Comments on certain past cryptographic flaws affecting fully encrypted censorship circumvention protocols

David Fifield
david@bamsoftware.com

September 12, 2023

Abstract

This article presents three retrospective case studies of cryptography-related flaws in censorship circumvention protocols: a decryption oracle in Shadowsocks "stream cipher" methods, non-uniform Elligator public key representatives in obfs4, and a replay-based active distinguishing attack exploiting malleability in VMess. These three protocols come from the family of "fully encrypted" circumvention protocols: their traffic in both directions is indistinguishable from a uniformly random stream of bytes (or at least, is supposed to be). Some of the flaws are fixable implementation errors; others are rooted in more fundamental design errors. Their consequences range from enabling passive probabilistic detection to complete loss of confidentiality. All have been fixed, mitigated, or superseded since their discovery.

My primary purpose is to provide an introduction of circumvention threat models to specialists in cryptography, and to make the point that while cryptography is a necessary tool in circumvention, it is not the sole or even most important consideration. Secondarily, I want to furnish a few instructive examples of cryptographic design and implementation errors in uncontrived, deployed protocols. While the flaws I discuss affected systems of significant social importance with millions of collective users, they are not well-known outside a small circle of specialists in circumvention.

1 Censorship circumvention

Censorship circumvention studies how to achieve communication between two network endpoints despite the interference of a censor. In a typical scenario, there is a client that wants to reach some destination (a web site, for example). The client's network access is mediated by a censor that tries to prevent the client from communicating with the destination in question. The censor is presumed, at a minimum, to block all network traffic that is addressed directly to or from the prohibited destination, and may also inspect and block other traffic with the aim of preventing any indirect access. The censor may, for example, check packet contents for transparent references to the blocked destination, or look for byte patterns that are characteristic of proxy protocols.

A system for circumvention, then, must satisfy at least two requirements. It must access the destination only indirectly (because direct access is blocked), and it must disguise the content of the communications stream (in particular, it must not expose the true destination address). Put simply: avoid address blocking, avoid content blocking.

The abstract solution to the circumvention problem is some sort of obfuscated proxy. "Proxy" means indirect access via an intermediary; "obfuscated" means using protocols and access patterns that hide the fact that a proxy is in use. Many circumvention systems have been invented, instantiating the general idea in many ways. Here I focus on a particular family of circumvention protocols: the fully encrypted protocols, or FEPs for short.[1] Protocols in this family use cryptography in an attempt to make every part of their communications stream (computationally) indistinguishable from a uniformly random sequence of bytes. In contrast to TLS, for example, with its partially plaintext handshake and recognizable record format, these protocols are designed to offer a censor no fixed byte patterns that might serve as features for classification. Fully encrypted protocols have a long history of use in censorship circumvention, going back at least to 2012 with the obfs2 protocol [3]. *Why* protocols of this type should be effective in circumvention is a separate question. I will return to this point later, but for now, let it suffice to know that they are, empirically, effective, and that distinguishability from random is considered a security bug.

Of course, a network protocol is more than a sequence of bytes, and real censors are not limited to passive distinguishability attacks. To place a circumvention protocol in the family of fully encrypted protocols is not to claim that it actually satisfies all the security properties you might want such a protocol to have [5], nor does it imply there are no ways to attack it (including, potentially, the very fact of its own high apparent randomness [14]). Taking packet boundaries and timing into account, performing replay attacks, port scanning—all these techniques and more are fair game for the censor. In this article I discuss only vulnerabilities and attacks of a cryptographic nature.

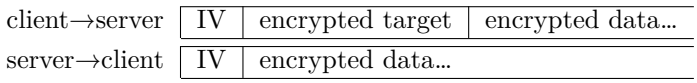
Given its focus, an article like this runs the risk of giving an exaggerated impression of the importance of content obfuscation in circumvention. It is a necessary element, to be sure, but circumvention is not, as one might think at first, purely a matter of encryption or steganography. In practice, obscuring the *content* of communication is comparatively easy—what's harder is stopping the censor from discovering and blocking endpoint *addresses*, particularly the addresses of proxy servers. This latter challenge demands separate consideration and dealing with it requires an almost orthogonal set of techniques. Even within the scope of content obfuscation, fully encrypted protocols represent just one approach among many. What I will present is a significant facet of a much larger field. It is good to remember, too, that while there are still advancements to be made, censorship circumvention is not an intractable or unsolved problem: it is used with success by millions daily.

