# Comments on certain past cryptographic faults affecting randomized censorship circumvention protocols

David Fifield

November 8, 2022

## Abstract

This talk presents three retrospective case studies of cryptography-related faults in censorship circumvention protocols: a decryption oracle vulnerability 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 so-called "randomized" or "look-like-nothing" protocols: their communications stream is indistinguishable from a uniformly random byte string (that is, when nothing goes wrong). Some of the faults are fixable implementation errors; some are rooted in more fundamental design flaws. Their consequences range from enabling probabilistic passive detection to complete loss of confidentiality. All have been fixed, mitigated, or superseded since their discovery.

I have two goals in giving this talk. The first is to furnish examples of surprising effects of cryptographic faults in non-contrived, significant, deployed protocols, as an aid in building motivation and intuition in learners of cryptography. My second goal is to acquaint specialists in cryptography with censorship circumvention threat models, and in particular, to support a claim that cryptography, while a necessary tool, is not the sole or even the primary consideration in circumvention.

Anyone who has taught beginning cryptography knows the experience of justifying adversary models and formal security reductions in response to well-meaning questions by students. Why care about chosen ciphertext attacks—won't it just decrypt to gibberish? Why is it a problem if an attacker learns *some* information about a message, as long as they cannot decrypt all of it? How would an attacker get access to an encryption/decryption oracle, anyway? In answer to questions like these it is good to have a few ready examples of how things have gone wrong in practical cryptographic deployments, to motivate the need for proofs and demonstrate that intuitions about security can be misleading.

In this talk I will present a few such examples that I have found instructive. They come from the field of censorship circumvention, and specifically from the subcategory of protocols that use cryptography to present a byte stream that is indistinguishable from uniform randomness. While the faults I will discuss affect systems that are socially important and have millions of collective users, I believe they are not well known outside a small group of specialists.

# 1 Censorship circumvention

Censorship circumvention studies how to enable communication between two network endpoints despite the interference of a censor. In a typical model, there is a client that wants to reach some destination (for example, a web site). 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 has the power to inspect other flows and block them based on any observable characteristic (such as, for example, whether they transparently refer to the blocked destination in their payloads).

A system for circumvention, then, must satisfy at least two requirements. It must access the destination only indirectly (generally, through a proxy server), because direct access is blocked; and it must hide the content of communications (in particular, it must not reveal the true target address in any traffic sent to a proxy). Put simply: avoid address blocking, avoid content blocking.

To meet these requirements, a large number of circumvention systems have been invented, tackling the problem in a variety of ways. In this talk I focus on a particular subfamily of circumvention protocols, sometimes known as "look-like-nothing" or "randomized" protocols. Protocols of this type use cryptography to ensure that every part of their communications stream is (computationally) indistinguishable from a uniformly random byte stream. In contrast to TLS, for example, which has plaintext fields in its handshake and a recognizable record format, these fully randomized protocols offer a censor no fixed byte patterns to serve as a basis for classification. *Why* protocols of this type are effective in circumvention is a separate question. I will return to this point later, but for now, let it suffice to state that they are, empirically, effective, and that distinguishability from random is considered a security bug.

Of course, real censors are not limited to passive examination of byte streams. The complicated world of networks provides an intermediary with a number of avenues of attack, which realistic circumvention threat models must take into account. Inspecting packet boundaries and measuring timing, replay attacks, port scanning—all these are fair game for the censor. In this talk I will only discuss vulnerabilities that have a cryptographic flavor.

A talk like this one runs the risk of giving an exaggerated impression of the importance of content obfuscation in circumvention. It is an indispensable element, to be sure, but circumvention is not, as one might think at first, purely a matter of steganography or encryption. In practice, obscuring the *contents* of communication is comparatively easy—what's harder is stopping the censor from discovering and blocking endpoint *addresses* (including those of proxy servers). Meeting this latter challenge requires a separate, almost orthogonal set of techniques. Even within the scope of content obfuscation, look-like-nothing is only one technique among many. What I will present is only one notable facet of a 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 every day.

# 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 lent it popularity in many areas subject to network censorship. (There are even businesses, known as 机场, “airports,” built around selling access to Shadowsocks servers.) Shadowsocks has many independently developed but compatible implementations. They, and the protocol itself, are under ongoing development. The vulnerability I will describe is in an older version of the protocol, now deprecated but still widely supported, known as “stream ciphers” methods. (The label is in contrast to the newer and currently recommended “AEAD ciphers” methods, which, while not without flaw, do not have this particular problem.) The effect of the vulnerability is total loss of confidentiality in past recorded sessions, to an attacker who is able to guess a short prefix of the plaintext. It works by tricking a Shadowsocks server into connecting to an attacker-controlled host and sending it the decryption of a past session. It was discovered in 2020 by Zhiniang Peng [5].

