usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Balboa: Bobbing and Weaving around Network Censorship

Marc B. Rosen, James Parker, and Alex J. Malozemoff, *Galois, Inc.*

https://www.usenix.org/conference/usenixsecurity21/presentation/rosen

This paper is included in the Proceedings of the
30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the
30th USENIX Security Symposium
is sponsored by USENIX.

Balboa: Bobbing and Weaving around Network Censorship

Marc B. Rosen
Galois, Inc.

James Parker
Galois, Inc.

Alex J. Malozemoff
Galois, Inc.

Abstract

We introduce Balboa, a link obfuscation framework for censorship circumvention. Balboa provides a general framework for tunneling data through existing applications. Balboa sits between an application and the operating system, intercepting outgoing network traffic and rewriting it to embed data. To avoid introducing any distinguishable divergence from the expected application behavior, Balboa only rewrites traffic that matches an externally specified *traffic model* pre-shared between the communicating parties. The traffic model captures some subset of the network traffic (e.g., some subset of music an audio streaming server streams). The sender uses this model to replace outgoing data with a pointer to the associated location in the model and embed data in the freed up space. The receiver then extracts the data, replacing the pointer with the original data from the model before passing the data on to the application. When using TLS, this approach means that application behavior with Balboa is *equivalent*, modulo small (protocol-dependent) timing differences, to if the application was running without Balboa.

Balboa differs from prior approaches in that it (1) provides a framework for tunneling data through arbitrary (TLS-protected) protocols/applications, and (2) runs the unaltered application binaries on standard inputs, as opposed to most prior tunneling approaches which run the application on non-standard—and thus potentially distinguishable—inputs.

We present two instantiations of Balboa—one for audio streaming and one for web browsing—and demonstrate the difficulty of identifying Balboa by a machine learning classifier.

1 Introduction

The continued increase in Internet censorship across the world [1] has spurred the research community to develop censorship resistant systems (CRSs). These systems seek to allow a party within a monitored region to access censored content. In this work we focus specifically on CRSs based on *link obfuscation*. Link obfuscation aims to allow communication between two or more parties such that a censor monitoring (or manipulating) the network should not be able to detect such communication. There are a wide array of such tools (see Khattak et al.'s systemization [18] for a detailed summary of CRSs—including those that focus on link obfuscation—as of 2016) but they tend to fall into two main categories: *look-like-nothing* approaches, which avoid detection by being hard to classify as any particular type of traffic, and *look-like-something* approaches, which generate traffic designed to look like a protocol the censor does not wish to block. Look-like-something approaches, themselves, generally fall within two camps: *mimicry* and *tunneling*.

In the *mimicry* approach, a CRS produces network traffic designed to closely match the network traffic of an existing implementation of the target protocol. Any differences between this implementation and the CRS constitute distinguishing features that a sufficiently powerful censor could target. In practice, CRSs that take the mimicry approach tend to produce network traffic with such distinguishing features. This led Houmansadr et al. [17] to conclude that mimicry approaches are "fundamentally flawed."

An alternative approach called *tunneling* directly runs a concrete implementation of the target protocol, addressing the key concern of the mimicry approach. To send data in this approach, the standard implementation is run with a non-standard input, which embeds the data to be sent. For example, DeltaShaper [3] is a CRS that tunnels user data through Skype by encoding data as simulated camera and microphone inputs. The receiving party extracts the data by processing the call's output. Even though Skype data is encrypted, Wright et al. [26] found that the sizes and timings of packets alone can still leak information about the plaintext. As a result, a censor who can observe the encrypted packets can determine that the inputs to the Skype call are not standard inputs (e.g., the audio sounds like a dial-up modem instead of somebody talking). While Barradas et al. [3] implemented techniques in DeltaShaper to try to mitigate this information leak, the same authors later showed [4] that the mitigation was insufficient, and that (given labeled training data) a censor could discover when DeltaShaper was in use.

In summary, mimicry approaches can be detected because a CRS is unlikely to perfectly match a concrete implementation of the target protocol, and tunneling approaches can be detected because the concrete implementation of the target protocol is not run on standard inputs.

1.1 Our Approach

In this work, we introduce Balboa, a link obfuscation framework that aims to address the above concerns by running a *concrete application* implementing the target protocol on *standard inputs*. The key insight is that if the communicating parties know *a priori* some subset of the expected network traffic then that network traffic does not actually need to be sent, and could instead be replaced by arbitrary data. Balboa handles this by sitting between the concrete application and operating system, intercepting outgoing and incoming network data. In addition, the communicating parties have a pre-shared *traffic model* which contains some subset of the expected network traffic. Whenever Balboa on the sender side intercepts outgoing data contained in the model, it replaces said data with a *pointer* to the appropriate location in the model; Balboa on the receiver side then "inverts" this procedure by using its own model to replace the pointer with the actual data.

This approach has two key features: (1) the applications themselves act *exactly the same* as if Balboa were not running, and (2) the sender can insert arbitrary data into the "freed up" bytes, since the pointer is much smaller than the data that would have been sent. Importantly, Balboa *does not* assume that the traffic model is complete (or even accurate). Instead, Balboa first checks to see whether outgoing traffic matches the traffic model before performing rewrite operations. If part of the outgoing traffic does not match the model, Balboa does not modify it.

Balboa relies on TLS to hide the fact that the application data itself changed—all other (non-timing) characteristics of the traffic (e.g., TLS record length) remain identical. In particular, Balboa uses debugging features found in most TLS libraries to extract the session key and uses this to decrypt and re-encrypt the intercepted TLS traffic.

Because Balboa only makes changes to the plaintext content of TLS-protected network traffic, the fact that Balboa is running is indistinguishable to a censor lacking the session key for the connection, modulo a small protocol-dependent timing delay. Importantly, unlike many censorship circumvention approaches, Balboa does not modify the TLS handshake at all. This makes it much more difficult for the many censors which have historically relied on TLS handshake fingerprinting [13] to identify Balboa.

As a concrete example, consider the setting where a client C streams music from an audio streaming server S. The two parties would like to use this channel to send covert data from S to C. Balboa assumes a trusted setup phase where both C and S agree on a symmetric key and playlist of songs; that is, C knows *a priori* some subset of the songs S will stream. On launch, S starts the audio streaming application (e.g., Icecast) with Balboa, which intercepts outgoing traffic produced by the application and replaces the audio data with a pointer to where in the playlist the given audio data corresponds. On C's end, Balboa intercepts incoming traffic to C's listening application (e.g., VLC) and replaces the data with the actual audio data (which C knows, as this info was pre-shared), before passing on the data to the listening application.

Because network reads/writes originate within the (unmodified) application, their lengths and behavioral characteristics—modulo slight timing differences introduced by the processing required by Balboa—*exactly* match that of the application running without Balboa. The Balboa framework also provides a generic signaling technique to allow clients and servers to covertly mutually authenticate each other. Because the server runs an unmodified application binary, it could even be providing a legitimate service (such as a public audio streaming channel in the above example). Normal clients can successfully connect to the Balboa-enabled server as usual, without detecting anything about its circumvention capability.

Table 1 provides a comparison of Balboa to several mimicry and tunneling approaches (see also our discussion of related work in §7). While Balboa is not the first CRS to use standard input to drive the channel, it is the first to provide a flexible framework while achieving significantly higher goodput than prior work.

Balboa, however, is not a panacea. It specifically relies on TLS and the fact that TLS is not being man-in-the-middled by a censor. In environments where TLS is expressly forbidden or actively man-in-the-middled (which occurs from time to time [8]), Balboa may be detectable. Also, like most CRSs, Balboa does not address the channel setup phase, the phase most often attacked by censors [23]. However, despite these drawbacks, Balboa offers a flexible framework for building circumvention channels, one which generalizes prior approaches and which can be adjusted, by varying the model or application, to the characteristics of the network environment in which it is being deployed.