[1] The "fully encrypted" terminology was introduced by Fenske and Johnson [5], who are formalizing this class of protocols and looking at applications beyond circumvention. These protocols have also been known as "looks-like-nothing" or "randomized" protocols.

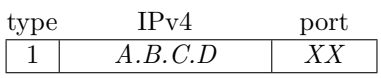
2 Shadowsocks stream ciphers decryption oracle

Shadowsocks is an encrypted proxy protocol. It is particularly commonly used in China, but its simplicity and effectiveness have made it popular in many other places as well. There are many independent but compatible implementations of Shadowsocks, which, like the protocol itself, are under ongoing development. The vulnerability I will describe is in an older variant of the protocol, now deprecated but still widely supported, known as "stream ciphers" methods. (The newer and currently recommended "AEAD ciphers" methods, while not perfect, do not have this particular problem.) In the worst case, the vulnerability leads to loss of confidentiality in past sessions, to an attacker who is able to guess a short prefix of their plaintext. It works by tricking a Shadowsocks server into sending it the decryption of a recorded ciphertext to an attacker-controlled host. The vulnerability was discovered in 2020 by Zhiniang Peng [11].

Shadowsocks stream ciphers methods [13] are easy to describe and implement. Only symmetric-key cryptography is used. Client and server share a key, derived from a password, which is used to encrypt all communications between them. The same key is used in both directions and across all sessions; only the initialization vector changes in each new session. The client connects to the server and sends it one continuous stream of ciphertext, which represents a target specification followed by upstream data to be sent through to the target. The server decrypts the target specification, makes a TCP connection to the target, then begins decrypting data from the client and sending it to the target, and encrypting data from the target and sending it to the client.



The target specification is 7 bytes long and encodes an IPv4 address and port:[2]



These protocol details are enough to imply the attack, though finding it takes a little creativity. The crucial observation is that there is no integrity protection on ciphertexts: even without knowledge of the shared secret key, an attacker can send data to a Shadowsocks server and the server will try to act on it. If you send random bytes to a Shadowsocks server, with probability 1/256 the first byte will decrypt to a 1, which will cause the server to interpret the decryption of the next 6 bytes as an IPv4 address and port. The server will connect to that address and send it the decryption (under its key) of whatever follows. A Shadowsocks server knows a secret key and has proxy functionality: abstractly, it is a machine for decrypting a ciphertext stream and sending it somewhere (the "somewhere" itself being part of the ciphertext). In other words, a Shadowsocks server can be seen as a kind of decryption oracle.

[2]Target specifications formats are taken from SOCKS [9, §4]. Besides type 1, IPv4, there are two other possibilities: type 3 is a hostname and type 4 is an IPv6 address. The attack is easiest with the IPv4 type, because addresses are short and of fixed length.

Not knowing the server's secret key, an attacker cannot produce ciphertexts of its own that will decrypt to anything meaningful. But what it can do is replay a past legitimate ciphertext. The full attack is to send the server a previously recorded encrypted Shadowsocks session, tweaking the first 7 bytes so that they decrypt to a specification of an address controlled by the attacker. Properly adjusting the first 7 bytes requires the attacker to know or guess the plaintext of those bytes. If the attacker can tolerate the data being sent to any TCP port, then 5 bytes are enough; if the attacker controls a /24 network, only 4 bytes are needed. Because the same key is used in both directions, both client→server and server→client streams can be attacked. It is often easier to guess the initial bytes in the server→client direction; for example, all HTTP/1.1 responses begin with the 7 bytes '`HTTP/1.`'. This is the beginning of a recorded server→client Shadowsocks session (minus the initialization vector):

```
7c20f534e986dbce37f555c6760ea24faa928f760db22438c8963c57e83b36fe
```

Sending this recorded ciphertext back to the server that produced it results in the server closing the connection. If we could peek at the server's logs, we would see a warning:

```
unsupported addrtype 72, maybe wrong password or encryption method
```

The server decrypted the first byte and got the value 72, which is the ASCII code for '`H`'—the first byte of '`HTTP/1.`'. Because 72 is not one of the known target specification types 1, 3, or 4, the server closes the connection.