Shadowsocks stream ciphers methods [7] are easy to describe and implement. Only symmetric-key cryptography is used. Client and server share a key, derived from a password, that is used to encrypt communications between them. The same key is used in both directions and for all sessions; only the initialization vector changes in each new connection. The client connects to the server and sends it one long ciphertext, whose content consists of a target specification and the data to send to that target. The server decrypts the target specification, makes a TCP connection to the given host, then decrypts the client’s data and forwards it to the target. The target’s response is proxied back to the client.

| | | | |
|---|---|---|---|
| client→server | IV | encrypted target | encrypted data… |
| server→client | IV | encrypted data… | |

The target specification is 7 bytes long and encodes an IPv4 address and port:[1]

| type | IPv4 | port |
|---|---|---|
| 1 | *A.B.C.D* | *XXYY* |

These few protocol details are enough to imply the attack, though it takes a little creativity to find. The crucial observation is that there is no integrity protection on ciphertexts: even without knowing the shared secret key, you 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, then with probability 1/256 the first byte will decrypt to a 1, which will cause the server to interpret whatever the next 6 bytes decrypt to as an IPv4 address and port, which the server will attempt to connect to and send the rest of the stream, decrypted under its key. The combination of being a proxy server and knowing a secret key makes a Shadowsocks server a machine for decrypting a ciphertext stream and sending it somewhere, the “somewhere” being part of the ciphertext. In other words, a Shadowsocks server is a particular kind of decryption oracle.

---

[1] There are two other target specification formats. Type 3 is a hostname and type 4 is an IPv6 address. The attack is easiest to perform with an IPv4 target.

The full attack is to replay a previously recorded encrypted Shadowsocks stream, tweaking the first 7 bytes so that they decrypt to a specification of an IPv4 address controlled by the attacker. Tweaking the first 7 bytes requires the attacker to know the plaintext of those bytes. If the attacker can tolerate the data being sent to any TCP port on a single host, then the first 5 bytes are enough; 4 bytes are enough if the attacker controls an entire /24 network. 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.`’. Below is a recorded server→client Shadowsocks session (minus the initialization vector, for clarity):

```
7c20f534e986dbce37f555c6760ea24faa928f760db22438c8963c57e83b36fe
```

