Free and Open
COMMUNICATIONS
https://foci.community on the Internet

Proteus: Programmable Protocols for Censorship Circumvention

Ryan Wails
U.S. Naval Research Laboratory
Georgetown University

Rob Jansen
U.S. Naval Research Laboratory

Aaron Johnson
U.S. Naval Research Laboratory

Micah Sherr
Georgetown University

Free and Open Communications on the Internet 2023(1), 50-66

Proteus: Programmable Protocols for Censorship Circumvention

Ryan Wails[1,2], Rob Jansen[1], Aaron Johnson[1], and Micah Sherr[2]

[1]U.S. Naval Research Laboratory
[2]Georgetown University

Abstract

We present the Proteus system for censorship circumvention. Proteus provides a programmable protocol environment in which new communication protocols can be expressed as concise and comprehensible specification files. This design allows clients and proxies to quickly respond to new censorship strategies just by installing new specification files. Proteus improves on prior programmable designs by improving host safety from malicious specifications, providing a specification language that is complete and comprehensible to non-specialists, and supporting multiple simultaneous protocols at a proxy for versioning and localization. This paper represents work in progress and provides an overview of the Proteus design, as well as examples showing that it can express existing encrypted protocols.

1 Introduction

Internet censorship is an increasingly common tool of political and social control. Consequently, anti-censorship communities have developed tools to circumvent censorship. One popular design for those tools is to relay traffic through proxies using an encrypted protocol [11, 15, 16, 21, 25, 26, 29]. However, if the censor can identify when connections are being proxied, they can block the use of those designs. Some proxy systems can be identified at the protocol level, that is, using an identifiable feature of the protocol messages, such as a header or a byte pattern [1, 24, 27, 28].

In response, Dyer et al. [6] proposed a *programmable* system for communication protocols. Their system, Marionette, provides a language and tools that make it easy to write and install new protocols at the client and proxy. This design allows new censorship methods to be quickly evaded by reconfiguring the proxy protocol. A variety of proxy protocols can be used by different proxies, making comprehensive censorship difficult to implement.

A programmable proxy system by itself does not provide a strategy to avoid detection by a censor—it only enables strategies to be quickly implemented and deployed. Censors may install blocking rules for deployed protocols, prompting the development of new protocols by evaders; this cycle is the so-called arms race interaction of censors and evaders. Evaders have some advantages in this race. As the initiators of connections, they are in a position to test and measure rules being applied by a censor, but conversely the censor cannot easily induce a potential evader to make proxied connections. Also, the population of network users is typically large and diverse relative to the authorities and professionals designing and enforcing the censorship regime. It is typically much easier to target evasion of a relatively small set of blocking rules than it is for a censor to block a potentially large variety of circumvention strategies.

Two case studies demonstrate the usefulness of programmable protocol systems. Bock et al. [3] measured protocol filtering being applied in Iran and identify a set of rules to recognize the allowed protocols (namely, DNS, HTTP, and HTTPS). Once such rules are discovered, a programmable circumvention tool could simply distribute updated protocol specifications containing any of the allowed fingerprints. In China, measurement studies have revealed targeted blocking of Shadowsocks [2, 27], which also affects other fully encrypted protocols such as obfs4 [29] and VMess [23]. The studies reveal a protocol blocklist being applied to connections to certain destinations outside the country. The inferred rules are simple, and a programmable design would allow circumvention systems like Shadowsocks to quickly distribute protocol modifications to evade them.

Despite its potential benefits, there exist obstacles to using the Marionette system in practice. First, Marionette poses a safety risk to clients and proxies. It executes *user-specified* plugin code in a generic Python runtime environment, making its hosts vulnerable to a malicious protocol distributor that crafts the protocol files to exploit vulnerabilities or abuse privileges of the runtime. Even non-malicious protocol implementations may contains bugs that present a risk to the host machines. Accepting such a threat would give distributed proxy networks, such as the Tor network [4], a single point of failure. Second, Marionette is not expressed in a self-contained language that is both available for use today and is accessible to developers and activists. Its custom specification language is defined only implicitly by the implementation of its interpreter, and the parsing and packaging of communications data must be implemented by plugins written in a standard programming language. Third, Marionette does not support multiple protocols and version upgrades. While new protocols can be developed to respond to changes in censorship rules, clients and proxies have to synchronously upgrade to the new protocols.

To address these weaknesses, we present the *Proteus* system. Proteus ensures safety by specifying a limited runtime system that prevents the protocol specification files from being maliciously used to exploit proxies or clients. Proteus also provides a comprehensible specification of the language for its protocol specification files. They are designed to be usable by ordinary programmers, and their message formatting component, which defines the format of individual protocol messages, requires little programming background to configure. Finally, we describe how multiple protocols can be simultaneously supported by a single Proteus proxy. As special cases of this, (1) protocol versioning can be used to respond to new censorship rules while still supporting existing clients, and (2) proxies can support clients in different locations with different strategies to evade their censors.

For a client and proxy to use Proteus to circumvent censorship, they must both be configured with the same specification files, and those files must specify a protocol that evades the techniques being applied by their censor. We do net expect the specification files to be designed by individual users. Instead, we expect that domain experts, such as the Tor Project, or activists, such as the Shadowsocks developers, will develop and distribute those files to their communities.

This paper describes work in progress on Proteus. We provide high-level descriptions of the runtime environment, a grammar for our programming language, and example protocol specifications that implement mocked versions of two existing encrypted protocols (namely, Shadowsocks and a Noise protocol [12]). Work is ongoing to fully implement Proteus and test it in target network environments. The working code repository for Proteus can be found at the following link: `https://github.com/unblockable/proteus`.

2 The Proteus System

The Proteus system is intended to enable fast reaction to a changing censorship environment. Its key design goals are (1) to enable pairwise communication, (2) to provide protocol programmability, (3) to provide safety from malicious protocol updates, and (4) to allow for graceful updates.

The basic functionality requirement is bidirectional communication between two parties. A particular focus is on enabling secure protocols that use cryptography to provide message confidentiality and integrity. While unencrypted protocols can be implemented, Proteus's library functions and parsing support are designed to facilitate cryptographic functionality, such as encryption, key exchange, and signatures.

Proteus communication protocols are programmable to allow its users to quickly adjust to changes in censorship rules and techniques. Proteus supports a wide range of different protocol state machines, message formats, and cryptographic primitives, which are commonly targets of censorship rules. Changing a protocol can easily be accomplished by updating a concise specification file which is written in a language that is designed to be familiar to programmers.

Proteus is designed to provide safety to its users by limiting the power of its execution environment thereby reducing the risk of protocol updates (relative to updating entire protocol executables). The execution environment can only interact with host operating systems through a limited set of system calls. Also, there is a limit on the memory consumed during protocol execution. Finally, the protocol specifications are expressed in a high-level language that enables inspection by the users before being installed.

2.1 Design

Proteus is designed to be used in a client-server setting. The client and proxy server communicate using a Proteus protocol designed to evade network censorship. The client is defined to be the party that initiates the connection, and the server must be running and waiting for connection attempts. Each side must possess the same Protocol Specification File (PSF) that provides the protocol specification. That PSF must be produced and distributed out-of-band, and in the setting of an adversarial censor, the PSF may need to be kept secret from the censor (for example, when specifying some distinctive but otherwise unknown protocol).

Proteus supports versioning and localization at the server. That is, the server may hold multiple PSFs and simultaneously support their multiple protocols. This feature allows the server to upgrade its protocol while remaining accessible to clients running previous protocol versions, as well as support protocols suitable for clients located in different censorship regimes. However, the method Proteus uses to choose the correct protocol requires that the supported protocols must have mutually compatible specifications to guarantee the server makes a correct protocol choice.

Multiple key setup assumptions can be used to facilitate secure communication. Keys can be provided as inputs at startup in addition to the PSFs, and then they can be used by the protocol. For example, a pre-shared symmetric key or a server public key can be provided as input by both sides to be used for encryption and authentication. Such keys must be distributed out-of-band, just as with the PSFs. Other keys may be negotiated during the protocol itself, such as ephemeral public keys or session symmetric keys, and the construction and use of those keys is specified directly in a PSF.

The system assumes that TCP is used as the underlying transport. Message delivery is assumed to be reliable and in-order. There is a notion of a connection between a pair of hosts, and it is opened by the client but may be closed by either side. The network stack may fragment messages, which should be tolerated by the protocol being used.

2.2 Abstract Model

We highlight the essential parts of the Proteus system using an *abstract model*. The Proteus abstract model consists of two

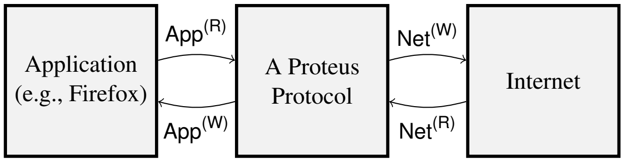


Figure 1: Relationship of the read and write buffers to a Proteus protocol. A protocol takes in data through the read buffers, and outputs data through the write buffers.

components: (1) a fixed-size execution environment *Env*, and (2) a protocol P to run inside of the execution environment. Protocol actions will be triggered from events defined by a set of possible events E. Each Proteus connection is handled by an independent pair of protocol instances (one instance for the client and one for the server).

The fixed-size execution environment $Env = (N, B)$ is an ordered pair determining the total state of a protocol execution: N is a positive integer that determines the size of the protocol's global state in bytes, and B is a positive integer that determines the buffer size limit in bytes.

These parameters define the global protocol memory $G = \{0, 1, \ldots, 255\}^N$ and four bounded buffers with which the protocol interacts: application read-only and write-only buffers $\mathsf{App}^{(\mathsf{R})}$ and $\mathsf{App}^{(\mathsf{W})}$, and network read-only and write-only buffers $\mathsf{Net}^{(\mathsf{R})}$ and $\mathsf{Net}^{(\mathsf{W})}$, each consisting of B bytes. The relationship of these buffers to the protocol is shown in Fig. 1.

Protocol $P = (F, \delta)$ is an ordered pair parameterized by *Env*. F is a finite set of functions $F_1, \ldots, F_k$. Each function takes as input the memory and buffer state and outputs new state, i.e., $F_i : \{0, 1, \ldots, 255\}^{N+4B} \to \{0, 1, \ldots, 255\}^{N+4B}$. Each function is a fixed-sized boolean circuit. $\delta : E \to F$ is a dispatch function that maps each event to an event handling function.