1.2 Our Contributions

To summarize, we make the following contributions:

- We introduce Balboa, an open-source framework for censorship circumvention which embeds data in TLS-protected traffic generated by an *unmodified* application binary. Balboa is designed to make it easy to spin-up new instantiations for different applications and protocols. While the high level idea of Balboa is relatively straightforward, realizing an implementation is quite

Scheme	Approach	Unmodified Binary	Standard Input	Does Not Require Encryption	Flexible	Goodput
FTE [9]	Mimicry	N/A		✓	✓	1.9–42 Mbps[*]
DeltaShaper [3]	Tunneling	✓				2.56 kbps
Freewave [17]	Tunneling	✓				19 kbps
Castle [15]	Tunneling	✓	✓			190 bps
Rook [24]	Tunneling	✓	✓	✓		26-34 bps
Protozoa [5]	Tunneling		✓			160–1400 kbps
Balboa (audio streaming) Balboa (web browsing)	Tunneling	✓	✓		✓	145 kbps[**] 8 Mbps[†]

[*] This range corresponds to an HTTP format on the low-end, and an SSH format on the high-end.
[**] When streaming an audio file encoded at 148 kbps.
[†] When downloading a video with bandwidth capped at 8 Mbps. In general, the goodput depends heavily on the assets being accessed by the client, and may be much lower, or higher, than the number reported here.

Table 1: Comparison of several look-like-something link obfuscation schemes versus Balboa. "Unmodified Binary" denotes those schemes that run an unmodified implementation of the target protocol under-the-hood, "Standard Input" denotes those schemes that run on input that matches the expected input of the implementation, "Does Not Require Encryption" denotes those schemes that do not rely on encryption for undetectability, "Flexible" denotes those schemes which provide frameworks for supporting various applications/protocols, and "Goodput" denotes the covert throughput of the scheme.

complicated due to the need to minimize the effect Balboa has on packet timings alongside avoiding subtle attack vectors; see §2 for the architecture description and §4 for implementation details.

- We describe two instantiations of Balboa (§3): one for audio streaming and one for web browsing. In the audio streaming case, Balboa is able to replace all of an audio stream with arbitrary data—when streaming an Ogg-Vorbis file with a bitrate of 148 kilobit/second this corresponds to a 148 kilobit/second channel. In the web browsing case, Balboa is able to replace all content transmitted via `HTTP` including HTML, CSS, image, audio, and video files.

- We provide a security analysis (§5) and evaluation (§6) of Balboa against both passive and active adversaries.

Because the Balboa framework is extensible to new protocols and new applications, we believe that its deployment could help enable censorship circumvention providers to evolve more quickly in response to developments of a censor's capabilities.

2 Architecture

Balboa provides a bidirectional[1] channel-based censorship circumvention framework for TLS-protected channels. The framework needs to be instantiated for specific applications/protocols. In this work we demonstrate two such instantiations: (1) audio streaming and (2) web browsing. We assume the censor monitors the network traffic between the two communicating parties and can use both passive and active attacks to identify the channel. We also assume a trusted setup phase where the communicating parties agree on some shared information: a symmetric key and a *traffic model* which encodes the particular plaintext data to replace (cf. §2.1).

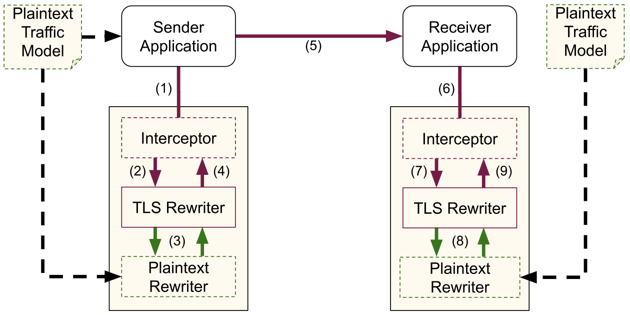

Figure 1: The Balboa architecture. Yellow denotes Balboa components, red denotes TLS-encrypted data, and green denotes plaintext data. Boxes with dashed lines denote instantiation-specific components of Balboa.

Figure 1 shows the overall Balboa architecture. Balboa sits between an application and the network, intercepting outgoing/incoming TLS streams (Steps 1 and 6). The intercepted stream is then fed to a *TLS rewriter* (Step 2), which extracts the underlying plaintext of the TLS stream. For outbound traffic, the plaintext is fed to a protocol-specific *plaintext rewriter* that replaces the plaintext with a pointer to the appropriate location in the traffic model and fills in the leftover bytes with any covert data to send (Step 3). For inbound traffic, the plaintext is again fed to a protocol-specific plaintext rewriter that extracts the covert data and replaces the model pointer

[1] The bidirectionality is dependent on the application and network protocol used; for example, our audio streaming instantiation only achieves a unidirectional channel.

with the pointed-to data (Step 8). The TLS rewriter then re-encrypts the (transformed) plaintext data before feeding it back to the calling application (Steps 4 and 9).

In what follows we walk through this architecture in more detail, discussing the relevant implementation considerations along the way.

2.1 Traffic Models

Balboa makes use of *traffic models* that capture some subset of the expected plaintext network traffic between the communicating parties, and Balboa assumes that the communicating parties have access to compatible models. While the traffic model structure is specific to a particular Balboa instantiation, *within a given Balboa instantiation* the particular traffic model may differ between each pair of communicating parties. For example, client C_1 talking to audio streaming server S may use a different traffic model than client C_2 talking to the *same* server S. We discuss the traffic model structures for our instantiations in §3.

Importantly, the traffic model need not be a model of the *entire* interaction between the parties. This allows parties to communicate N bytes of data without needing the model to be of size $O(N)$. In addition, for bidirectional instantiations of Balboa, the traffic model could even be *learned* by the client, who could then update the server on the traffic model to use. For example, for web browsing—assuming some base traffic model—the client could collect a set of assets available on the server to use as its traffic model and inform the server on which assets to use going forward.

Additionally, the traffic model need not be static. For example, in the audio streaming setting, the server could dynamically generate audio from a seed and send that seed along with covert data to the client. The Balboa client could then replicate the dynamically-generated music that the server is sending. For web browsing, the server could be running a blog in which the articles are automatically generated from some seed (enabling them to be replaced with covert data for a Balboa client), while comments (which can be posted by arbitrary users) can be sent through unmodified.

2.2 Potential Deployment Scenarios

Due to Balboa's use of both a shared key and traffic model between the communicating parties, we believe Balboa's ideal deployment scenario is one in which a small trusted set of clients (such as a select set of journalists) are aware that a given server is Balboa-enabled. Recall that the Balboa-enabled server functions exactly as a server would without Balboa running, and thus this server could provide a service to the public at large. For example, the server could be a programming blog, providing the set of trusted clients a reasonable alibi for accessing the server.

2.3 Intercepting TLS Data

Balboa needs to intercept outgoing TLS data (Step 1, Figure 1) in order to rewrite the underlying plaintext before sending it to the receiver, and needs to intercept incoming TLS data (Step 6, Figure 1) to extract the covert data before sending the (original) plaintext on to the application. In Balboa, we use dynamic linker features to manipulate network traffic by intercepting calls to `libc` system call wrappers. This approach has two distinct advantages over other approaches: (1) since we are directly running an unmodified version of the application, the network traffic characteristics exactly match those of the application (besides slight timing differences), and (2) the approach is more amenable to adding support for additional applications (or additional application versions) since we can largely treat the application as a black box and do not depend on the application's source code.

2.3.1 Implementing Dynamic Library Injection

On Linux, Balboa takes advantage of the `LD_PRELOAD` option to `ld.so` to perform dynamic library injection[2]. The dynamic linker causes calls to `read()`, `write()`, `sendmsg()`, `writev()`, among others, to be captured by Balboa instead of performing their usual action inside the C standard library. Balboa's injection library is tuned to the particular protocol to specify (1) which network connections to intercept (e.g., based on IP address or port number), and (2) which plaintext rewriter to use for the particular protocol/application.

This approach does have several subtle considerations that complicate the implementation, which we discuss below.

Performance considerations. Because Balboa performs in-band network traffic rewriting, it operates on the "hot path", and thus any delay imposed by Balboa's processing may be directly visible to a censor monitoring the connection. Thus, it is vital that Balboa is as efficient as possible. As a result, Balboa's rewriter code is designed to be low-latency. We achieve this primarily by avoiding memory allocation alongside implementing a high-performance logging library (see §4), among other standard techniques. We discuss specific performance numbers in §6.2.

Recursive calls. Balboa may invoke `libc` functions as part of its operation. If such a call occurs within an *intercepted* `libc` function this could cause an infinite loop. Balboa mitigates this by maintaining a flag in thread-local storage to see whether control has already entered an injected function call. If so, then the `libc` routine that Balboa replaced is transparently called instead.

[2] Balboa additionally works on macOS using `DYLD_INSERT_LIBRARIES` (and other features of the macOS dynamic linker) instead of `LD_PRELOAD`.

