NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

Defense Implementation for Website Fingerprinting Attacks on Nginx Web Server

Panagiotis P. Kokkinakos

Supervisor: Konstantinos Chatzikokolakis, Associate Professor

ATHENS

JULY 2019

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Υλοποίηση Άμυνας για Website Fingerprinting Attacks στον Nginx Web Server

Παναγιώτης Π. Κοκκινάκος

Επιβλέπων: **Κωνσταντίνος Χατζηκοκολάκης,** Αναπληρωτής Καθηγητής

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2019

BSc THESIS

Defense Implementation for Website Fingerprinting Attacks on Nginx Web Server

Panagiotis P. Kokkinakos
S.N.: 1115201400069

SUPERVISOR: Konstantinos Chatzikokolakis, Associate Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Υλοποίηση Άμυνας για Website Fingerprinting Attacks στον Nginx Web Server

Παναγιώτης Π. Κοκκινάκος
Α.Μ.: 1115201400069

Επιβλέπων: Κωνσταντίνος Χατζηκοκολάκης, Αναπληρωτής Καθηγητής

ABSTRACT

Website Fingerprinting attack gives a passive adversary the ability to know which sites a client visits, even when the packages that are being exchanged between the client and the site are encrypted. This is possible by analyzing the network traffic between those two, and extracting network patterns that are unique to each site.

For this kind of attacks, we implement an application-level defense called ALPaCA (*Application Layer Padding Concerns Adversaries)*, as proposed by Giovanni Cherubin, Jamie Hayes, and Marc Juarez. We implement ALPaCA as a Rust library, and develop an Nginx module which uses ALPaCA to protect the sites for which it is enabled.

In this thesis, we implement the first Website fingerprinting defense which can be used on a web server.

The defense's purpose is to lower the adversary's predictive accuracy as of which site the client visits, by altering the network traffic, and specifically the packages from the site's server towards the client.

The code of this thesis can be found at the following links:
Nginx Module
ALPaCA Library

SUBJECT AREA: Web Defense
KEYWORDS: website fingerprinting, privacy, anonymity

ΠΕΡΙΛΗΨΗ

Η επίθεση Website Fingerprinting δίνει σε κάποιον παθητικό επιτιθέμενο την δυνατότητα να ξέρει ποιους ιστοτόπους επισκέπτεται κάποιος πελάτης, ακόμα και όταν τα πακέτα που ανταλλάσσονται μεταξύ του πελάτη και του ιστοτόπου είναι κρυπτογραφημένα. Αυτό είναι δυνατό μέσω της ανάλυσης της διαδικτυακής κίνησης μεταξύ αυτών των δύο, και της εξαγωγής μοτίβων δικτύου που είναι μοναδικά για κάθε ιστότοπο.

Για αυτό το είδος επίθεσης, υλοποιούμε μια άμυνα επιπέδου εφαρμογής που ονομάζεται ALPaCA (*Application Layer Padding Concerns Adversaries)*, όπως προτάθηκε από τους Giovanni Cherubin, Jamie Hayes, και Marc Juarez. Υλοποιούμε το ALPaCA σαν βιβλιοθήκη της Rust και αναπτύσσουμε ένα module του web server Nginx το οποίο χρησιμοποιεί το ALPaCA για να προστατεύσει τους ιστοτόπους για τους οποίους είναι ενεργοποιημένο.

Στην πτυχιακή αυτή, υλοποιούμε την πρώτη άμυνα για Website Fingerprinting επιθέσεις που μπορεί να χρησιμοποιηθεί σε web server.

Ο σκοπός της άμυνας είναι να μειώσει την ακρίβεια πρόβλεψης του επιτιθέμενου όσο αφορά τον ιστότοπο που επισκέπτεται ο πελάτης, τροποποιώντας την διαδικτυακή κίνηση, και συγκεκριμένα τα πακέτα από τον server του ιστότοπου προς τον πελάτη.

Ο κώδικας της πτυχιακής βρίσκεται στους ακόλουθους συνδέσμους:
Nginx Module
ALPaCA Library

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Web Defense

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: website fingerprinting, ιδιωτικότητα, ανωνυμία

To my family.

