Scrambling for lightweight censorship resistance

Joseph Bonneau and Rubin Xu

University of Cambridge

Abstract. In this paper we propose scrambling as a lightweight method of censorship resistance, in place of the traditional use of encryption. We consider a censor which can only block banned content by scanning it while in transit (for example using deep-packet inspection), instead of attacking the communication endpoints (for example using address filtering or taking servers offline). Our goal is to greatly increase the workload of the censor by scrambling all data during communication, while maintaining reasonable workloads for the endpoints of the communication network. In particular, our goal is to make it impossible for the censor to effectively accelerate the de-scrambling procedure over what may be achieved by commodity PCs or mobile phones at the endpoints, a goal which we term *high-inertia* scrambling. We also aim to achieve this using the standard JavaScript runtime environment of modern browsers, requiring no distribution or installation of censorship-resistance software.

1 Introduction

Traditional approaches to censorship resistance include steganography and cryptography (including anonymity networks which usually bypass censorship by encrypting all data). Each has the fundamental problem of requiring some metadata to be communicated in the clear prior to communication of potentially censored content, namely the steganographic or cryptographic software which is not normally built in to web browsers.

Several proposals exist for censorship resistance through steganography, or careful hiding of banned content amongst innocuous content [8,5,4]. All of these proposals require special software for communicating parties. Steganography's drawbacks also include poor efficiency and a lack of robustness against a censor who may alter communications.

Other proposals involve the use of cryptography for censorship resistance, making it computationally infeasible for a censor to distinguish between banned and innocuous content. Proposals include storage and publication systems which "entangle" different types of content, making it impossible to delete banned content only [14], or overlay networks which protect all traffic cryptographically [3]. A general-purpose anonymity-network like Tor [7] can also be used for censorship resistance, as all traffic is encrypted.

While cryptographic solutions offer strong resistance to censorship, the protocols are complex and again require special client software and secure distribution of a set of trusted public keys to end-users. For example, censors in Iran have recently attempted to block Tor by blocking the bit sequence of its public Diffie-Hellman parameters. The global Certificate Authority system underpinning TLS/SSL encryption, which is built-in to all web browsers, has recently been called into question for anti-censorship purposes due to the large number of government-controlled CAs and the recently-leaked existence of commercially available equipment to perform real-time middleperson attacks on TLS sessions given access to a CA's private key [12].

We focus on resisting *passive censors*, who will merely try to halt the transmission of banned content, and *active censors*, who may try to investigate and intimidate the end-points of communication. Cryptography frustrates passive censorship by making it computationally infeasible for censors to tell the difference between banned information and legal information. However, encryption is not strictly necessary for resisting passive censorship if secrecy of the banned content is not important. Cryptography may also be insufficient because, as we have outlined, it still requires distribution of trusted keys and decryption can often be accelerated by a censor (which is of considerable importance for censorship to scale to deep packet inspection of significant levels of Internet traffic).

Against a passive adversary who has potentially compromised the root of trust for cryptographic communication, we propose censorship without encryption but with simpler *scrambling*. We only require that endpoints have a trustworthy computation environment (for example, a modern web browser) which can compute a publicly available de-scrambling algorithm, which in practice can be transmitted as in-page JavaScript which de-scrambles data received through AJAX requests. This is similar to the function of many existing dynamic web pages.

Faced with a large volume of such scrambled messages (whether encrypted or due to simpler scrambling), a censor will be unable to block its desired set of data without either de-scrambling all data or over-censoring. Thus, a scrambling function which is sufficiently difficult to de-scramble can frustrate a central censor while not preventing endpoints from communicating. We are particularly interested in scrambling which cannot be practically accelerated, which we term high-inertia scrambling. That is, the censor must do as much work as all communication endpoints are willing to do and cannot "cheat" by investing in custom hardware. Another way of framing the problem is that we want de-scrambling to be optimised to run in the web browsers of commodity PCs, and have no significantly faster implementation on another platform.