Signal safety. Several functions that Balboa intercepts are considered *signal-safe* by the POSIX standard. As a result, an application might call any of these functions from inside a signal-handler. Balboa mitigates this issue via the same recursive call mechanism described above. That being said, Balboa is not perfectly signal-safe—more extensive testing and implementation work is necessary to ensure full signal safety.

Limitations of dynamic library injection. Because we use dynamic library injection, Balboa does not work on applications that do not use dynamic library calls to perform network operations (such as applications written in Go)[3]. In addition, because we only intercept POSIX (and Linux) network APIs, we restrict ourselves to Unix-like operating systems; in particular, we do not have Windows support for Balboa. However, this could potentially be added using DLL injection techniques; we leave this to future work.

2.4 Extracting TLS Key Material

In order for Balboa to manipulate TLS data it must first learn the TLS key material. It does so by taking advantage of debugging features available in most modern TLS libraries.

`SSLKEYLOGFILE`. When working with an application using GnuTLS, NSS[4], or Rustls[5], Balboa constructs a named pipe and passes it to the application using the `SSLKEYLOGFILE` environment variable. The application sends a serialized form of the TLS master secret to Balboa which can use it for further processing.

OpenSSL. OpenSSL does not support the `SSLKEYLOGFILE` environment variable. Thus, when working with an application that dynamically links to OpenSSL, Balboa uses `LD_PRELOAD` to inject a shim over the `SSL_new()` function that configures a callback to receive the TLS key material. For applications that *statically* link to OpenSSL, we rely on the application itself to support `SSLKEYLOGFILE`; this is the case for many applications, including curl, among many others.

Because Balboa treats the application's TLS library as a gray-box—that is, the only requirement beyond using `libc` system call wrappers is that the TLS library supports dumping the TLS key material in some way—Balboa has a single TLS rewriter codebase that works with OpenSSL, GnuTLS, NSS, and Rustls. Since Balboa is very weakly-coupled to the application's TLS library, it makes it easy to extend support to additional applications, as well as additional TLS libraries. As an example, no code changes were required to get Balboa working for Rustls once we implemented GnuTLS support.

A significant benefit of extracting TLS key material from the library itself is that Balboa *does not modify the TLS handshake*. This prevents a whole class of attacks that censors commonly employ to detect CRSs [13]. One downside however is that Balboa cannot make any active changes to the TLS traffic until the key information has been emitted. Fortunately, every TLS library that we looked at releases the TLS master secret by the time a TLS Application Record is sent or received, which is sufficient for Balboa's needs.

2.5 Processing Intercepted TLS Data

Once Balboa has intercepted the TLS data, the next steps are to: (1) decrypt the data, (2) rewrite the resulting plaintext, and (3) re-encrypt the plaintext to either send over the wire or return to the application. We describe each of these steps in turn.

2.5.1 Decrypting TLS Data

Balboa decrypts incoming and outgoing TLS data (Steps 2 and 7, Figure 1) identically. How decryption works depends on the particular TLS version and cipher suite used. In particular, Balboa currently only supports TLS 1.2 and stream cipher suites (see §A and §B for a discussion on how we can support TLS 1.3 and non-stream cipher suites, respectively, although we leave the implementation to future work). To decrypt, Balboa scans the intercepted TLS data for Application Data records, ignoring other record types[6]. Once it has found an Application Data record, it reads the explicit nonce for the record (if there is one[7]). Armed with the explicit nonce, Balboa performs an unauthenticated decryption of the bytes. As these bytes are decrypted, they are sent to the plaintext rewriter for processing. After the payload has been processed, Balboa reads the (original) MAC of the incoming record, and checks that it is correct. If it is, Balboa generates a new MAC for the rewritten record, and if not, Balboa generates an invalid MAC. While the above gives the high-level idea, we discuss some subtleties with this approach in §2.5.4.

2.5.2 Rewriting the Plaintext

Given the extracted plaintext data, Balboa either rewrites the plaintext to make room for covert data (Step 3, Figure 1) or extracts the covert data and rewrites the plaintext to recover the original data (Step 8, Figure 1). Rewritten bytes are then forwarded on for re-encryption.

[3] We note that Balboa *still* works even if the TLS library is statically linked, as long as the TLS library supports extracting the TLS key material through the `SSLKEYLOGFILE` environment variable.

[4] Mozilla's TLS library, used in Firefox and Thunderbird, among other software.

[5] A TLS library written in Rust: https://github.com/ctz/rustls

[6] Balboa also looks for Alert records. If an Alert record is observed, Balboa transparently passes traffic to the application without modifying it.

[7] In TLS 1.2, the ChaCha20-Poly1305 cipher takes the approach that is standard in TLS 1.3 of having no explicit nonce sent over the wire.

How rewriting is performed is protocol (and possibly application) specific and must be designed on a per-protocol basis. This is the key point at which Balboa is configurable. We have implemented two instantiations of Balboa—audio streaming and web browsing—which we discuss in §3.

2.5.3 Re-encryption

The final step is to re-encrypt the plaintext before sending it either over the wire (Step 4, Figure 1) or to the application itself (Step 9, Figure 1). For the former case, we could simply re-encrypt using the extracted TLS master secret; however, this leaves open the possibility that a censor that man-in-the-middles the TLS connection could extract the user data. We thus re-encrypt using a key k' derived from the TLS master secret mk and the pre-shared key k. That is, $k' \leftarrow \mathrm{KDF}(mk\|k)$, where KDF is a key derivation function (BLAKE3 in our case). Besides this change, re-encryption operates the same for Steps 4 and 9.

2.5.4 Handling Partial Reads and Writes

In order to be as faithful to the application's behavior as possible, Balboa rewrites TLS data immediately upon intercepting a system call. If a system call returns an error (such as `EWOULDBLOCK`), then Balboa forwards that response on to the caller[8]. The immediate rewriting, however, results in several implementation complications, which we elaborate on below.

Handling partial writes. For performance purposes, TLS libraries optimistically try to `write()` as much data as possible. In practice, this means that Balboa gets to see at least one full TLS record in a single intercepted `write()`. However, if the application's TLS library attempts to write more bytes than there is room for in the kernel's buffer, then the kernel reports that only a partial write occurred. Balboa handles this by performing unauthenticated decryption until the MAC is received. Figure 2 provides an illustrated example, where it takes three `write()`s to emit a complete TLS record.

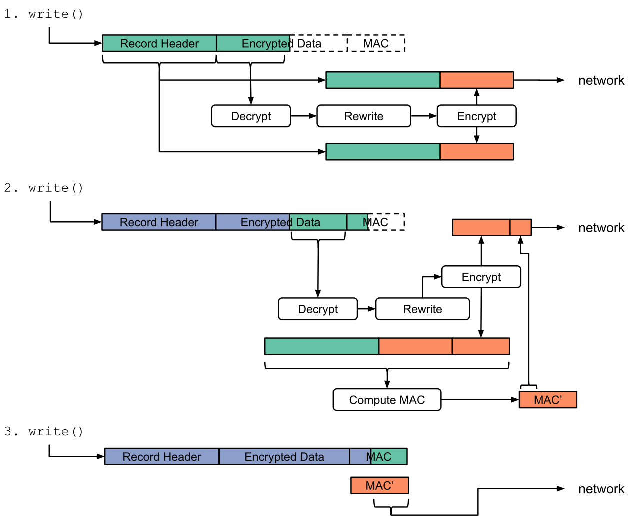

Figure 2: Processing outgoing TLS records. We consider a scenario where it takes three calls to the `write()` function for the application to write the full TLS record. Green denotes data written during a given `write()` call, purple denotes prior written data, and orange denotes data computed by Balboa.

Handling partial reads. Handling `read()`s is more complicated as *the number of bytes that `read()` returns may depend on censor-controlled network conditions*. As a result, unlike with `write()`s, where we know that we should see whole chunks at a time, with `read()`s a censor could manipulate the TCP connection such that each successful `read()` *only yields one byte*. In order to cope with this, we designed Balboa to be able to decide what byte to return to the application given only a single incoming byte alongside any previously observed traffic. In particular, when processing one byte at a time Balboa does not necessarily have access to the given TLS record's MAC (that is, it may not be contained in the data acquired for the particular `read()` function call made by the application), and so it cannot authenticate the TLS record until all bytes of the TLS record have been received. However, Balboa must provide *something* to the application on each `read()` call, and this something must be the re-encrypted plaintext data if the MAC is indeed correct. Balboa addresses this conundrum by *assuming* that the TLS record is valid, up until the last byte of the incoming MAC, providing an invalid value for the last MAC byte if it turns out that the incoming MAC was incorrect.