AKNOWLEDGMENTS

I would like to thank my supervisor, Prof. Konstantinos Chatzikokolakis, for giving me the chance to work on this subject and for his support and encouragement throughout this time.

CONTENTS

LIST OF FIGURES

PREFACE

The work for this thesis was done between November 2018 and July 2019 in Athens. The project was developed on a Linux machine, Rust and C were the programming languages that were used and it was tested using version 1.14.2 of the Nginx web server. For the development of this project it was of great importance to get familiar with the Nginx modules and the Rust programming language.

1. INTRODUCTION

Website fingerprinting attacks are a great threat towards client anonymity, and are particularly effective against *.onion* sites which are anonymous web servers hosted over Tor. A passive adversary can learn which site has been visited by the client, even if they browse through an anonymity network such as Tor and the communication is encrypted. High level features such as the number of requests the browser makes to download a page, the order of these requests and the size of each response, induce distinctive low level features observed in the network traffic. The attack is often formulated as a classification problem. The adversary collects samples of traffic traces from many websites by performing web requests. Previously mentioned features are extracted, and a machine learning classifier is trained to classify websites using those features. When a client browses a web page, the adversary collects the tragic, extracts the features and passes them in the classifier. In the case of a closed-world experiment, the visited website is definitely one of those used to train the classifier, so a prediction about which site was visited is made. In the case of an open-world experiment, the classifier tells the adversary if a site from a specific set was visited.

Most of the existing defenses are often about the network-level, which makes their deployment unrealistic, because of the fact that they may often require great changes of Tor or the TCP stack. ALPaCA, however, is an application-level defense, which is more natural and far easier to implement. There have been proposed two versions of ALPaCA: Deterministic ALPaCA (*D-ALPaCA*) and Probabilistic ALPaCA (*P-ALPaCA*). The purpose of the defense is to lower the adversary's classifier's accuracy by altering the traffic features in order to make the sites less fingerprintable.

In this thesis, we implemented both ALPaCA versions as a Rust library and develop an Nginx module which uses ALPaCA to protect the sites being served by the server.

2. NGINX

Nginx is a web server which can also perform as a reverse proxy, load balancer and more. It uses a single-threaded, modular, event-driven, asynchronous architecture which enables it to outperform other popular web servers and excels at serving static content. It optimizes the usage of memory, CPU and network; therefore it can often serve 10 times more requests compared to the Apache web server. One of its most important features is that the content can be served to the client in chunks. This means that the response starts getting sent to the client before it has been produced or processed in its entirety, which decreases the server response time dramatically.

2.1 Configuration File

It is important to understand some basic concepts about the Nginx's configuration file. There are four *contexts* (called *main*, *server*, *upstream*, and *location*) which can contain directives with one or more arguments. Directives in the main context apply to everything; directives in the server context apply to a particular host/port; directives in the upstream context refer to a set of backend servers; and directives in a location context apply only to matching web locations (e.g., "/", "/images", etc.) A location context inherits from the surrounding server context, and a server context inherits from the main context.

2.2 Nginx Modules

Nginx modules are the building blocks of Nginx and they do most of the work a web server can be associated with. Anyone can develop any kind of module, but Nginx comes with a set of modules implementing most of its functionality called core modules. Nginx modules are developed using the C programming language.

There are 3 types of modules:

- *Handlers* process a request and produce an output
- *Filters* process and manipulate the output produced by a hander
- *Load Balancers* choose a backend server to send a request to, when more than one backend server is available

At server startup, each handler attaches itself to particular locations defined in the configuration file. Each location can only be associated with one handler. There is a default handler which is typically used to serve static files.

If the handler is a reverse proxy to a set of backend servers, a load balancer can also be attached to the location. Nginx comes with two load balancing modules.

If the handler does not produce an error, the filters are called. Multiple filters can be attached into each location, so that a response can be manipulated in different ways. The order of their execution is determined at compile-time. Filters have the classic "chain of responsibility" design pattern: one filter is called, does its work, and then calls the next filter, until the final filter is called. Then, Nginx sends the response.

