Autosonda: Discovering Rules and Triggers of Censorship Devices

Jill Jermyn
Columbia University

Nicholas Weaver
International Computer Science Institute
University of California at Berkeley

Abstract

Using censorship to forbid access to certain content on the internet is very common in the world today. Some censorship mechanisms are well-studied, however there remain a large number of techniques that remain unknown. Furthermore, many censorship implementations are dynamic as they attempt to prevent against new circumvention techniques. Current tools often tell us when something is censored, but don't do an automated analysis of the approach nor provide clues about the rule sets used by censorship devices. This paper presents Autosonda, a tool for discovering and studying decision models of censorship devices. Through network traffic alone, Autosonda fingerprints censorship devices by discovering their models and mechanisms for how they enforce rule sets. The strength of Autosonda is demonstrated in a study that we present of 76 web filters currently in use in the New York City metropolitan area. In our study we encounter a great variety of behavior and implementation techniques for blocking prohibited web content. Not only does Autosonda help us to find implementation flaws and rule sets, it also allows us to find circumvention paths for 100% of our test subjects. Being able to perform this type of detailed analysis automatically and at scale is a large contribution for understanding censorship and how device behavior can be classified.

1 Introduction

Censorship devices control the type of content that can be accessed, viewed, or published over a network. Censorship most typically refers to nation-state internet censorship, where countries or regions of countries set rules of the censorship devices to prohibit types of content accessed by citizens within the boundaries of that country's networks. It can also be employed through means such as web proxies, typically by using commercial filtering software, or even enterprise data leak prevention. Although the extent to which a network is censored differs on a country or enterprise basis, most censors act based on characteristics of the network traffic they examine. These characteristics can be protocol-dependent, such as packet header fields, or protocol-independent, such as simply looking for a keyword in a packet's content. Censors use these traffic characteristics to make decisions on how they handle the traffic that they intend to censor, typically by blocking it, modifying its content, or injecting packets into the network stream.

Internet censorship is extremely pervasive in the world today. In 2015 more than 70% of countries employed some type of censorship [11]. Although censorship is so common, many people oppose it. According to an Internet Society survey, 83% of people worldwide believe that access to the internet is considered a basic human right [12]. As a result, censorship evasion techniques have become very popular among citizens in censored countries. The Tor anonymity network [3] and encrypted VPN connections are two examples of tools that have been commonly used for censorship evasion. In 2014 an estimated 400 million people were using VPNs to circumvent censorship or increase privacy [17]. Because such tools are widespread, censorship device rule implementations are frequently updated to adapt to and prevent these anticensorship mechanisms. Once censorship device rules change, anticensorship tools must be updated to evade the censors using new techniques. This creates very frequent reactive behavior between censorship rule creators and engineers of the anticensorship tools.

In order for developers to improve upon their anticensorship tools, they must understand something about how the censorship device makes decisions and how its rules are implemented. However, developers cannot directly access the device rule implementation; from their point of view, the device is just a black box. However, they can reverse engineer rules by specially crafting traffic and probing the censor to examine its output. Such approaches have been applied to specific censors, such as the Great Firewall of China [21], however the large majority of censorship techniques and how often they change remain unknown. Existing methods to date for discovering censorship techniques are mostly manual, which are time consuming, not scalable, and are not feasible in the long run, given the dynamic nature of censor implementations. Censor devices could be activated on a range of behavior, but finding the exact trigger is often too arduous a task to be done manually at scale. Yet, such knowledge is necessary for properly analyzing censorship devices, comparing their models, and designing evasion techniques. Although there currently exist tools for discovering *when* something is censored, we do not have tools to tell us *how* the censorship is done and the details and implementation of the rule enforcement.

In this paper we present Autosonda, a tool used for automated rule reverse engineering and fingerprinting of censorship middleboxes. Our solution uncovers the model and mechanism used by a censor for making decisions on how to handle traffic, as well as identifies the censor's approach used to block content that is deemed prohibited. To identify feature triggers, our tool runs a series of protocol-aware probe tests at each layer and then narrows down its search space to uncover how a particular censor rule is implemented. Our tests are based on those used in previous manual analysis of censors to understand their decision making process. The goal of Autosonda is to capture that knowledge into a tool that performs the tests and analysis automatically and at scale. It can be used not only to create implementation fingerprints of censors, but to also track how the fingerprints change over time and compare them with fingerprints of other devices. The following sections will discuss the architecture and implementation of our system and provide results from our experiments on web filters.