Figure 3 provides an illustrated example of how Balboa handles this. In the figure, the application makes three calls to the underlying `read()` function to read the full TLS record. In the first `read()`, Balboa has not yet received the MAC so cannot actually validate that the incoming TLS record is valid. It thus assumes it is, sending back the re-encrypted plaintext data to the application. In the second `read()`, Balboa receives a portion of the MAC. Again, it cannot assume the MAC is correct, but must provide the plaintext data alongside a portion of the MAC to the application. In this case, it computes the expected MAC (MAC$'$ in the figure) and passes the requisite portion of MAC$'$ to the application. Finally, in the third `read()`, Balboa receives the full MAC. It does an equality check between this MAC and its precomputed one: if these MACs are equal then the TLS record is valid, and Balboa sends the rest of the MAC on to the application. Otherwise, it sends the inverse of the last byte of the MAC to force the application to receive an invalid MAC (which is what the application would have received in the case where Balboa

[8] An alternative approach would be to perform multiple, e.g., `read()`s upon intercepting a `read()`. However, such an approach would potentially alter the TCP flow control in a sufficient way to be identifiable to a censor.

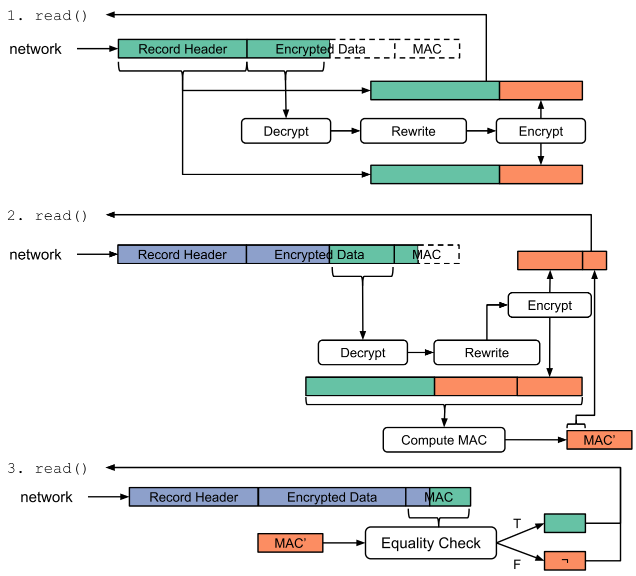

Figure 3: Processing incoming TLS records. We consider a scenario where it takes three calls to the `read()` function for the application to read the full TLS record. Green denotes data read during a given `read()` call, purple denotes prior read data, and orange denotes data computed by Balboa.

was not used).

2.6 Signaling

While the above steps allow parties to communicate using Balboa, an important step is for the parties to signal that they want to send/receive data in the first place. Balboa's signaling protocol allows the client and server to authenticate to each other, and is designed to be secure even against active probes made by the censor. We assume a secret key k has been pre-shared between the client and server, and use that—in conjunction with the TLS master secret—to derive a server key k_S and client key k_C.

2.6.1 How the Client Authenticates the Server

We re-use the existing certificate mechanism in TLS for the client to authenticate the server. Balboa clients are provisioned with a pinned public key certificate which is validated against the signature that the server sends in its Server Key Exchange TLS Handshake record. If the signature does not match, Balboa enters a transparent pass-through state and makes no modification to the traffic.

2.6.2 How the Server Authenticates the Client

The main challenge with signaling is for the server to authenticate the client. Balboa's protocol has two settings: (1) one in which it assumes that the server waits for a TLS Application Data record from the client before it sends any Application Data itself (as is the case in HTTP and other protocols), and (2) one in which it does not make this assumption.

Setting #1. When Balboa intercepts the client's first Application Data record, it leaves the plaintext untouched but replaces the MAC T with $T \oplus k_C$. Because the client has already verified the server's certificate as part of the key exchange, the censor is unable to distinguish between T and $T \oplus k_C$.

On the server, Balboa looks for the incoming client-sent Application Data record. Balboa then checks to see whether T or $T \oplus k_C$ is a valid MAC for the given record. If T is a valid MAC then the server assumes it is dealing with a non-Balboa client and enters a transparent pass-through state in which it performs no traffic modification. If $T \oplus k_C$ is a valid MAC, then signaling has succeeded and the rewriting stages can proceed as normal. If neither T nor $T \oplus k_C$ is a valid MAC, then Balboa passes an intentionally invalid MAC to the application and enters a transparent pass-through state. This case may occur if the censor has tampered with the connection, and by passing an invalid MAC to the application, Balboa causes it to respond as it would ordinarily to an invalid MAC.

Setting #2. If the client does *not* always send an Application Data record before the server, then Balboa proceeds as follows. Balboa on the server starts by transparently passing-through all outgoing Application Data records. When Balboa on the client receives these records, it also transparently passes them on to its application.

Balboa on the client performs the same operation as in Setting #1 on the first client-sent Application Data record. The client has now successfully completed its *outgoing* signaling efforts, and can now freely perform its normal plaintext rewriting and re-encryption processes on its outgoing traffic.

Because TLS (and TCP) are *full-duplex* protocols, there is no ordering relationship between client-to-server messages and server-to-client messages. The client can immediately proceed with its normal outgoing rewriting processes because the ordering constraints of TCP and TLS ensure that the server sees the Application Data message with the mangled MAC before it sees any messages sent after that. However, when a message comes in from the server, the client does not know whether that message was sent before or after the server saw the client's initial signaling message (in the form of the mangled MAC). As a result, the client does not know which key (namely, the standard TLS master secret or the derived re-encryption key) to use to decrypt the message. In addition, the client does not know whether to attempt to rewrite the message. To reiterate, the problem is the following: the client knows that the server is a Balboa-server, and it has told the server that it is a Balboa-client, but because incoming and outgoing messages have no ordering relationship, the client does not know whether the server *knows* that the client is a Balboa-client.

We solve this problem by having the server *acknowledge* that it received the client's initial signal. By having the server signal on its outgoing half of the duplex connection, any subsequent messages that it sends will arrive after its acknowledgement message. The server sends its acknowledgment message by replacing the MAC T on an outgoing Application Data record with $T \oplus k_S$. After sending this message, the server can start its normal Balboa operations. The client scans incoming Application Data records and performs the same check from above to find an Application Data record where the MAC is $T \oplus k_S$. After observing that message, the client is free to start rewriting incoming traffic from the server.

2.6.3 Security of Signaling

Several CRSs [7, 11] use a signaling technique based on Telex [27] which modifies the Client Random field of the TLS Client Hello message. While this change is indistinguishable to a censor, we do not use this technique because it would require us to re-implement many more pieces of TLS, and it would not work with our method of using TLS libraries' debugging features to extract TLS key material. In addition, Telex's signaling scheme does not offer forward secrecy: a censor can record network traffic and then, if at any point in the future they compromise the server, they would be able to go back through the recorded traffic and determine which connections used signaling.

In contrast, Balboa's signaling scheme inherits the forward secrecy of TLS: because the key material that Balboa uses to perform signaling is based on the ephemeral key of the TLS connection, any future compromise of the server would not reveal which connections had signaling. As a result, Balboa's shared covert signaling secret has the same security properties of Telex's public key: any client with the key can authenticate itself to the server, but the key does not allow any client (except for the sever) to identify which clients are using the key.

3 Balboa Instantiations

We have implemented two instantiations of Balboa: one for audio streaming and one for web browsing. We describe each in turn.

3.1 Audio Streaming

This instantiation supports Ogg Vorbis audio streaming traffic generated by an Icecast instance, with the client running a media player such as VLC[9]. The traffic model in this case is a single Ogg Vorbis file containing a concatenation of audio files.

Our rewriter works specifically for Ogg Vorbis traffic. Vorbis is a free and patent-free audio coding format (similar to MP3), and Ogg provides a container format for transmitting Vorbis streams. Icecast streams audio data to the client in an `HTTP/1.0` response which does not terminate. Ogg data itself is broken up into *pages*, each of which starts with an Ogg page header.

When the rewriter encounters an Ogg page, it determines whether the page is a candidate to be rewritten. A page is "rewriteable" if its body can be found in the source audio (i.e., the traffic model). Because an Ogg page might not fit entirely in a single TLS record, the rewriter sometimes has to decide whether a page is rewriteable before seeing it in its entirety. To get around this, the rewriter searches for audio data prefixed by what it has learned is in the body. It then uses the CRC32 checksum present in the original Ogg page to determine whether its guess of the audio data was correct. If the rewriter is unable to find a match, then it passes the page through unmodified.