Suppose an attacker is able to guess the plaintext prefix '`HTTP/1.`' (which is `485454502f312e` in hexadecimal) and controls a host at 203.0.113.5:8000 (whose target specification is `01cb0071051f40`). The attacker XORs out the known plaintext and XORs in their own target specification, changing the first 7 bytes to `7c20f534e986db` ⊕ `485454502f312e` ⊕ `01cb0071051f40` = `35bfa115c3a8b5`:

```
35bfa115c3a8b5ce37f555c6760ea24faa928f760db22438c8963c57e83b36fe
```

On receiving this modified ciphertext, the Shadowsocks server connects to the attacker's host and sends it the decryption of the remainder of the stream.

Minor details depend on what stream cipher the server is configured to use. With a block cipher in CTR mode, every plaintext byte after the modified prefix is decrypted accurately. In CFB mode, the second block will be garbage, because of the attacker's modification of the first block, but every block after that will be decrypted correctly. The attacker has a special advantage when the server uses CFB mode. Because of the self-synchronizing property, the attacker can concatenate (fragments of) multiple past sessions and have them all decrypted in sequence: one guessed plaintext serves to decrypt all past and future sessions.

Some Shadowsocks server implementations have (for other reasons) a replay filter that prevents the use of previously seen initialization vectors.[3] The attack can be modified to work even in the presence of such a filter [6].

[3] https://github.com/shadowsocks/shadowsocks-org/issues/44

Shadowsocks was designed with the goal of access, not confidentiality, integrity, or any other security property. But here, a lack of integrity protection led to a loss of confidentiality. Technically, a loss of user privacy does not, in itself, count as failure in a strictly access-oriented threat model—but the fact that it provides an efficient test to identify Shadowsocks servers certainly does. (Having identified a server, a censor can block its IP address, making it useless for circumvention.) The problems with Shadowsocks stream ciphers methods ran too deep to be fixed. They have been deprecated, and users encouraged to transition to newer "AEAD ciphers" methods (which fortunately had already been designed and deployed, in response to other, less severe vulnerabilities[4]).

3 obfs4 non-uniform public key representatives

obfs4 is another fully encrypted protocol whose origin is in circumvention. For many years, it has been the most-used circumvention protocol used to access Tor. The protocol is more sophisticated than Shadowsocks, featuring client authentication, a Diffie–Hellman exchange (X25519) with ephemeral session keys and forward secrecy, and options for traffic shaping [15]. Unlike Shadowsocks, obfs4 is not itself a proxy protocol: it delegates proxying to another layer, such as Tor. The original implementation of obfs4 is in a program called obfs4proxy.

obfs4's use of public-key cryptography (in the initial Diffie–Hellman exchange) adds a complication, namely that ephemeral public keys (points on Curve25519 [1]) must be exchanged, and these, like every other part of the protocol, should be indistinguishable from random. Encoding (x, y) coordinates as binary strings does not meet the requirement, because an observer can check whether x and y satisfy the curve equation $y^2 = x^3 + 486662x^2 + x$. Neither does it suffice to compress the point to just its x-coordinate, since an observer can check whether the quantity $x^3 + 486662x^2 + x$ is a square in $\mathrm{GF}(2^{255} - 19)$, which is true of every x on the curve, but only half of random strings. obfs4 solves the problem using Elligator [2], a mapping between elliptic curve points and bit string "representatives" that are indistinguishable from random. obfs4proxy originally used the Elligator implementation from the agl/25519 package [8], one of the few implementations available in 2014, when obfs4proxy was created.

There turned out to be minor oversights in agl/25519, and the way it was integrated into obfs4proxy, that caused public key representatives to be distinguishable from random in three different ways, some rather subtle. In what follows, recall that Curve25519 is defined over $\mathrm{GF}(q)$, with $q = 2^{255} - 19$. I will number bits starting from 0, in order from least to most significant. The ways public key representatives differed from random were:

1. Representatives were not always canonical.
2. Bit 255 (the most significant bit) was always zero.
3. Only points from the large prime-order subgroup were represented.

[4] https://github.com/shadowsocks/shadowsocks-org/issues/30

The distinguishing attacks enabled by these features are passive and probabilistic. A binary classifier built on them will have 100% sensitivity (obfs4 never falsely marked as random), but less than 100% specificity (random sometimes falsely marked as obfs4). Taking advantage of them does not require a censor to send its own traffic, but because a single observation is not conclusive, it would need to correlate multiple observations before marking an endpoint as obfs4 with high confidence.