Proteus protocols are *event driven*, which is a common programming paradigm for message passing and network protocols. Events are generated and enqueued as application and network transmissions occur. Events are processed in a loop where each event invokes an event-handling function F_i determined by δ. The event-handling loop is shown in Alg. 1.

Events are assumed to occur atomically and may be generated concurrently as the protocol is executed (e.g., an implementation of the Proteus runtime could run Alg. 1 in one thread of execution and monitor for events in another thread). The set of possible events E is given in Table 1. The most common events are EV-APP and EV-NET, which occur when new data is made available by the application or communicating party. Other events are used to handle connection initialization, termination, and errors.

2.3 Implementation

The abstract model is useful for understanding how Proteus protocols work, but does not describe how these protocols are specified or the details of the protocol runtime environment. Here we describe the *Proteus language* that is used to define the set of event handling functions $F_1, \ldots, F_k$ described in the abstract model, which fully specifies a protocol. These function definitions are stored in a single source code file, the protocol specification file (PSF). In order for the language to be both simple and safe, we intentionally limited its capabilities. For example, Proteus programs have no way of dynamically managing memory. To enable complex functionalities necessary for transport protocols, such as encryption, a *standard library* of functions is provided for programs to use. Because transport protocols heavily involve message serialization and parsing, Proteus has facilities and standard library functions to simplify message formatting.

Algorithm 1 Main event-handling loop for Proteus protocols.

▷ Initialization:

$G \leftarrow 0^N$
$\mathsf{App}^{(\mathsf{R})} \leftarrow 0^B$
$\mathsf{Net}^{(\mathsf{R})} \leftarrow 0^B$
$\mathsf{App}^{(\mathsf{W})} \leftarrow 0^B$
$\mathsf{Net}^{(\mathsf{W})} \leftarrow 0^B$
EV-INIT is placed on the event queue.

▷ Event Processing:

repeat
    The next event e is popped from the event queue. Execution is paused if an event is not yet available.
    The event handler function is obtained: $f \leftarrow \delta(e)$.
    Let S be shorthand notation for the state of the execution, $S \equiv (G, \mathsf{App}^{(\mathsf{R})}, \mathsf{Net}^{(\mathsf{R})}, \mathsf{App}^{(\mathsf{W})}, \mathsf{Net}^{(\mathsf{W})})$. The event handler for e is invoked and state is updated: $S \leftarrow f(S)$.
    Data added to the application write-only buffer $\mathsf{App}^{(\mathsf{W})}$ is written to the application. The buffer is reset: $\mathsf{App}^{(\mathsf{W})} \leftarrow 0^B$. The same action is then applied to the network write-only buffer.
until e = EV-TERM

2.3.1 Proteus Language

Proteus protocols are expressed in a PSF consisting of (1) protocol message definitions (described further in § 2.3.3), (2) global state variables, and (3) event handling functions. This layout is depicted in Fig. 2. We define a custom language which is used to write Proteus protocols. The syntax of the language is designed to be familiar to Rust programmers and the language has typical low-level language semantics. A parsing expression grammar recognizing the language is given in Appendix C. The language is designed to be simple, minimal, easily edited, and interpreted at runtime. A variety of standard programming language constructs are supported, including: variable declaration and assignment; basic logical and arithmetic operations; branching execution with `if` and `match` statements; type casting; standard library function invocation; and repeated evaluation with statically-bounded `for` loops. The language is statically typed and statically

Table 1: Description of events defining the event set E.

Event	Description
EV-INIT	The initialization event will always occur exactly once at the very beginning of every protocol execution.
EV-APP	New data was written from the application into the application read buffer $\mathsf{App}^{(R)}$.
EV-NET	New data was written from the network into the network read buffer $\mathsf{Net}^{(R)}$.
EV-TIMER	A timer expired.
EV-SIGQUIT	The execution process received a quit or kill signal.
EV-PANIC	The execution process encountered an unrecoverable error, such as an out-of-memory error.
EV-APP-CLOSE	The application closed its side of the connection.
EV-NET-CLOSE	The network closed its side of the connection.
EV-TERM	The final termination event. This event occurs exactly once at the very end of a protocol execution. It fires (1) immediately following EV-SIGQUIT, (2) immediately following EV-PANIC, or (3) after both the application and network connections are closed.

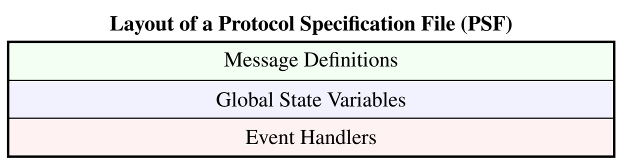


Figure 2: Schematic overview of PSF that consists of: (1) protocol message formats, (2) global variables used by event handlers; and (3) event handling functions.

allocated, with simple function-level lexical scoping and lifetimes (except for global variables, which have global scope and static lifetime). Listing 1 shows a simple example of code written in the Proteus language.

We intentionally limit the Proteus language to include only a small number of basis functionalities in order to promote *safety* of the language and execution environment. Specifically, we exclude: dynamic memory allocation; function or class declaration; template and macro metaprogrammming[1]; exceptions or exception handling; arbitrary system calls; infinite or dynamic loops; jumps; floating point arithmetic; pointer arithmetic; concurrency; and explicit memory dereferencing. Proteus programs cannot consume more than a fixed amount of memory, and procedure execution times may be measured before the procedures are invoked. Unsafe memory operations are disallowed to prevent this common source of programming errors. Many of these choices coincide with common standards for writing safety-critical code [9].

To limit host-machine access, only the narrow set of necessary system calls is allowed by the runtime. These trusted system calls related to network communication are made available through the standard library of functions available to Proteus programs, which we describe next.

[1] We do allow a limited number of trusted standard library functions to be defined with template types and macros to improve code concision.

```
let n: u16 = 0;

for n in 1..=100 {
  if n % 15 == 0 {
    log("fizzbuzz");
  } else if n % 3 == 0 {
    log("fizz");
  } else if n % 5 == 0 {
    log("buzz");
  }
}
```

Listing 1: A simple example showing the "fizzbuzz" program implemented in the Proteus language. The syntax closely follows that of the Rust language.

2.3.2 Standard Library

Because Proteus programs are fairly limited in what they can express, a *standard library* is defined to provide common and required functionalities for communication protocols. Standard library details and functions are further described in Appendix B. Categories of functions include:

I/O Related: These functions are used to manipulate the communication buffers. Functions include `buffer_length()`, `buffer_peek()`, `buffer_pop()`, `buffer_push()`, `buffer_close()`, and `buffer_close_all()`.

Utility: Utility functions are also provided for operations such as getting the value of an environment variable or setting a timer. Functions include `getenv()` (which retrieves the value of an environment variable), `log()`, `arm_timer()`, `disarm_timer()`, `get_timer()`, and `get_random_bytes()`.

Message Formatting: Special functions are provided to format and parse protocol messages. These functions are described further in § 2.3.3.

Cryptographic: A number of cryptographic facilities must be provided to support common operations, such as encryption and message authentication. We assume a standard set of functionalities in the standard library, such as those provided by the RustCrypto packages [20].

2.3.3 Message Formatting

Message formatting constitutes an central part of the Proteus language. The Proteus language includes *message definition* functionality, where the layout and binary encoding of protocol messages can be defined. The syntax for protocol message formats is contained in the Proteus language grammar grammar (Appendix C). An example of a message format specification is shown in Listing 2. This example describes a protocol message called `EncryptedMessageFormat` with 3 fields: (1) `PayloadSize`, (2) `EncryptedPayload`, and (3) `MACTag`. The order of enumeration in the format specifier defines the order that these fields appear in the serialized message. Each field

```
DEFINE EncryptedMessageFormat
{ NAME: PayloadSize; TYPE: u16 },
{ NAME: EncryptedPayload;
  TYPE: [u8; PayloadSize.value] },
{ NAME: MACTag; TYPE: [u8; 16] };
```

Listing 2: Example protocol message definition with 3 fields—`PayloadSize`, `EncryptedPayload`, and `MACTag`—that are serialized by their order of declaration and types.

has a corresponding `type` parameter, which determines how the field is represented in binary format. *Arrays* with type t and length ℓ are denoted $[t;\ \ell]$. Array lengths may be concretely defined (e.g., 2 elements), or defined using a simple unambiguous expression (e.g., for the `EncryptedPayload` field, the size is set to `PayloadSize.value`, which indicates that the `PayloadSize` field stores the length of the field. An example of message formatting and parsing is shown below:

```
// Dummy values. In practice, the payload and MAC tag
// would be set by an encryption function.
let payload: [u16; 30] = [0; 30];
let mac: [u16; 16] = [0; 16];
let payload_size: u16 = 30;

// Sets the value of each field.
let fields: Fields = create_fields();
set_field(&fields, "PayloadSize", payload_size);
set_field(&fields, "EncryptedPayload", payload);
set_field(&fields, "MACTag", mac);

// The fields can then be serialized according to
// the EncryptedMessageFormat into a byte string.
let (success, b): (bool, Bytes) =
  format(&EncryptedMessageFormat, &fields);

// This invocation parses byte string b
// following the EncryptedPayloadSpec format and
// stores the result in f2.
let (_, f2): (_, Fields)
  = parse(&EncryptedMessageFormat, &b);

// Then fields can be accessed:
let x: u16 = 0;
get_field(&f2, "PayloadSize", &x);
```

Message formatting and parsing is designed to be easy and flexible. For example, in a new protocol version, a message format could be extended simply by adding new lines specifying fields' names and types.

2.4 Versioning

Version upgrading and *localization* are important aspects of circumvention protocol design that are often overlooked. Proteus enables graceful protocol upgrades and does not require all clients and servers to update PSFs in lockstep. Instead, servers can be simultaneously provisioned with multiple protocol versions; multiple PSFs may be executed independently and in parallel using a view a single set of read buffers. State

Time:	t_0	t_1	t_2
Events:	Receive bytes b_1	Receive bytes b_2	
Protocols	↓	↓	
P_1 :	`parse(`m_1, b_1`)` $\xrightarrow{✗}$	$\perp$	
P_2 :	`parse(`m_2, b_1`)` $\xrightarrow{✓}$	`parse(`m_4, b_2`)` $\xrightarrow{✗}$	$\perp$
P_3 :	`parse(`m_3, b_1`)` $\xrightarrow{✓}$	`parse(`m_5, b_2`)` $\xrightarrow{✓}$	`send(data)`

Figure 3: Diagram of protocols P_1–P_3 simultaneously parsing incoming bytes until only one protocol P_3 remains, indicating that P_3 was the correct protocol version. ✓ denotes when a message was successfully parsed, ✗ denotes when a parsed failed, and $\perp$ denotes protocol termination.

is independently maintained for each of the running protocols. This process continues until all-but-one of the protocols have quit, or until a protocol tries to modify any one of the buffers. In the case when all-but-one have quit, the remaining protocol is determined to be the selected protocol version and continues to run. If one of the protocols modifies buffer state, then this protocol is chosen as the correct version and all other running protocol instances are immediately terminated.