A typical request processing cycle goes as follows:

A client sends an HTTP request. Nginx chooses the appropriate handler based on the location configuration. A load balancer (if present) picks a backend server. The handler produces the output and passes it to the first filter. The first filter manipulates the response and passes it to the next, until it reaches the last one. Finally, the response is sent to the client.

2.3 Filters

Filter modules are the ones that interest us for our project. They manipulate responses generated by handlers. Header filters manipulate the HTTP headers, and body filters manipulate the response content.

The really interesting part about the filter chain is that each filter doesn't wait for the previous filter to finish; it can process the previous filter's output as it's being produced. Most of the time, the response is chunked into buffers which are linked with each other, constructing a buffer chain. Sometimes there are more than one buffer chains. Filters operate on *one chain at a time*, which means that if there are many chains, a filter is called more than one time for a single response.

Basically, a body filter does one of the following:

- Keeps a buffer intact and passes it to the next filter
- Deletes the buffer from the buffer chain
- Modifies the buffer's contents

In the next section, we will briefly run through the key components of an Nginx filter module without going into too many details.

ngx_command_t: It is a typedef of ngx_command_s, for defining a module directive. A static array of ngx_command_t, containing the directives of a module is passed to Nginx. The array is terminated by a ngx_null_command. ngx_command_t has the following fields.

- **Directive Name**
  An ngx string for the name of the directive.
- **Bitmask**
  Indicates where the directive will be configured (e.g. HTTP, server or location block in the Nginx configuration file). The bitmask also indicates how many and what arguments the directive takes.
- **Set Function pointer**
  A set handler function for saving the directive arguments. Nginx has several pre-defined set functions for saving various directive arguments like boolean, string etc... A custom handler can also be specified.
- **Configuration Structure**
  This specifies the configuration structure passed to the directive handler. If a module directive is configured in the server context/block of the Nginx configuration file, then the server context offset (NGX_HTTP_SRV_CONF_OFFSET) should be specified here. The handler function use this information for locating the right module configuration.
- **Parameter offset**
  This is where the parameter for the module configuration is located. The set handler function will save the directive argument here.
- **Post**
  A secondary function pointer can be specified that will be called after the earlier set function handler has saved the directive argument. This field can also hold a default value that can be used by some of the Nginx pre-defined set functions.

ngx_http_module_t: Module context, a static data structure that defines the handlers for the creation and initialization of a module's configuration struct. It includes handlers that can run pre and post configuration.

A module can have its own configuration struct that contains the parameters it requires. The function handlers defined here are for the creation and merging of the module configuration struct. There are separate pairs of function handlers for the module configuration that appear in Nginx 's main configuration block, server configuration block and location block. There are also two handlers that can run pre and post configuration.

For those handlers that are not needed, NULL can be specified. For example, if a module only has directives in Nginx's location block and it doesn't require merging

values from higher levels, the function handler for creating a location configuration can be specified, while all others set to NULL.

ngx_module_t: This structure is a typedef of ngx_module_s and it defines the module. It is a global variable for each module. At the top of the structure are version information that can be filled by using a macro NGX_MODULE_V1. There are also several unused fields for future extensions at the bottom of the struct that can be filled with NGX_MODULE_V1_PADDING.

For the remaining fields, we are interested in only 3 of them. The rest are handlers that can be called at various points in the Nginx cycle. These are set to NULL. The 3 fields that concern us are as follow.

- **void *ctx;**
  This takes the module context (ngx_http_module_t) which contains the function handlers for creating module configuration struct and merging module configuration.
- **ngx_command_t *commands;**
  This takes a pointer to an array of ngx_command_t. Each ngx_command_t defines a directive that the module takes.
- **ngx_uint_t type;**
  This defines the type of module, such as NGX_CORE_MODULE, NGX_HTTP_MODULE etc.

3. ALPACA

As mentioned earlier, ALPaCA is an application-level, server-side defense for the purpose of tackling website fingerprinting attacks, which comes in two versions; P-ALPaCA and D-ALPaCa. The fact that it doesn't deal with the network layer, makes it attractive and easy to implement. Most existing defenses pad the traffic with extra network packets in order to hide the site's features. This however, is difficult to implement, as significant changes need to be applied to the underlying network protocols. Therefore, in ALPaCA the padding is added to the actual contents of the page, which is more natural and practical. By doing that, it alters the site's high level feature's, which leads to the alteration of its low level features as well.