2 Related Work

Discovery of censorship techniques has been the topic of much recent research, yet there are still many techniques that remain unknown. Tschantz et al. offer a nice overview of known censorship mechanisms in [18]. Techniques used in Chinese censorship have received particular attention, such as how China censors Web accesses [8], Tor [20], Skype [14], the location of its censoring modules [22], and even how it discovers and blocks privacy tools [9]. Countries such as Bangladesh, Bahrain, India, Iran, Malaysia, Russia, Saudi Arabia, South Korea, Thailand, and Turkey have received some preliminary analysis in [19]. In [4], Aase et al. extend the analysis of censorship mechanisms by focusing on finding motivation, resources, and time elements of censorship. From an ISP perspective, Clayton studied the British Telecom CleanFeed blocking system in [7].

Although there has been significant work on discovering censorship mechanisms, the bulk of the techniques have been performed manually, which is the primary motivation for our work, and have been targeted to specific types of censors. The authors of [13] propose an approach for fingerprinting censorship devices that serves as a motivation for our work. However, the research was limited only to the Great Firewall of China and the analysis was performed manually. We used the results of this research to validate our results obtained from Autosonda on the Great Firewall.

Some existing tools similar to Autosonda have been developed for different purposes. ooniprobe [2] is an app that measures internet censorship and performance. It can be used to discover which websites are blocked by a censor and if there is a system on the network that can be responsible for censorship or surveillance. However,

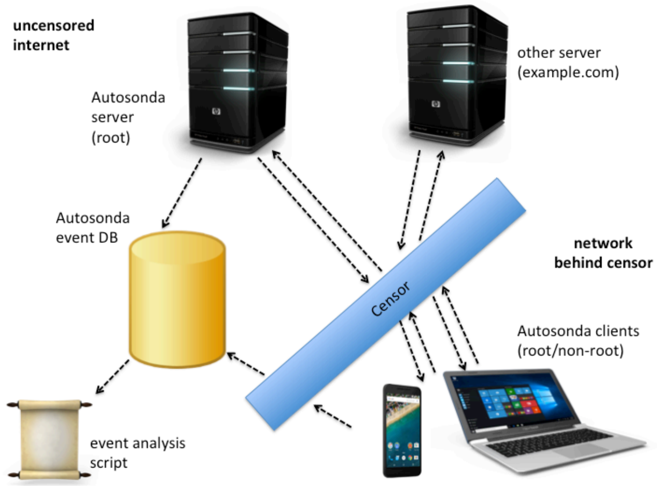

Figure 1: Autosonda architecture

it doesn't focus on finding the exact mechanism that is used to censor a particular site, for example how Autosonda narrows in on regular expressions used by censors. The uses of ooniprobe are different than those of Autosonda, and these tools could be used together to do a full analysis of where there exists censorship, which sites are blocked, and how the filtering mechanisms work. Netalyzr [15] runs a series of tests that probe a network for both measurement and debugging in order to discover a wide range of properties of users' internet access.

3 Tool design and implementation

Figure 1 shows the architecture of Autosonda. It consists of a client device located within a censored country or behind a filtering agent and a set of custom servers running on Amazon EC2 [1] outside of the censored region. The client and server execute a series of tests that craft traffic for the purpose of bidirectional probing the censorship device to discover various attributes about its decisions for traffic handling. The client and the server both log events that are stored either locally or on a database outside of the censored region. Once the tests are completed, we do postanalysis of the event logs to determine the results by comparing the actual events with the expected events.

The goal of Autosonda is to create an implementation fingerprint of a censorship device. This involves discovery in three different categories: *model, mechanism*, and *technique*. Model refers to discovery of network traffic features that the censorship device keys in on. Some examples are IP destination address and the Host header field of an HTTP GET request. Autosonda examines features at each layer of the network stack to determine which trigger a censor. Because there are an exponential number of values to test, it is not feasible to discover the entire censor model. However, it is also not necessary to discover the entire model in order to create an implementation fingerprint of a censor. For example, a particular rule set might include a rule such as "block traffic if byte 8 is 0." It could potentially take an exponential number of tries to find such a rule, yet knowledge of it doesn't necessarily impact a significant amount of client traffic. Instead, we aim to discover a subset of the censor's model according to which traffic features have been identified in existing literature, such as [18], as the most impactful. For each of these features, we take protocol layers and semantics into account. For example, if we speculate that TCP port is a feature of interest, we could test the censor by sending packets with different port numbers to see if and how its behavior changes. Another important aspect of a censor's model is its maintenance of state. Autosonda runs a series of tests to determine if the censor maintains state at all and, if so, at which network layer. For each state that is maintained in a censor, there is a point when that state expires. Autosonda runs additional tests to determine the timeout period of a state.

