



SAPIENZA  
UNIVERSITÀ DI ROMA

# Design and Implementation of a Sponsored CAPTCHA Platform: Integration of CAPTCHaStar for Display Advertising and Resiliency Assessment of Adversarial Attacks

**Faculty of Information Engineering, Computer Science and Statistics**  
**Master Course in Computer Science**

Bryan Ferracuti  
1914350

Advisor  
Prof. Gabriele Tolomei

Co-Advisor  
Prof. Mauro Conti

A/Y 2021/2022



## Abstract

In recent years, several websites have used CAPTCHAs to increase their security by denying access to automated bots. This project presents a system that allows website owners to adopt CAPTCHAs that produce passive revenue through integrated advertising. Online advertising allows content providers, or publishers, to monetise their websites by reserving ad spaces. Advertisers compete for and compensate publishers for these spots. The proposed platform blends the two paradigms by using CAPTCHAs as ad spaces to allow website owners to act as publishers. Specifically, a recent innovation known as CAPTCHaStar is used since it satisfies the image-centric requirements of this project, and thus, makes it simple to display ads.

This idea is brought to life via two parallel thesis works. The objectives of this thesis were to design the system's general architecture, adapt CAPTCHaStar to perform smoothly in the advertising domain, build the system's challenge handling mechanisms, and evaluate the security of this new CAPTCHA framework. While the objective of the complementary work was to design the overall architecture of the system (in conjunction with this thesis) and handle all processes associated with online advertising.

CAPTCHaStar presents several white pixels known to as stars within a black square, the location of which changes according on where the mouse cursor is. The user must move the cursor until a recognizable shape is formed by the stars. However, in the original work that presented this innovation, the generation process was given pictures that already contained just white and black pixels. An important challenge to address in this study is how to seamlessly derive a CAPTCHaStar challenge from brand-representing images.

Every CAPTCHA system has an inherent trade-off between security and usability. Complex and secure challenges would be more difficult for a computer to solve, just as they would be for a human. This may make users feel uneasy and discourage them from using the website. Thus, the security and robustness of the proposed CAPTCHA system must be addressed in this study.

The thesis is divided into six main parts, the first three of which were elaborated in conjunction with the author of the complementary thesis: an introduction to the use of **CAPTCHAs in Online Advertising**; a clear definition of the goals and **scope of the project**; a detailed description of the **application's design architecture**; an in-depth analysis of the practical **implementation**; an examination of the **CAPTCHA generation** mechanism developed; an **assessment of the security** and resiliency to automated attacks of the challenges.



# Contents

|  |           |
|--|-----------|
| <b>Contents</b>                                | <b>v</b>  |
| <b>1 CAPTCHaStar! in Online Advertising</b>    | <b>1</b>  |
| 1.1 CAPTCHaStar! . . . . .                     | 1         |
| 1.2 Online Advertising . . . . .               | 4         |
| <b>2 Project Scope</b>                         | <b>7</b>  |
| 2.1 Problem Definition . . . . .               | 7         |
| 2.2 Users and Operations . . . . .             | 8         |
| <b>3 Architecture</b>                          | <b>9</b>  |
| 3.1 Design Choices . . . . .                   | 9         |
| 3.2 Main Components . . . . .                  | 12        |
| 3.3 Additional Considerations . . . . .        | 15        |
| <b>4 Implementation</b>                        | <b>17</b> |
| 4.1 Technical Choices . . . . .                | 17        |
| 4.2 Services . . . . .                         | 19        |
| 4.3 APIs . . . . .                             | 24        |
| 4.4 Front-End . . . . .                        | 26        |
| 4.5 Performance . . . . .                      | 27        |
| <b>5 CAPTCHA Generation</b>                    | <b>31</b> |
| 5.1 CAPTCHaStar Algorithm . . . . .            | 31        |
| 5.2 Custom Logo Binarization . . . . .         | 33        |
| 5.3 Results & Considerations . . . . .         | 37        |
| <b>6 Security Assessment</b>                   | <b>43</b> |
| 6.1 Sponsored CAPTCHaStar Resiliency . . . . . | 43        |
| 6.2 Ad-hoc Adversarial Attack . . . . .        | 44        |
| <b>7 Conclusions</b>                           | <b>49</b> |

|          |                            |           |
|----------|----------------------------|-----------|
| 7.1      | Obtained Results . . . . . | 49        |
| 7.2      | Future Work . . . . .      | 50        |
| <b>A</b> | <b>Code</b>                | <b>53</b> |
|          | <b>Bibliography</b>        | <b>59</b> |

# 1. CAPTCHaStar! in Online Advertising

In recent years, a number of websites have employed *CAPTCHA* technologies to guarantee their security by ensuring that the users accessing their services are not automated bots. The field of CAPTCHAs is currently dominated by Google's services, which provides them in an easy-to-access and free manner. The goal of this thesis is to develop an alternative platform that allows website owners to use a CAPTCHA that generates passive revenue from an embedded *Ad*, in addition to the protection provided.

## 1.1 CAPTCHaStar!

In general, CAPTCHA (Completely Automated Public Turing Test to Tell Computers and Humans Apart) is a challenge-response security mechanism. The CAPTCHA test is intended to verify that the user is a human and not a computer trying to access a password-protected account, thereby protecting other users from spam and other concerns.

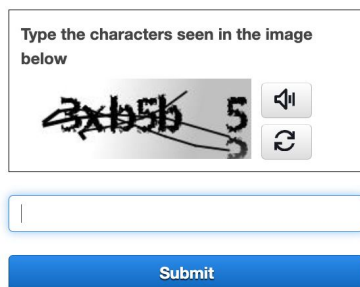


Figure 1.1: A Text-based CAPTCHA

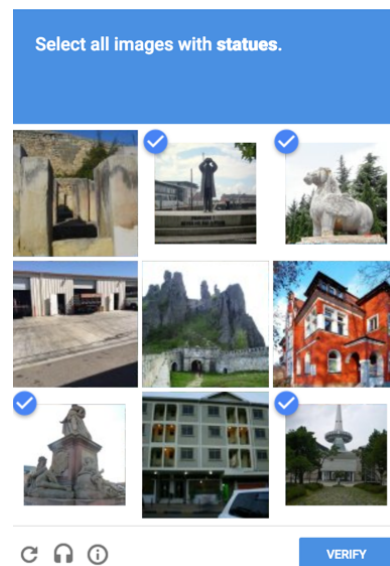


Figure 1.2: An Image-based CAPTCHA

A recent CAPTCHA innovation presented by Conti et al.[1] called *CAPTCHaStar!* is being adopted in this project. Because it focuses the user’s attention on images and shapes, this is an excellent method for showing ads that make use of images.