ALPaCA pads both the HTML of the page, and each of its objects such as the images and the CSS stylesheets. In order to pad binary objects such as images, we can simply append random bytes to the end of the file, without affecting it in any undesirable way. To pad text objects such as HTML and CSS, we add random data into a comment.

ALPaCA chooses a list of sizes T called target, which specifies the number and the size of the objects of the morphed page. The target is chosen in a different way, depending on the ALPaCA version we use. Then, the original objects are padded to match the sizes in T. If the target objects are more than the original objects, fake ALPaCA objects which match the remaining target objects in size are created and added to the HTML.

Next, we describe the steps in order to morph a page.

3.1 Morphing a Page

At first, we choose the target T. We keep a list M containing the morphed objects, and a list P containing the sizes in T that have not been used. We sort the original objects in ascending size order, and do the following: for each original object, we find the smallest size t in T to which the object can be padded (i.e., for which size(object) ≤ t). We remove all the sizes in T smaller than the size we found, pad the object to that size and then we add the morphed object to M. All the sizes removed from T, except the one we used, are added into P. When all the original objects have been padded, the remaining T sizes are also added into P. Then, we create new fake ALPaCA objects, which contain random bytes, according to the sizes in P. Fake ALPaCA objects are downloaded by the browser, but are styled as "hidden", therefore they do not show on the page. Finally, we pad the HTML to a target.

In the next section, we explain the different ways P-ALPaCA and D-ALPaCA select the target T.

3.2 D-ALPaCA

D-ALPaCA (Deterministic-ALPaCA) decides deterministically how much a page and its objects have to be padded. We have to supply three parameters: λ, σ and *max_s*, where *max_s* should be a multiple of σ. The number of objects in the page is padded to the next multiple of λ which is equal or greater than the original objects' number, and the size of each object is padded to the next multiple of σ which is equal or greater than its original size. If there is a need to create fake ALPaCA objects, their size is sampled uniformly at random from {σ, 2σ, ..., *max_s*}. Finally, the html is padded to the next multiple of σ.

3.3 P-ALPaCA

P-ALPaCA (Probabilistic-ALPaCA) uses distributions to generate the target T. We have to supply three probability distributions: Dn, Dh and Ds. Dn defines the number of objects a page has, Dh defines the size of the HTML, and DS defines the size of each object. P-ALPaCA samples the target T using these distributions, and morphs the original page as described previously.

The defense first samples the number of objects n for the morphed page according to Dn. Then, it samples the size of the morphed HTML from Dh, and n sizes from Ds. It attempts to morph the original page to T, and if morphing fails the procedure is repeated. Sampling from the distribution can always produce very large targets T, so a parameter *max_bandwidth* is supplied. The total page size has to be smaller than or equal to this parameter. If not, the procedure is repeated. Setting a very low *max_bandwidth* value in order to avoid extremely high bandwidth overheads is not the correct approach, as this would set a limit to the page size, removing the possibility that it is morphed to resemble a larger site.

4. IMPLEMENTATION

We implemented the ALPaCA defense as a Rust library. We also developed an Nginx module which uses the library in order to protect the sites that have it enabled, from website fingerprinting attacks. We chose the Rust programming language because of its compatibility with C code. The fact that Rust doesn't use complicated features like garbage collection makes it very easy to communicate with C APIs. Rust code is compiled into a library, exposing certain functions which can be called from the C code, in our case from our Nginx module.

In the next sections we describe the implementation and functionality of both the Nginx module and the Rust library, and how they communicate with each other.

4.1 Module Implementation

Our Nginx module which is called **ngx_http_alpaca_module** is a filter. It attaches itself in the chain of filters, and operates on the response generated by an Nginx handler. At a high level, what is does is the following: At first it receives the HTML to be sent to the client, passes it to a library function, gets the modified HTML and passes it onto the next filter. Then, for each one of the requested objects, it receives its contents, calls a library function, receives the extra padding for the object and appends it to the objects' buffer chain (which contains its contents as mentioned earlier).