For each feature that triggers a censor, there is a certain mechanism that is used to look for that feature in the network traffic. There could be a particular regular expression implemented in one of the censor's rules that tries to match on a field in a packet header. Autosonda tries to uncover the mechanisms used for various features in a censor's model by utilizing a fuzzing-like approach. Fuzzing is a software testing technique that provides different types of input to a program to test how it responds. It typically uses unexpected or random data as input, while Autosonda uses a more protocol-aware approach for crafting its input data. For different protocols, Autosonda takes tokens in client messages and applies fuzzing to the delimiters and values of each of these tokens. An example of Autosonda's fuzzing technique is shown in Figure 2. This example assumes that the censor is keying in on the 'GET' keyword; perhaps it wants to censor every HTTP GET request that it receives. Once Autosonda knows that the 'GET' keyword is a feature of interest, it runs a series of tests to determine the censor's rule for that keyword. In the example, we start with a typical 'GET' and then try to change the capitalization to 'GeT' and then the spacing between 'GET' and the forward slash. Discovering the censor's mechanism is particularly important because of protocol ambiguity. There are often nuances in syntax that are not perfectly specified in protocol specification, leading to ambiguity. Such ambiguities are difficult for middlebox developers to handle when accepting and parsing input and often lead to poorly implemented rules.

Finally, technique refers to the action taken by the censor to prohibit censored content. In the case of the Great Firewall, this could be sending a TCP RST packet to the client that requests forbidden content. It could also be modification of a packet's content or dropping of certain packets once they reach the censor.

```
GET / HTTP/1.1\r\nHost: www.a.com\r\n\r\n
GeT / HTTP/1.1\r\nHost: www.a.com\r\n\r\n
GET/ HTTP/1.1\r\nHost: www.a.com\r\n\r\n
```

Figure 2: Example of a fuzzing approach to determine the rule implemented to match GET

When creating our set of tests, we made several initial assumptions. The first is that there exists some form of censorship on the network and that we have a given string that triggers a censorship event, most typically a censored URL. We get this information by testing URLs that have a high probability of being censored and only include networks that show signs of censorship. We further assume root access on the control servers for our experiments. However, Autosonda can run two sets of tests on the client: one with root access and one without. Therefore, even if it is not possible to have root on the client device, there is still a substantial amount of censorship device discovery that can be done. These tests could be useful, for example, in a scenario where the client is an unrooted smartphone.

Test Sets: Because we focus on web filtering and internet censorship, Autosonda primarily tests TCP, UDP, HTTP and DNS protocols. However, functionality can be extended by simply adding additional tests for other protocols. As mentioned above, Autosonda starts with a censored URL and discovers the censor's model by executing a series of tests to find features that are of interest to the censor. When filtering occurs for a particular domain, the censor typically identifies that domain by URL or IP address. Autosonda first tries to determine which, or both, of these features the censor looks at. Other types of filtering, such as with data leak protection, as well as other types of censorship could also use keywords in data content as triggers. Although we didn't have the opportunity to test Autosonda on these types of filters, its approach could easily be extended to do so.

4 Experiments and Results

To demonstrate Autosonda's utility and use it to discover models of web filtering devices, we performed experiments on 76 censored wifi networks over several months in 2017 in the New York City metropolitan area. The wifi networks were all open (not password protected) and located in establishments such as banks, community institutions, clothing stores, grocery stores, home furnishing stores, restaurants/fast food chains, and medical clinics, to name a few. For the purpose of our experiments, we labeled the networks as censored if we were not able to retrieve content from the number one most popular Adult category site in Alexa's top 500 sites by category [5]. The experiments involved connecting a client mobile device to a wifi network that enforces censorship via web filtering. The client then ran Autosonda to probe the web filter with traffic and gather data about its filtering mechanism. There were three test servers, located on EC2 in the United States, that our clients corresponded with over the series of tests. The client and server tests were implemented in Java and Python, and we used Scapy to specially craft network traffic. Autosonda's tests are designed to discover the model, mechanism, and technique used by a censor only by examining network traffic from both the client and server. There was no physical access or remote direct control of any of these devices or communication with network system administrators during our experiments. We specifically defined the model, mechanism, and technique of the web filters as follows.

The model of a web filter is characterized by the feature that triggers the censorship of a URL. For these experiments we focused on two types of triggers: URL and IP address. We discovered that all of the web filters we tested censored by maintaining a blacklist of one of these two characteristics. As discussed in Section 3, mechanism refers to implementation details for how the web filter performs its censorship. For our experiments, we considered several characteristics for identifying mechanism. First, we looked at how protocol-specific the implementation of the censor is, for example the port and protocols it censors. We then looked at how the censor handles protocol ambiguities such as multiple Host headers in an HTTP request and how it responds to HTTP GET and DNS fuzzing. How the censor maintains state, for which protocols, and for how long were also part of our censor fingerprinting. Finally, we wanted to know if the censor reassembles IP fragments and TCP segments.