CAPTCHaStar prompts the user with some white pixels known as *stars* within a black square, the location of which varies depending on where the pointer is. The user must move the pointer until the stars form a discernible shape.

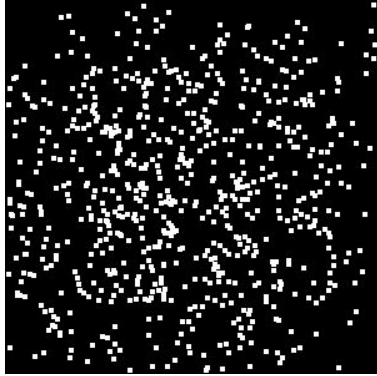


Figure 1.3: An incorrect configuration

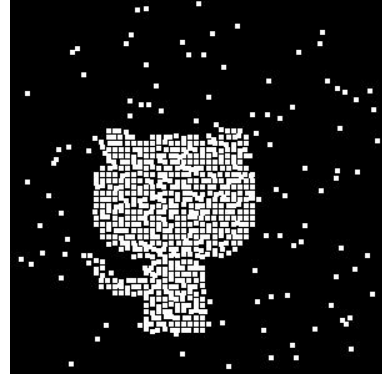


Figure 1.4: A correct configuration

An essential aspect of any CAPTCHA system that needs emphasis is the following: both the requirements of usability for humans and resistance to automated attacks must be satisfied. A more complex and secure CAPTCHA would be more challenging for a machine to solve, just as it would be for a person. This may make visitors uncomfortable and discourage them from utilizing the website. Therefore, it is essential to evaluate the trade-off between security and usability. Regarding CaptchaStar, this topic was fully covered in the original publication [1]; a brief summary is provided below.

## Usability

CAPTCHaStar’s usability was evaluated using many of the metrics introduced by Yan and colleagues[2] such as the success rate, completion time, and ease of comprehension. The authors conducted this study on 250 people of varying ages, genders, educational backgrounds, and more. Six CAPTCHaStar and two text-based challenges were distributed to the participants, for a total of eight CAPTCHAs.

According to the results presented in Figure 1.5, text-based challenges (T7 and T8) achieve a success rate of 62.7% and 46.9% in approximately 20 seconds, whereas CAPTCHaStar challenges attain a higher success rate but with a larger average completion time. In general, participants comments are positive, highlighting how CAPTCHaStar is a “sort of a game” and more engaging than typical



text-based CAPTCHAs. When the optimal combination of parameter settings is used, CAPTCHaStar users can solve the challenge in less than 30 seconds with a success rate of more than 90%.

|                |         | CAPTCHaStar |      |      |      |      |      | Text |      |
|----------------|---------|-------------|------|------|------|------|------|------|------|
| Test           |         | T1          | T2   | T3   | T4   | T5   | T6   | T7   | T8   |
| Succ. Rate (%) |         | 78.7        | 90.2 | 90.6 | 50.4 | 85.1 | 76.6 | 62.7 | 46.9 |
| Difficulty     |         | 1.9         | 2.4  | 2.6  | 3.4  | 2.9  | 3.1  | 2.4  | 2.7  |
| Succ.          | Avg (s) | 14.4        | 17.5 | 22.2 | 54.1 | 30.2 | 28.5 | 11.0 | 14.9 |
|                | Std     | 9.8         | 9.3  | 15.8 | 33.5 | 20.2 | 19.7 | 5.4  | 6.1  |
| Fail           | Avg (s) | 14.7        | 18.2 | 33.1 | 49.0 | 38.8 | 40.0 | 12.6 | 21.2 |
|                | Std     | 13.5        | 10.7 | 21.2 | 33.5 | 26.6 | 25.6 | 8.8  | 17.8 |

Figure 1.5: Survey results for CAPTCHaStar and text-based CAPTCHAs

## Security

In their assessment of the resiliency of the CAPTCHaStar challenges, the authors analyzed both traditional and ad-hoc attacks. While the results of traditional attacks are encouraging, the findings of ad-hoc heuristics and machine learning attacks are more difficult to discern.

- CAPTCHaStar’s results for **traditional attacks** (e.g., random choice, pure relay, and so on) are comparable to those of standard CAPTCHAs, showing that it is adequately resilient to these types of attacks;
- The primary idea behind the **heuristic** is to collect all of the states for each challenge (corresponding to each location of the pointer) and assign each a score that quantifies the dispersion of the stars. The candidate solution is the state that minimizes this value. Depending on the exact definition of dispersion, different results are achieved. In conclusion, the efficacy of this approach can be successfully reduced by increasing the number of noisy stars (stars that move at random and are not part of the shape to reconstruct) in the challenge while maintaining acceptable usability values;
- To test how resilient the system is to attacks based on **machine learning**, the authors trained two classifiers (a Random Forest and a Support Vector Machine) with the goal of predicting whether a state was a solution or not. To do this, they constructed a dataset consisting feature vectors representing various states of different challenges, with each vector bearing its own label.

Although the data indicates that these models are capable of addressing the problems (78% success rate), the average time needed is somewhat lengthy (420 seconds, compared to the average 30 of a human user). This indicates that the task of a bot automatically finding the solution state of a CAPTCHaStar challenge using this strategy is tough to accomplish in a short period of time and with limited resources.

## 1.2 Online Advertising

Online advertising is a form of company promotion in which advertising messages are sent through the Internet to attract customers. Due to the expansion of Internet users and Internet technology, a growing number of companies have started to promote their products and services online.