4.1.1 Configuration

The filter is controlled by eight directives which are placed in the Nginx's configuration file:

- **apaca_prob**
  It takes **on** or **off** as its argument, depending on whether the user wants to use the probabilistic version
- **alpaca_deter**
  It takes **on** or **off** as its argument, depending on whether the user wants to use the deterministic version
- **alpaca_dist_html_size**
  Its argument is the distribution to be used for the probabilistic version in order to sample the size of the HTML
- **alpaca_dist_obj_number**
  Its argument is the distribution to be used for the probabilistic version in order to sample the number of objects

- **alpaca_dist_obj_size**
  Its argument is the distribution to be used for the probabilistic version in order to sample the size of each object
- **alpaca_obj_num**
  Its argument is the λ parameter for the deterministic version. The number of objects in the morphed HTML will be a multiple of it
- **alpaca_obj_size**
  Its argument is the σ parameter for the deterministic version. The sizes of objects in the morphed HTML will be a multiple for it
- **alpaca_max_obj_size**
  The *max_s* parameter for the deterministic version. It has to be a multiple of σ.

The arguments for the alpaca_dist_* directives can either be a known distribution with its parameters from the list below:

- LogNormal/mean,std_dev**2
- Normal/mean,std_dev**2
- Exp/lambda
- Poisson/lambda
- Binomial/n,p
- Gamma/shape,scale

Or a file which contains values and a probability for each value in ascending probability order.

ALPaCA can be used in both server and location contexts. It can also be used together with fastcgi_pass (for dynamic content) and proxy_pass (for proxying upstream servers), but only if embedded images are static and accessible locally. A sample nginx.conf is below:

```
server {
    listen       80;
    server_name  www.example.com;
    root /var/www;

    # ALPaCA can be configured at the server context
    # Use the probabilistic method
    alpaca_prob on;
    # Path to the distribution file, relative to root
    alpaca_dist_html_size   /dist/dist1.dist;
    # Known distribution
    alpaca_dist_obj_number  Normal/20.0,1.0;
    # Known distribution
    alpaca_dist_obj_size    Normal/1071571.0,1000.0;

    # but also at a location contenxt
    #
    location /foo/ {
        # Use the deterministic method

        alpaca_deter on;
        alpaca_obj_num 5;
        alpaca_obj_size 50000;
        alpaca_max_obj_size 100000;
    }

    # It works for dynamically generated content (but embedded images/css need to
be static)
    location ~ \.php$ {
        include snippets/fastcgi-php.conf;
        fastcgi_pass unix:/var/run/php/php7.2-fpm.sock;
        fastcgi_param SCRIPT_FILENAME $request_filename;
    }

    # It also works with proxy_pass, however embedded images/css still need
to be  accessible in the local files  ystem.
    # IMPORTANT: Accept-Encoding should be set to "" so that the upstream server
returns raw html
    location /proxy/ {
        proxy_pass http://www.upstream.com/;
        proxy_set_header Accept-Encoding "";
    }
}
```

Figure 1: Nginx example configuration

4.1.2 Functionality

As mentioned earlier, each filter module contains a header filter which manipulate the HTTP headers, and a body filter which manipulates the response content. The headers manipulation comes in handy, because we need to change the headers of the requested fake ALPaCA objects. The name of a fake ALPaCA object when requested is **__alpaca_fake_image.png**. However, because of the fact that the image does not actually exist in the filesystem, Nginx sets the response's status to 404, which means that the file was not found. In order to force the browser to download the random content we send for this requests, we change the response's status to 200 in the header filter.

As for the body filter, it does the following: If the response is HTML, we collect the whole response in a buffer and determine which version of ALPaCA to use according to the directives' arguments. Then, we call the ALPaCA library function which corresponds to the determined version, along with some other parameters such the suitable arguments of each version, the Nginx root, the HTML's path in the filesystem and the current size of the HTML. The function returns the morphed HTML, which is put it into a new buffer chain and we pass the chain onto the next filter.