This process is shown in Fig. 3. In this example, 3 protocols—P_1, P_2, and P_3—are executed, each of which is configured with a separate set of message format definitions. Two events occur which correspond to receiving bytes from a client. The client (not depicted) is using protocol P_3. Each protocol uses a different series of message formats m_i when parsing the messages. In the shown example, protocol P_1 tries parsing the first string of received bytes b_1 with an incompatible message format m_1 and quits upon failure (a parse failure can occur if, for example, a field does not contain an expected value). For protocols P_2 and P_3, both m_2 and m_3 were compatible message formats for the first received byte string, so execution proceeds. When b_2 arrives, P_2 encounters a parsing error using format m_4 and quits, whereas P_3's parsing with format m_5 is successful. At time t_2, P_3 is the only protocol version running and is the version selected to communicate with the client.

For the Proteus versioning scheme to work as intended, Proteus protocol versions should be unambiguously determined by a client's messages before the server is required to respond. Many transport protocols transmit a version number in the first message, which is accordant with our design.

2.5 Design Capabilities

The Proteus system contains the low-level building blocks necessary to realize high-level protocol capabilities. We now describe some of the capabilities that are commonly found in real-world protocols and that can be achieved in Proteus.

Message Format: A protocol message is typically composed of multiple fields that contain important information to assist the receiver in parsing the message and to communicate protocol state. For example, a `length` field is often used to communicate the total length of the message. Additional information is commonly communicated in distinct message fields, such as the message type, the protocol version, a human-readable protocol greeting string, binary flags, cryptographic counters or nonces, reserved (unused) or padding bytes, message authentication codes, and application data. We are capable of expressing any number of such fields and of specifying the order in which the fields should occur within a given message by writing PSFs in the Proteus language.

Protocol Behavior: Network protocols are commonly separated into multiple protocol phases, and our language allows us to express multiple of such phases. During a *handshake* phase, specific message types are sent between the communicating parties to, for example, negotiate protocol versions, negotiate ciphersuites, and exchange cryptographic key material. The handshake phase may encompass several messages in multiple rounds of communication. Our standard library enables us to express precisely how data communicated during the handshake phase should be processed, e.g., to enable encryption. During a *data* phase, the primary focus is sending application data, possibly using an encryption method established during the handshake and possibly sending diagnostics in parallel. Finally, during a *shutdown* phase, protocols can close a connection by sending an error message or performing other termination procedures. Proteus allows us to express the logic for establishing such protocol phases.

Cryptographic Behavior: Encrypted protocols contains logic for establishing a secure communication channel. Cryptographic logic can be quite complex; for example, a ciphersuite commonly involves algorithms for key exchange, encryption, and message authentication. We support cryptographic logic through a standard library of functions, including cryptographic functions such as those supported in RustCrypto [20]. For example, Proteus allows us to express a key exchange procedure using ECDH in the Curve25519 group with the SHA256 hash function, or that encryption should be performed with a ChaCha20 stream cipher with a Poly1305 authentication tag. Functions that require auxiliary data, such as key material when constructing an ephemeral DH key, can obtain it from a peer using messages exchanged during a handshake phase as previously described.

2.6 Design Limitations

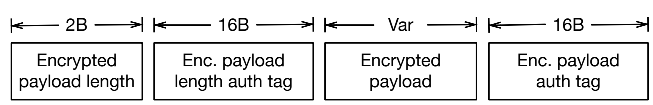
Although the Proteus system offers a large degree of flexibility due to its focus on safety and simplicity, some complex network protocols cannot be represented. For example, the file transfer protocol [14] multiplexes protocol messages over multiple connections and cannot be replicated in Proteus because every client-server session is isolated to a single connection and protocol instance. Some real-world network protocols use the host's persistent storage to maintain protocol state. TLS, for example, authenticates certificates with certificate stores located on disk. Proteus restricts system call usage from within an protocol, and hence this functionality could not be reproduced. Point-to-point transport protocols designed for censorship circumvention tend to have simple designs, leading us to believe that Proteus may be useful to program a number of protocols despite these limitations.

3 Proteus Examples

In this section, we show by example how an evader can specify and then easily modify encrypted network protocols using Proteus. We highlight salient elements of Proteus programs here and list the PSF source files in their entirety in Appendix A.

3.1 Shadowsocks

As an example, we first describe the Shadowsocks [21] obfuscation protocol as implemented in Proteus. Our implementation is *not* designed to be interoperable with Shadowsocks—it only has the same flow characteristics. To an observing third party, Shadowsocks flows have no structure and are indistinguishable from a stream of random bytes. The Shadowsocks protocol is fairly simple: each message consists of an encrypted length and an encrypted payload, where encryption is performed using an authenticated encryption with associated data (AEAD) scheme. AEAD ciphers simultaneously provide encryption and authentication, with the encryption operation outputting both a ciphertext and a *tag*, the latter of which is used by the decryption function to authenticate the ciphertext. Shadowsocks messages follow the format:



Specifying Shadowsocks in Proteus is straightforward. We first define protocol message definitions for the encrypted length (and tag) and encrypted payload (and tag). Separate message definitions are necessary since the encrypted length field needs to first be decrypted in order to determine how many bytes are required for the payload. We specify these message definitions as follows:

```
DEFINE EncLenFmt // encrypted length
{ NAME: EncPayloadLen;    TYPE: [u8; 2]; }
{ NAME: EncPayloadLenTag; TYPE: [u8; 16]; };

DEFINE EncPayloadFmt // the payload
{ NAME: EncPayload;    TYPE: [u8; *]; }
{ NAME: EncPayloadTag; TYPE: [u8; 16]; };
```

Listing 3: Message definitions for Shadowsocks

Following the Shadowsocks specification, we use two bytes for the encrypted payload length and 16 bytes for all tags. As with Shadowsocks, we use the ChaCha20 stream cipher with (16 byte) Poly1305 message authentication codes.

The PSF file also defines event handlers for the events described in Table 1:

```
SET_HANDLER( EV_NET, evNetRead );
SET_HANDLER( EV_APP, evAppRead );
...
```

The main operation of our Proteus-based Shadowsocks implementation is described in the `evNetRead()` and `evAppRead()` handlers (see § A.1 for their full descriptions). `evNetRead()` computes the length of an `EncLenFmt` message, $2+16=18$ bytes, and calls `pop()` on Net$^{(R)}$ to read 18 bytes off of the network read buffer. The `parse()` function then casts those bytes into an `EncLenFmt` message:

```
let encLen: Fields =
  match parse(&EncLenFmt, &encLenBytes) {
    (true, v) => v, ... };
```

Given the resulting message, the `decrypt()` function is called to obtain the payload size, *pl* (in plaintext). The handler then reads another *pl* bytes from Net$^{(R)}$ and calls `parse()` on the returned bytes to obtain the `EncPayloadFmt` message:

```
let payload: Fields = match
  parse(&EncryptedPayloadFmt, &encPayload) {
    (true, v) => v, ... };
```

Because the length of the `EncPayload` field in the `EncPayloadFmt` message is not known before receiving and decrypting the encrypted payload length, the `*` size indicator in the message definition is necessary (see Listing 3). This tells the `parse()` function to first assign all other fields (here, just the fixed-sized `EncPayloadTag` field) before assigning the remaining bytes in the buffer to the `EncPayload` field.

Finally, the `decrypt()` function is called again to obtain the plaintext payload. The decrypted payload is then pushed to the App$^{(W)}$ buffer for reading by the application.

The `evAppRead()` event handler performs the mirror operations with respect to `evNetRead()`: it reads bytes from App$^{(R)}$ (data sent by the application) and encrypts (1) the number of bytes read, and (2) the read bytes, both using ChaCha20-Poly1305. It then calls `format()` to construct the `EncLenFmt` and `EncPayloadFmt` messages[2]:

```
let encLenSpec: [u8; *] = match format(&EncLenFmt,
    &format![("EncPayloadLen", encLen),
        ("EncPayloadLenTag", encLenTag) ] )
        { (true,v) => v, ... };

let encPayloadSpec: [u8; *] =
  match format(&EncryptedPayloadFmt,
    &format![("EncPayload", encPayload),
        ("EncPayloadTag", encPayloadTag) ])
        { (true,v) => v, ... };
```

[2]The `format![...]` construct used in this example is syntactic sugar to create a format object with the specified fields.

The `format()` function returns the byte-representation of the messages, which are then pushed to the Net$^{(W)}$ buffer for transport over the network.

3.2 Modifying Shadowsocks

Wu et al. recently exposed a number of heuristics used by the Great Firewall (GFW) in China to detect and block Shadowsocks [27]. Essentially, the GFW looks for and blocks apparently high-entropy connections that are not TLS or HTTP. However, Wu et al. note that the GFW's approach to blocking Shadowsocks is brittle. In particular, connections are allowed if the first 6 bytes of the first packet of a flow are all printable characters (printable bytes are in the range `0x20`–`0x7E`).

Modifying the Proteus implementation of Shadowsocks to bypass GFW's censorship is thus trivial. The `EncLenFmt` message definition can be modified as follows:

```
DEFINE EncLenFmtV2
{ NAME: FixedPreamble;    TYPE: [u8; 6] }, // <-- New
{ NAME: EncPayloadLen;    TYPE: [u8; 2] },
{ NAME: EncPayloadLenTag; TYPE: [u8; 16] };
```

where `FixedPreamble` will be populated with a 6 byte alphanumeric string. Additionally, the `pop()` call in `evNetRead()` needs to read 6 more bytes than in our original Shadowsocks implementation. In total, expressing the modified Shadowsocks PSF file requires only a short patch (see Listing 5 in § A.2).

Proteus makes prototyping other packet encoding strategies easy, too. If instead of printable characters, the packet's ratio of `0`s to `1`s (the packet's so-called popcount) should be altered, a biased string could be inserted into the packet's fields. We posit that Proteus's adaptability is well-suited for the censorship arms race. The ability to easily modify protocols' structure enables evaders to quickly counter new changes in behavior of the censorship system.