Sending this recorded ciphertext back to the same 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 the attacker guesses the full 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 $\texttt{7c20f534e986db} \oplus \texttt{485454502f312e} \oplus \texttt{01cb0071051f40} = \texttt{35bfa115c3a8b5}$:

```
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.

What can we learn from this attack? The developers of Shadowsocks emphasize that it is a tool for access (i.e., circumvention), not for secrecy or any other security property. Still, such a complete loss of confidentiality is surely unexpected by most Shadowsocks users, and unintended by the protocol designers. It is severe enough to threaten even the stated goal of access, as it provides an effective active-probing test to identify Shadowsocks servers (which can be blocked by IP address). The lesson, perhaps, is that it is hard to skimp on security without giving up more than you intend: you need integrity protection even when you “don’t need” integrity protection. In this case, the problems with Shadowsocks stream ciphers methods ran too deep to be fixed. They could only be deprecated, and users encouraged to transition to the newer “AEAD ciphers” methods (which fortunately had already been designed and deployed, in response to other, less severe vulnerabilities).

# 3 obfs4 non-uniform public key representatives

obfs4 is another fully randomized circumvention protocol [8]. For many years it has been the most-used Tor pluggable transport. The protocol is more sophisticated than Shadowsocks, featuring client authentication, a Diffie–Hellman exchange (X25519) with ephemeral session keys and forward secrecy, and affordances for traffic shaping. Unlike Shadowsocks, obfs4 is not itself a proxy protocol; it delegates proxying to some other protocol, usually Tor. Also unlike Shadowsocks, there is only one main implementation of obfs4, called obfs4proxy.

obfs4's use of public-key cryptography (in its Diffie–Hellman exchange) adds a complication to the design, namely that ephemeral public keys (points on Curve25519) must be exchanged, and these, like every other part of the protocol, must 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 used the Elligator implementation from the agl/25519 package [3], one of the few implementations of Elligator available in 2014, when obfs4proxy was created.

There turned out to be minor oversights in agl/25519, and the way it was integrated with obfs4proxy, that caused public key representatives to be distinguishable from random in three 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, 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.

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 uniform), but less than 100% specificity (uniform sometimes falsely marked as obfs4). Taking advantage the obfs4 distinguishers does not require a censor to send any 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 these 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 in 254 bits; this will be relevant for the second 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. This fact made possible a passive distinguisher: observe a representative $r$, interpret it as a field element, square it, then take the square root using the same non-canonical square root algorithm.[2] With an affected version of obfs4proxy, the output of the square-then-root operation would always match the input. With random strings, the same test matches only half the time.

An error of this kind is hard to detect, because it does not cause interoperability problems. The Elligator map takes every representative and its negative to the same elliptic curve point, so key exchanges continue to work. The problem was fixed in obfs4proxy-0.0.12 in December 2021, which replaced the agl/ed25519 package with a custom 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 (by the definition 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.

**Most significant bit always zero.** Elligator representatives are formally defined as bit strings [2, §5.4]—specifically, in the case of Curve25519, bit strings of length 254. But in programming interfaces and network protocols, it is more practical to work with byte arrays, not 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 extra 2 bits (bits 254 and 255), and the answer is easy: 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. I have described how bit 254 was sometimes set, which was its own bug, but nothing ever touched bit 255. The result is that bit 255 of public key representatives was always equal to 0.

This error may be attributed to an unclear API boundary, combined with

---

[2] Suggested by Biryuzovye Kleshni: https://github.com/Yawning/libelligator/issues/1.

the fact that the most straightforward representation of a bit string in many programming languages is as 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 zero-bit 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 2 most significant 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 prime-order subgroup. In another easy-to-overlook subtlety, it is important that Elligator representatives not be constrained to representing the prime-order subgroup, for the simple reason that random strings do not always map to points on that subgroup. Older versions of obfs4proxy only sent representatives of points on the prime-order subgroup. A test for this behavior was to multiply observed points by $p$, and check whether the result was always the identity element.

One way to fix the issue is to add a randomly selected low-order point to public key points before encoding. Thanks to the special structure of X25519 private keys, the low-order component conveniently disappears during key exchange. This strategy was adopted in obfs4proxy-0.0.12, but a minor implementation error prevented it from working until a further correction was made in obfs4proxy-0.0.14 in September 2022.

There is no evidence that these features were ever used by censors to identify and block obfs4 connections. One can speculate as to why. It could be that censors were simply ignorant—the distinguishers do take some skill to detect, after all. Another possibility is that censors did, in fact, make use of them, but were never caught in the act by someone in a position to document it. Allow me to suggest another idea: censors, for whatever reason, find it unappealing to do stateful processing over multiple flows, of the kind needed to take advantage of distinguishers like these. I will return to this idea at the end of the talk.

## 4 VMess header malleability

VMess is another encrypted proxy protocol, developed as part of V2Ray [6]. V2Ray is not itself a protocol; it is a framework for constructing proxies by composing and layering multiple protocols. Like the other protocols, VMess is fully encrypted and intended to be indistinguishable from random. It is like Shadowsocks in its use of symmetric cryptography with a shared secret (in this case, a 16-byte UUID specific to each user).