If the response is CSS or image, we check if the ALPaCA GET parameter is present in the request's arguments. The ALPaCA GET parameter is a parameter that is appended by Rust to the images' and CSS file's references in the HTML when it is morphed. It indicates the target size of the object. An image reference in the HTML after it has been morphed should look like this: *<img src="image1.png?alpaca-padding=10000">.* This means that the image's content should be padded to 10000 bytes. If the *alpaca-padding* is present to the request's arguments, we call the library function which is responsible for the objects' padding, passing the type of the object, the request's arguments and the original size of the object as arguments. It returns the extra padding for the object, so we put it in a new buffer and pass it onto the next filter. The same procedure is being followed if the request is about a fake ALPaCA object. Its original size is set to 0, because of the fact that it doesn't exist yet, so the extra padding bytes that are returned from the function are actually as many as the number which the *alpaca-padding* parameter indicates.

A core difference between the HTML's and the objects' requests, is how we treat the original response. In the case of HTML, we don't pass the original response to the next filter, but the modified one. In the case of objects, we pass the whole original response to the next filter, and then we pass the extra padding bytes as well.

If any kind of error occurs during the morphing, we pass the original response onto the next filter.

4.2 Library Implementation

The Rust library is where the actual ALPaCA functionality takes place. It exposes three functions which are called from the module: *morph_html_Palpaca* which morphs an HTML according to the probabilistic version, *morph_html_Dalpaca* which morphs an HTML according to the deterministic version, and *morph_object* which returns the extra padding for an object. There is also a struct called *Object* which represents an HTML, CSS or image object, containing its type, its content its target size and its position in the HTML if it is CSS or image.

At first, no matter which version we use, we parse the objects' references contained in the HTML. This is done by using the *select* Rust library. For each CSS and image reference we find, we look for it in the filesystem in order to get its content's size. Then we create an *Object* object to keep track of it, and append it to an *Object* array. When the parsing is finished, the array contains an *Object* object for every CSS and image found in the HTML in ascending size order.

4.2.1 P-ALPaCA Implementation

In the case of the P-ALPaCA version, we have to parse the distributions passed from the module. If the distribution is a known one, we use the *rand_distr* Rust library to sample values. If a distribution file has been given the sampling is done like this:

We parse the file and create two vectors, one containing the values and the other containing the probability for each value. Then, we produce a random number between 0 and 1, and start summing the probabilities until the sum is greater than the random number. The sampled value is the one that corresponds to the probability that caused the sum to become greater than the random number.

After parsing the distributions, we try to morph the HTML *PAGE_SAMPLE_LIMIT* times, which is currently set to 10. The morphing is done according to the P-ALPaCA algorithm described earlier, using the given distributions to sample the target T. We try to sample each value SAMPLE_LIMIT times, which is set to 30. For example, if we want to sample the number of objects in the HTML after the morphing and the original HTML has n objects, we will keep sampling values until the sampled value is greater than or equal to n. If no such value has been sampled after SAMPLE_LIMIT times, the morphing fails. If the morphing is successful, a vector containing the morphed objects and the fake ALPaCA objects as *Object* objects is available. Each *Object* object contains the position in the HTML of the object it represents, so we add the *alpaca-padding* parameter to each original object's reference in the HTML. Then, we append the fake ALPaCA objects' references to the end of the HTML, creating *img* HTML elements styled as hidden: *<img src="/__alpaca_fake_image.png?alpaca-padding=target_size" style="visibility:hidden">*

Finally, we sample a target size for the HTML, pad it to it and return a pointer to the morphed HTML. To add the padding, we simply add the extra bytes to the end of the HTML in a comment. The target size has to be at least 7 bytes greater than the original size, because of the 7 bytes we need to add the comment indication. The same applies to the CSS padding as well, where the target size has to be at least 4 bytes greater than the original.

4.2.2 D-ALPaCA Implementation

For D-ALPaCA, the same procedure is followed, apart from the part where the target T is made. The number of objects is padded to the next multiple of λ which is greater than or equal to the original number of objects. Each original object is padded to the next multiple of σ which is greater than or equal to its original size. For each fake ALPaCA object, we sample a size between σ and *max_s.* Finally, the HTML's target size is the next multiple of σ.