3.3 Noise

To further illustrate the language's versatility, we express a Noise-based [12] protocol in Proteus; see Listing 6 in § A.3. Noise is a protocol framework that provides building blocks for constructing secure cryptographic protocols. In Listing 6, we present a Proteus-based implementation of a Noise protocol in which a client with knowledge of a server's (e.g., bridge's) public key performs a Diffie-Hellman exchange (with server authentication) and derives an ephemeral key, which it then uses to exchange messages via an AEAD cipher. This corresponds to the `NK` handshake pattern as described in the Noise specification [12].

For brevity, we omit a full explanation of our Noise-based protocol, and instead highlight some of the core functionalities that were expressed in Proteus. As shown in Listing 6, we use built-in crypto primitives—namely, `DH()` and `HMAC()`—to implement Noise's key chaining and derivation algorithms.

We also separate out the logic in the `evNetRead()` and `evAppRead()` handlers based on whether the protocol is in the handshake or data transmission phase. Much of the code in Listing 6 is fairly formulaic and mostly consists of sequences of calls to `parse()` and `format()`. In total, it took less than 4 hours to express a Noise-based protocol in Proteus.

4 Related Work

Programmable Obfuscation: Format-transforming encryption (FTE) is a programmable obfuscation system that takes a regular expression as input and then modifies a data stream such that it passes the regular expression [5]. A primary use-case of FTE is to create a data stream that mimics the format of well known application protocols such as HTTP. Although FTE can modify a data stream to impose a defined structure, it offers little control over protocol semantics or the statistical properties of the obfuscated traffic.

Marionette extends FTE to improve the programmability of protocol semantics and statistical traffic properties [6]. Similar to Proteus, Marionette defines protocol state machines (called *models*) which can capture the state of a channel between multiple rounds of communication and can drive responses to particular actions such as errors. Marionette uses a domain-specific language to specify a series of *templates* that will, as in FTE, insert the bytes necessary to impose a defined structure on outgoing messages. However, this language is not specified outside of the implementation of the interpreter making it difficult even for domain experts to write correct code using the language. Comparatively, the Proteus language is specified and designed to be easy to write for both domain experts and non-specialists. Furthermore, Marionette is designed such that its language calls out to plugins written in a standard programming language to implement important data processing functionality, posing significant safety risks to users and proxy operators. In contrast, the Proteus language is intentionally limited to a core set of functions necessary to implement common functionality, and this isolation improves safety and reliability of both Proteus and the protocols it runs. Finally, unlike Proteus, Marionette does not support multiple simultaneous protocols and version upgrades.

Anti-censorship researchers activists have developed other tools offering aspects of programmability [10, 22]; however, these projects tend to lack formal documentation and maturity, making a rigorous evaluation difficult.

Programmable Anonymous Communication: Flexible Anonymous Networks (FAN) is a programmable network design that separates the software architecture from deployed functionalities [18, 19]. A FAN can be programmed by compiling functionalities (e.g., adding, removing, or modifying hook functions) using LLVM into portable RISC-V object files that get packaged and distributed as a plugin and then loaded by network nodes and executed in a sandbox using just-in-time compilation. This approach effectively changes the code that runs inside of the anonymity network nodes.

Similar to FAN, Bento is an architecture that proposes to use middleboxes to bring network function virtualization to anonymity networks [17]. Rather than modifying Tor's internal functions as FAN does, Bento runs as a separate process, and runs arbitrary user-defined functions in a secure enclave while interacting with Tor using its control interface.

Like our approach, FAN and Bento seek to provide better modularity to more quickly adapt to new requirements. However, they both raise significant security and trust questions since a user or plugin programmer can cause arbitrary code execution on network nodes. Our approach is more isolated and measured, focusing on providing a small standard library of functions that focus on censorship circumvention protocol behavior rather than a fully general software architecture.

5 Discussion and Future Work

Proteus is compatible with several deployment models. In coordinated systems like Tor, Proteus can enable the authorities to quickly disseminate new circumvention protocols. In loosely organized systems like Shadowsocks, Proteus could foster an ecosystem of individual experimentation to evade censorship rules as they appear in different locales.

Proteus could be seamlessly swapped into several existing systems. For existing protocols that can be expressed in the Proteus language, such as obfs4 and Shadowsocks, Proteus can work in partial deployment at only the server or client side. Moreover, on the server side, it can be used to simultaneously support improved circumvention techniques and legacy clients who have not yet upgraded.

The safety of Proteus could make automatic updating desirable for systems that adopt it. Currently, in security-conscious proxy systems like Tor, updates cannot be forced on proxy operators to limit the risk of a malicious or mistaken developer. However, this limits the speed of the arms race to how fast operators can be made to install upgrades with new evasion strategies. Proteus safety features could make pushing protocol updates no more objectionable than how the Tor authorities currently push their hourly network consensuses.

Work currently ongoing in Proteus includes completing a complete specification of the language, developing a prototype implementation, and testing it in target network environments. Possible improvements to its design include the ability to create multiple TCP connections, support for UDP, and providing more support for traffic shaping through padding bytes and added delays. Another aspect of Proteus that may be improved is error handling. Allowing Proteus protocols to implement normalized or randomized responses to errors may improve its resistance to detection via active probing [8].

Availability

Proteus is actively developed at the time of this work's publication. The Proteus source code is maintained and updated at the following link:
https://github.com/unblockable/proteus.

Acknowledgments

This work was partially supported by the Office of Naval Research (ONR), the Defense Advanced Research Projects Agency (DARPA) (including under Contract No. FA8750-19-C-0500), and the Georgetown University Callahan Family Professor Chair Fund. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] S. Afroz, D. Fifield, M. C. Tschantz, V. Paxson, and J. Tygar, "Censorship arms race: Research vs. practice," in *HotPETs: Workshop on Hot Topics in Privacy Enhancing Technologies*, 2015. eprint: https://petsymposium.org/2015/papers/afroz-censor-eval-hotpets2015.pdf.

[2] Alice, Bob, Carol, J. Beznazwy, and A. Houmansadr, "How China detects and blocks Shadowsocks," in *ACM IMC: ACM Internet Measurement Conference*, ACM, 2020. DOI: 10.1145/3419394.3423644.

[3] K. Bock, Y. Fax, K. Reese, J. Singh, and D. Levin, "Detecting and evading censorship-in-depth: A case study of Iran's protocol filter," in *FOCI: USENIX Workshop on Free and Open Communications on the Internet*, USENIX Association, 2020. eprint: https://www.usenix.org/system/files/foci20-paper-bock.pdf.

[4] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The Second-Generation Onion Router," in *USENIX Security Symposium*, USENIX Association, 2004. eprint: https://www.usenix.org/legacy/publications/library/proceedings/sec04/tech/full_papers/dingledine/dingledine.pdf.

[5] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton, "Protocol misidentification made easy with format-transforming encryption," in *ACM CCS: ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2013. DOI: 10.1145/2508859.2516657.

[6] K. P. Dyer, S. E. Coull, and T. Shrimpton, "Marionette: A programmable network traffic obfuscation system," in *USENIX Security Symposium*, USENIX Association, 2015. eprint: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-dyer.pdf.

[7] B. Ford, "Parsing expression grammars: A recognition-based syntactic foundation," in *ACM POPL: ACM SIGPLAN Symposium on Principles of Programming Language*, 2004. DOI: 10.1145/964001.964011.

[8] S. Frolov, J. Wampler, and E. Wustrow, "Detecting probe-resistant proxies," in *NDSS: Network and Distributed Systems Security Symposium*, Internet Society, 2020. DOI: 10.14722/ndss.2020.23087.

[9] G. J. Holzmann, "The power of 10: Rules for developing safety-critical code," Computer, 2006. DOI: 10.1109/MC.2006.212.

[10] "Operator foundation." Accessed 2023-06-30, Operator Foundation. (2023), [Online]. Available: https://operatorfoundation.org/.

[11] "Outline client." Accessed 2023-03-15, Jigsaw. (2023), [Online]. Available: https://github.com/Jigsaw-Code/outline-client.

[12] T. Perrin, *The Noise protocol framework*, version 34, 2018. eprint: https://noiseprotocol.org/noise.pdf.

[13] "pest." Accessed 2023-01-11. (2023), [Online]. Available: https://pest.rs/. Archived: http://archive.today/3Gfsr.

[14] J. Postel and J. Reynolds, *File transfer protocol (FTP)*, Request for Comments (RFC) 959, 1985. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc959.

[15] "Project V." Accessed 2023-03-15, Project V. (2023), [Online]. Available: https://www.v2ray.com/en/.

[16] "Psiphon." Accessed 2023-03-15, Psiphon Inc. (2023), [Online]. Available: https://psiphon.ca/ur/index.html.

[17] M. Reininger, A. Arora, S. Herwig, N. Francino, J. Hurst, C. Garman, and D. Levin, "Bento: Safely bringing network function virtualization to Tor," in *ACM SIGCOMM Conference*, ACM, 2021. DOI: 10.1145/3452296.3472919.

[18] F. Rochet, O. Bonaventure, and O. Pereira, "Flexible anonymous network," in *HotPETs: Workshop on Hot Topics in Privacy Enhancing Technologies*, 2019. eprint: https://petsymposium.org/2019/files/hotpets/proposals/rochet-fan.pdf.

[19] F. Rochet and T. Elahi, "Towards flexible anonymous networks," arXiv, 2022. DOI: 10.48550/arXiv.2203.03764.

[20] "Rust Crypto," Rust Crypto. (2023), [Online]. Available: https://github.com/RustCrypto.

[21] "Shadowsocks." Accessed 2023-01-11, Shadowsocks. (2023), [Online]. Available: https://shadowsocks.org/. Archived: https://archive.ph/EIB3f.

[22] "trojan." Accessed 2023-06-30, trojan-gfw. (2023), [Online]. Available: https://github.com/trojan-gfw/trojan.

[23] "VMess." Accessed 2023-03-15, V2Ray. (2023), [Online]. Available: https://www.v2ray.com/en/configuration/protocols/vmess.html.

[24] L. Wang, K. P. Dyer, A. Akella, T. Ristenpart, and T. Shrimpton, "Seeing through network-protocol obfuscation," in *ACM CCS: ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2015. DOI: 10.1145/2810103.2813715.

[25] B. Wiley, "Circumventing network filtering with polymorphic protocol shapeshifting," PhD Dissertation, The University of Texas at Austin, 2016. eprint: https://blanu.net/Dissertation.pdf.