In common with steganographic and cryptographic censorship-resistance schemes, scrambling can only be effective if some non-banned content is scrambled as well, preventing the censor from simply blocking any content which looks scrambled. However, scrambling possesses two potential advantages for web content. First, a large amount of content is already effectively scrambled in the form of obfuscated Javascript which unpacks page content dynamically in client browsers. Second, due to the ability of all current web browsers to de-scramble content, a large site such as Google or Wikipedia could scramble all served content without seriously inconveniencing any of its users. This would make blocking such a large site effectively all-or-nothing for the censor.

2 Scrambling

For our purposes, a scrambling function S is any function[1] which takes an input x and produces an output y from which no information about x can be recovered more efficiently than running the de-scrambling function S^{-1}. Encryption can be thought of as a special type of scrambling for which S^{-1} is intractable to compute without the secret key k. We are instead interested, however, in scrambling functions for which S^{-1} is either obvious[2] or is transmitted as a header along with the scrambled data y, enabling data to be transmitted in scrambled form without any key management.

2.1 Required properties of a scrambling function

– **One-way** An adversary given $y = S(x)$ should not be able to compute any information about x in a more efficient way than completely running $S^{-1}(y)$.
– **Randomised** Computing $y_1 = S(x, r_1)$ and $y_2 = S(x, r_2)$ with different random seeds $r_1 \neq r_2$ must produce different outputs $y_1 \neq y_2$. Given y_1 and y_2, an adversary should be unable to determine if they represent the same input by any more efficient method than computing S^{-1}. If one-wayness is satisfied this implies an **indistinguishability** property, namely that an adversary given y_1, y_2 cannot determine if $S^{-1}(y_1) = S^{-1}(y_2)$ without computing both de-scramblings.
– **Universal and compact de-scrambling** S^{-1} must be computable from a compact description on a widely available computing platform. This ensures that code for S^{-1} can be transmitted along with y to enable the scheme to be used by clients with no special set-up. In practice, JavaScript is an obvious choice due to its ubiquitousness and common use to de-pack websites.
– **Difficult to accelerate** It should not be possible for an adversary seeking to compute S^{-1} on n inputs $y_1 \ldots y_n$ to do so using less than $\Theta(n)$ times more resources than an ordinary user computing S^{-1} once on commodity hardware. This should include resources of electricity, power, memory, and computation time. Fortunately, bulk hardware acceleration for JavaScript is not an active area of research.
– **Variable strength** It should be possible to parameterise S such that the resulting S^{-1} can require any desired amount of resources to compute. Given with the difficulty of acceleration, this enables users to devote any available idle resources to computing S^{-1} and force the censor to perform as much work as possible.

[1] Technically speaking, S is likely not to be a true mathematical function but a randomised multivalued function which can map the same input to many possibly outputs.
[2] A classic scrambling function is ROT-13, a Caesar cipher with a fixed alphabetic shift of 13 which is used to hide "spoilers" in online forums.

– **Asymmetric cost** Computing S should be significantly cheaper than computing S^{-1}. This is necessary to prevent bottlenecks if many users are accessing content from one server and to effectively push out all work the censor is being forced to do to the network edges.
– **Adjustable resource usage** To avoid the problems of proof-of-work systems, for which adversaries can realistically expend more computation time than legitimate users [9], it may be desirable to make computing S^{-1} utilise other resources. In particular, memory latency-bound functions [6] are considerably more difficult for an adversary to compute relative to most end-user machines.
– **Human-interactive de-scrambling** A particularly useful example of the above idea is to require human computational abilities in addition to standard machine computation to compute S^{-1}. This can be achieved by incorporating CAPTCHA-solving into S^{-1}. This is not a complete solution, as CAPTCHA-farming and other attacks exist, but puts additional burden on the censor.

3 Outline of a practical implementation

We imagine a practical implementation will be built using the AJAX architecture used in many modern dynamic web pages. A site with some potentially censored material will include JavaScript which will fetch the banned material M in multiple blocks, compute $S^{-1}(M)$ and display the results.

A key design tool is to use a package transform [11], which makes it impossible to compute any information about M until all blocks are available. While we are not aware of any implementations of package transform in pure JavaScript[3], one can be constructed relatively by computing the BEAR-encryption [2] of M using an all-zero key. This can be implemented using standard symmetric primitives like AES and SHA, for which widely-used JavaScript libraries are now available [13].