### Entities

Online advertising's main concern is to allow online content providers (*publishers*) to generate revenue by reserving particular *advertising slots* on their websites. Companies wishing to promote themselves (so-called, *advertisers*) compete for these slots and are ready to pay publishers for them. This system entails a number of actors, many of which are separate businesses, that manage different aspects of advertising on behalf of advertisers and/or publishers:

- A Sell-Side Platform, often referred to as **SSP** for short, is a platform that enables web publishers to manage their ad slots, have those populated with advertising, and get revenue as a result;
- **Ad Exchanges** are online markets that allow advertisers, publishers, and others to buy and sell ad inventory by auctioning off slots to anybody who is willing to participate;
- Demand-Side Platform, commonly abbreviated as **DSP**, is a system that assists advertisers in locating and purchasing slots from the marketplace.



Figure 1.6: Entities of Online Advertising

## Real Time Bidding

Real-time bidding (RTB) is the method through which businesses purchase and sell advertisements online via automated auctions. Real-time bidding simplifies advertising by enabling advertisers to place hundreds of thousands of ads online in a matter of seconds rather than contacting individual publishers. The entire pipeline of most RTB systems is shown in section 1.2.

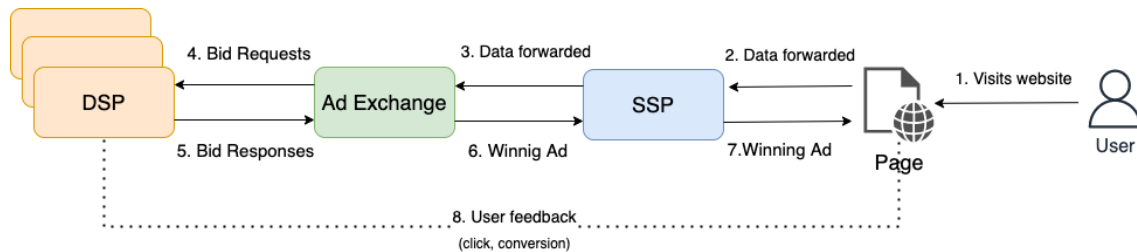


Figure 1.7: RTB Pipeline Overview

1. When a user visits a web page, an ad slot on the publisher's website is created;
2. The information regarding the slot, along with the user's data, is sent to the publisher's SSP;
3. The SSP forwards this information to an Ad Exchange;
4. The slot auction is forwarded by the Ad Exchange to all available DSPs;
5. If the DSP decides to participate, a bid is generated and submitted;
6. The winner is chosen by the Ad Exchange based on the DSP bids, and the winning ad is forwarded to the SSP;
7. The winning advertising is sent by the SSP to the publisher's website, where it is displayed;
8. The user feedback is gathered, determining whether the user clicked the Ad and whether the Ad resulted in a *conversion* (e.g., the user bought something from a website store).



## 2. Project Scope

### 2.1 Problem Definition

The purpose of the system is to create a platform that integrates CAPTCHA services with an online advertising mechanism. When a publisher wants to challenge a user with a CAPTCHA, an ad slot gets allocated. The slot is thereafter auctioned off on the platform itself. The winning ad is a challenge based on the image provided by the winning advertiser, as opposed to a conventional advertisement (e.g., banner). In particular, since CAPTHaStar is able to show potentially any kind of image, we want to use it to show images that can be considered ads and thus create revenue for a publisher that decides to use this kind of CAPTCHA.

Aside from the creation of the actual platform, the two main goals are to recreate an embedded RTB system (to provide a mechanism for selecting the ad to show as a challenge) and adapt the CAPTHaStar tool for such a purpose. This thesis is focused on the CAPTHaStar development, while the RTB solution is presented in the complementary thesis [7]. It is important to note that the general design and primary architectural decisions were made in cooperation between the two theses since they would have a significant influence on the work of each project. For this reasons, this chapter and chapter 3 are shared between the two.



Figure 2.1: Base logo

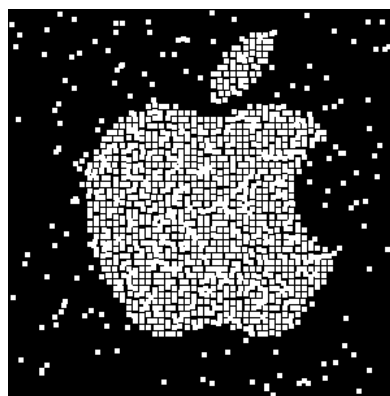


Figure 2.2: Sponsored CaptchaStar

## 2.2 Users and Operations

This chapter will outline the many identified actors and the operations they must be capable of performing.

### Publisher

The publisher is the website owner who wants to display sponsored CAPTCHAs. On the platform, he can perform the following operations:

- create his own publisher profile, which grants access to the publisher's protected area;
- register a websites from the publisher's protected area, using the corresponding domain;
- create a sponsored CAPTCHA slot by invoking the appropriate APIs whenever a visitor attempts to access an area that requires protection.

### Advertiser

The advertiser is a business that desires to promote itself and is willing to pay for the publisher's advertising slots. On the platform, he can perform the following operations:

- create his own advertiser profile, which grants access to the advertiser's protected area;
- create a CaptchaStar challenge providing an image representing his ad;
- create an ad campaign providing the budget and the CAPTCHA to be shown when winning a slot in the RTB process.

### Visitor

The visitor to the publisher's website is the actor who initiates the RTB auction and receives a CAPTCHA challenge. Finally, he submits his own answer to the platform for evaluation. If it is acceptable, the system displays the original ad image that was used to produce the challenge and enables the visitor access to the restricted area.

## 3. Architecture

This chapter will cover the system’s design and the decisions that led to it. The technical and implementation specifics, related to this thesis, will be discussed in chapter 4.

### 3.1 Design Choices

The system is designed to be organized in *Microservices*, an architectural paradigm that aims to structure an application as a collection of independent services. As a consequence, our services are individually deployable and loosely connected, making them simple to manage, scale, and test. This allows us to implement each service separately, using any technology we see appropriate.