Non-canonical representatives. The final step of the Elligator inverse map (which takes an elliptic curve point to its bit string representative) involves taking a square root in the finite field [2, §5.3]. An instantiation of Elligator is parameterized by what might be called a "canonical" square root function, one with the property that $\sqrt{a^2} = \sqrt{(-a)^2}$ for all field elements a [2, §5.1]. That is, we designate just over half the field elements as "non-negative," and the image of the square root function consists of exactly those elements. A convenient definition of "non-negative" for Curve25519, suggested by its authors, is the lower half of the field, the elements $\{0, 1, \ldots, (q-1)/2\}$. When there are two options for a square root, take the smaller of the two [2, §5.5]. Observe that the smaller option fits into 254 bits; this will be relevant for the next distinguisher.

The agl/ed25519 Elligator implementation did not do this canonicalization of the final square root; instead it mapped a given input systematically to either its negative or non-negative root.[5] This fact made possible a passive distinguisher: observe a representative, interpret it as a field element, square it, then take the square root using the same non-canonical square root algorithm.[6] With representatives produced by an affected version of obfs4proxy, the output of the square-then-root operation would always match the input. With random strings, the output would match only half the time.

An error of this kind is hard to detect, because it does not cause interoperability problems. The Elligator forward map takes every representative and its negative to the same elliptic curve point, so key exchanges continue to work even when an implementation systematically produces only one of two possible preimages. The problem was fixed in obfs4proxy-0.0.12 in December 2021,[7] which replaced the agl/ed25519 package with a new implementation.

Because the order of the finite field of Curve25519 is close to a power of 2, bit 254 of the binary representation of field elements is very nearly a sign bit: virtually all "negative" elements (using the convention above) have bit 254 equal to 1. Another way to understand the non-canonical square root issue is as a correlation: bit 254, by itself, had a nearly uniform 0–1 distribution, but it was not independent of the lower-order bits. This interpretation leads us to the next distinguishing feature, which has to do with what data type an Elligator inverse map function should output, and who is responsible for performing certain operations.

[5] https://github.com/agl/ed25519/pull/12, https://github.com/agl/ed25519/issues/27
[6] Suggested by Biryuzovye Kleshni: https://github.com/Yawning/libelligator/issues/1.
[7] https://gitlab.com/yawning/obfs4/-/merge_requests/3

Most significant bit always zero. Elligator representatives are formally defined as bit strings [2, §5.4]—in the case of Curve25519, bit strings of length 254. But in programming interfaces and network protocols, it is often more practical to work with byte arrays than bit strings. An elliptic curve point representative therefore needs to be encoded as an array of 32 bytes (256 bits). The question, then, is what to do with the two extra bits (bits 254 and 255), and there's an easy answer: randomize them when encoding and ignore them when decoding. Whether this auxiliary randomization is done in the caller or the callee is an implementation detail—but it must be done somewhere.

Neither obfs4proxy (the caller) nor agl/ed25519 (the callee) randomized the high-order bits of the byte array. Above, I have described how bit 254 was sometimes set and sometimes not, in a systematic way that was distinguishable from random. Besides that, nothing ever touched bit 255, with the result that the most significant bit of public key representatives was always unset.[8]

This error may be attributed to an unclear API boundary, combined with the fact that the most natural representation of a bit string in many programming languages is a byte array. Conceptually, an implementation of Elligator over Curve25519 needs a second pair of encoding and decoding functions (however trivial) that convert between 254-bit bit strings and 32-byte byte arrays. This need is not obvious from a formal description of the map.

The always-zero distinguisher was fixed, along with the one already discussed, in obfs4proxy-0.0.12. The revised Elligator implementation takes an additional "tweak" parameter, which is used to randomize the high-order bits.

Only points from the large prime-order subgroup. The elliptic curve Curve25519 has order $8p$, where p is a large prime number. In other words, the cofactor of the curve is 8. To avoid small-subgroup attacks, the base point of the X25519 key exchange algorithm is chosen to generate the subgroup of order p. In addition, secret key scalars are "clamped"—forced to be a multiple of 8—such that multiplying *any* point by a secret key results in a point on the order-p subgroup. In another easy-to-overlook subtlety, it is important that Elligator representatives *not* be constrained to representing only points on the large prime-order subgroup, for the simple reason that random strings do not always map to that subgroup. Older versions of obfs4proxy applied the Elligator inverse map to points on the large prime-order subgroup, so its representatives were distinguishable.[9] A test for this behavior was to multiply a set of observed points by p, and check for the result always being the identity element.

