Comments on certain past cryptographic faults affecting randomized censorship circumvention protocols

David Fifield

draft November 1, 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 All the issues I mention have been fixed, mitigated, or superseded since their discovery.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, deployed protocols of some importance, in order to help build motivation and intuition in learners of cryptography. The second goal is to increase appreciation of censorship circumvention threat models among specialists in cryptography, and in particular, to help explain why the problems I will cover did not have as big a negative effect as one might predict. Practical circumvention is a topic in which cryptography is a necessary tool, but is not the only or even the most important consideration.

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 do we care about chosen ciphertext attacks—won't it just decrypt to gibberish? Why is it a problem for an attacker to learn *some* information about a message, as long as they can't decrypt the whole thing? How would an attacker get access to an encryption/decryption oracle, anyway? In response 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 interesting and instructive. They come from the field of censorship circumvention, 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 easily 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 that controls part of the network paths between them. In a typical model, a client resides in a network that is completely controlled by a censor, and wants to reach some destination (for example, a web site) that is located outside the censor's network but which the censor wants to prevent access to. The censor is presumed, at a minimum, to block all network traffic that is addressed directly to or from the prohibited destination, and additionally has the power to inspect and block other flows based on any other observable characteristics (such as, for example, whether they contain references to the blocked destination in the contents of their packets).

Figure.

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 static features to serve as a basis for classification. *Why* protocols of this type are effective for circumvention is an interesting question: I will return to this point later, but for now, let it suffice to know that they are, empirically, effective.

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. Attacks based on inspecting packet boundaries or measuring timing, replays, port scanning—all these are fair game for the censor, but in this talk I will only cover topics that have a cryptographic flavor.

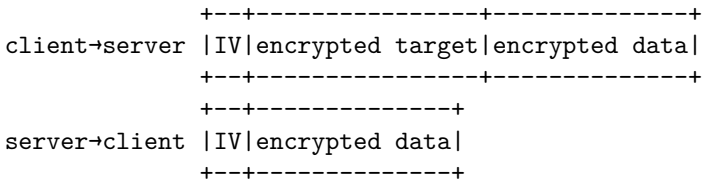
A presentation like this one runs the risk of giving an exaggerated impression of the importance of content obfuscation in circumvention. It is an indispensable

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 different, almost orthogonal set of techniques. Even within the scope of content obfuscation, look-like-nothing is only one technique among many. Keep in mind that what I will present is just one interesting facet of a larger field. It is good to remember, also, that while there are still advancements to be made, censorship circumvention is not an unsolved problem: it is used with success by millions every day.

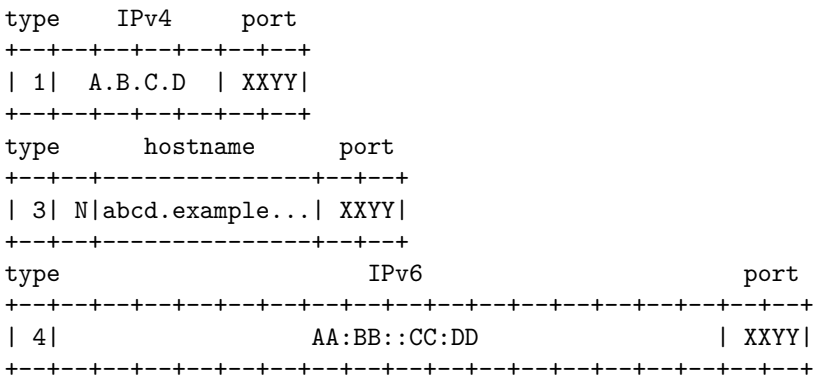
2 Shadowsocks stream ciphers decryption oracle

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 businesses centered around managing and selling access to Shadowsocks servers.) Shadowsocks has many independently developed, but compatible implementations, and they and the protocol itself are under ongoing development. The vulnerability I will tell you about is in an older version of the protocol, which is 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, though not without flaw, do not have this particular problem.) The consequence 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 forwarding a decrypted session there. It was found and first described in 2020 by Zhiniang Peng [7].