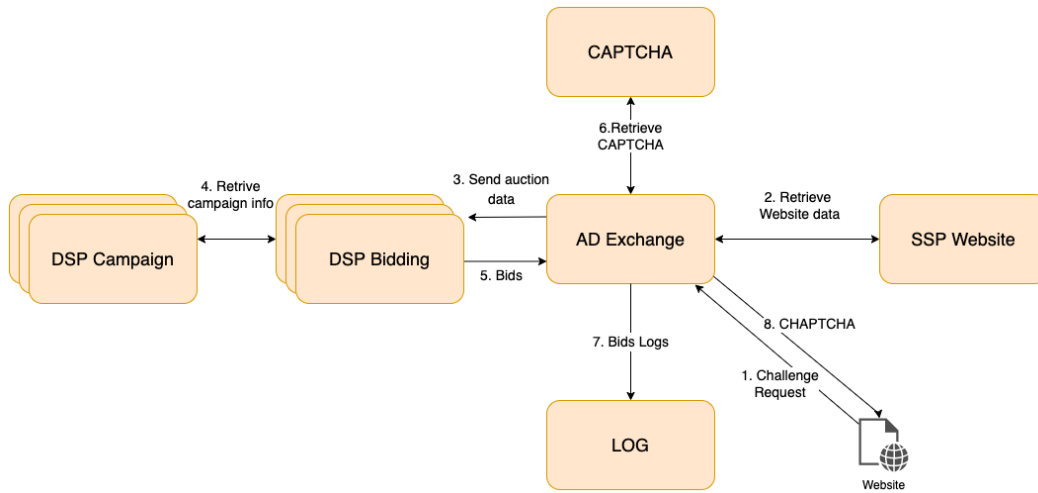


Figure 3.1: High-level Architecture of the system

Figure 3.1 depicts the major application components and their interactions throughout the main application flow: the initial modelling starts from the standard RTB model consisting of DSP, Ad Exchange, and SSP. Yet, the DSP functions are split into two independent services with distinct roles: DSP Campaign and DSP Bidding, whose respective tasks are to manage the data of the ad campaigns and to bid on behalf of the advertisers.

The fact that the Ad Exchange, instead of the SSP, handles the initial request for a challenge is a significant deviation from the original model. Since all RTB

processes can be managed in the same system, there is no reason to include the SSP service in the auction itself; it is solely responsible for managing the data on the publishers' websites. Finally, we add the CAPTCHA and Log modules. The CAPTCHA module creates, stores, and manages CAPTCHA challenges while the Log module keeps record of all auction activities.

## Functionalities

Taking a closer look, the capabilities that are currently being implemented are as follows:

- **RTB Flow:** an ad slot is produced when a user visits a publisher's website. Its data is sent to the *AD Exchange* service through the associated API. The AD Exchange runs a *DSP Bidding* instance for each eligible ad campaign after collecting the website's data from the *SSP Website* service. Each instance of DSP Bidding retrieves campaign data from the *DSP Campaign* service. Once all the required data has been gathered by the respective DSP bidding instance, the adequate bid is submitted to the AD Exchange service. This service identifies the winning campaign and fetches the corresponding CAPTCHA challenge. Finally, it sends the information to the web page for display the CAPTCHA challenge. At this stage, there are two optional operations that may be performed:
  - **Check Solution:** upon selecting his proposed solution, the visitor submits it to the CAPTCHA service through the proper API. As soon as the CAPTCHA service determines whether it is acceptable, it returns a response. If the answer is positive, the system also provides the original ad image from which the challenge is based on so that it may be shown to the user;
  - **Log Click:** if the challenge is completed successfully, the original image is displayed. If the user clicks on the ad image, the *Log* service is triggered to record the click event;
- **Log Auction:** once the AD Exchange service has established the winning campaign, it invokes the Log service to log the information regarding the auction;
- **Add CAPTCHA:** the advertiser, through CAPTCHA service, can upload an image which is converted into a CAPTCHaStar challenge;
- **Add Campaign:** the advertiser, through the DSP Campaign service, can create an ad campaign by providing some required information such as budget and the sponsored CAPTCHA to use;



- **Add Website:** the publisher, through the SSP-Website service, can register a website by providing some required information.

## Auction

When determining the kind of Auction and Bidding Strategy to employ, we sought to minimize the coupling between the **DSP Bidding** and **AD Exchange** services. We choose to charge the advertiser for every auction won. This choice was taken to build an interface between DSP and AD Exchange: the auction accepts a bid (the amount a DSP is willing to spend for a predetermined slot) as input, regardless of the DSP's strategy or objective (maximize clicks, impressions and so on). Each eventual kind of DSP is responsible for developing its own logic to offer the Ad Exchange with the amount it is ready to pay for a given slot. Thus, in the future, other types of campaigns (and related DSPs) will be able to be developed without disrupting the auction's logic.

## 3.2 Main Components

This section will go over each component of the system in terms of its role and responsibilities.