4.2.3 Object Padding Implementation

In order to add the extra padding for the object requests (CSS, images) we have the *morph_object* function. As arguments, it receives the object's type, its original size and the request's query which contains the *alpaca-padding* parameter. The target size is extracted from the parameter, and it has to be greater than or equal to the original size. The number of padding bytes is calculated by subtracting the original size from the target size, and a vector of random padding bytes is created which is returned by the function.

5. CHALLENGES

In this section, we describe the challenges we faced during the Nginx module and the Rust library implementation. A challenge that involved both the library and the module implementation was the fact that Rust frees automatically any variable that goes out of scope. This means that we had to find a way to tell Rust not to free the morphed HTML and the extra padding for the objects after they are returned to the module. The solution was the following: We move the contents we want to return to the heap in a buffer, and then tell Rust to “forget” it, therefore not calling a destructor for the buffer when it goes out of scope.

```
pub fn as_prt(self) -> *const u8 {
  let mut buf = self.content.into_boxed_slice();
  let data = buf.as_mut_ptr();
  std::mem::forget(buf);

  data
}
```

Figure 2: Return contents from Rust without freeing them afterward

After we are done processing the returned contents in the module, we have to free this memory. This cannot be done from the module because the memory was not allocated in it; therefore we created a custom *free_memory* function in the library. When the contents are returned to the module, they get hard-copied to a buffer. Then, *free_memory* is called passing the contents that were returned from the library and their size as arguments, and it frees the memory allocated for them.

```
pub extern “C” fn free_memory(data: *mut u8, size: &usize) {
  let s = unsafe { std::slice::from_raw_parts_mut(data, *size) };
  let s = s.as_mut_ptr();

  unsafe {
    Box::from_raw(s);
  }
}
```

Figure 3: Free memory allocated by Rust

5.1 Module Implementation Challenges

Learning how to develop an Nginx module was a challenge itself, because of the fact that modules are extremely customizable. A big burden is placed on the programmer o define exactly how and when the module should run. Apart from that, although the module’s functionality was pretty straight forward (most of the work is done in the library), we faced the following challenges:

As mentioned earlier, sometimes there are more than one buffer chains containing the response. This means that a filter is called more than one time for a single response. In the case where we have an HTML response, we had to find a way to collect the whole response to a buffer in order to pass it to the library function. The buffer of course cannot be declared inside the *body_filter* function, as its variables are reinitialized for every call. Fortunately, Nginx allows a module to keep state information per request through a request context data structure defined by the module. We created the *ngx_http_alpaca_ctx_t* struct which stores the state of processing a response for the module. The struct, along with other variables, contains a buffer in which the whole response is stored, and its size. At the start of each request, we initialize the request's context, and for each chain link that is received, we iterate through its buffers and copy the contents into the context's buffer. When the last chain link is passed in the module, the buffer contains the whole response so this is when we call the library functions. At first the memory we allocated for the whole response's buffer had size *content_length_n* which can be found in the response's headers and contains the size of the whole response. This however created a new challenge.

If Nginx is used as a proxy server, *content_length_n* doesn't contain the actual size of the HTML, as it is unknown. Considering this case, we had to modify the code in order to support it, so if the *content_length_n*'s value is -1, we allocate 1000 bytes of memory for the response's buffer. Now, when we copy a buffer's contents from the chain link into our buffer, if the copy causes the size to be greater than the buffer's capacity, we double the size. Of course we keep track of both its capacity and its actual size. Making this modification, solved our problem, and also enabled the module to work with dynamically generated HTML response as well.

5.2 Library Implementation Challenges

In the implementation of the ALPaCA library, we faced the following challenges:

During the ALPaCA morphing we had to append the *alpaca-padding* parameter to the original objects' references in the HTML. For this purpose we needed a high level HTML parser so the *select* Rust library did the work. It generates a document from a given HTML string, which contains nodes corresponding to the HTML elements, which can be accessed and modified. The *position* field in the *Object* struct which indicates the position of the object reference in the HTML, is actually the index of its corresponding node, so after the morphing, in order to append the parameter to each reference, we had to modify the corresponding's node's HTML.