Shadowsocks stream ciphers methods [9] are easy to describe and implement. Only symmetric key cryptography is used. Client and server share a symmetric key (derived from a password), which 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 choice of cipher (supported options include ChaCha20 and AES in CFB or CTR mode) must be settled in advance, out of band. 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. Any response from the target is proxied back to the client.



The target specification can take three forms, which are distinguished by a one-byte type prefix. Type 1 is IPv4, type 3 is a hostname, and type 4 is IPv6.



These 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 attempt to act on it. As a simple example, if you send a 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, to which the server will attempt to connect and send the rest of the stream (decrypted under the server's key). The combination of being a proxy server and knowing a secret key means that a Shadowsocks server is *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.

The full attack is to replay a previously recorded encrypted Shadowsocks stream, tweaking the first 7 bytes so 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. If all but a few bytes are known, the unknown ones can be brute-forced over multiple connection attempts. 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 real 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`

What happened is 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 `7c20f534e986db` ⊕ `485454502f312e` ⊕ `01cb0071051f40` = `35bfa115c3a8b5`:

`35bfa115c3a8b5ce37f555c6760ea24faa928f760db22438c8963c57e83b36fe`

On receiving the 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 limited goal of access, as it provides an effective active-probing test for Shadowsocks servers (which then can be blocked by address). The lesson, perhaps, is that it is hard to skimp on security properties 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 developed, in response to other known vulnerabilities).

3 obfs4 non-uniform public key representatives

obfs4 is another fully randomized circumvention protocol [10]. For several years it has been the most-used Tor pluggable transport. The protocol is considerably more sophisticated than Shadowsocks's, featuring client authentication, a Diffie–Hellman exchange (X25519) with ephemeral session keys and forward secrecy, and affordances for traffic padding. Unlike Shadowsocks, obfs4 is not itself a proxy protocol. Its only job is content obfuscation; it delegates proxying to some other service, usually Tor. Also unlike Shadowsocks with its multiple implementations, there is only one main implementation of obfs4, called obfs4proxy.

obfs4's use of public-key cryptography (its Diffie–Hellman exchange) adds a notable complication to the design, namely that ephemeral public keys (i.e., points on Curve25519) must be conveyed on the network, 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 $y^2 = x^3 + 486662x^2 + x$. Even compressing the point to just its x-coordinate does not work, because an observer can check whether the quantity $x^3 + 486662x^2 + x$ is a square in $\mathrm{GF}(2^{255} - 19)$, which is true for every x on the curve, but only half of random strings.

As a solution to this problem, obfs4 encodes its public keys using Elligator [2], which is designed for exactly this purpose: in the forward direction it maps bit strings to elliptic curve points, and in the inverse direction it maps (a subset of) elliptic curve points to bit strings that are indistinguishable from random (provided the input points are selected uniformly). The bit string encoding of a point is called the point's "representative." obfs4proxy used the Elligator implementation from the agl/25519 package [5], which was 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 in its interaction with 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 $q = 2^{255} - 19$. We number bits starting from 0. The three ways in which 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, not the whole curve.

The distinguishing attacks enabled by these features are passive and probabilistic. A binary classifier built on these features will have 100% sensitivity (obfs4 never falsely marked as uniform), but less than 100% specificity (uniform sometimes falsely marked as obfs4). Unlike the Shadowsocks vulnerabilities, taking advantage of obfs4 public key representative non-uniformity does not require the censor to send any of its own traffic; but because a single observation is not conclusive, the censor will generally 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 we have two options for a square root, we take the smaller of the two [2, §5.5]. (Note that smaller option always fits in 254 bits; this will be relevant to the next distinguisher.)

The agl/ed25519 Elligator implementation did not do this canonicalization of the final square root; instead it mapped a given input to either its negative or non-negative root, in a consistent manner. This fact makes 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.[1] With an affected version of obfs4proxy, the output of the square-then-root operation always matches the input. For random strings, the input and output will match 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 output point, so key exchanges still work. It was fixed in obfs4proxy-0.0.12 in December 2021, which replaced the agl/ed25519 package.

Because the order of the finite field of Curve25519 is so close to a power of 2, bit 254 of the binary representation of 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]. In the case of Curve25519, specifically, bit strings of length 254. But in programming interfaces and network protocols, it is more practical to work with byte arrays, not bit streams. An elliptic curve point representative needs to be encoded as an array of 32 bytes (256 bits). The question, then, is what to do with the 2 extra 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.