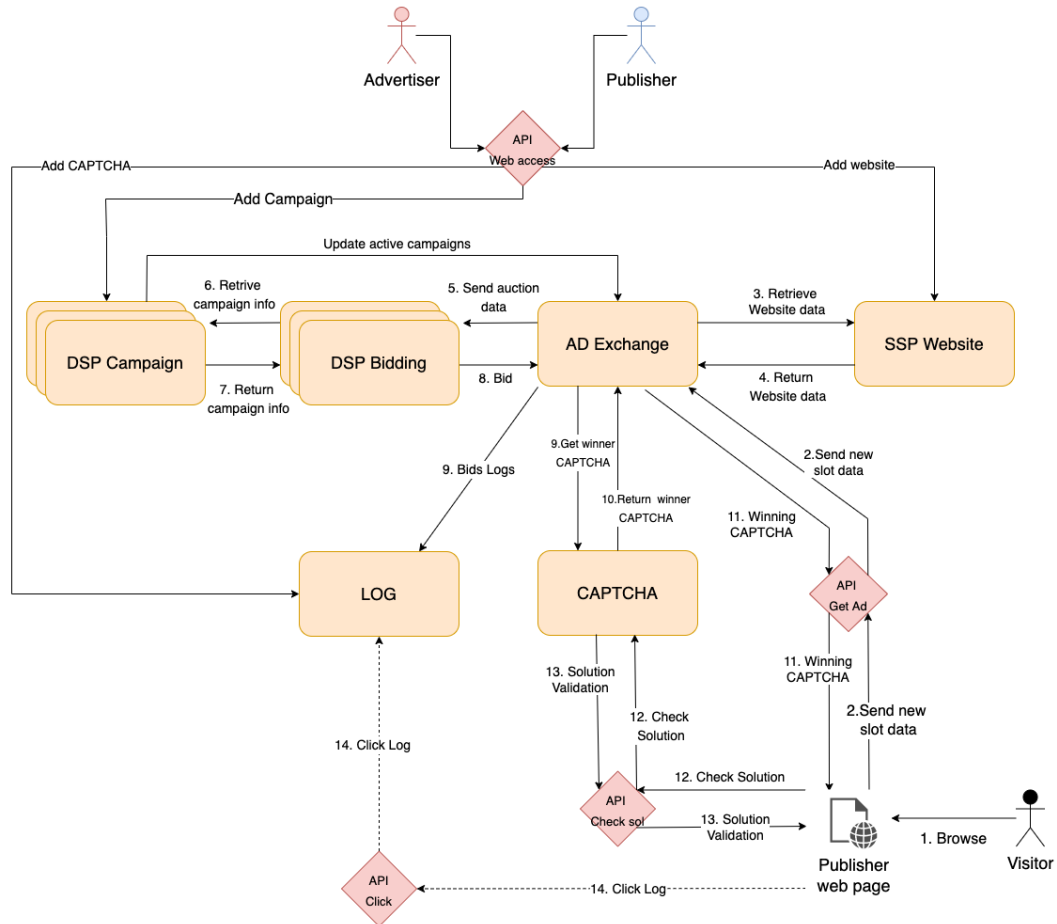


Figure 3.2: Architecture Overview

### SSP Website

This module's responsibility is to handle the data of a website. The current functionalities are:

- **Add Website:** it allows the publisher to register a new website by providing several data. This data is stored in a dedicated storage;
- **Get Site Data:** provides all of the information regarding a website.

## Ad Exchange

This module's responsibility is to start and handle the RTB auction. This module offers two functionalities:

- **RTB Auction:** it receives a request for a challenge from a web page, retrieves the data of the website from the SSP Website service, retrieves all the available campaigns from its storage, and invokes a DSP Bidding instance for each one. Once it has received the bids and corresponding CAPTCHA challenge identifier from each DSP, it establishes the winner (the one with the highest bid), retrieves the corresponding CAPTCHA from the CAPTCHA service, and sends it back to the web page. After deciding the winner, it also calls the Log service to store information about the auction, such as the participants, their bids, and the winner;
- **Update Active Campaigns:** it overwrites the active campaigns in the storage with the ones received (see section 3.3).

## DSP Bidding

This module's responsibility is to handle the bidding process for each auction from the point of view of a single ad campaign. It offers the following functionality:

- **Get Bid:** after retrieving information about the specific campaign, it estimates the optimal bid value and sends it back to Ad Exchange.

## DSP Campaign

This module's responsibility is to handle the data of an ad campaign. Its functionalities are:

- **Add Campaign:** given the required data about the campaign, such as budget, captcha, and so on, it creates an ad campaign;
- **Get Campaign Data:** it provides all of the information regarding a specific campaign.

## CAPTCHA

This module's responsibility is to create, handle, and store all of the CAPTCHaStar challenges. It offers three main functionalities:

- **Add Captcha:** given an image, it creates a new CAPTCHaStar challenge and saves the challenge along with the relative solution in the dedicated storage. See chapter 5 for details about the generation process;
- **Get Captcha:** it retrieves a specific challenge from the storage;
- **Get Default Captcha:** it returns one of the default non-sponsored CAPTCHaStar challenges (see section 3.3 for details about the default CAPTCHA);
- **Check solution:** given a proposed solution and a CAPTCHA challenge identifier, it checks whether the solution is acceptable; if so, it also sends back the ad image from which the challenge was generated.

## Log

This module's responsibility is to handle the Logs from the various auctions. It offers two main functionalities:

- **Log Auction:** given all the data of an Auction, this functionality log this information into its dedicated storage.
- **Update On Click:** given an auction identifier, it updates the winner's record belonging to the selected auction to "clicked".

## APIs

Four APIs were designed to serve different purposes:

- **Web Access:** it allows the advertiser and publisher users to handle their private area through a website;
- **Get Ad:** it allows a publisher's website to require a CAPTCHaStar challenge through an RTB auction mechanism;
- **Check Solution:** it allows a publisher's website to check whether the solution to a specific CAPTCHaStar challenge provided by a user is correct;
- **Click:** it allows a publisher's website to log the click of a user on a specific CAPTCHaStar challenge after resolution.

### 3.3 Additional Considerations

During the process of modeling the problem, numerous simplifications were employed in order to aid the work for the thesis. First, we decided to associate a single CAPTCHA with each campaign. Nevertheless, it is easy to adapt the storage of each campaign such that it is associated with a list of CAPTCHAs rather than a single one. Security should also be considered, since each CAPTCHA may now be shown an endless number of times with the same solution coordinates. In real life, this problem might be addressed in a variety of ways: re-create a CAPTCHA once it has been served, generate a set of challenges for each campaign at regular intervals, and so forth. When it comes to the data used, we don't have any information about the visitor or the website, such as the category. However, the logs are intentionally designed such that they may be utilized effortlessly for training a model that matches our goals and data. (see [7]).

In addition, several decisions were made for more technical reasons. As a sort of cache, store the active campaigns redundantly in a specialized storage of the Ad Exchange module (rather than retaining them just in the DSP Campaign storage). This choice was made in an effort to reduce the number of interactions between components and accelerate the whole auction process. Lastly, a fallback mechanism was created to offer a default non-sponsored CAPTCHA in various situations: when it is not possible to deliver a sponsored challenge, including when no campaigns are willing to participate in the auction, or when errors occur during the auction process.



## 4. Implementation

This chapter will address the technical decisions and implementation specifics related to the development of the architecture discussed in the previous chapter.

### 4.1 Technical Choices

The objective of the implementation process was not to build a simple prototype but rather a platform that may serve as a solid foundation for future enhancements and advancements. As a consequence, the system was created to operate in an environment that was as realistic as reasonable, taking into consideration the limitations of a thesis project.