Another challenge was the fact that we had to find the filesystem path of each object in the HTML in order to access it. The path of each object in the HTML is relative to the root of the Nginx server. For example, if the root is */var/www*, and an image's url path is

/images/image1.png, then the image's filesystem path is */var/www/images/image1.png*. To complicate things, if the objects' url path doesn't begin with a slash, we also have to consider the HTML's base url path in the equation. Finally, we have to resolve the dots in the path, and be careful not to end up with a filesystem path that accesses files outside the Nginx root. For these reasons, we had to pass both the Nginx root and the html's url path to the library functions. The algorithm to resolve an object's filesystem path can be found in the next page.

```
fn filesystem_path(root: &str, relative: &str, html_path: &str) ->
Option<String> {
   if relative.starts_with("https://") || relative.starts_with("http://") {
     return None;
   }
   let mut fs_relative = String::from(relative);
   if !fs_relative.starts_with('/') {
   let base = Path::new(html_path).parent().unwrap().to_str().unwrap();
   if !base.ends_with('/') {
     fs_relative.insert(0,'/');
   }
   fs_relative.insert_str(0,base);
   }
   // Resolve the dots in the path so far
   let components: Vec<&str> = fs_relative.split("/").collect();
   // Stack to be used for the normalization
   let mut normalized: Vec<String> = Vec::with_capacity(components.len());
   for comp in components {
     if comp == "." || comp == "" {continue;}
     else if comp == ".." {
       if !normalized.is_empty() {
         normalized.pop();
       }
    }
    else {
      normalized.push("/".to_string()+comp);
    }
   }
   let mut absolute: String = normalized.into_iter().collect();
   absolute.insert_str(0,root);
   Some(absolute)
}
```

Figure 4: Resolve filesystem path for an object

6. CONCLUSIONS

Website fingerprinting attacks pose a big threat in clients' anonymity, and there is definitely a need for protection against them. In this thesis we implemented ALPaCA defense as a Rust library and developed an Nginx module which uses it. As was proven, ALPaCA is a realistic and easy to deploy defense with very satisfying results. Users who operate *.onion* sites have the incentives to use such a defense, as most of the time those sites provide sensitive information and clients may not want to be associated directly with it. Apart from that, *.onion* sites can be distinguished from regular sites with more than 90% accuracy. Finally, the fact that ALPaCA is a server-side defense makes it easier to use, because it only needs to be enabled by the site's operator without the client's interference.

REFERENCES

[1] *Cherubin, Giovanni*, Hayes, Jamie, and Juarez Marc. Website Fingerprinting Defenses at the Application Layer. In *Proceedings on Privacy Enhancing Technologies* 2017.

[2] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Peeka-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. *In Proceedings of the 2012 IEEE Symposium on Security and Privacy*.

[3] JUAREZ, M., AFROZ, S., ACAR, G., DIAZ, C., AND GREENSTADT,R. A Critical Evaluation of Website Fingerprinting Attacks. *In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 263–274.

[4] J. Hayes and G. Danezis. k-fingerprinting: a Robust Scalable Website Fingerprinting Technique. In *USENIX Security Symposium*. USENIX Association, 2016.

[5] A. Kwon, M. AlSabah, D. Lazar, M. Dacier, and S. Devadas. Circuit fingerprinting attacks: passive deanonymization of tor hidden services. In *USENIX Security Symposium*, pages 287–302. USENIX Association, 2015.

[6] X. Cai, R. Nithyanand, T. Wang, R. Johnson, and I. Goldberg. A Systematic Approach to Developing and Evaluating Website Fingerprinting Defenses. In *ACM Conference on Computer and Communications Security (CCS)*, pages 227–238. ACM, 2014.

[7] Cherubin Giovanni. 2017. Bayes, not Naïve: Security Bounds on Website Fingerprinting Defenses. *Proceedings on Privacy Enhancing Technologies* 2017 (2017).

[8] Rebekah Overdorf, Marc Juarez, Gunes Acar, Rachel Greenstadt, Rachel Greenstadt. How Unique is Your .onion?An Analysis of the Fingerprintability of Tor Onion Services. 2017

[9] Shuai Li, Huajun Guo, Nicholas Hopper. Measuring Information Leakage inWebsite Fingerprinting Attacks and Defenses. CCS'18, October 15-19, 2018, Toronto, ON, Canada