The techniques of web filters are the means by which the filters perform their censoring. The types of techniques that we observed in the experiments were modification of HTTP responses and modification of DNS responses. In addition to identifying the model, mechanism, and technique for our 76 censored networks, we also observed the vendor of each web filter when possible (through identifying features of network traffic). We also took note of the percentage of censors whose filtering rules we were able to bypass, which for these experiments was 100%. Overall we found that the approaches taken by web filters are not robust and are easily breakable with a slight change in protocol attributes or using a different protocol to transmit data.

Table 1 shows the web filter vendors we encountered for each category of device. We were often able to infer the vendors by traffic that we received, such as in response messages, but there were several filters that did not give us clues about vendor information (labeled as unknown). About half of the web filter vendors we observed in the DNS filtering category were OpenDNS. Norton ConnectSafe and Amazon were about 15% each. 20% of the devices in the Host header filtering category were Fortinet and 30% were Cisco Meraki.

Device category	**Number of devices**	**Web filter vendor**
DNS filtering	21	OpenDNS, Skydns, Savvis, Amazon, Fortinet, Conversant, Norton Connectsafe
Host header filtering	44	SonicWall, AT&T, Juniper, Fortinet, Cisco Meraki, ZScaler, Global Technology Associates, BHI, 12 unknown
Host header lookup	11	OpenDNS, Cisco Scansafe, Squid Proxy, Wayport, 1 unknown

Table 1: Web filter vendors encountered for each category of device

4.1 Results

Model and Technique: We divided our 76 web filters by the primary traffic characteristic with which they were triggered and their approach to triggering. Autosonda was able to break down the filters into two main categories and one subcategory: DNS filtering, Host header filtering, and Host header lookup. 21 of the filters (27.63%) maintained a DNS blacklist and performed all of their censorship via DNS (**DNS filtering** category). Each of these filters monitored DNS requests and compared the name against a blacklist. If a match was found in the blacklist, the filter overwrote the DNS response to direct the HTTP request to a static block page. The block page is typically maintained by the vendor of the web filter. This category of filters only censored requests that contained the URL of a censored webpage. They did not censor requests containing Host header values of censored URLs when sent to the destination IP address of our servers on EC2. 44 of the 76 web filters (57.89%) censored based on the HTTP Host header of a GET request (**Host header filtering** category). These filters checked the GET Host header against a blacklist of URLs to determine if the response should be blocked. The remaining 11 filters (14.47%) could be classified as a subcategory of the DNS blacklist category. They censored by ignoring the destination IP of an HTTP request and instead doing a DNS lookup of the Host header value in the HTTP request (**Host header lookup** category). Once the filter received a DNS response, it searched for the returned IP in a blacklist of IP addresses. If found, it created a new response with source IP of the Host header URL, wrote content describing that the page is blocked, and sent this message to the client. Source IP addresses of responses coming from censored URLs or our EC2 servers were left unmodified by filters in the Host header filtering category. However, since requests were redirected to static pages with the DNS filtering category, responses received by our clients were from the source IP address of the static pages for this category.

Mechanism: To discover implementation details of our web filters, we ran the tests described above (Section 4). Our tests for understanding how protocol-specific the filter's implementation were involved crafting HTTP requests with slight variations in HTTP attributes. Web filters likely assume that all HTTP requests will be using TCP on port 80. When we sent exactly the same HTTP request using UDP, all of our filters allowed the request and response without any censorship. Thus, the filters were only examining TCP communications. Although HTTP requests and responses are only expected to be on port 80, it is still useful to know that the filters are only looking at TCP traffic. However, when we sent requests using TCP on port 9900, a port not typically used with HTTP, our results differed quite a bit. Most filters did not care that the requests were not on port 80; they censored them anyway. Others, 17 out of the 49 filters that censored requests to our servers, only inspected requests going to port 80 and allowed those going to 9900 without censoring them. None of our filters expected query keywords in the URI field of the HTTP GET request and they all allowed responses from censored URLs if the request included a query in the URL string [16].