One fix is to add a random point on the order-8 subgroup to each public key point before encoding. Thanks to the clamping of X25519 private keys, the low-order component disappears during key exchange. This fix was adopted in obfs4proxy-0.0.12, but an implementation error prevented it from working until a further correction was made in obfs4proxy-0.0.14 in September 2022.[10]

[8] https://bugs.torproject.org/tpo/anti-censorship/team/91
[9] https://bugs.torproject.org/tpo/anti-censorship/pluggable-transports/lyrebird/40007
[10] https://gitlab.com/yawning/obfs4/-/merge_requests/9

There is no evidence that censors used any of these distinguishers to identify and block obfs4 connections. We may speculate as to why not. It could be that censors were simply ignorant—the distinguishers do take some skill to detect, after all. Or perhaps censors did, in fact, use them, but were never caught in the act by someone in a position to document it. (We can be sure, at least, that it is not because obfs4 is too small a target to be worth a censor's attention, since censors *do* try to discover and block obfs4 servers in other ways—by attacking address distribution mechanisms, for example.) I will suggest another possibility: censors, for whatever reason, find it unappealing to do the kind of stateful processing over multiple flows that is needed to take advantage of distinguishers like these. I will return to this idea in the final section.

4 VMess header malleability

VMess is a fully encrypted proxy protocol, one of the native protocols of the V2Ray proxy framework [12]. Like Shadowsocks, it is based on symmetric cryptography and a shared secret (a 16-byte UUID). Unlike Shadowsocks and obfs4, VMess is not always exposed on the wire: it may be used only as an inner proxy protocol inside some other transport protocol, like WebSocket or QUIC.

The VMess implementation in V2Ray was vulnerable to an active probing attack to identify VMess servers, caused by the way data structures are represented in the encrypted protocol header, and interactions between the protocol parser and the network. The vulnerability was reported in 2020 by GitHub users p4gefau1t and studentmain [10]. For my understanding, I am grateful for the English summary by GFW Report [7]. I will present a simplified version (omitting details of overcoming replay protection).

A VMess client request consists of a 16-byte authentication header, a command block (which contains the target specification among other things), and a data stream. The authentication header is a MAC, keyed by the user's random UUID, of a recent timestamp, which must be within 2 minutes of the current time. The command block is encrypted with AES in CFB mode and a key derived from the UUID and the timestamp. The data stream is encrypted with a key and initialization vector stored in the command block. The tolerance interval on timestamps permits an attacker to replay authenticated requests for a short time, which is the first part of the attack.

16 bytes	variable	variable
auth info	encrypted command block	encrypted data…

The command block contains many fields. I will highlight just the ones needed to understand the attack, particularly the variable-length padding and address fields and the final hash. The address length N depends on the address type.

39.5 bytes	4 bits	1 byte	N bytes	P bytes	4 bytes
other fields	P	address type	address	padding	hash

The encrypted command block has a limited form of integrity protection. There is a field at the end with a hash of the preceding bytes, working something like a

MAC-then-encrypt construction (except that the hash is not keyed by a secret). The problem is that the overall length of the command block is variable, because of the variable-length padding and address fields it contains. The receiver of a command block cannot locate the hash field to verify it, without parsing the very data the hash is meant to protect.

The idea of the active probing attack is to repeatedly replay an authentic client request, taking advantage of the lack of integrity in the command block to modify the field P that indicates how much padding there is. Record a legitimate encrypted command block, and set the 4 ciphertext bits that correspond to the P field to 0000. Send the modified command block to the suspected VMess server, but stop just before the variable-length fields. Then resume sending, one byte at a time (the values are not important), with a delay after each byte, until the server closes the connection. The hash verification will almost certainly eventually fail, because of the changes to the ciphertext, but a vulnerable VMess server will not disconnect until it has read the full N bytes of address, P bytes of padding, and 4 bytes of hash. Repeat the attack another 15 times, setting the bits that correspond to P to $0001, 0010, \ldots, 1111$ in turn. Record how many bytes the server receives, in each attempt, before closing the connection. The sign of a VMess server is that the maximum and minimum byte counts differ by 15, with every value in between represented.