The VMess implementation in V2Ray was vulnerable to an active probing attack to identify VMess servers, caused by the representation of structures in the encrypted protocol header and the protocol parser's interaction with the network. The vulnerability was reported in 2020 by GitHub users p4gefau1t and studentmain [4]. For my understanding, I am grateful for the English summary of the vulnerability posted by GFW Report [1]. I will present a simplified version here (omitting details of overcoming replay protection).

A VMess client request consists of a 16-byte authentication header, followed by a command (which consists of a target specification and other data), then a stream of data. 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 timestamp. The data stream is encrypted with a key and initialization vector from the command block. The tolerance interval on timestamps permits an attacker to replay authenticated requests within a short time, which is the first part of the attack.

| 16 bytes | variable length | variable length |
|---|---|---|
| auth info | encrypted command | encrypted data... |

The command block contains many fields. I will highlight only those that are necessary for understanding the attack.

| 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. A field at the end contains a hash of the preceding bytes, like a MAC-then-encrypt construction (except that the hash is not keyed by a secret). The problem is that the length of the command block is variable because of the address and padding 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 vulnerability enables an active probing attack to identify VMess servers. The idea is to replay an authentic client request, taking advantage of the lack of integrity in the command block to modify the field $P$, which indicates the amount of padding. Begin sending a command, but stop just before the variable-length fields. Then resume sending, one byte at a time, and count how many bytes the server receives before closing the connection. The hash verification will almost certainly eventually fail, because of the attacker's changes to the ciphertext, but the VMess server will not disconnect until it has read a full $N$ bytes of address, $P$ bytes of padding, and 4 bytes of hash. The attacker repeats the attack 16 times, tweaking the bits of $P$ each time to cycle through all four-bit values. The sign of VMess is that the server reads $X + P$ bytes in each attempt, where $X$ is constant and $P$ ranges through $\{0, \ldots, 15\}$.

V2Ray developers mitigated this vulnerability within a few day by having the server disconnect after a timeout even if it had not yet received a hash to verify, and not close a connection immediately after a hash verification failure.[3]

---

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

## 5 Does it matter?

The cryptographic problems I have described did not spell the end of the protocols involved—why not? There are a number of likely explanations, some superficial and some more fundamental. The common element among them is that cryptography is rarely the weak link in censorship resistance, which more crucially hinges on toher factors that are difficult to quantify and model, such as a censor's priorities, resources, and (especially) costs. Through a purely cryptographic lens is not the right way to view censorship circumvention protocols.

Certainly censors are limited by various mundane constraints. It takes time and expertise to discover and understand protocol flaws, and even when known, time and resources are needed to deploy countermeasures against them. This fact can explain why the vulnerabilities in this talk were not exploited for censorship, but it falls short of explaining why look-like-nothing protocols work at all, despite their simplistic strategy for disguising traffic and censors' having had years to study them. Why do censors not, for example, detect and block flows with unusually high entropy? I will suggest a way to think about it. If you model a censor as a binary classifier, which decides, for each unit of network traffic, whether to block or allow, you must take into account that, besides the inherent (e.g. computational and maintenance) costs of operating the classifier, there are costs for misclassification, in both directions. False negatives—failing to block what should be blocked—obviously work against the censor's purpose, but false positives—accidental overblocking—have their costs too, which are hard to quantify, but may take the form of user anger and generally lessened utility of the network. It may be that it is a difficult task to build a classifier that detects a large fraction of fully randomized traffic without falsely detecting too much unrelated traffic, especially given the unequal base rates of circumvention and non-circumvention traffic. Censorship researchers *do* see concerted attempts by censors to block circumvention using these and other protocols, but for sufficiently advanced protocols, it is usually not the protocols themselves that are attacked, but other elements of the system.

An even more fundamental observation is that, according to available evidence, censors find it easier and cheaper to effect censorship by blocking addresses than by blocking protocols, and prefer to do the former whenever possible. Protocol obfuscation only has to be good enough to be more expensive than address blocking, a threshold which look-like-nothing evidently reaches. A good thought experiment for an aspiring circumvention protocol designer is this: suppose you had a perfect steganographic communications protocol, immune to all blocking on the basis of content. What proxy or proxies do you communicate with, and what stops the censor from blocking them by address? If the addresses of proxies are to be kept secret, how do legitimate users learn them, without the censor also learning them? There are reasonable answers to these questions: for example, one can compartmentalize proxy addresses over small groups of users to limit exposure; tie proxy access to some scarce real-world resource; change proxy addresses frequently; or co-locate proxies with important network infrastructure that the censor is reluctant to block even when its addresses are known. A practical circumvention system must have some answer to these questions, which usually constrains the design more than does the choice of protocol.

## 6 Thanks

Special thanks go to Loup Vaillant, whose efforts in documenting and explaining Elligator, and encouraging better and safer implementations, have been nothing short of 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 bug [1]. I thank Adam Langley, Zhiniang Peng, Loup Vaillant, Xiaokang Wang, and Yawning Angel for comments on a draft of this talk.

## References

[1] Anonymous. Summary on recently discovered V2Ray weaknesses, June 2020. URL: https://gfw.report/blog/v2ray_weaknesses/en/.

[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. URL: https://elligator.cr.yp.to/papers.html.

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

[4] 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. URL: https://github.com/v2ray/v2ray-core/issues/2523.

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

[6] Project V. VMess, February 2021. URL: https://www.v2fly.org/en_US/developer/protocols/vmess.html.

[7] Shadowsocks. Stream ciphers, June 2022. URL: https://shadowsocks.org/guide/stream.html.

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