If the rewriter does decide to rewrite an Ogg page, it replaces the Version field, which is normally a '0', with '*'[10]. This Version field signals to the receiver that it should attempt to rewrite the page. Next, the rewriter replaces the Bitstream Serial Number with the byte offset in the original audio data to which the data in the page corresponds. With the page header modified, the rewriter can replace the *entire* audio data component with covert data.

To rewrite an Ogg page on the receiver's side, we first check whether the page corresponds to covert data by checking that the Version field in the page header is the magic number '*'. If so, we extract the data and then replace it with the actual audio data using the location specified in the Bitstream Serial Number.

3.2 Web Browsing

This instantiation handles web browsing between a Firefox client and an Apache web server. We consider a traffic model in which the communicating parties share a directory of shared assets, such as HTML, images, video files, etc., and currently only support a unidirectional covert channel between the server and client.

Our rewriter works by parsing the `HTTP` request made by the client and storing the `HTTP` version, method, request URI, and headers. For example, a request to `https://example.com/dir/index.html` might have a version of `HTTP/1.1`, a `GET` method, `/dir/index.html` as the request URI, and header values for fields such as `Host`, `User-Agent`, and `Cookie`. Similarly, when the server receives the `HTTP` request, its rewriter parses and stores the request information. The server's rewriter waits until an `HTTP` response

[9] We have in addition validated that Balboa works for several other media players, including Audacious, Rhythmbox, etc.

[10] The choice of '*' is arbitrary.

is sent in reply. The rewriter parses the response to extract the status code and headers. If the status code indicates success and the request URI matches a shared asset, the body of the HTTP response is overwritten with covert data. In addition, to indicate to the client that rewriting has occurred, the third byte of the \r\n\r\n bytes between the response header and body is rewritten to 0xff. When the client receives the response, its rewriter parses the response. If the 0xff byte is present, the rewriter extracts the covert data and replaces it with the shared asset data.

HTTP allows partial downloads of files, which is often used for streaming audio or video files. Our HTTP rewriter supports this functionality by first checking for a 206 Partial Content status code. It then checks for Content-Range headers in the HTTP response and rewrites the shared asset with the appropriate position offset and length based on values in the range header.

4 Implementation

We have implemented Balboa alongside rewriters for audio streaming and web browsing. Balboa is implemented in Rust and is available at https://github.com/GaloisInc/balboa under an Apache 2.0/MIT dual-license.

Code organization. Balboa is comprised of several Rust crates that correspond to the components depicted in Figure 1:

- injection contains the core code and traits for injecting code into a shared library. A rewriter for Balboa needs to provide implementations of the associated traits for the particular application being injected.

- tlsRewriter contains code for rewriting the TLS records, and handles the decryption and re-encryption required. We have tested the rewriter with the following TLS libraries: OpenSSL, GnuTLS, and Rustls.

- rewriter contains the traits for implementing protocol-specific (plaintext) rewriters. An instantiation of Balboa needs to provide implementations of these traits.

Because Balboa must contend with partial reads (cf. §2.5.4), it can be tedious to manually write a state machine to perform byte manipulations. To remedy this, the rewriter and tlsRewriter components are written as coroutines. Coding in this style makes the rewriter implementations smaller and easier to develop.

For our audio streaming rewriter, we implemented the rewriter described in §3.1 and implemented wrapper code for injecting Balboa into VLC and Icecast. This wrapper code is reusable across multiple multimedia clients; for example, the wrapper code works for Audacious, Rhythmbox, MPlayer, and mpv, among others, without requiring a single line of code to be changed from the original VLC implementation.

Our web browsing rewriter proceeded similarly: we implemented the rewriter described in §3.2 and implemented wrapper code for injecting Balboa into Firefox and the Apache Web Server. We have additionally tested the Firefox injector on curl.

High speed logging. We developed a highly-performant logging library called Stallone (available at https://github.com/GaloisInc/stallone) to facilitate debugging Balboa both during implementation and for any potential future deployment. Due to the careful performance considerations required, we could not use existing logging libraries, as those add overheads of hundreds of microseconds per log entry, which would add noticeable delay to a running Balboa instance. We thus designed Stallone from scratch, taking inspiration from the NanoLog library [29]. Compared to NanoLog, Stallone does not rely on the CPU's timestamp counter, which might not be stable or valid in cloud environments or in any situation where the user does not know what exact CPU model they are working with [28]. In addition, Stallone uses stable identifiers for log record types and stores the mapping between log record identifiers and log record metadata (such as the message and line number) in a special section of the binary, eliminating the need for this information to be dumped online. Stallone is written in Rust and is capable of logging messages at an overhead of around 10 nanoseconds, and as such may be of independent interest.

5 Security Analysis

In this section we discuss the security of Balboa versus a censor that controls all network traffic between the communicating parties, and either passively monitors the network or actively manipulates, blocks, or injects packets. Due to the heavy systems engineering and subtle implementation details required in building Balboa—alongside a lack of security definitions within the field of censorship circumvention—we forgo a formal (i.e., "provable security") treatment of Balboa. Instead, given the relative simplicity of the cryptography inside Balboa, we focus more closely on the practical security of the implementation (and the timing channel that it produces).

Identifying the signaling protocol. Balboa's signaling protocol (cf. §2.6) replaces the original MAC of a TLS record with a one-time-pad of the MAC and a key derived from the TLS master secret and the pre-shared secret. Because the master secret is chosen pseudorandomly for each connection, and because the censor does not know the pre-shared secret, the new MAC is indistinguishable from the original to a censor.

However, Balboa's signaling protocol does leave open the possibility of a timing channel resulting from the need to compute the modified MAC and check equality when an invalid

MAC is encountered. We minimize this channel by precomputing the KDF as soon as the TLS master secret is known, reducing the online cost to a single XOR operation.

Manipulating the TLS channel. Balboa alters the TLS channel by replacing the plaintext data in a given TLS record. This replacement is indistinguishable from standard application traffic, assuming the security of TLS. However, due to restrictions on reading from the network (cf. §2.5.4), Balboa currently requires the use of a *stream* cipher suite. Thus, an active censor could force a particular cipher suite to be used, one that is not supported by Balboa. Thus, Balboa *only* operates for specific supported cipher suites, and otherwise operates in pass-through mode. This however leaves open the possibility of a denial-of-service attack where a censor actively enforces that only non-stream cipher modes are negotiated. We view such an attack as highly unlikely, given that 81% of TLS connections use stream cipher suites [2]. However, even in this case we can resort to supporting non-streaming modes as discussed in §B.

A sufficiently powerful censor may be able to man-in-the-middle the TLS connection and thus recover the covert data. Such attacks are not unrealistic [8]. While we cannot prevent such a censor from *identifying* that Balboa is in use, we prevent the censor from acquiring the covert data by re-encrypting it using a different key than the TLS master secret, as specified in §2.5.3.

Manipulating the application itself. A censor could try to use traffic manipulation or injection to force Balboa to enter an invalid state, producing behavior that is distinguishable from what the underlying application would have done. We carefully designed Balboa such that whenever it reaches a failure mode it reverts to pass-through mode such that any observer sees the underlying application behavior directly.

Identifying timing differences. The main difference between running the application with or without Balboa is the timing differences introduced by Balboa. We discuss the effects these timing differences have on classifying Balboa for audio streaming and web browsing in §6.

Identifying plaintext traffic model differences. A censor may try to identify Balboa by identifying differences between a particular traffic model and the baseline behavior of the network environment. As an example, if an audio streaming service streams the same song over and over the traffic pattern may differ sufficiently from other audio streaming services found on the network. Note that this attack is *external* to whether Balboa is deployed. That is, if the user's behavior varies significantly from behavior in the baseline network environment, a (sufficiently powerful) censor could detect this *whether or not Balboa was running at all*[11]. Thus, it is important to choose an appropriate traffic model instantiation for the particular deployment environment of Balboa, and this choice is one that needs to be made with the particular deployment environment in mind (e.g., the expected audio from a stream in Country A may differ from that in Country B).

Mimicking a client. A censor can try to determine a Balboa server by acting as a client. Assuming the censor does not have the required shared key to allow it to signal the server, the probability it successfully guesses the modified MAC and hence passes the signaling protocol is negligible.