RFC 7230 [10] states that HTTP requests with multiple Host headers should be rejected with a 400 response. To see if filters properly implemented HTTP, we ran various tests in which the client device sent an HTTP request with multiple Host headers and changed the ordering of the hosts. These tests yielded interesting results. For our 55 filters that examined Host header, 26 only looked at the first Host header, 27 only looked at the last Host header, and two looked at both. Such poor implementations can lead to severe security problems, clearly demonstrated in our experiments with bypassing security policies, and also with HTTP cache poisoning [6].

We saw a lot of variation among web filter vendors for how they handle TCP segmentation and IP fragmentation reassembly. Of our 44 filters that were not DNS censored, eleven did not reassemble TCP segments and seven different filters did not reassemble IP fragments. Five filters had short timeout periods for fragments and segments, just below two seconds, even though the timeout period of HTTP request state was more than 8.5 seconds for all of the filters. For practicality purposes, we did not test state expiry for more than 8.5 seconds.

Since filters in the DNS filtering category censor right at DNS lookups, the mechanism tests for HTTP are not relevant to devices in this category. However, we did perform some additional tests for these devices to understand more about how the filters are implemented. The first test was a DNS fuzzing approach where we made DNS lookups for a censored URL and changed capitalization and domain extensions of the URL. Although capitalization didn't influence any of the filters, changing of domain extension, for example from .com to .org, did manage to return the correct IP address of a URL that should have been filtered for three of the filters. As with any of these filtering approaches, the success of the approach relies on the robustness of the blacklist, which is difficult and time-consuming to maintain. Another important attribute that we wanted to discover with the DNS filtering category is how it handles custom DNS responses messages. For these tests we created our own DNS server and forwarded requests for censored domains to the server from our clients. Fourteen of the web filters modified our DNS responses, while six did not.

The last group of Autosonda's tests take a fuzzing approach to HTTP GET requests in order to test the strength of filters' regular expression matching. Using Scapy, we created request messages with slight modifications to see how the web filters responded for 76 tests. The tables in Appendix A show a subset of the 76 tests that we performed in this group, along with the number of filters that were bypassed with each request. With just simple modifications, many of the web filters allowed censored content to bypass their policies. Among all the filters we see a great deal of variation in behavior. These results really demonstrate the utility of Autosonda, that it is able to discern slight variations and find trends in censor behavior by trying many tests that modify subtle details of traffic. These results give us many clues about how filters implement their regular expressions. Note that actual retrieval of prohibited content depends on the implementation of the server. A server can intentionally be liberal in the formatting of HTTP requests that it accepts or it can also unintentionally contain bugs in its rules for parsing requests. Regardless, our goals were to test the implementation of the filter rather than the server, so our control servers returned an HTTP 200 response and content for any request it received.

The categories that allowed the most bypasses were long hostname tests, Host word tests, \r\n tests, and after Host tests. The results for long hostname tests give us intuition that most filters do not search entire hostnames for censored URL substrings, however the Host substring tests yield mostly censored results. A likely explanation is that filters are looking at the first or last part of the hostname to search for a censored URL. When the URL occurs in the middle of a long string, such as in the long hostname tests, the filters usually do not find it.

Most filters hardcode the Host word in their regular expressions, according to the results in Table 5. It also seems that their regular expressions are rather particular, since we see a majority of filters bypassed with strings that change the spacing and characters in and around the Host word. However, changes in capitalization seem to be caught by most of the filters, which we can see when we tried "HoSt". Filters' regular expressions just before Host are also particular. We can see in Table 8 that nearly all of the filters were bypassed by removing the \r\n and spacing before Host. Filters are likely using these tokens as delimiters to split sections of the GET request. Unsurprisingly, filters were easily bypassed in our after Host tests, Table 10. In these request strings we placed the censored URL in the X-id field and then modified tokens just after Host. Clearly most of the filters look only at the Host field for a censored URL and ignore everything afterward. Thus, with special implementation of a server, a client could craft a request by putting a censored URL after the Host field and bypass the policies of the filter.

Web filter vendors: A very interesting result that we noticed is the diversity of behavior among web filters of the same vendor. For example, we observed 13 Cisco Meraki web filters and found that only two of them considered both Host fields when multiple Hosts were used in a request, whereas the other 11 filters only looked at the last Host. Similarly, two different Meraki filters used a timeout period of less than two seconds for IP fragments, where the other 11 filters had a timeout of over 8.5 seconds. Some logical explanations for behavior diversity are that some of the filters run more updated software or perhaps there are differences in customer preferences. Not only did we see subtle variations in implementation among the same vendors, but we also noticed completely different approaches taken to filtering by the same vendors. Fortinet filters, for example, accounted for eight of our Host header filtering category devices. These devices solely considered an HTTP request Host header when making filtering decisions. Yet, we found two filters also by Fortinet that performed only DNS filtering and completely ignored the HTTP Host field.