[26] B. Wiley, "Dust: A blocking-resistant Internet transport protocol," 2011. eprint: https://www.freehaven.net/anonbib/cache/wileydust.pdf.

[27] M. Wu, J. Sippe, D. Sivakumar, J. Burg, P. Anderson, X. Wang, K. Bock, A. Houmansadr, D. Levin, and E. Wustrow, "How the Great Firewall of China detects and blocks fully encrypted traffic," in *USENIX Security Symposium*, USENIX Association, 2023. eprint: https://www.usenix.org/conference/usenixsecurity23/presentation/wu-mingshi.

[28] D. Xue, R. Ramesh, A. Jain, M. Kallitsis, J. A. Halderman, J. R. Crandall, and R. Ensafi, "OpenVPN is open to VPN fingerprinting," in *USENIX Security Symposium*, USENIX Association, 2022. eprint: https://www.usenix.org/system/files/sec22-xue-diwen.pdf.

[29] Yawning Angel. "Obfs4 (the obfourscator)." (Jan. 2019), [Online]. Available: https://github.com/Yawning/obfs4/blob/master/doc/obfs4-spec.txt.

Appendices

A Proteus Programs

This appendix contains source code listings for Proteus programs referred to in this work. Each program was syntactically checked against the Proteus grammar given in Appendix C.

A.1 Shadowsocks

```
/*
 * Event handlers
 */
SET_HANDLER( EV_INIT, evInit );
SET_HANDLER( EV_NET, evNetRead );
SET_HANDLER( EV_APP, evAppRead );
SET_HANDLER( EV_TIMER, nullHandler );
SET_HANDLER( *, exitHandler );    // everything else goes to exitHandler

/*
 * Message formats
 */

// encrypted length
DEFINE EncLenFmt
{ NAME: EncPayloadLen; TYPE: [u8; 2] },
{ NAME: EncPayloadLenTag; TYPE: [u8; 16] };

// the payload
DEFINE EncryptedPayloadFmt
{ NAME: EncPayload; TYPE: [u8; *] },
{ NAME: EncPayloadTag; TYPE: [u8; 16] };

/*
 * Global variables
 */

GLOBALS {
  let mut Key :[u8;32] = [0u8; 32];
  // initializes nonces (counters) for incoming and outgoing traffic
  let mut OutNonce :u64 = 0u64;
  let mut InNonce :u64 = 0u64;
}

/*
 * Event handlers
 */

fn evInit() {
    // grab key from environment variable
    match getenv<[u8; 32]>("chacha20_key", &global.Key) {
        (false, _) => panic(),
    };
}


fn evNetRead() {
    // compute size of the EncLenFmt frame
    let encPayloadLenSize :u64
        = get_field_size(&EncLenFmt, "EncPayloadLen");

    let encPayloadLenTagSize :u64
        = get_field_size(&EncLenFmt, "EncPayloadLenTag");

    let expectedLen :u64 = encPayloadLenSize + encPayloadLenTagSize;

    let blocking: bool = true;

    // block until there are at least `expectedLen` bytes to read
    let encLenBytes: [u8; *] = buffer_pop(&RB_net, expectedLen, blocking);

    let encLen: Fields = match parse(&EncLenFmt, &encLenBytes) {
        (true,v) => v,
        (false,_) => {
            buffer_close_all();  // close connection on error
            return;
        }
    };

    let encPayloadLen: [u8; 2] = [0u8; 2];
    match getField<[u8; *]>(&encLen, "EncPayloadLen", &encPayloadLen) {
        false => panic(),
    };

    let encPayloadLenTag: [u8; 16] = [0u8; 16];
    match getField<[u8; *]>(&envLen, "EncPayloadLenTag", &encPayloadLenTag) {
        false => panic(),
    };

    // decrypt returns (num_bytes,val) tuple
    let (_, l) : (_, u64) = match decrypt<u64>(
        "chacha20-poly1305",
        &global.Key, // key
        &encPayloadLen, // message
        encPayloadLenSize, // message size
        &global.InNonce, // nonce and (next line) tag
        &encPayloadLenTag) {
            (true,v) => v,
            (false,_) => {
                buffer_close_all();  // close connection on error
                return;
            }
        };

    global.InNonce = global.InNonce + 1u64;

    // then, grab the encrypted payload and parse it
    let encPayload: [u8; *] = buffer_pop(&RB_net, l, blocking);
    let payload: Fields = match parse(&EncryptedPayloadFmt, &encPayload) {
        (true,v) => v,
        (false,_) => panic(),
    };

    let payload_buffer: BytesMut = get_buffer(65536u64);
    let encPayload: [u8; *] = match getField<[u8; *]>(
        &payload_buffer, "EncPayload", &payload_buffer) {
            false => panic(),
        };

    let tag_buffer: BytesMut = get_buffer(16u64);
    let encPayloadTag: [u8; *] = match getField<[u8; *]>(
        &payload, "EncPayloadTag", &tag_buffer) {
            false => panic(),
        };

    // and decrypt it
    let (_, plaintext) : (_, [u8; *]) = match decrypt<[u8; *]>(
        "chacha20-poly1305",
        &global.Key, // key
        &encPayload,
        &encPayloadSize,
        &global.InNonce,
        &encPayloadTag)
    {
            (true,v) => v,
            (false,_) => {
                buffer_close_all();  // close connection on error
                return;
            }
    };

    global.InNonce = global.InNonce + 1u64;

    // send the results to the app
    buffer_push(&WB_app, &plaintext);
}


fn evAppRead() {
    // grab data from buffer (from the application)
    let l :u16 = buffer_length(&RB_app);
    let data :[u8; *] = buffer_pop(&RB_app, l, false);

    // encrypt the length
    let (encLen, encLenTag): ([u8; 2], [u8; 16]) = match encrypt(
        "chacha20-poly1305",
        &global.Key,
        &l,
        2u64,
        &global.OutNonce ) {
            (true,ciphertext,tag) => (ciphertext, tag),
            (false,_,_) => panic(),
        };
    global.OutNonce = global.OutNonce + 1u64;

    // encrypt the payload
    let (encPayload, encPayloadTag): ([u8; *], [u8; 16]) = match encrypt(
        "chacha20-poly1305",
        &global.Key,
        &data,
        l,
        global.OutNonce ) {
            (true,ciphertext,tag) => (ciphertext, tag),
            (false,_,_) => panic(),
        };

    global.OutNonce = global.OutNonce + 1u64;

    // produce the frames
    let encLenSpec: [u8; *]= match format(&EncLenFmt,
        &format![
            ("EncPayloadLen", encLen),
            ("EncPayloadLenTag", encLenTag)
        ]
        ) {
            (true,v) => v,
            (false,_) => panic(),
    };

    let encPayloadSpec: [u8; *] = match format(&EncryptedPayloadFmt,
        &format![
            ("EncPayload", encPayload),
            ("EncPayloadTag", encPayloadTag),
        ]
        ) {
            (true,v) => v,
            (false,_) => panic(),
    };

    // send them on the wire
    buffer_push(&WB_net, &encLenSpec);
    buffer_push(&WB_net, &encPayloadSpec);
}
```

```
fn nullHandler() {
}

fn exitHandler() {
    exit(0u32);
}
```

Listing 4: PSF for Shadowsocks in AEAD mode

A.2 Modifying Shadowsocks to Bypass GFW Censorship

Modifying the Proteus implementation of Shadowsocks (see Listing 4) to bypass blocking by the GFW is straightforward. Listing 5 describes the complete patch for adding a six byte alphanumeric constant ("`123456`") to the beginning of Shadowsocks messages.

```
@@ -13,6 +13,7 @@

 // encrypted length
 DEFINE EncLenFmt
+{ NAME: FixedPreamble; TYPE: [u8; 6] },
 { NAME: EncPayloadLen; TYPE: [u8; 2] },
 { NAME: EncPayloadLenTag; TYPE: [u8; 16] };

@@ -45,6 +46,8 @@


 fn evNetRead() {
+    let fixed_preamble_size :u64
+        = get_field_size(&EncLenFmt, "FixedPreamble");
     // compute size of the EncLenFmt frame
     let encPayloadLenSize :u64
         = get_field_size(&EncLenFmt, "EncPayloadLen");
@@ -52,7 +55,8 @@
     let encPayloadLenTagSize :u64
         = get_field_size(&EncLenFmt, "EncPayloadLenTag");

-    let expectedLen :u64 = encPayloadLenSize + encPayloadLenTagSize;
+    let expectedLen :u64 = encPayloadLenSize +
+    encPayloadLenTagSize + fixed_preamble_size;

     let blocking: bool = true;

@@ -169,6 +173,7 @@
     // produce the frames
     let encLenSpec: [u8; *]= match format(&EncLenFmt,
         &format![
+            ("FixedPreamble", "123456"),
             ("EncPayloadLen", encLen),
             ("EncPayloadLenTag", encLenTag)
         ]
```

Listing 5: Modifications to the Shadowsocks PSF (see Listing 4) to achieve reduced entropy

A.3 Noise

Noise [12] is a protocol framework and does not specify wire formats. We adapt Noise to a "wire" protocol by prepending a length field in front of every message.

Noise does not correspond to a particular protocol, and instead is a framework for specifying secure protocols via *handshake patterns*. We use the `NK` handshake pattern, which is defined as:

NK:

$\leftarrow s$ (sent out-of-band)

$\ldots$

$\rightarrow e, es$

$\leftarrow e, ee$

This corresponds to the case where the client knows apriori the server's public key (`s`) and uses it to perform a DH exchange (with server authentication) with the server.