Mimicking a server. A censor could also mimic a server, flagging any client that connects and produces a TLS record with an invalid MAC. Balboa thwarts this attack by verifying the public-key signature in the TLS connection against a pinned public-key. If this verification fails, then Balboa enters a pass-through mode, and the connection appears as normal to the server.

6 Evaluation

There are several avenues in which we evaluate Balboa: goodput and detectability. As discussed in §5, the ability for a censor to identify Balboa depends in part on any delay introduced by the tool over the baseline performance of the application. Thus, we focus our detectability evaluation on (1) producing microbenchmarks for the delay introduced by our two instantiations of Balboa, and (2) building classifiers for Balboa under various network latency settings to investigate whether a passive censor could detect Balboa.

6.1 Goodput

Because Balboa tunnels data through existing channels, the goodput of Balboa closely matches the throughput of the cover channel. In particular, for audio streaming we can replace 98% of cover data. Thus, when streaming an audio file encoded at X kbps ($X = 148$ or $X = 160$ is standard), we achieve a goodput of $.98 \cdot X$. For web browsing the computation is more complicated, as the percentage of data we can replace depends on the size of the cover asset. For example, if the asset is a blank HTML page we would achieve a very low goodput as there is no cover data to replace. However, for the "real-world" assets we have tested against (everything from single HTML pages to video files) we have found that we can replace 62–99% of cover data.

[11] Whether such an attack is feasible in practice depends heavily on the censor and what their false positive threshold is.

6.2 Microbenchmarks

As discussed in §5, Balboa introduces timing delays due to the processing required to rewrite TLS records and perform plaintext rewriting. To measure this delay, we ran Balboa on a standard laptop (Intel Core i7-6820HQ @ 2.7 GHz) for both our audio streaming and web browsing rewriters, tracking the cost of each rewrite operation for the sender and receiver. Each rewrite consists of decrypting the TLS data (encrypted under the `AES128-GCM-SHA256` or `AES256-GCM-SHA384` cipher suites), rewriting the plaintext, and re-encrypting—that is, a rewrite consists of all the processing done by Balboa upon intercepting a `read()` or `write()` from the underlying application.

Audio streaming. We gathered data while streaming a 10 second Ogg Vorbis audio file encoded at a bitrate of 148 kbps. For the sender (i.e., Icecast), we see an average delay of $122\mu s$. The delay seen on the receiver depends on the particular client application we are running; for example, for VLC we see an average delay of $36\mu s$ and for MPlayer we see an average delay of $20\mu s$. The additional delay imposed by the sender is largely due to (1) the CRC computation required when replacing the plaintext Ogg data, and (2) the computation of the GCM tag required when re-encrypting the plaintext.

Web browsing. We gathered data for two scenarios: using curl to download a video file and using Firefox to browse several links on a website containing a small subset of Wikipedia. For the sender (i.e., Apache), we see an average delay across both scenarios of roughly $89\mu s$. For curl we see an average delay of $90\mu s$, and for Firefox we see an average delay of $216\mu s$. The reason we see a higher delay than audio streaming is that the web browsing rewriter needs to store `HTTP` requests and thus requires allocations.

6.3 Timing Analysis

The introduced delays have security implications, as a sufficiently powerful censor may be able to classify Balboa-enabled traffic due to these delays. To determine the effect of these timing differences on the ability to classify Balboa, we ran several experiments on both our audio streaming and web browsing instantiations. For all of our experiments, we generated 130 `pcap` traces[12] between two Ubuntu 18.04 docker containers with and without Balboa enabled, using `tc` to control the average latency and its standard deviation in our simulated network. We generated traces for latencies between 0 ms (the “ideal” scenario) and 30 ms (the average latency in the United States[13]). We then built classifiers to try to distinguish the Balboa-enabled versus -disabled traffic, using `tcptrace` [21] to extract TCP statistics to train on. Our classifiers used random forests due to the success similar classifiers have had on distinguishing prior censorship circumvention systems [4]. For each scenario we trained classifiers using 10-fold stratified cross-validation using Scikit-learn [22].

Note that all of these experiments occurred in an idealized setting with no additional network traffic, and thus represent a *best case scenario for a censor*. In a real-world deployment successfully applying such a classifier would be much more difficult. We additionally ran our experiments with a number of additional clients whose network data was not analyzed by the classifier. This mimics a setting where the censor attempts to identify the use of Balboa among a larger set of innocuous traffic. We found—as expected—that this setting *decreases* the classifier's accuracy. As an example, for VLC with zero latency and four additional clients, we achieve a classifier accuracy of only 66%, versus 84% when a single client is used. Thus, to model the best case scenario for a censor we consider the single-client setting.

Audio streaming. For audio streaming we investigated the potential to identify Balboa running across four different media players: VLC, MPlayer, Audacious, and mpv. Each trace comprised of a client connecting to an Icecast server, streaming a 10 second song, and then exiting. Table 2 presents the accuracy, precision, and recall of our classifier for different latencies against these different media players. For each scenario we trained 1000 classifiers, with the presented results being the average and standard deviation of these classifiers.

We find at the extreme end—where there is zero latency in the simulated network—the classifier is able to distinguish Balboa traffic across the various media players with between 66% and 84% accuracy, with the key features being the average TCP window advertisement seen and data transmit time. This suggests that even the slight delay introduced by Balboa is enough to affect some network statistics (albeit in an unrealistic network setting). However, as we increase the realism of the network (by increasing the average latency as well as the standard deviation) we see the accuracy of the classifier quickly drop to a point where it is essentially no better than random guessing. This makes sense given that the delays introduced by Balboa become part of the noise of the network latency.

Another interesting feature of Table 2 is that the classifer accuracy varies depending on the media player. This suggests (perhaps not surprisingly) that different media players present different “network footprints”. To validate this, we additionally ran our classifier to see if we could distinguish two different media players, both with Balboa disabled. We

[12] We generated packet captures on an Intel Xeon Silver 4114 CPU @ 2.20GHz with 40 cores and 512 GB of memory. Doing so enabled us to generate packet captures more quickly, by running multiple trials in parallel. We (informally) verified that running parallel trials did not impact our results by comparing the results against a small number of non-parallel runs.

[13] According to https://www.verizon.com/business/terms/latency/ as of March, 2021.

Latency (ms)	Accuracy	Precision	Recall
0 ± 0	0.84 ± 0.07	0.87 ± 0.09	0.80 ± 0.11
5 ± 1	0.72 ± 0.08	0.76 ± 0.10	0.66 ± 0.13
5 ± 3	0.63 ± 0.09	0.67 ± 0.11	0.55 ± 0.14
10 ± 1	0.67 ± 0.09	0.71 ± 0.12	0.59 ± 0.13
10 ± 3	0.67 ± 0.09	0.70 ± 0.11	0.61 ± 0.14
10 ± 5	0.59 ± 0.09	0.61 ± 0.11	0.51 ± 0.14
30 ± 1	0.64 ± 0.09	0.67 ± 0.11	0.57 ± 0.14
30 ± 3	0.56 ± 0.09	0.57 ± 0.12	0.47 ± 0.15
30 ± 5	0.57 ± 0.09	0.58 ± 0.12	0.49 ± 0.14
30 ± 10	0.50 ± 0.09	0.50 ± 0.12	0.41 ± 0.14

(a) VLC

Latency (ms)	Accuracy	Precision	Recall
0 ± 0	0.68 ± 0.09	0.72 ± 0.12	0.60 ± 0.13
5 ± 1	0.50 ± 0.10	0.50 ± 0.12	0.41 ± 0.14
5 ± 3	0.51 ± 0.09	0.51 ± 0.12	0.41 ± 0.14
10 ± 1	0.55 ± 0.10	0.56 ± 0.12	0.47 ± 0.15
10 ± 3	0.53 ± 0.09	0.54 ± 0.12	0.45 ± 0.14
10 ± 5	0.52 ± 0.09	0.53 ± 0.12	0.42 ± 0.14
30 ± 1	0.53 ± 0.10	0.54 ± 0.13	0.44 ± 0.14
30 ± 3	0.49 ± 0.10	0.49 ± 0.12	0.40 ± 0.14
30 ± 5	0.50 ± 0.09	0.50 ± 0.12	0.41 ± 0.13
30 ± 10	0.49 ± 0.09	0.48 ± 0.12	0.39 ± 0.14