Bypassing filtering mechanisms: Our experiments enabled us to bypass 100% of the web filters we tested. Although all of the approaches that we took for bypassing were protocol-related, it is also worth mentioning that since all of the web filters we studied were blacklist-based and did not do content filtering, data could easily be transferred through any URL/IP not on the blacklist. Although Autosonda's tests did not account for content filtering, its fuzzing approach could be utilized to handle these types of implementations.

Since all filters in our DNS filtering category worked by modifying DNS responses, they were easily bypassed by sending HTTP GET requests directly to an IP address rather than performing a DNS lookup of a URL. Similarly, HTTP requests made to our servers on EC2 with a censored Host header also successfully returned responses without modification for the DNS filtering category. As mentioned above, sending HTTP requests over UDP bypassed filters 100% of the time, as did adding query keywords to the URI field of requests and using multiple Host headers. Our HTTP request fuzzing approach also easily bypassed filters for the Host header filtering category. For 76 different tests, we saw that individual filters failed to censor up to 65 of those tests for SonicWall, 60 for Cisco Meraki, 52 for ZScaler, 52 for Juniper, 38 for AT&T, and up to 51 for unknown vendors.

The web filters on which we ran our experiments were likely programmed to block access to specific categories of websites, which is why they keep blacklists of URLs or IP addresses to block. A blacklist approach is difficult to maintain, since websites are constantly changing and content can be moved around to different sites, easily allowing one to bypass the filter. To even attempt to keep blacklists up-to-date, they need to be frequently pushed updates. During our experiments we ran some additional tests to see how robust the blacklists were for our test filters. To do this, we downloaded the Alexa top 100 Adult category sites on the web and tried to connect to them through each filter that blocked the number one most popular site. Not one of the filters we tested blocked access to all 100 of the sites, and some blocked as low as 31 sites. Also interesting to see was that different filters of the same vendor blocked different subsets of sites.

Limitations: Using Autosonda over time to fingerprint sophisticated censorship devices could potentially lead to a few complications if a censor actively tries to evade a censorship discovery tool. A censor could block traffic to or from the EC2 servers or try to fingerprint Autosonda by looking for specific types of tests performed in sequence. However, Autosonda could be extended to periodically modify the IP addresses of its servers or to randomize its testing.

5 Conclusion

Discovering censorship decision models is useful to find evasion techniques and strengthen implementations. However, it is difficult to perform the analysis in depth and at scale. We introduce Autosonda, a tool used for automated rule reverse engineering and fingerprinting of censorship middleboxes. Autosonda is used to uncover the model, mechanism, and technique when access to the device is only available through network traffic probing. Through a series of tests across multiple protocols, Autosonda characterizes devices according to their decision models, techniques for enforcing censorship, and discovers clues about the regular expressions used by these devices for rule enforcement. The value and effectiveness of our tool is demonstrated by using it in a study of 76 web filters, where we discover a variety of implementation and decision-making techniques. Autosonda enabled us to find methods for bypassing 100% of the filters we studied and categorize common implementation flaws and rule sets for popular device vendors.

6 Acknowledgements

This work was supported by NSF CNS-1518918, CNS-1237265, and CNS-1514509.

References

[1] https://aws.amazon.com/ec2/.

[2] ooniprobe - Measure Internet Censorship & Performance. https://ooni.torproject.org/post/ooni-mobile-app/.

[3] The Tor Project. https://www.torproject.org/.

[4] AASE, N., CRANDALL, J. R., DIAZ, A., KNOCKEL, J., MOLINERO, J. O., SAIA, J., WALLACH, D. S., AND ZHU, T. Whiskey, weed, and wukan on the world wide web: On measuring censors' resources and motivations. In *FOCI* (2012).

[5] ALEXA. The top 500 sites on the web by category, 2017. http://www.alexa.com/topsites/category/Top/Adult.

[6] CHEN, J., JIANG, J., DUAN, H., WEAVER, N., WAN, T., AND PAXSON, V. Host of troubles: Multiple host ambiguities in http implementations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1516–1527.

[7] CLAYTON, R. Failures in a hybrid content blocking system. In *International Workshop on Privacy Enhancing Technologies* (2005), Springer, pp. 78–92.

[8] CLAYTON, R., MURDOCH, S. J., AND WATSON, R. N. Ignoring the great firewall of china. In *International Workshop on Privacy Enhancing Technologies* (2006), Springer, pp. 20–35.