In recent years, companies of all kinds have considered a transition to cloud computing. There are several explanations for this phenomenon. Scalability, flexibility, lower entry costs, ease of access, subscription and pay-per-use, and so on, drive enterprises to transition from conventional to **cloud-based platforms** (for more insight, see [3]). It was therefore appropriate to make use of the advantages of the cloud computing paradigm. Specifically, we chose to focus on *Amazon Web Services* (AWS), one of the most prominent cloud service providers. Due to the limited budget of an academic project, we had to stick to the AWS platform's free-tier plan. This approach enabled us to construct the full system but with some constraints, such as restricted traffic volume, storage capacity, accessible services, etc.

According to the cloud design principles, a modern application should be decomposed into loosely connected services that are as serverless as possible (see section 3.1). A serverless architecture helps to design and execute apps without worrying about infrastructure. This implies that in our scenario, AWS handles all server administration, provisioning, and scalability. This permits us to concentrate on the system's functionality and structure.

## AWS Services

This section will outline the key services used when designing the application among those provided by AWS.

- **Cloudfront:** a CDN service mainly used to efficiently serve static content;
- **S3:** an object storage service often used to store the static content of a website;
- **API Gateway:** a service that allows developers to create, publish, and generally handle endpoint APIs to enable communication between the front and back end of any application;
- **Cognito:** a user identity and data synchronization service;
- **Lambda:** a service which provides developers with the ability to deploy event-driven functions without handling the provisioning;
- **Dynamo DB:** a fully managed, key-value NoSQL database designed to work with applications with high-performance needs;
- **RDS:** a collection of managed services that make it simple to set up, operate, and scale relational databases in the cloud;
- **Cloudwatch:** a monitoring service that provides data to check your applications, respond to performance changes and optimize resource utilization. In addition, it also provides the ability to schedule events and triggers.



## 4.2 Services

This section will describe in detail all the services discussed in section 3.2. The functionalities closely related to the complementary thesis [7] will not be addressed here.

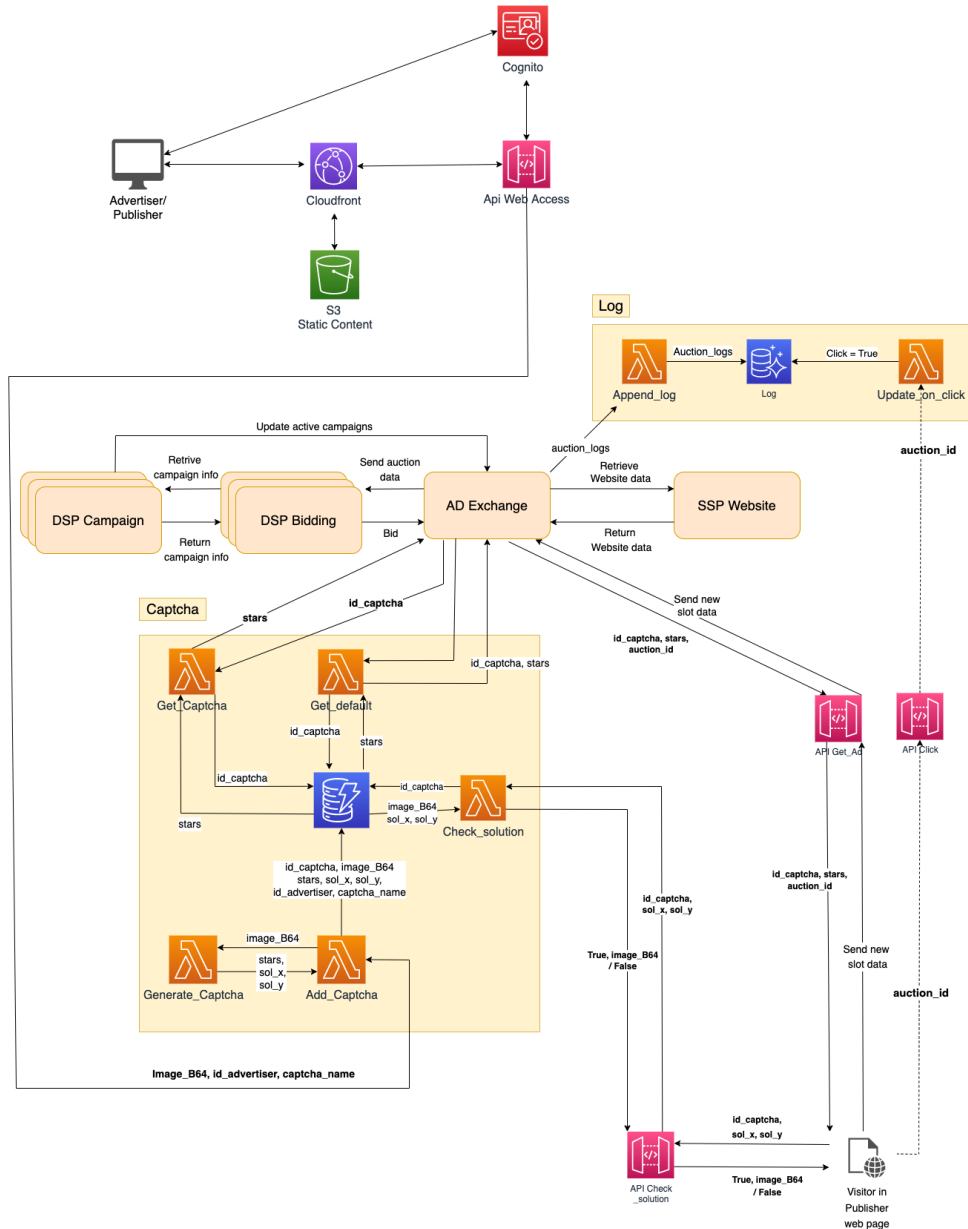


Figure 4.1: System Overview

## CAPTCHA

This module is responsible for managing the CAPTCHA challenge data for advertisers. All of this module's functionality is implemented using Python-written Lambda Functions that use a DynamoDB database to store and retrieve all data.