(b) MPlayer

Latency (ms)	Accuracy	Precision	Recall
0 ± 0	0.82 ± 0.05	0.85 ± 0.07	0.78 ± 0.08
5 ± 1	0.73 ± 0.06	0.75 ± 0.07	0.71 ± 0.09
5 ± 3	0.68 ± 0.06	0.70 ± 0.07	0.63 ± 0.09
10 ± 1	0.68 ± 0.06	0.70 ± 0.07	0.63 ± 0.10
10 ± 3	0.59 ± 0.07	0.61 ± 0.08	0.53 ± 0.10
10 ± 5	0.63 ± 0.07	0.65 ± 0.08	0.56 ± 0.10
30 ± 1	0.65 ± 0.06	0.68 ± 0.08	0.59 ± 0.10
30 ± 3	0.56 ± 0.06	0.57 ± 0.08	0.48 ± 0.10
30 ± 5	0.59 ± 0.07	0.61 ± 0.08	0.52 ± 0.10
30 ± 10	0.56 ± 0.07	0.58 ± 0.08	0.49 ± 0.10

(c) Audacious

Latency (ms)	Accuracy	Precision	Recall
0 ± 0	0.66 ± 0.09	0.69 ± 0.11	0.61 ± 0.13
5 ± 1	0.53 ± 0.09	0.54 ± 0.12	0.44 ± 0.14
5 ± 3	0.57 ± 0.09	0.58 ± 0.12	0.48 ± 0.14
10 ± 1	0.55 ± 0.09	0.57 ± 0.12	0.46 ± 0.14
10 ± 3	0.49 ± 0.09	0.48 ± 0.13	0.38 ± 0.14
10 ± 5	0.53 ± 0.09	0.54 ± 0.13	0.43 ± 0.14
30 ± 1	0.53 ± 0.09	0.53 ± 0.12	0.43 ± 0.14
30 ± 3	0.53 ± 0.09	0.54 ± 0.12	0.44 ± 0.14
30 ± 5	0.52 ± 0.10	0.53 ± 0.13	0.42 ± 0.15
30 ± 10	0.50 ± 0.10	0.49 ± 0.13	0.40 ± 0.14

(d) mpv

Table 2: Accuracy, precision, and recall of classifying Balboa-generated traffic versus baseline for various latency settings against various media players (VLC, MPlayer, Audacious, and mpv). Values are given in "mean ± standard deviation" format.

found that regardless of which media players we compared against, we achieved a 99–100% accuracy for all latency and standard deviation settings.

Web browsing. For web browsing we investigated the potential to identify Balboa using two different clients: curl and Firefox. For curl, each trace comprised of downloading a 13.6 MB video and then exiting. For Firefox, each trace comprised of a Selenium script accessing three different web pages scraped from Wikipedia, sleeping three seconds between each web page access. The assets for the three web pages totaled 8.9 MB and included HTML, javascript, image, and CSS files. As with the audio streaming case, Table 3 presents the accuracy, precision, and recall of our classifier across different latencies.

While the accuracies for web browsing tend to be higher than in the audio streaming case, this makes sense given the larger average delay introduced by Balboa. However, we reiterate that these results are for an *ideal* setting for the censor and the accuracies are still sufficiently low given the base rate fallacy.

7 Related Work

The literature is rich with different approaches to building censorship resistant systems (CRSs); we refer the reader to existing systematization of knowledge papers [18, 23] for a more thorough overview of the field than what we can provide here.

A CRS can be viewed as comprising two key components: *communication establishment* and *conversation*. Balboa addresses the second, which is where most of the academic literature has focused [18, §5.5]. In particular, Balboa corresponds to an "access-centric" scheme using the terminology of Khattak et al. [18]. We thus focus on such schemes in this section. Access-centric schemes can be subdivided into four[14] main categories, which we discuss in turn.

Mimicry. These approaches send data by mimicking some cover protocol. A representative example is *format-transforming encryption* [9] and its variants [10, 19], which operate by mapping ciphertexts to regular expressions or

[14] Khattak et al. [18] differentiate between *tunneling* approaches and *covert channel* approaches whereas we view these as the same, since any covert channel approach necessarily needs to "tunnel" its traffic through some existing application.

Latency (ms)	Accuracy	Precision	Recall
0 ± 0	0.66 ± 0.01	0.68 ± 0.01	0.61 ± 0.01
5 ± 1	0.69 ± 0.01	0.71 ± 0.01	0.64 ± 0.01
5 ± 3	0.69 ± 0.01	0.71 ± 0.01	0.64 ± 0.01
10 ± 1	0.66 ± 0.01	0.68 ± 0.01	0.61 ± 0.01
10 ± 3	0.66 ± 0.01	0.68 ± 0.01	0.60 ± 0.01
10 ± 5	0.65 ± 0.01	0.67 ± 0.01	0.58 ± 0.01
30 ± 1	0.69 ± 0.01	0.71 ± 0.01	0.66 ± 0.02
30 ± 3	0.67 ± 0.01	0.69 ± 0.01	0.62 ± 0.01
30 ± 5	0.62 ± 0.01	0.63 ± 0.01	0.55 ± 0.02
30 ± 10	0.57 ± 0.01	0.59 ± 0.01	0.49 ± 0.02

(a) Firefox

Latency (ms)	Accuracy	Precision	Recall
0 ± 0	0.96 ± 0.04	0.97 ± 0.05	0.95 ± 0.06
5 ± 1	0.71 ± 0.08	0.74 ± 0.11	0.65 ± 0.13
5 ± 3	0.66 ± 0.09	0.69 ± 0.12	0.58 ± 0.14
10 ± 1	0.79 ± 0.08	0.81 ± 0.09	0.77 ± 0.12
10 ± 3	0.70 ± 0.08	0.73 ± 0.10	0.64 ± 0.13
10 ± 5	0.63 ± 0.09	0.67 ± 0.12	0.56 ± 0.14
30 ± 1	0.86 ± 0.07	0.90 ± 0.08	0.82 ± 0.11
30 ± 3	0.62 ± 0.09	0.65 ± 0.12	0.55 ± 0.14
30 ± 5	0.62 ± 0.09	0.65 ± 0.12	0.55 ± 0.14
30 ± 10	0.67 ± 0.08	0.71 ± 0.11	0.58 ± 0.13

(b) curl

Table 3: Accuracy, precision, and recall of classifying Balboa-generated traffic versus baseline for various latency settings against various web clients (curl and Firefox). Values are given in "mean ± standard deviation" format.

context-free grammars that can encode, e.g., common network protocols like HTTP. The well-known "Parrot is Dead" paper [16] argues that such approaches are doomed to fail due to the difficulty of accurately mimicking a given protocol, although as discussed below (and in §1) even tunneling approaches suffer the same challenges.

Tunneling. These approaches try to avoid the "weaknesses" of the mimicry approach by running the actual application under-the-hood. Such approaches include Freewave [17], DeltaShaper [3], and Castle [15]. However, as several researchers have shown [14, 25, 26], even these approaches are susceptible to distinguishing attacks due the protocol distribution differences between the circumvention system itself and the underlying application when run on its own. This weakness appears inherent due to the inability to perfectly mimic the real world application behavior, or let alone know what such a "real world distribution" is in the first place. Balboa aims to minimize this gap by having such real world application behavior be a parameter specified by the user of the tool.

Concurrently with this work, Barradas et al. [5] introduced Protozoa, a tunneling approach which uses WebRTC as its communication medium. Protozoa shares several similarities to Balboa, in that it uses a form of rewriting to replace WebRTC traffic with user data. However, Protozoa is specific for WebRTC and requires modifications to the application source code, reducing the flexibility of the tool as application versions change, an attack vector exploited in practice [13]. It also does not replace the original video on the receiver side, potentially leaving the approach open to traffic analysis attacks.

Traffic manipulation. These approaches manipulate traffic to circumvent known censors. Recent approaches, such as Geneva [6], have proven successful at circumventing existing nation-state censors in several countries. However, the security model is fundamentally different (and weaker) than the one considered by both Balboa and tools in the mimicry and tunneling space: traffic manipulation approaches generally assume a weak censor that monitors traffic using a firewall or deep packet inspection device, whereas Balboa considers a potentially active censor that can apply more powerful capabilities. (Whether this more powerful censor is a realistic threat in practice is an orthogonal question.)

Destination obfuscation. These approaches, which include Tor and refraction networking protocols [7, 20, 27], focus on hiding the destination website from a censor, and borrow from the mimicry and tunneling literature in how they obfuscate the channel itself (e.g., Tor uses a "pluggable transport" infrastructure for link obfuscation).