[9] ENSAFI, R., FIFIELD, D., WINTER, P., FEAMSTER, N., WEAVER, N., AND PAXSON, V. Examining how the great firewall discovers hidden circumvention servers. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference* (2015), ACM, pp. 445–458.

[10] FIELDING, R., AND RESCHKE, J. Hypertext transfer protocol (http/1.1): Message syntax and routing. rfc 7230 (proposed standard), June 2014.

[11] FREEDOM HOUSE. Freedom on the Net 2015, 2015.

[12] INTERNET SOCIETY. Global Internet User Survey, 2012. http://www.internetsociety.org/surveyexplorer/.

[13] KHATTAK, S., JAVED, M., ANDERSON, P. D., AND PAXSON, V. Towards illuminating a censorship monitor's model to facilitate evasion. In *Presented as part of the 3rd USENIX Workshop on Free and Open Communications on the Internet* (2013).

[14] KNOCKEL, J., CRANDALL, J. R., AND SAIA, J. Three researchers, five conjectures: An empirical analysis of tom-skype censorship and surveillance. In *FOCI* (2011).

[15] KREIBICH, C., WEAVER, N., NECHAEV, B., AND PAXSON, V. Netalyzr: illuminating the edge network. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement* (2010), ACM, pp. 246–259.

[16] LEACH, P. J., BERNERS-LEE, T., MOGUL, J. C., MASINTER, L., FIELDING, R. T., AND GETTYS, J. Hypertext transfer protocol–http/1.1.

[17] MARCELLO MARI. How Facebook's Tor service could encourage a more open web. The Guardian, December 2014.

[18] TSCHANTZ, M. C., AFROZ, S., PAXSON, V., ET AL. Sok: Towards grounding censorship circumvention in empiricism. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 914–933.

[19] VERKAMP, J.-P., AND GUPTA, M. Inferring mechanics of web censorship around the world. In *FOCI* (2012).

[20] WINTER, P., AND LINDSKOG, S. How china is blocking tor. *arXiv preprint arXiv:1204.0447* (2012).

[21] WINTER, P., AND LINDSKOG, S. How the great firewall of china is blocking tor. In *FOCI* (2012).

[22] XU, X., MAO, Z. M., AND HALDERMAN, J. A. Internet censorship in china: Where does the filtering occur? In *International Conference on Passive and Active Network Measurement* (2011), Springer, pp. 133–142.

A Appendix: HTTP Fuzzing Results

Tables 2-12: Fuzzed HTTP GET request tests and number of web filters that were bypassed out of the 44 that examined HTTP request strings (the first %s is replaced by a censored URL; the second, when present, is replaced with a test id)

HTTP GET request	**Number of web filters bypassed**
GET / HTTP/1.1\r\n Host: xxxxxxxxxxxxx%sxxxxxxxxxxxxxxxxxxx\r\nX-id:%s\r\n\r\n	34
GET / HTTP/1.1\r\nHost:xxxxxxxxxxx%sxxxxxxxxx xxxxxxxxxxxx\r\nX-id:%s\r\n\r\n	33
GET / HTTP/1.1\r\nHost:xxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxx\r\nX-id:%s\r\n\r\n	35

Table 2: long hostname tests

HTTP GET request	**Number of web filters bypassed**
GeT / HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	16
/ HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	28
/ HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	29
get / HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	20
XXX / HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	24
GE / HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	24

Table 3: GET word tests

HTTP GET request	Number of web filters bypassed
GET / HTTP/\r\nHost: %s\r\nX-id:%s\r\n\r\n	21
GET / http/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	18
GET / HTTP/1.\r\nHost: %s\r\nX-id:%s\r\n\r\n	20
GET / HTT/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	22
GET / /1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	21
GET / /1\r\nHost: %s\r\nX-id:%s\r\n\r\n	22
GET / HTTP/ \r\nHost: %s\r\nX-id:%s\r\n\r\n	19
GET / /\r\nHost: %s\r\nX-id:%s\r\n\r\n	20
GET / HtTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	6
GET / /11.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	21
GET / XXXX/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	22
GET / HTTP9\r\nHost: %s\r\nX-id:%s\r\n\r\n	21
GET / HTTP\r\nHost: %s\r\nX-id:%s\r\n\r\n	20
GET / \r\nHost: %s\r\nX-id:%s\r\n\r\n	20
GET / HTTP/9\r\nHost: %s\r\nX-id:%s\r\n\r\n	20

