_fake ; SEMANTIC:        RANDOM (416) } ;
{ FORMAT:       handshake_subprotocol_server_1 ; FIELD:
    msg_mac ; SEMANTIC:     RANDOM (16) } ;
{ FORMAT:       data ; FIELD:   length ; SEMANTIC:      LENGTH
    } ;
{ FORMAT:       data ; FIELD:   type ; SEMANTIC:
    FIXED_BYTES (0x01) } ;
{ FORMAT:       data ; FIELD:   padding_length ;
    SEMANTIC:      PADDING_LENGTH } ;
{ FORMAT:       data ; FIELD:   payload ; SEMANTIC:
    PAYLOAD } ;
{ FORMAT:       data ; FIELD:   padding ; SEMANTIC:
    PADDING } ;
{ FORMAT:       data ; FIELD:   msg_mac ; SEMANTIC:     RANDOM
    (16) } ;
@SEGMENT.SEQUENCE
{ ROLE: SERVER ; PHASE: HANDSHAKE ; FORMAT:
    greeting_server } ;
{ ROLE: CLIENT ; PHASE: HANDSHAKE ; FORMAT:
    greeting_client } ;
{ ROLE: CLIENT ; PHASE: HANDSHAKE ; FORMAT:     handshake1 } ;
{ ROLE: SERVER ; PHASE: HANDSHAKE ; FORMAT:     handshake2 } ;
{ ROLE: CLIENT ; PHASE: HANDSHAKE ; FORMAT:
    handshake_subprotocol_client_0 } ;
{ ROLE: SERVER ; PHASE: HANDSHAKE ; FORMAT:
    handshake_subprotocol_server_0 } ;
{ ROLE: CLIENT ; PHASE: HANDSHAKE ; FORMAT:
    handshake_subprotocol_client_1 } ;
{ ROLE: SERVER ; PHASE: HANDSHAKE ; FORMAT:
    handshake_subprotocol_server_1 } ;
{ ROLE: CLIENT ; PHASE: DATA ; FORMAT:  data } ;
{ ROLE: SERVER ; PHASE: DATA ; FORMAT:  data } ;
@SEGMENT.CRYPTO
PASSWORD = "659edb08-1d3c-4658-a32c-814350c5167f" ;
CIPHER = AES128GCM ;
ENCRYPT handshake1 FROM handshake1
```

```
{ PTEXT : extra ; CTEXT : extra ; MAC : NULL },
{ PTEXT : reserved ; CTEXT : reserved ; MAC : NULL } ;
ENCRYPT handshake2 FROM handshake2
{ PTEXT : extra ; CTEXT : extra ; MAC : NULL },
{ PTEXT : reserved ; CTEXT : reserved ; MAC : NULL } ;
ENCRYPT data FROM data
{ PTEXT : padding_length ; CTEXT : padding_length ; MAC : NULL
    },
{ PTEXT : payload ; CTEXT : payload ; MAC : NULL },
{ PTEXT : padding ; CTEXT : padding ; MAC : NULL } ;
@SEGMENT.OPTIONS
SEPARATE_LENGTH_FIELD = true ;
```