Other related work. Several CRSs either require a specific version of an application (such as meek [12]) or otherwise need to mimic the TLS handshake in some way. However, Frolov and Wustrow [13] showed that this mimickry is often easily identifiable due to cleartext header information sent in the initial Client Hello message of a TLS connection—that is, this information must exactly match what an innocuous (and popular) application would produce. With this in mind, the authors introduce a tool, uTLS, for automatically mimicking existing TLS implementations.

Balboa avoids the need for a tool like uTLS by running the (unmodified) application under-the-hood and leaving the TLS handshake untouched. As long as the underlying protocol used by the application remains unchanged between versions, the application can be updated without affecting Balboa. In particular, unlike tools like meek [12], Balboa does not need to come bundled with a particular version of an application.

Acknowledgments

This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-19-C-0085. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force and DARPA. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

References

[1] Freedom on the net. https://www.freedomonthenet.org/explore-the-map?mapview=trend. Accessed February 10, 2020.

[2] The ICSI certificate notary. https://notary.icsi.berkeley.edu/. Accessed January 28, 2020.

[3] Diogo Barradas, Nuno Santos, and Luís Rodrigues. DeltaShaper: Enabling unobservable censorship-resistant TCP tunneling over videoconferencing streams. *Privacy Enhancing Technologies*, 2017(4):1–18, 2017.

[4] Diogo Barradas, Nuno Santos, and Luís Rodrigues. Effective detection of multimedia protocol tunneling using machine learning. In *USENIX Security Symposium*. USENIX, 2018.

[5] Diogo Barradas, Nuno Santos, Luís Rodrigues, and Vítor Nunes. Poking a hole in the wall: Efficient censorship-resistant internet communications by parasitizing on WebRTC. In *Computer and Communications Security*. ACM, 2020.

[6] Kevin Bock, George Hughey, Xiao Qiang, and Dave Levin. Geneva: Evolving censorship evasion strategies. In *Computer and Communications Security*. ACM, 2019.

[7] Cecylia Bocovich and Ian Goldberg. Slitheen: Perfectly imitated decoy routing through traffic replacement. In *Computer and Communications Security*. ACM, 2016.

[8] Catalin Cimpanu. Kazakhstan government is now intercepting all HTTPS traffic. *ZDNet*, July 2019.

[9] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Protocol misidentification made easy with Format-Transforming Encryption. In *Computer and Communications Security*. ACM, 2013.

[10] Kevin P. Dyer, Scott E. Coull, and Thomas Shrimpton. Marionette: A programmable network-traffic obfuscation system. In *USENIX Security Symposium*. USENIX, 2015.

[11] Daniel Ellard, Alden Jackson, Christine Jones, Victoria Ursula Manfredi, Timothy Strayer, Bishal Thapa, and Megan Van Welie. Rebound: Decoy routing on asymmetric routes via error messages. In *Local Computer Networks*. IEEE, 2015.

[12] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. Blocking-resistant communication through domain fronting. *Privacy Enhancing Technologies*, 2015(2), 2015.

[13] Sergey Frolov and Eric Wustrow. The use of TLS in censorship circumvention. In *Network and Distributed System Security*. The Internet Society, 2019.

[14] John Geddes, Max Schuchard, and Nicholas Hopper. Cover your ACKs: Pitfalls of covert channel censorship circumvention. In *Computer and Communications Security*. ACM, 2013.

[15] Bridger Hahn, Rishab Nithyanand, Phillipa Gill, and Rob Johnson. Games without frontiers: Investigating video games as a covert channel. In *European Symposium on Security & Privacy*. IEEE, 2016.

[16] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The parrot is dead: Observing unobservable network communications. In *Symposium on Security & Privacy*. IEEE, 2013.

[17] Amir Houmansadr, Thomas Riedl, Nikita Borisov, and Andrew Singer. I want my voice to be heard: IP over voice-over-IP for unobservable censorship circumvention. In *Network and Distributed System Security*. The Internet Society, 2013.

[18] Sheharbano Khattak, Tariq Elahi, Laurent Simon, Colleen M. Swanson, Steven J. Murdoch, and Ian Goldberg. SoK: Making sense of censorship resistance systems. *Privacy Enhancing Technologies*, 2016(4):37–61, 2016.

[19] Daniel Luchaup, Kevin P. Dyer, Somesh Jha, Thomas Ristenpart, and Thomas Shrimpton. LibFTE: A toolkit for constructing practical, format-abiding encryption schemes. In *USENIX Security Symposium*. USENIX, 2014.

[20] Milad Nasr, Hadi Zolfaghari, and Amir Houmansadr. The waterfall of liberty: Decoy routing circumvention that resists routing attacks. In *Computer and Communications Security*. ACM, 2017.

[21] Shawn Ostermann. Tcptrace. https://tcptrace.org, 2005.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer,

R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[23] Michael Carl Tschantz, Sadia Afroz, Anonymous, and Vern Paxson. SoK: Towards grounding censorship circumvention in empiricism. In *Symposium on Security & Privacy*. IEEE, 2016.

[24] Paul Vines and Tadayoshi Kohno. Rook: Using video games as a low-bandwidth censorship resistant communication platform. In *Workshop on Privacy in the Electronic Society*. ACM, 2015.

[25] Liang Wang, Kevin P. Dyer, Aditya Akella, Thomas Ristenpart, and Thomas Shrimpton. Seeing through network-protocol obfuscation. In *Computer and Communications Security*. ACM, 2015.

[26] Charles V. Wright, Lucas Ballard, Scott E. Coull, Fabian Monrose, and Gerald M. Masson. Uncovering spoken phrases in encrypted voice over IP conversations. *ACM Transactions on Information and System Security (TISSEC)*, 13(4):1–30, 2010.

[27] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman. Telex: Anticensorship in the network infrastructure. In *USENIX Security Symposium*. USENIX, 2011.

[28] Oliver Yang. Pitfalls of TSC usage. https://oliveryang.net/2015/09/pitfalls-of-TSC-usage/, 2017.

[29] Stephen Yang, Seo Jin Park, and John Ousterhout. Nanolog: A nanosecond scale logging system. In *2018 USENIX Annual Technical Conference*. USENIX, 2018.

A Supporting TLS 1.3

One nice feature of TLS 1.2 is that handshake records are distinct from application records, and are distinguished by early bytes in the record header. However, this is not the case for TLS 1.3: handshakes may occur at any time during a given connection and are distinguished by the last encrypted byte of the encrypted payload. As a result, when operating on incoming TLS 1.3 records, Balboa does not know whether the record should be rewritten or not.

Our proposed solution to this problem is to add functionality to the sender's plaintext rewriter to let it rewrite the first byte of a TLS 1.3 handshake record (which contains the TLS record handshake type) into a form that the receiver's rewriter can distinguish from the rewriting of the first plaintext byte of an Application Data record. As an example, the rewriter could set the high-order bit of the first byte of the record in an HTTP request to denote that it is Application Data and not a Handshake record. Balboa could then use this information to determine whether to proceed with rewriting.

B Supporting CBC-mode Ciphers

While Balboa's current implementation only supports stream ciphers, it is possible for Balboa to intercept TLS traffic encrypted with a CBC-mode cipher and still operate under the restriction that incoming traffic can be processed one byte at-a-time. We leave the implementation of the below approach as future work.

To avoid the numerous number of attacks on CBC-mode, modern TLS libraries use a randomly generated initialization vector (IV) for each TLS record. Balboa can take advantage of this as follows. For outgoing traffic, Balboa can replace the TLS record IV with the encryption (under a stream cipher, with an IV from the sequence number) of the first block of plaintext. It can then proceed in this fashion, replacing each subsequent block of ciphertext with the stream-cipher-encryption of the next block of plaintext. The last block of CBC-encrypted ciphertext can be replaced with random bytes. The MAC can be handled as in the case for stream ciphers.

On the incoming side, because the incoming plaintext is encrypted with a stream cipher, Balboa can decrypt it one byte at a time. To re-encrypt the traffic with a CBC-mode cipher for the application, Balboa can pick a new random IV to encrypt the block with, and emit this random IV. Even with the one byte at a time requirement, by the time Balboa emits any encrypted bytes of the plaintext it would have already observed a full block of plaintext, enabling it to generate the encrypted bytes. Balboa can then rewrite the outgoing MAC, in the same manner as it would for stream ciphers, to generate a MAC that matches the ciphertext that it just outputted (if the incoming MAC is valid).