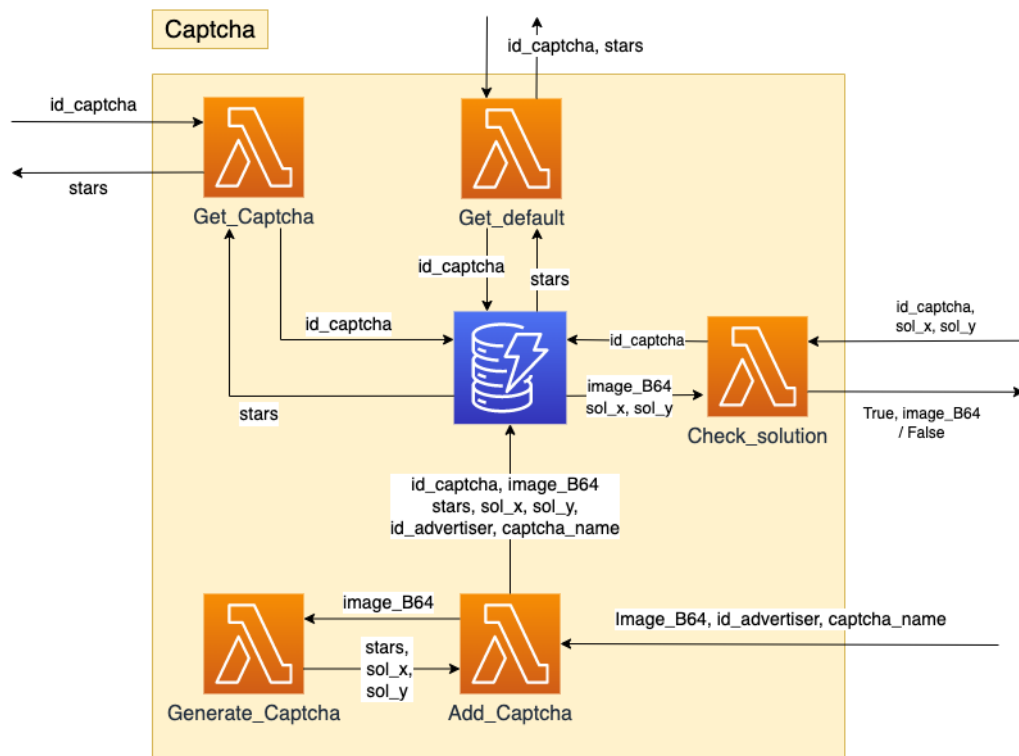


Figure 4.2: Captcha Service

### Add CAPTCHA

**Inputs:** id advertiser (string), image base64 (string), captcha name (string)

**Description:** it invokes the Generate Captcha function with image base64 as input and retrieves: stars, sol x and sol y. Then, it stores everything in the Database.

**Outputs:** None

The input image is encoded in base64 to be easily and securely handled during the various communications. *Stars* is the name used to refer to the string which encodes the CaptchaStar challenge. Finally, *sol x* and *sol y* indicate the cursor position for the solution of the challenge.

### Generate CAPTCHA

**Inputs:** image base64 (string)

**Description:** it performs the binarization algorithm explained in section 5.2 on the original image. Then, it generates the final Captcha challenge and solution using the CaptchaStar algorithm (see chapter 5).

**Outputs:** stars (string), sol x (int), sol y (int)

### Get CAPTCHA

**Inputs:** id captcha (string)

**Description:** it retrieves all the data associated with the CAPTCHA.

**Outputs:** stars (string)

Since the solution is checked from a separate API call, there is no need to send it back from this function.

### Get Default

**Inputs:** None

**Description:** it randomly chooses one of the default, non-sponsored CAPTCHA challenges to send back.

**Outputs:** id captcha (string), stars (string)

### Check Solution

**Inputs:** id captcha (string), sol x (int), sol y (int)

**Description:** it retrieves all the data associated to the challenge from the storage and checks whether the proposed solution is acceptable.

**Outputs:** correct (boolean), image base 64 (string)

An acceptable solution denotes that the suggested coordinates are within a certain range of the actual solution. If the proposed coordinates are valid, the original picture is also supplied as an output to be shown on the front-end.

### Captcha DynamoDB Table

This module's data is stored in a DynamoDB database named **captcha-db-cs**. This decision was taken for two major reasons: we wanted to boost the system's performance (a crucial factor in RTB) and we wanted a database that could easily accommodate data structure modifications. Even though the structure is not fixed, the following is how we stored our records:

| Key        | Value         | Value          | Value | Value | Value |
|------------|---------------|----------------|-------|-------|-------|
| Id Captcha | Id Advertiser | Original Image | Sol X | Sol Y | Stars |

The value used as the key is the Id Captcha because, in our use-case, when querying this database, we expect to search based on the Id Captcha and not the Id Advertiser. The Id Captcha is composed via a string concatenation:

$$id\_captcha = captcha\_name + " - " + id\_advertiser$$

## Log

This module is responsible for managing the log information. All of this module's functionality is developed using Python-written Lambda Functions that use a RDS database to store and retrieve all data. We will cover the functionality associated with this thesis work; for the remainder, see [7].

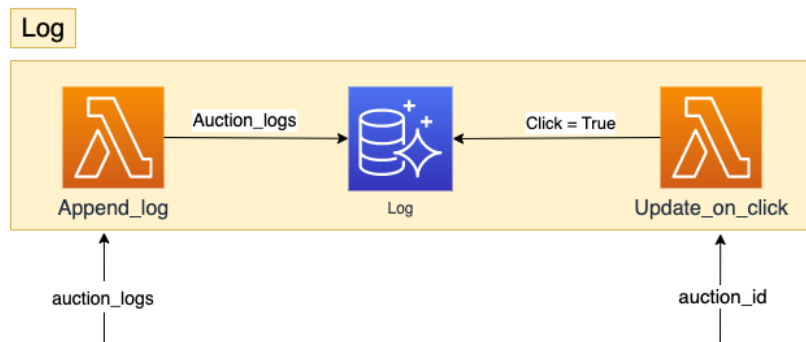


Figure 4.3: Log Service

## Update On Click

**Inputs:** auction id (string)

**Description:** it sets the *click* field of the winner of the specific auction to True.

**Outputs:** None

## Log RDS Database

We chose to implement this database as a relational one (rather than a DynamoDB, like the others discussed thus far) because, during the Update On Click function, we have to perform a slightly complex query (search for the winner of a specific auction). This would be difficult to do using a key-query and would not take advantage of the benefits of NoSQL databases. The storage is arranged in the following way:

| Primary Key            | Attributes  | Type    |
|------------------------|-------------|---------|
| Id<br>(auto increment) | id campaign | Varchar |
|                        | bid         | Float   |
|                        | auction id  | Varchar |
|                        | domain      | Varchar |
|                        | hour        | Integer |
|                        | weekday     | Integer |
|                        | os          | Varchar |
|                        | browser     | Varchar |
|                        | region      | Varchar |
|                        | city        | Varchar |
|                        | site data   | List    |
|                        | win         | Boolean |
|                        | click       | Boolean |

### 4.3 APIs

The API Gateway serves as the entry point for invoking the various microservices. The API Gateway was designed as a RESTful API to make use of standard HTTP methods for accessing our endpoints and to make the system as lightweight as possible.