V2Ray developers mitigated the vulnerability within a few days. In v4.23.4, the server disconnects after a timeout even if it has not yet received the full hash, and does not close a connection immediately when hash verification fails.[11]

5 The bigger picture

I intend for this article to help calibrate a practical mental model of censorship and circumvention. For that, we need to step back from close examination of cryptographic flaws and take a wider view of how cryptography fits into circumvention. The core criterion by which a circumvention system is judged is blocking resistance. It is by this criterion that we decide what counts as a vulnerability (what might get the system blocked), and model how difficult or expensive a vulnerability might be to exploit. The notion of blocking resistance is hard to characterize fully—depending as it does partially on the unknown resources and preferences of sometimes volatile censors—but one thing we may say is that it is not solely cryptographic. Usually, other factors matter more, such that minor errors in the cryptography or its implementation do not much diminish overall blocking resistance. This is not to excuse shoddy crypto: developers should take seriously their responsibility to minimize risk of harm to users. It is only to say that when when the goal is not getting blocked, focusing too narrowly on cryptography can leave one open to more elementary vulnerabilities.

Intuitions about what is easy or hard for a censor to do can be misleading. An illuminating question is: why should fully encrypted protocols be useful for

[11] https://github.com/v2ray/v2ray-core/releases/tag/v4.23.4

circumvention at all? These protocols originated in circumvention, and persist in that use because they continue to work. But isn't the fact that they have no identifiable plaintext features, and overall high randomness, itself a fingerprint? To be sure, it is—but experience shows it not as easy to exploit for blocking as one might initially think. The first evidence of the detection of fully encrypted protocols purely by passive measurement of randomness appeared only in 2021, in China [14], around a decade after the debut of that family of protocols. And even that showed signs of being somewhat tentative, being limited to certain foreign IP address ranges and a subsampled fraction of traffic [14, §6]. It is not that censors haven't wanted or tried to block fully encrypted protocols: the censorship system in China had been discovering proxies of obfs4's predecessor protocols obfs2 and obfs3 by *active probing* as early as 2013 [4, §5.4].[12] And since before that, censors have attacked circumvention proxies (fully encrypted or otherwise) by attacking the systems used to distribute proxy addresses, which has the advantage of being more general: a proxy, once discovered, can be blocked once and for all by its IP address, with no need for online protocol classification. We should not assume that censors always act rationally, but to judge by their behavior, censors find other means of attacking fully encrypted protocols more attractive (simpler, more reliable, less expensive) than what might seem to be the more obvious way of detecting their unusually high randomness.

It would seem that, in deploying a simple strategy for content obfuscation (fully encrypted protocols), circumvention developers hit on a weak spot of censors. As to the question of why censors might have difficulty blocking fully encrypted protocols, I cannot give a definite answer, but I can suggest a way of thinking about it. Network censorship devices are often modeled as binary classifiers: for each unit of traffic (packet, connection), they decide whether it should be blocked or allowed to pass. But censorship is not only a game of indistinguishability, it is also a game of *costs*. Put another way, circumvention is more than a matter of blending in with "normal" traffic (however that may be defined). If a circumvention protocol blends in perfectly with traffic the censor regards as unimportant, its blocking resistance is low, because the censor can block all traffic of that type without regret. Conversely, even an imperfect imitation of traffic the censor values highly may have high blocking resistance, because the censor has to be careful not to block the normal traffic in the course of trying to blocking circumvention traffic. Think of a censor not as trying to classify traffic correctly per se, but to maximize utility when classification decisions are coupled to costs. It may be that it hard to build a classifier that correctly detects a large fraction of fully encrypted proxy traffic without also falsely detecting too much other traffic. The task is made more difficult by the unequal base rates of non-circumvention and circumvention traffic.

There are costs to misclassification in both directions. False negatives (failing to block what should be blocked) obviously work against the censor's purpose, but false positives (blocking what should have been allowed) have costs too.

[12] Because of this, obfs4 was designed not only to be fully encrypted but also to resist active probing attacks [15, §2].

The costs of false positives are hard to quantify, but may take the form of, for example, generally higher levels of discontent, and people being unable to get work done because the resources they need are blocked. Censors differ in their preference for overblocking versus overblocking. The same censor may even have different preferences at different times; for example, it may choose to exchange more false positives for fewer false negatives around sensitive political events.