Table 4: HTTP word tests

HTTP GET request	Number of web filters bypassed
GET / HTTP/1.1\r\n%s\r\nX-id:%s\r\n\r\n	38
GET / HTTP/1.1\r\nHost%s\r\nX-id:%s\r\n\r\n	36
GET / HTTP/1.1\r\n Host: %s\r\nX-id:%s\r\n\r\n	23
GET / HTTP/1.1\r\nXXX: %s\r\nX-id:%s\r\n\r\n	29
GET / HTTP/1.1\r\nH: %s\r\nX-id:%s\r\n\r\n	38
GET / HTTP/1.1\r\nHostwww.%s\r\nX-id:%s\r\n\r\n	37
GET / HTTP/1.1\r\nHoSt: %s\r\nX-id:%s\r\n\r\n	20
GET / HTTP/1.1\r\n %s\r\nX-id:%s\r\n\r\n	38
GET / HTTP/1.1\r\nXXXX: %s\r\nX-id:%s\r\n\r\n	36
GET / HTTP/1.1\r\n: %s\r\nX-id:%s\r\n\r\n	37
GET / HTTP/1.1\r\nHost www.%s\r\nX-id:%s\r\n\r\n	37
GET / HTTP/1.1\r\nHost %s\r\nX-id:%s\r\n\r\n	37

Table 5: Host word tests

HTTP GET request	Number of web filters bypassed
GET / HTTPx/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	21
GET / HTTP /1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	21
GET / HTTP/ 1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	28
GET / HTTP/1.1x\r\nHost: %s\r\nX-id:%s\r\n\r\n	18
GET / HTTP/x1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	19
GET / HTTP/1.1 \r\nHost: %s\r\nX-id:%s\r\n\r\n	16

Table 6: spacing after HTTP tests

HTTP GET request	Number of web filters bypassed
GET / HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\nAccept: text/html, */*;q=0.8\r\n\r\n	0
GET / HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\nAccept:text/html,application/xhtml+xml, application/xml;q=0.9,*/*;q=0.8\r\n\r\n	0
GET / HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\nAccept: text/html\r\n\r\n	0
GET / HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\nAccept: text/html,application/xhtml+xml, application/xml;q=0.9,*/*;q=0.8\r\n\r\n	0
GET / HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\nAccept:text/html\r\n\r\n	0
GET / HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\nAccept:text/html,*/*;q=0.8\r\n\r\n	0

Table 7: Accept tests

HTTP GET request	Number of web filters bypassed
GET / HTTP/1.1\rHost: %s\r\nX-id:%s\r\n\r\n	21
GET / HTTP/1.1\nHost: %s\r\nX-id:%s\r\n\r\n	2
GET / HTTP/1.1 Host: %s\r\nX-id:%s\r\n\r\n	36
GET / HTTP/1.1Host: %s\r\nX-id:%s\r\n\r\n	34

Table 8: \r\n tests

HTTP GET request	Number of web filters bypassed
GET / HTTP/1.1\r\nHost: [host without extension]\r\n\r\n	23
GET / HTTP/1.1\r\nHost: %s#\r\nX-id:%s\r\n\r\n	20
GET / HTTP/1.1\r\nHost:%s\r\nX-id:%s\r\n\r\n	4
GET / HTTP/1.1\r\nHost:www.%s\r\nX-id:%s\r\n\r\n	6
GET / HTTP/1.1\r\nHost:x%s\r\nX-id:%s\r\n\r\n	20
GET / HTTP/1.1\r\nHost: www.%s\r\nX-id:%s\r\n\r\n	1
GET / HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	1
GET / HTTP/1.1\r\nHost: www.%s\r\nX-id:%s\r\n\r\n	0
GET / HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	0

Table 9: hostname tests

HTTP GET request	Number of web filters bypassed
GET / HTTP/1.1\r\nHost:.com\r\nX-id:%s\r\n\r\n	36
GET / HTTP/1.1\r\nHost: \r\nX-id:%s\r\n\r\n	29
GET / HTTP/1.1\r\nHost:a.com\r\nX-id:%s\r\n\r\n	35
GET / HTTP/1.1\r\nHost:\r\nX-id:%s\r\n\r\n	30

Table 10: after Host tests

HTTP GET request	**Number of web filters bypassed**
GET/ HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	26
GET z HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	24
GET ? HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	21
GET HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	26
GET /HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	12
GETHTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	28
GET/HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	27
GET**HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	25
GET /xHTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	13
GET HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	22
GETx/ HTTP/1.1\r\nHost: %s\r\nX-id:%s\r\n\r\n	29

Table 11: Request-URI tests

HTTP GET request	**Number of web filters bypassed**
GET / HTTP/1.1\r\nHost:[host+host]\r\n\r\n	20
GET / HTTP/1.1\r\nHost: [host+host]\r\n\r\n	22

Table 12: Host substring tests