As it happens, neither obfs4proxy (the caller) nor agl/ed25519 (the callee) randomized the high-order bits of the byte array. I have just described how agl/ed25519 was sometimes setting bit 254, which was a different sort of bug, but nothing ever touched bit 255. The result is that bit 255 of public key representatives was always equal to 0, a property that requires no fancy test to distinguish from random.

This error can be attributed to an unclear division of responsibilities, coupled with the fact that the most reasonable representation of a bit string in many programming languages is a byte array. Conceptually, a practical implementation of Elligator over Curve25519 needs a *second* pair of encoding and decoding functions (even if they are fairly trivial) that convert between 254-bit bit strings and 32-byte byte arrays—but this need is not obvious from reading a formal description of the map.

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

This distinguisher was fixed, along with 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; i.e., the cofactor of the curve is 8. To avoid small-subgroup attacks, the base point of X25519 key exchange algorithm is chosen to generate the subgroup of order p. On top of that, secret key scalars are "clamped," forced to be a multiple of 8, which ensures that multiplying *any* point—whether on or off the prime-order subgroup—results in a point on the prime-order subgroup.

But for the purpose of Elligator representation, it is important that the points represented *not* be constrained to the prime-order subgroup, for the simple fact that random strings do not always map to points on that subgroup. Older versions of obfs4proxy had this problem: only representatives of points on the prime-order subgroup were ever sent. A test for this behavior is to multiply observed represented points by p, and check whether the resulting is always the identity element of the elliptic curve group.

One fix for the issue is to add a randomly selected low-order point to public key points, before running them through the Elligator inverse map to obtain a representative. Conveniently, thanks to the special structure of X25519 private keys, the low-order component disappears during the key exchange, so no further algorithmic changes are required. This strategy was adopted in obfs4proxy-0.0.12, but a minor implementation error prevented it from working, until a further correction in obfs4proxy-0.0.14 in September 2022.

The Elligator encoding is, in principle, a solution for the needs of obfs4. And yet, the integration had faults that (again, in principle) undermined the goal. In situations like these, it is good to consider what might be done differently to produce better outcomes. I know I am not alone in having had singular difficulty in reading the Elligator paper. I could not have dreamed of attempting an implementation, and I think many other practitioners were in the same boat. An early reference implementation, a set of test vectors, would have gone a long way to head off errors in production implementations. The existence of a few independent implementations might have served as a cross-check for discrepancies.

I will remark that there is no evidence that any of these features were ever used by censors to identify and block obfs4 connections. There are many possible reasons 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.) One thing that is sure is that it is not the case that obfs4 presents too small a target to be worth the effort of blocking, since there is evidence that the censorship infrastructure in China systematically harvests obfs4 proxy addresses through their distribution channels (and blocks them by IP address, thus avoiding the need for on-line protocol identification),

and censor-originated probes targeted obfs4's predecessor protocols going back several years [3, §5.4]. Allow me to suggest the possibility that censors find it unappetizing to do stateful processing over multiple flows, the kind of processing needed to take advantage of distinguishers like these in past versions of obfs4. I will return to this idea at the end of the talk.

Upgrade progress of public deployments.

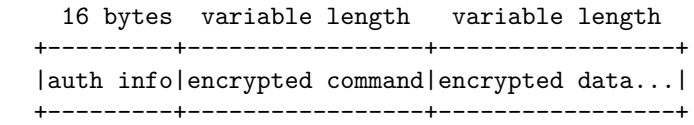
4 VMess header malleability

VMess is another encrypted proxy protocol, developed as part of V2Ray [8]. V2Ray is not itself a protocol; rather it is a framework for constructing proxies by composing and layering multiple protocols. Like the other protocols that have been discussed, VMess is intended to be fully encrypted and indistinguishable from a random stream. It is similar to Shadowsocks in its use of symmetric cryptography and a secret shared between client and server (in this case, a 16-byte UUID specific to a single user), and in that it has native support for specifying a proxy destination. Unlike Shadowsocks and obfs4, VMess is not intended to always be exposed on the wire (although it can be). In a V2Ray-composed proxy, VMess may only be used as the inner proxy protocol inside some other transport protocol, such as WebSocket or QUIC.