The corresponding Proteus definition file is presented in Listing 6.

```
/**
 * Event handlers
 */
SET_HANDLER( EV_INIT, evInit );
SET_HANDLER( EV_NET, evNetRead );
SET_HANDLER( EV_APP, evAppRead );
SET_HANDLER( EV_TIMER, nullHandler );
SET_HANDLER( *, exitHandler );    // everything else goes to exitHandler


/*
 * Message formats
 */

// corresponds to -> e, es
DEFINE Handshake1
{ NAME:  InitiatorEphemeralKey; TYPE: [u8; 45] };

// corresponds to <- e, ee
DEFINE Handshake2
{ NAME: EncResponderEphemeralKey; TYPE: [u8; 56] },
{ NAME: EncResponderEphemeralKeyTag; TYPE: [u8; 16] };

// an encrypted message (after handshaking)
DEFINE EncryptedPayloadSpec
{ NAME: PayloadSize; TYPE: u16 },
{ NAME: EncPayload; TYPE: [u8; *] },
{ NAME: EncPayloadTag; TYPE: [u8; 16] };


/*
 * Global variables
 */
GLOBALS {
    let mut CompletedHandshake :bool = false;
    let mut OutNonce :u64 = 0u64;
    let mut InNonce :u64 = 0u64;
    let mut IsInitiator :bool = false;
    let mut ck :[u8;32] = [0u8; 32];
    let mut k :[u8;32] = [0u8; 32];

    // some constants we'll need later
    let byte01 :[u8;1] = [ 0x01; 1 ];
    let byte02 :[u8;1] = [ 0x02; 1 ];

    // key material
    let mut EphemeralDHKeyPub :[u8;56] = [0u8; 56];
    let mut EphemeralDHKeyPri :[u8;56] = [0u8; 56];
    let mut ServerStaticDHKeyPub :[u8;56] = [0u8; 56];
    let mut StaticDHKeyPub :[u8;56] = [0u8; 56];
    let mut StaticDHKeyPri :[u8;56] = [0u8; 56];
}

/*
 * Event handlers
 */

 fn evInit() {
    // let's figure out if we're the initiator or responder
    // by looking for the is_initiator environment variable

    let mut tmp: [u8;1] = [0u8; 1];

    global.IsInitiator = match getenv<[u8; 1]>("is_initiator", &tmp) {
        (true,_) => true,
        (false,_) => false
    };

    // initialize hash according to InitializeSymmetric(...) func from Noise
    let h :[u8;32] = hash(sha256, "Noise_NK_25519_ChaChaPoly_SHA256");
    global.ck = h;
    global.k = h;

    // compute an ephemeral key
    (global.EphemeralDHKeyPub, global.EphemeralDHKeyPri) = genDHKeyPair(56u64);

    match getenv<[u8; 56]>("server_pub", &global.ServerStaticDHKeyPub) {
        (false,_) => panic(),
    };

    if global.IsInitiator == true {
        // compute initial key by first doing DH...
        let input_key_material :[u8;56]
            = match DH(global.EphemeralDHKeyPri, global.ServerStaticDHKeyPub) {
                (true,v) => v,
                (false,_) => panic(),
        };
        // update the global key according to Noise HKDF() function
        let temp_k :[u8;32] = hmac(sha256, global.ck, input_key_material);
        global.ck = hmac(sha256, temp_k, global.byte01);
        global.k = hmac(sha256, concatenate(global.ck,global.byte02));

    } else {
        // we're the server, grab key pair from environment variable
        match getenv<[u8;56]>("dhkey_pub", &global.StaticDHKeyPub) {
            (false,_) => panic(),
        };
        match getenv<[u8;56]>("dhkey_pri", &global.StaticDHKeyPri) {
            (false,_) => panic(),
        };
    }
}


fn evAppRead() {
```

```
    if !global.CompletedHandshake {
        // before we do anything else, we need to complete the handshake

        if global.IsInitiator {

            // send Handshake1 message to responder
            let handshake1Fields :Fields = create_fields();
            set_field( &handshake1Fields,
                "InitiatorEphemeralKey", global.EphemeralDHKeyPub);

            let handshake1 :[u8; *] = match format(&Handshake1, &handshake1Fields) {
                (true,v) => v,
                (false,_)  => panic(),
            };

            buffer_push( &WB_net, handshake1 );

            // wait for response from responder
            let handshake2KeySize :u64
                = get_field_size(&Handshake2,"EncResponderEphemeralKey");
            let handshake2TagSize :u64
                = get_field_size(&Handshake2,"EncResponderEphemeralKeyTag");
            let frameContents :[u8; *]
                = match buffer_pop(&RB_net, handshake2KeySize+handshake2TagSize, true) {
                    (true,v) => v,
                    (false,_) => panic(),
                };

            // parse response (a handshake2 message)
            let handshake2 :Fields = match parse(&Handshake2, frameContents) {
                (true,v) => v,
                (false,_) => {
                    buffer_close_all();  // close connection on error
                    return;
                }
            };

            let encResponderEphemeralKey :BytesMut = get_buffer(64u64);

            if !get_field<[u8; 56]>(&handshake2,
                "EncResponderEphemeralKey", &encResponderEphemeralKey) {
                  buffer_close_all();
                  return;
            };

            let encResponderEphemeralKeyTag :BytesMut = get_buffer(32u64);

            if !getField<[u8; 16]>(&handshake2,
                "EncResponderEphemeralKeyTag", &encResponderEphemeralKeyTag) {
                  buffer_close_all();
                  return;
            };

            // decrypt to get the responder's ephemeral key
            let responderEphemeralKey :[u8; 56] = match decrypt(
                "chacha20-poly1305",
                global.k,
                &encResponderEphemeralKey,
                handshake2KeySize,
                &global.InNonce,
                &encResponderEphemeralKeyTag) {
                    (true,v) => v,
                    (false,_) => {
                        buffer_close_all();
                        return;
                    }
            };

            global.InNonce = global.InNonce + 1u64;

            // update the global key according to Noise HKDF() function
            let input_key_material :[u8;56] = responderEphemeralKey;
            let temp_k :[u8;32] = hmac(sha256, global.ck, input_key_material);
            global.ck = hmac(sha256, temp_k, global.byte01 );
            global.k = hmac(sha256, global.ck, global.byte02 );

            global.CompletedHandshake = true;

        } else {

            // we're the responder, so we'll wait for the initiator to send Handshake1
            let handshake1KeySize :u64
                = get_field_size(&Handshake1, "InitiatorEphemeralKey");

            let frameContents : Bytes
                = match buffer_pop(&RB_net, handshake1KeySize, true) {
                    (true,v) => v,
                    (false,_) => panic(),
                };

            // parse response (a handshake1 message)
            let handshake1 :Fields = match parse(&Handshake1, frameContents ) {
                (true,v) => v,
                (false,_) => {
                    buffer_close_all();
                    return;
                }
            };

            let initiatorPK :BytesMut = get_buffer(64u64);

            if !get_field<[u8; 56]>(&Handshake1, "InitiatorEphemeralKey", &initiatorPK) {
                buffer_close_all();
                return;
            };

            // compute our first DH
            let input_key_material :[u8;56]
                = match DH(global.StaticDHKeyPri, initiatorPK) {
                    (true,v) => v,
                    (false,_) => panic(),
            };
            let temp_k :[u8;32] = hmac(sha256, global.ck, input_key_material);
            global.ck = hmac(sha256, temp_k, global.byte01 );
            global.k = hmac(sha256, global.ck, global.byte02 );

            // send our ephemeral key, encrypted, to the initiator
            let (encPayload, encPayloadTag) : ([u8; *], [u8; 16]) =
            match encrypt(
                "chacha20-poly1305",
                &global.k,
                &global.EphemeralDHKeyPub,
                56u64,
                &global.OutNonce ) {
                    (true,ciphertext,tag) => (ciphertext, tag),
                    (false,_,_) => panic(),
            };

            global.OutNonce = global.OutNonce + 1u64;

            let handshake2Fields :Fields = create_fields();
            set_field( &handshake2Fields,
                "EncResponderEphemeralKey", encPayload );
            set_field( &handshake2Fields,
                "EncResponderEphemeralKeyTag", encPayloadTag );

            let handshake2 :Bytes = match format(
                &Handshake2, &handshake2Fields ) {
                    (true,v) => v,
                    (false,_)  => panic(),
            };

            buffer_push( &WB_net, handshake2 );

            // and compute our final global key
            let input_key_material2 :[u8;56]
                = match DH(global.EphemeralDHKeyPri, initiatorPK) {
                    (true,v) => v,
                    (false,_) => panic(),
            };
            let temp_k :[u8;32] = hmac(sha256,global.ck,input_key_material2);
            global.ck = hmac(sha256,temp_k,global.byte01);
            global.k = hmac(sha256, global.ck, global.byte02);

            global.CompletedHandshake = true;
        }
    } else {
        // handshake completed, so send data in AEAD

        // get data from app
        let l :u64 = buffer_length(&RB_app);
        let data :Bytes = match buffer_pop(&RB_app, l, true) {
            (true,v) => v,
            (false,_) => panic(),
        };

         // encrypt the payload
        let (encPayload, encPayloadTag): ([u8; *], [u8; 16]) = match encrypt(
            "chacha20-poly1305",
            &global.k,
            &data,
            l,
            &global.OutNonce ) {
                (true,ciphertext,tag) => (ciphertext, tag),
                (false,_,_) => panic(),
        };
        global.OutNonce = global.OutNonce + 1u64;

        // put the ciphertext in its frame
        let encryptedPayloadFields :Fields = create_fields();
        set_field( &encryptedPayloadFields, "PayloadSize", l );
        set_field( &encryptedPayloadFields, "EncPayload", encPayload );
        set_field( &encryptedPayloadFields, "EncPayloadTag", encPayloadTag );
        let encPayloadSpec :Bytes
            = match format(&EncryptedPayloadSpec, &encryptedPayloadFields) {
                (true,v) => v,
                (false,_)  => panic(),
        };

        // and send it!
        buffer_push( &WB_net, encPayloadSpec );
    }
}


fn evNetRead() {
    if !global.CompletedHandshake {
        // before we do anything else, we need to complete the handshake

        if global.IsInitiator {

            // send Handshake1 message to responder
            let handshake1Fields :Fields = create_fields();
            set_field( &handshake1Fields,
                "InitiatorEphemeralKey", global.EphemeralDHKeyPub);
            let handshake1 :Bytes = match format(&Handshake1, &handshake1Fields) {
                (true,v) => v,
                (false,_)  => panic(),
            };
            buffer_push( &WB_net, handshake1 );

            // wait for response from responder
            let handshake2KeySize :u64
                = get_field_size(&Handshake2,"EncResponderEphemeralKey");
            let handshake2TagSize :u64
                = get_field_size(&Handshake2,"EncResponderEphemeralKeyTag");
```

```
            let frameContents :Bytes
                = match buffer_pop(&RB_net, handshake2KeySize+handshake2TagSize, true) {
                    (true,v) => v,
                    (false,_) => panic(),
                };

            // parse response (a handshake2 message)
            let handshake2 :Fields = match parse(&Handshake2, frameContents) {
                (true,v) => v,
                (false,_) => {
                    buffer_close_all();  // close connection on error
                    return;
                }
            };

            let encResponderEphemeralKey :BytesMut = get_buffer(64u64);

            if !get_field<[u8; 56]>(&handshake2,
                "EncResponderEphemeralKey", &encResponderEphemeralKey) {
                buffer_close_all();
                return;
            };

            let encResponderEphemeralKeyTag :BytesMut = get_buffer(32u64);

            if !getField<[u8; 16]>(&handshake2,
                "EncResponderEphemeralKeyTag", &encResponderEphemeralKeyTag) {
                buffer_close_all();
                return;
            };

            // decrypt to get the responder's ephemeral key
            let responderEphemeralKey: [u8; *] = match decrypt(
                "chacha20-poly1305",
                global.k,
                &encResponderEphemeralKey,
                handshake2KeySize,
                &global.InNonce,
                &encResponderEphemeralKeyTag) {
                    (true,v) => v,
                    (false,_) => {
                        buffer_close_all();
                        return;
                    }
            };

            global.InNonce = global.InNonce + 1u64;

            // update the global key according to Noise HKDF() function
            let input_key_material :[u8;56] = responderEphemeralKey;
            let temp_k :[u8;32] = hmac(sha256, global.ck, input_key_material);
            global.ck = hmac(sha256, temp_k, global.byte01);
            global.k = hmac(sha256, global.ck, global.byte02);

            global.CompletedHandshake = true;

        } else {

            // we're the responder, so we'll wait for the initiator to send Handshake1
            let handshake1KeySize :u64
                = get_field_size(&Handshake1, "InitiatorEphemeralKey");

            let frameContents :Bytes
                = match buffer_pop(&RB_net, handshake1KeySize, true) {
                    (true,v) => v,
                    (false,_) => panic(),
                };

            // parse response (a handshake1 message)
            let handshake1 :Fields = match parse(&Handshake1, frameContents ) {
                (true,v) => v,
                (false,_) => {
                    buffer_close_all();
                    return;
                }
            };

            let initiatorPK :BytesMut = get_buffer(64u64);

            if !get_field<[u8; 56]>(&Handshake1,"InitiatorEphemeralKey", &initiatorPK) {
                buffer_close_all();
                return;
            };

            // compute our first DH
            let input_key_material :[u8;56]
                = match DH(global.StaticDHKeyPri, initiatorPK) {
                    (true,v) => v,
                    (false,_) => panic(),
                };

            let temp_k :[u8; 32] = hmac(sha256, global.ck, input_key_material);
            global.ck = hmac(sha256, temp_k, global.byte01 );
            global.k = hmac(sha256, global.ck, global.byte02 );

            // send our ephemeral key, encrypted, to the initiator
            let (encPayload, encPayloadTag): ([u8; *], [u8; 16]) = match encrypt(
                "chacha20-poly1305",
                &global.k,
                &global.EphemeralDHKeyPub,
                56u64,
                &global.OutNonce ) {
                    (true,ciphertext,tag) => (ciphertext, tag),
                    (false,_,_) => panic(),
                };

            global.OutNonce = global.OutNonce + 1u64;

            let handshake2Fields :Fields = create_fields();
            set_field( &handshake2Fields,
                "EncResponderEphemeralKey", encPayload );

            set_field( &handshake2Fields,
                "EncResponderEphemeralKeyTag", encPayloadTag );

            let handshake2 :Bytes = match format(
                &Handshake2, &handshake2Fields ) {
                    (true,v) => v,
                    (false,_)  => panic(),
            };
            buffer_push( &WB_net, handshake2 );

            // and compute our final global key
            let input_key_material2 :[u8;56]
                = match DH(global.EphemeralDHKeyPri, initiatorPK) {
                    (true,v) => v,
                    (false,_) => panic(),
            };
            let temp_k :[u8;32] = hmac(sha256,global.ck,input_key_material2);
            global.ck = hmac(sha256,temp_k,global.byte01);
            global.k = hmac(sha256, global.ck, global.byte02);

            global.CompletedHandshake = true;
        }
    } else {
        // handshake completed, so grab the data off of the wire
        let payloadSizeLen :u64
            = get_field_size(&EncryptedPayloadSpec,"PayloadSize");
        let encPayloadTagLen :u64
             = get_field_size(&EncryptedPayloadSpec,"EncPayloadTag");

        let l :u16 = buffer_peek(&RB_net, payloadSizeLen);

        let frameContents :Bytes = match buffer_pop(
            RB_net,
            payloadSizeLen + l + encPayloadTagLen,
            true) {
                (true,v) => v,
                (false,_) => panic(),
            };

        // parse it
        let encryptedPayload: Fields = match parse(&EncryptedPayloadSpec, &frameContents ) {
            (true,v) => v,
            (false,_) => {
                buffer_close_all();  // close connection on error
                return;
            }
        };
        let encPayload: BytesMut = get_buffer(65535u64);
        if !get_field<[u8; *]>(&EncryptedPayloadSpec, "EncPayload", &encPayload )
        {
            panic();
        };

        let encPayloadTag: BytesMut = get_buffer(32u64);
        if !getField<[u8; 32]>(&EncryptedPayloadSpec, "EncPayloadTag", &encPayloadTag ) {
            panic();
        };

        // decrypt returns (num_bytes,val) tuple
        let plaintext :[u8; *] = match decrypt(
            "chacha20-poly1305",
            &global.k,
            &encPayload,
            l,
            &global.InNonce,
            &encPayloadTag) {
                (true,v) => v,
                (false,_) => {
                    buffer_close_all();  // close connection on error
                    return;
                }
        };
        global.InNonce = global.InNonce + 1u64;

        // send the results to the app
        buffer_push( &WB_app, plaintext );
    }
}


fn nullHandler() {
}

fn exitHandler() {
    exit(0u32);
}
```