Besides the costs of misclassification, there are "overhead" costs associated with running a censorship system. These include the costs of buying hardware, training personnel, and researching detection rules. To these we may add the costs of actually performing classification, which are measured in CPU cycles and megabytes of memory. Not all classifiers are equally efficient: ones that only need to look at single packets in isolation, or just the first few packets of a connection, are cheaper to run than ones that require computing statistics over long packet sequences or correlating measurements across many connections. This may help explain why probabilistic classifiers, like those in the section on obfs4, have not found much purchase among censors (even apart from the risk of incorrect classification).

6 Thanks

The vulnerabilities I discovered myself are the second and third in the obfs4 section. Credit for the others goes to Zhiniang Peng, Biryuzovye Kleshni, p4gefau1t, and studentmain.

Special thanks go to Loup Vaillant, whose efforts in documenting and explaining Elligator, and encouraging better and safer implementations, have been heroic. The detailed information at elligator.org was essential to my understanding of Elligator and how it is used in obfs4. I am grateful to GFW Report for an English summary of the VMess header malleability vulnerability [7]. I thank Adam Langley, Zhiniang Peng, studentmain, Loup Vaillant, Xiaokang Wang, and Yawning Angel for comments on a draft of this article.

References

[1] Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography*. Springer, 2006. https://cr.yp.to/ecdh.html#curve25519-paper.

[2] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: Elliptic-curve points indistinguishable from uniform random strings. In *Computer and Communications Security*. ACM, 2013. https://elligator.cr.yp.to/papers.html.

[3] Roger Dingledine. Obfsproxy: the next step in the censorship arms race, February 2012. https://blog.torproject.org/obfsproxy-next-step-censorship-arms-race/.

[4] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. Examining how the Great Firewall discovers hidden circumvention servers. In *Internet Measurement Conference*. ACM, 2015. https://ensa.fi/active-probing/.

[5] Ellis Fenske and Aaron Johnson. Security notions for fully encrypted protocols. In *Free and Open Communications on the Internet*, 2023. https://www.petsymposium.org/foci/2023/foci-2023-0004.php.

[6] David Fifield. Decryption vulnerability in Shadowsocks stream ciphers, February 2020. https://github.com/net4people/bbs/issues/24.

[7] GFW Report. Summary on recently discovered V2Ray weaknesses, June 2020. https://gfw.report/blog/v2ray_weaknesses/en/.

[8] Adam Langley. Implementing Elligator for Curve25519, December 2013. https://www.imperialviolet.org/2013/12/25/elligator.html.

[9] Marcus D. Leech, David Koblas, Ying-Da Lee, LaMont Jones, Ron Kuris, and Matt Ganis. SOCKS protocol version 5. RFC 1928, March 1996. https://www.rfc-editor.org/info/rfc1928.

[10] p4gefau1t and studentmain. vmess协议设计和实现缺陷可导致服务器遭到主动探测特征识别(附PoC) *vmess protocol design and implementation flaws can lead to servers being identified by active probing (with PoC)*, May 2020. https://github.com/v2ray/v2ray-core/issues/2523.

[11] Zhiniang Peng. Redirect attack on Shadowsocks stream ciphers, February 2020. https://github.com/edwardz246003/shadowsocks/tree/ba5df18abf6792d0599c36a9e6c3398e7d0c1fd8.

[12] Project V. VMess, February 2021. https://github.com/v2fly/v2fly-github-io/blob/90cfaec7ee5e62fae5cc8351be7da7da82b3ed4e/docs/en_US/developer/protocols/vmess.md.

[13] Shadowsocks. Stream ciphers, May 2023. https://github.com/shadowsocks/shadowsocks-org/blob/8c44e881ec382037b113c0945c90e49d1801c6ae/docs/doc/stream.md.

[14] Mingshi Wu, Jackson Sippe, Danesh Sivakumar, Jack Burg, Peter Anderson, Xiaokang Wang, Kevin Bock, Amir Houmansadr, Dave Levin, and Eric Wustrow. How the Great Firewall of China detects and blocks fully encrypted traffic. In *USENIX Security Symposium*. USENIX, 2023. https://gfw.report/publications/usenixsecurity23/en/.

[15] Yawning Angel. obfs4 (the obfourscator), January 2019. https://gitlab.com/yawning/obfs4/-/blob/obfs4proxy-0.0.14/doc/obfs4-spec.txt.