As a first pass, the server simply computes:

$$G_1, G_2, \dots G_n = \mathrm{BEAR}_0(M_1, M_2, ...M_n)$$

And sends the G blocks to the client using individual AJAX requests. The client can undo the package transform once all of the G blocks are fetched.

This proposal already serves are a simple scrambling system as the cleartext blocks of M are not transmitted, but any client can still de-scramble M. By artificially delaying the transmission of the blocks, this system can force a censor to maintain a significant amount of state, as all blocks from all active transfers must be cached to enable de-scrambling. However, it is still not asymmetric or variable in cost, and admits significant acceleration by hardware implementation of the cryptographic primitives.

[3] Practical desktop libraries do exist for all-or-nothing encryption based using package transforms [10].

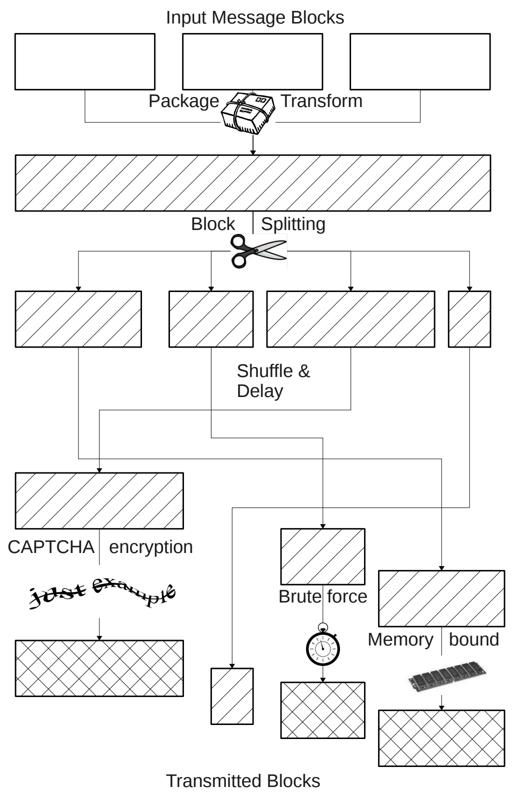

Fig. 1. Multi-stage scrambling of an input message. The package transform ensures that all steps must be undone to recover the original message.

To add these properties, we use a technique similar to that proposed by Anderson to enable multi-party decryption [1]. Any transform applied to any of the G blocks must be undone prior to inverting the package transform and recovering the original message. This enables each block to be scrambled in a different way, allowing us to compose an arbitrary number of scrambling techniques.

A solid implementation would first perform a computationally-intensive scrambling for one or more blocks. This can be easily done in an asymmetric way by encrypting the block with a random 128-bit AES key, discarding 20 of the bits and transmitting the rest, forcing the de-scrambling routine to search over the rest. Another block could be transmitted encrypted by an AES key which is derived from the result of a CAPTCHA (likely with additional key-strengthening to frustrate brute-force of the input space). A third block could be transmitted encrypted by a key computed by a memory-bound function. Finally, some blocks could be transmitted with a de-scrambling function making heavy use of quirks in JavaScript as well as its built-in floating point, date, and string processing libraries, forcing a censor to implement all of these in any accelerated environment. A diagram of this implementation is shown in Figure 1.

3.1 Future desirable properties

We believe our simple proposal can be improved upon in many interesting and exotic ways. We propose two here as areas for future exploration:

- **Busy beaver traps** Assuming the censor will attempt to compute S^{-1} for a large amount of content seen, it may be useful to introduce some functions S^{-1} into the system which in fact contain infinite loops and will never terminate. It can be assumed that end-users will simply give up on these and re-communicate the desired content, or perhaps will have agreed out-of-band on some messages to ignore. If some important content is scrambled with intentionally very high de-scrambling cost, then it may be difficult for the censor to detect which content will result in infinite loops. The existence of busy beaver functions in complexity theory which do an enormous amount of work before terminating despite their short descriptions indicates the difficulty for a censor detecting these traps.
- **Re-scrambling** It may be very useful if scrambled content y can be re-scrambled by a re-scrambling function S', such that if $y = S(x)$ and $y' = S'(y)$, we have both $x = S^{-1}(y)$ and $x = S^{-1}(y')$. Ideally, computing $S^{-1}(y)$ or $S^{-1}(y')$ will be of similar difficulty and an adversary cannot determine that y and y' are scramblings of the same input without computing S^{-1}. Such re-scrambling would fit nicely with the existing theory of mix networks and re-mailers. If re-scrambling is considerably cheaper than initial scrambling, then it may also be a technique for asymmetric strength scrambling, as a server could scramble x once and then repeatedly re-scramble each time it must be transmitted.

Acknowledgements

We thank Steven Murdoch for providing inspiration for this project, as well as Ross Anderson and Richard Clayton for helpful comments.

References

1. Ross Anderson. The Dancing Bear: A New Way of Composing Ciphers. In Bruce Christianson, Bruno Crispo, James Malcolm, and Michael Roe, editors, *Security Protocols*, volume 3957 of *Lecture Notes in Computer Science*, pages 231–238. Springer Berlin / Heidelberg, 2006. 10.1007/11861386_26.
2. Ross Anderson and Eli Biham. Two practical and provably secure block ciphers: BEAR and LION. In Dieter Gollmann, editor, *Fast Software Encryption*, volume 1039 of *Lecture Notes in Computer Science*, pages 113–120. Springer Berlin / Heidelberg, 1996. 10.1007/3-540-60865-6_48.

3. Michael Backes, Marek Hamerlik, Alessandro Linari, Matteo Maffei, Christos Tryfonopoulos, and Gerhard Weikum. Anonymity and Censorship Resistance in Unstructured Overlay Networks. In *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems: Part I*, OTM '09, pages 147–164, Berlin, Heidelberg, 2009. Springer-Verlag.
4. Arati Baliga, Joe Kilian, and Liviu Iftode. A web based covert file system. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 12:1–12:6, Berkeley, CA, USA, 2007. USENIX Association.
5. Sam Burnett, Nick Feamster, and Santosh Vempala. Chipping away at censorship firewalls with user-generated content. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 29–29, Berkeley, CA, USA, 2010. USENIX Association.
6. Jon Crowcroft, Tim Deegan, Christian Kreibrich, Richard Mortier, and Nicholas Weaver. Lazy Susan: dumb waiting as proof of work. Technical Report UCAM-CL-TR-703, University of Cambridge, 2007.
7. Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The next-generation onion routern. In *USENIX 07: Proceedings of 13th USENIX Security Symposium*. USENIX Association, 2004.
8. Nick Feamster, Magdalena Balazinska, Greg Harfst, Hari Balakrishnan, and David Karger. Infranet: Circumventing Web Censorship and Surveillance. In *Proceedings of the 11th USENIX Security Symposium*, pages 247–262, Berkeley, CA, USA, 2002. USENIX Association.
9. Ben Laurie and Richard Clayton. Proof-of-work proves not to work. *WEIS 04: The Third Workshop on the Economics of Information Security*, 2004.
10. Barath Raghavan. Staple project. `http://sysnet.cs.williams.edu/staple/`, 2009.
11. Ronald L. Rivest. All-or-Nothing Encryption and the Package Transform. In *Proceedings of the 4th International Workshop on Fast Software Encryption*, FSE '97, pages 210–218, London, UK, 1997. Springer-Verlag.
12. Christopher Soghoian and Sid Stamm. Certified Lies: Detecting and Defeating Government Interception Attacks Against SSL . `http://ssrn.com/abstract=1591033`, 2010.
13. Emily Stark, Michael Hamburg, and Dan Boneh. Symmetric Cryptography in Javascript. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 373–381, Washington, DC, USA, 2009. IEEE Computer Society.
14. Marc Waldman and David Mazières. Tangler: a censorship-resistant publishing system based on document entanglements. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, CCS '01, pages 126–135, New York, NY, USA, 2001. ACM.