Listing 6: PSF for the NK Noise handshake pattern

B Proteus Standard Library and Runtime

This appendix contains details regarding the functions provided by the Proteus standard library and their implementation.

B.1 Functionality

```
SECTION - I/O RELATED
================================================================================

NAME

  buffer_length - get number of bytes available in a buffer

SYNOPSIS

  fn buffer_length(b: &Buffer) -> usize;

DESCRIPTION

  Gets the number of bytes present in a buffer.

RETURN VALUE

  The number of bytes present in the buffer. No error value is specified.

--------------------------------------------------------------------------------

NAME

  buffer_peek - get a copy of n bytes from a buffer without removing them

SYNOPSIS

  fn buffer_peek(b: &ReadBuffer, n: usize) -> (bool, Bytes);

DESCRIPTION

  Gets the first n bytes of data present in the buffer. The buffer is not
  modified as a result of this operation.

RETURN VALUE

  buffer_peek() returns a pair of values. The first element of the pair
  indicates if the full peek could be performed (true if so, otherwise the value
  is false). The second element contains the copied data from the buffer
  (with length equal to the minimum of n and buffer_length(b)).

--------------------------------------------------------------------------------

NAME

  buffer_pop - removes bytes from a buffer

SYNOPSIS

  fn buffer_pop(b: &mut ReadBuffer, n: usize, blocking: bool) -> (bool, Bytes);

DESCRIPTION

  Removes the first n bytes from buffer b. If there are fewer than n bytes
  available, this function does not modify the buffer. If the blocking parameter
  is set to true, this function blocks until there is n bytes of data in the
  buffer.

RETURN VALUE

  For the first return value, returns true if the pop was successful (i.e., at
  least n bytes were removed from the buffer); false otherwise. The second
  argument returns the bytes that were removed.

--------------------------------------------------------------------------------

NAME

 buffer_push - adds bytes to a buffer.

SYNOPSIS

  fn buffer_push(b: &mut WriteBuffer, data: Bytes) -> bool;

DESCRIPTION

  Adds the input data to the buffer, if there is enough room in the buffer.
  Otherwise, the buffer is not modified.

RETURN VALUE

  True if the data was successfully added to the buffer; false otherwise.

--------------------------------------------------------------------------------

NAME

  buffer_close - close the connection associated with a buffer.

SYNOPSIS

  fn buffer_close(b: &mut Buffer);

DESCRIPTION

  Closes the connection associated with the given buffer. This operation is
  analogous to calling close on a buffer.

RETURN VALUE

  N/A

--------------------------------------------------------------------------------

NAME

  buffer_close_all - closes all connections

SYNOPSIS

  fn buffer_close_all();

DESCRIPTION

  Equivalent to calling:

  ```
  buffer_close(rb_app);
  buffer_close(rb_net);
  buffer_close(wb_app);
  buffer_close(wb_net);
  ```

RETURN VALUE

  N/A

SECTION - UTILITY FUNCTIONS
==

NAME

  getenv - gets an environment variable

SYNOPSIS

  fn getenv<T>(name: &str, value: &mut T) -> bool;

DESCRIPTION

  Gets en environment variable of type T. The value is stored in the `value`
  argument if the variable is defined and can be cast to type T.

RETURN VALUE

  Returns true if the environment variable was successfully stored in the value
  argument; false otherwise.

NAME

  get_random_bytes - generates a number of random bytes
                     (not suitable for cryptographic use)

SYNOPSIS

  fn get_random_bytes(n: usize) -> Bytes;

DESCRIPTION

  Generates n uniformly random bytes and returns them. The bytes are not
  necessarily cryptographically strong.

RETURN VALUE

  The n randomly sampled bytes.

NAME

  log - logs a string

SYNOPSIS

  fn log(line: &str);

DESCRIPTION

  Writes the input line out to the system log (defined as stderr).

RETURN VALUE

  N/A

NAME

  panic - exits the program due to an error

SYNOPSIS

  fn panic();

DESCRIPTION

  Closes the program and network connections associated with the program

RETURN VALUE

  N/A

NAME

  exit - exits the program cleanly