V2Fly?

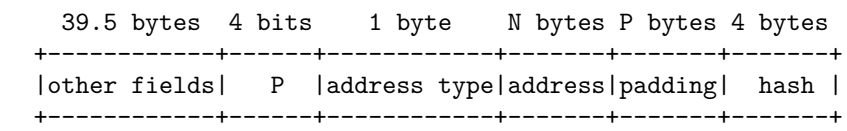
The VMess implementation in V2Ray was vulnerable to an active probing attack to identify VMess servers, caused by the representation of structures in the encrypted protocol header and the protocol parser's interaction with the network. The vulnerability was reported in 2020 by GitHub users p4gefau1t and studentmain [6]. For the description here, I am grateful for the English summary of the vulnerability posted by GFW Report [1]. I will present a somewhat simplified version here (omitting some details of overcoming replay protection). The vulnerability was mitigated in release v4.23.4 by changing how the network layer reacts to malformed client requests.[2]

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 related to the command. The authentication header is a MAC of a recent timestamp (a count of seconds, within 2 minutes of the current time), keyed by the user's random UUID. The command block is encrypted with AES in CFB mode, using a key derived from the UUID and the timestamp. The data stream is encrypted using a key and initialization vector from the command block. The tolerance interval on timestamps in the authentication header permits an attacker to replay previously observed authenticated requests within a short time—this is the first part of the attack.



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

The command block is composed of many fields. I will omit all but the ones that are necessary for understanding the attack.



The command block is encrypted using a stream cipher, but it also has a limited form of integrity protection. A field at the end contains a hash of the preceding bytes, something like a MAC-then-encrypt construction. (The hash is not keyed by a secret, so it is actually more like "hash-then-encrypt," but the distinction does not matter for the attack.) The problem is that the length of the command block is variable, due to the address and padding fields it contains. The receiver of a command block cannot locate the hash to verify it without parsing the very data the hash is meant to protect, which defeats the purpose of integrity protection.

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 four-bit field P, which indicates the size of the padding field. Stop sending the command just after the address type field, before the variable-length part of the command block begins. Then resume sending, one byte at a time, slowly, and count how many bytes are sent before the server closes the connection. Behind the scenes, the VMess server is waiting to receive a full N bytes of address specification, P bytes of padding, and 4 bytes of hash. The hash verification will almost certainly fail, because of the attacker's changes to the ciphertext, but the server will not disconnect until it has read to what it thinks is the end of the hash field. The attacker repeats the replay attack 16 times, tweaking the bits corresponding to P each time, so as to cycle through every four-bit value. The sign for a VMess server is if the server reads $X + P$ bytes in each attempt before disconnecting, where X is constant and P ranges through $\{0, \ldots, 15\}$.

V2Ray developers mitigated this vulnerability 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. These actions are along the lines of the recommendations of Frolov et al. [4, §VI] for probe-resistant proxies.

Mention VLess? (No longer aspires to look fully randomized?)

5 Does it matter?

The cryptographic faults described in this talk were not devastating to the protocols involved—why?

TODO

6 Thanks

Special thanks go to Loup Vaillant, whose efforts in documenting and explaining Elligator, and encouraging safer implementations, have been nothing short of heroic. The detailed, complete, and understandable information at elligator.org critical to my understanding of how Elligator is used in obfs4.

I also want to thank Yawning Angel for fixes in obfs4proxy; Meskio of the Tor Project for reports on the rate of obfs4 bridge upgrades; and GFW Report for the summary of the VMess header malleability vulnerability [1].

Others

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] 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. URL: https://ensa.fi/active-probing/.

[4] Sergey Frolov, Jack Wampler, and Eric Wustrow. Detecting probe-resistant proxies. In *Network and Distributed System Security*. The Internet Society, 2020. URL: https://www.ndss-symposium.org/ndss-paper/detecting-probe-resistant-proxies/.

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

[6] 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 actively probed for distinguishing features (with PoC)*, May 2020. URL: https://github.com/v2ray/v2ray-core/issues/2523.

Check English translation.

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

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

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

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