```

```
SYNOPSIS

  fn exit(exit_code: u32);

DESCRIPTION

  Closes the program and network connections associated with the program,
  returning the specified exit code.

RETURN VALUE

  N/A

NAME

  arm_timer - adds a timer event to the queue

SYNOPSIS

  fn arm_timer(k: u8, t: usize);

DESCRIPTION

  Creates a timer with number k that expires t seconds from the calling time.
  If called with a number of a timer that was already set, this function resets
  the timer.

RETURN VALUE

  N/A

NAME

  disarm_timer - disables a set timer

SYNOPSIS

  fn disarm_timer(k: u8);

DESCRIPTION

  Disables the kth timer, if it was previously armed. Otherwise, this function
  has no effect.

RETURN VALUE

  N/A

NAME

  get_timer - get the number of the last timer that expired

SYNOPSIS

  fn get_timer() -> u8;

DESCRIPTION

  On a timer expired event, this function can be used to get the number of the
  timer that expired.

RETURN VALUE

  The number of the last timer that expired.

NAME

  concatenate - join two byte objects together

SYNOPSIS

  fn concatenate(b1: &Bytes, b2: &Bytes) -> Bytes;

DESCRIPTION

  The output is a copy of b1 and b2 joined together in sequence.

RETURN VALUE

  b1 || b2

NAME

  get_buffer - gets a new mutable buffer

SYNOPSIS

  fn get_buffer(capacity: usize) -> BytesMut;

DESCRIPTION

  Returns an empty BytesMut container with the specifier capacity.

RETURN VALUE

  An empty BytesMut container.

SECTION - PROTOCOL MESSAGE MANIPULATION
==

NAME

  create_fields - create a new empty field object

SYNOPSIS

  create_fields() -> Fields;

DESCRIPTION

  Creates an initialized, empty Fields object. The Fields object is used to
  store and retrieve message field values by name for message formatting.

RETURN VALUE

  A new Fields object.

NAME

  set_field - set a field value

SYNOPSIS

  set_field<T>(fields: &mut Fields, name : &str, value: T);

DESCRIPTION

  Sets the field with the given name to by of type T and the given value.

RETURN VALUE

  N/A

NAME

  get_field - get a field value

SYNOPSIS

  get_field<T>(fields: &Fields, name: &str, value: &mut T) -> bool;

DESCRIPTION

  Gets the value of the field with the specified name and type and stores it in
  the value argument. Fails on type mismatch or if the name was not set.

RETURN VALUE

  true if the value fetch was successful; false otherwise.

NAME

  get_field_size - gets the size of a field in a message format.

SYNOPSIS

  get_field_size<T>(format: &MsgFormat, name: &str) -> usize;

DESCRIPTION

  Returns the statically-defined size of a field with the given name.

RETURN VALUE

  The size of the field, or 0 if the field was not present or defined to have
  variable size.

NAME

  format - try to create a formatted byte string

SYNOPSIS

  format(format: &MsgFormat, fields: &Fields) -> (bool, Bytes);

DESCRIPTION

  Attempts to format a byte string according to the specified message format and
  included field values.

RETURN VALUE

  (true, b) for a formatted byte string b if formatting was successful.
  (false, empty bytes) if formatting was not successful.

NAME

  parse - try to parse a byte string into the specified fields.

SYNOPSIS

  parse(format: &MsgFormat, data: &bytes) -> (bool, Fields);

```

```
DESCRIPTION

  Inverse function of format. Organizes the byte string into fields that can
  be retrieved with the get_field function.

RETURN VALUE

  For the first return value, true if the message could be successfully parsed;
  false otherwise. If the parse was successful, the returned Fields object will
  contain the parsed fields. Otherwise, it will be empty.

SECTION - CRYPTOGRAPHIC FUNCTIONS
==

We assume that ProtoSpec supports a standard set of cryptographic
functionalities, for example, those specified in the RustCrypto library
<https://github.com/RustCrypto>.
```

Listing 7: Listing of standard library functions

B.2 Implementation Details

Here we give brief, disparate remarks on implementation details related to the standard library and Proteus runtime.

Proteus programs must run using a fixed amount of memory for execution safety; however, some of the standard library functions have seemingly dynamic behavior. The standard library implementation must either (1) statically allocate all needed memory at initialization, or (2) monitor used memory, reallocating when needed but never exceeding a threshold.

Some standard library functions have the capability to *block* execution. Specifically, the network I/O function `buffer_pop()` exposes a parameter that causes execution to block until data is received. Blocking behavior is somewhat at odds with Proteus's event-driven model; we assume that in the case of `buffer_pop()` that the blocking call "intercepts" incoming EV-NET events out-of-order. More general issues still exist, for example, if a connection is closed during a blocking call, which may lead to a deadlock. We are still exploring ways to achieve a balance between different network programming paradigms that leads to easy programming.

C Proteus Grammar

The parsing expression grammar (PEG) [7] recognizing the Proteus language is given in Listing 8. The grammar is written for the `pest` library [13], which is a Rust package used for implementing performant parsers from PEGs.

```
WHITESPACE = _{ " " | "\t" | NEWLINE }
COMMENT = _{ "//" ~ (!"\n" ~ ANY)* | BLOCK_COMMENT }
BLOCK_COMMENT = _{ "/*" ~ (!("*/") ~ ANY)* ~ "*/" }

identifier = @{ (("_"|ASCII_ALPHA)~("_"|ASCII_ALPHANUMERIC)*) }
dotted_identifier = { identifier ~ ("." ~ identifier)* }
compound_identifier = { "(" ~ dotted_identifier ~ ("," ~ dotted_identifier)+ ~ ")"  }
basic_or_compound_identifier = { dotted_identifier | compound_identifier }

numeric_type = { "u8" | "u16" | "u32" | "u64" | "i8" | "i16" | "i32" | "i64" }
basic_type = { numeric_type | "bool" | "char" }
concrete_array_type = { "[" ~ type ~ ";"  ~ numeric_literal ~ "]" }
dynamic_array_type = { "[" ~ type ~ ";" ~ "*" ~ "]" }
flexible_array_type = { "[" ~ type ~ ";" ~ disj ~ "]" }
array_type = { concrete_array_type | dynamic_array_type }
custom_type = { identifier }
type = { basic_type | array_type | custom_type }
compound_type = { "(" ~ type ~ ("," ~ type)+ ~ ")"  }
basic_or_compound_type = { type | compound_type }

template_type = { "<" ~ basic_or_compound_type ~ ("," ~ basic_or_compound_type)* ~ ">" }

numeric_literal = @{ ASCII_DIGIT+ }
hex_literal = @{ "0x" ~ ASCII_HEX_DIGIT+ }
typed_numeric_literal = ${ hex_literal | (numeric_literal~numeric_type) }

string_literal = ${ "\"" ~ inner ~ "\"" }
inner = @{ char* }
char = {
  !("\"" | "\\") ~ ANY
  | "\\" ~ ("\"" | "\\" | "/" | "b" | "f" | "n" | "r" | "t")
  | "\\" ~ ("u" ~ ASCII_HEX_DIGIT{4})
}

basic_literal = { "true" | "false" | typed_numeric_literal | string_literal
  | numeric_literal}

array_literal = { "[" ~ literal ~ ";"  ~ numeric_literal ~ "]" }
literal = { basic_literal | array_literal | compound_literal }
compound_literal = { "(" ~ literal ~ ("," ~ literal)+ ~ ")"  }
basic_or_compound_literal = { basic_literal | compound_literal }

function_literal = { identifier ~ template_type? ~ (("(" ~ disj? ~ ")") | compound_disj) }

loc = { (dotted_identifier ~ "[" ~ disj ~ "]") | basic_or_compound_identifier }

disj = { (conj ~ ("||" ~ conj)*) }
compound_disj = { "(" ~ disj ~ ("," ~ disj)+ ~ ")"  }
basic_or_compound_disj = { disj | compound_disj }
conj = { bit_or ~ ("&&" ~ bit_or)* }
bit_or =  { bit_xor ~ ("|" ~ bit_xor)* }
bit_xor = { bit_and ~ ("^" ~ bit_and)* }
bit_and = { equal ~ ("&" ~ equal)* }
equal = { rel ~ (("==" | "!=") ~ rel)* }
rel = { bit_shift ~ ( ( "<="  | "<" | ">=" | ">" ) ~ bit_shift)* }
bit_shift = { sum ~ (("<<" | ">>" ) ~ sum)* }
sum = { product ~ (("+" | "-") ~ product)* }
product = { unary ~ (("*" | "/" | "%") ~ unary)* }
unary = { ("&" ~ unary) | ("!" ~ unary) | ("-" ~ unary) | factor }
factor = { function_literal | format_comprehension | "(" ~ disj ~ ")" | loc | literal }

block = { "{" ~ (stmt ~ ";")* ~ (stmt ~ ";"?)? ~ "}" }

match_arm = {
  (basic_or_compound_literal|basic_or_compound_identifier) ~
  "=>" ~ (basic_or_compound_disj|block)
}

match_stmt = { "match" ~ basic_or_compound_disj ~
    "{" ~ match_arm ~ ("," ~ match_arm)* ~ ","? ~ "}"
}

if_stmt = { "if" ~ basic_or_compound_disj ~ block ~
  ("else" ~ "if" ~ basic_or_compound_disj ~ block)* ~
  ("else" ~ block)?
}

assignment_rhs = { if_stmt | match_stmt | block | basic_or_compound_disj }

assignment_stmt = { (loc ~ "=" ~ assignment_rhs ) }
decl_stmt = { "let" ~ "mut"? ~ basic_or_compound_identifier ~
  ":" ~ basic_or_compound_type ~ ("=" ~ assignment_rhs)? }

return_stmt = { "return" ~ basic_or_compound_disj? }

format_tuple = { "(" ~ string_literal ~ "," ~ basic_or_compound_disj ~ ")" }

format_comprehension = {
  "format!" ~ "[" ~ format_tuple ~ ("," ~ format_tuple)* ~ ","? ~ "]"
}

for_stmt = {
  "for" ~ identifier ~ "in" ~ numeric_literal ~ ".." ~ "="? ~ numeric_literal ~
  block
}

stmt = {
  | decl_stmt
  | assignment_stmt
  | if_stmt
  | match_stmt
  | for_stmt
  | return_stmt
  | block
  | basic_or_compound_disj
}


handler_stmt = {
  "SET_HANDLER" ~ "(" ~ ("*" | identifier) ~ "," ~ identifier ~ ")"
}

handler_part = { (handler_stmt ~ ";")* }

msg_format_part = { (msg_format_stmt ~ ";")* }
msg_field_name = { "NAME" ~ ":" ~ identifier }
msg_field_type = { "TYPE" ~ ":" ~ (basic_type | array_type | flexible_array_type) }
msg_field = { "{" ~ msg_field_name ~ ";" ~ msg_field_type ~ "}" }
msg_format_stmt = { "DEFINE" ~ identifier ~ msg_field ~ ("," ~ msg_field)* }

global_part = { ("GLOBALS" ~ "{" ~ (decl_stmt ~ ";")* ~ "}")? }

fn_def = { "fn" ~ identifier ~ "(" ~ ")" ~ block }
fn_part = { fn_def* }

psf = {
  SOI ~
  handler_part ~
  msg_format_part ~
  global_part ~
  fn_part ~
  EOI
}
```

Listing 8: Parsing expression grammar recognizing the Proteus language.