#### Web Access

This collection of endpoints is used by advertisers and publishers to access different platform features through the dedicated website (eventually, this API could be split into two different ones, each dedicated to a type of user). In my work, I developed the following endpoint:

- **Add Captcha:** this resource, through a POST method, allows an advertiser to add a CAPTCHA in the system by calling the **Captcha - Add Captcha** function.

All of the resources under this API can be accessed only by authenticated users (being publishers or advertisers); this is accomplished by the **AWS Cognito** service.

#### AWS Cognito

AWS Cognito is a service that offers the web app with authentication, authorization, and user management features. First, two user pools were created, one

for advertisers and one for publishers. Cognito also offers hosted authentication pages for our application via which users may sign up, sign in, and recover their password.

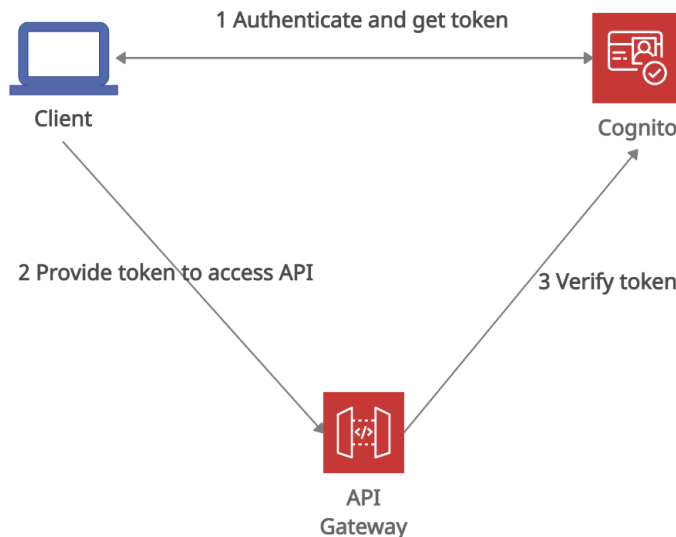


Figure 4.4: Cognito's Implicit Grant Flow

The system implements the OAuth 2.0 *Implicit Grant Flow* (supported by Cognito), which means that once a user logs in, he obtains a token and a cookie for authorization and session management. When the application accesses a protected API that needs the user to be authorized, it includes the token previously acquired in the request's authorization header. This allows Cognito to confirm the identity of the user attempting to access the API. Finally, when the user attempts to logout, the application deletes the session cookies and token.

## Get Ad

This collection of endpoints is used by the publisher's website to perform a variety of actions for each ad slot.

- **Get Ad:** it is invoked whenever the publisher's website needs a sponsored CAPTCHA through a GET method, it starts an auction for the ad slot by calling the **Ad Exchange - Slot handler** function;
- **Check Sol:** it is invoked when a visitor proposes a solution for a challenge through a GET method, given the challenge identifier and proposed solution,

it checks whether it was correct by calling the **Captcha - Check Solution** function. If it was, it also sends back the ad image used to generate the challenge:

- **Click:** it is invoked if a visitor clicks on an ad after solving the corresponding challenge through a PUT method, it updates the corresponding entry in the Log storage by invoking the **Log - Update On Click** function.

## 4.4 Front-End

This section will focus on the front-end of the system that advertisers and publishers use to administer their individual campaigns and websites (see section 2.2). Note that not all front-end actions, such as monitoring ad campaigns, checking all active campaigns, adding CAPTCHA, etc., were developed since we deemed them unnecessary to the completion of this thesis. However, the following framework was established to facilitate the future integration of similar features. The graphic below depicts how the user interacts with the front-end application.

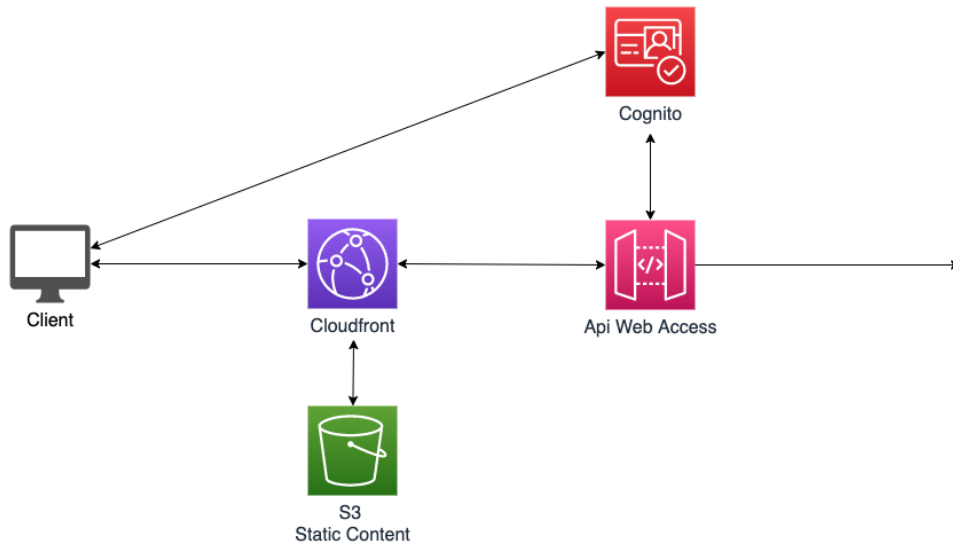


Figure 4.5: Front-end



## Static-Content S3 Bucket

All of the application's static files, including pictures, JavaScript, HTML, and CSS, are being stored in an S3 Bucket. When requested, these static items are fetched by the Cloudfront distribution and provided to the client. This can improve the scalability, reliability, and speed of our static storage.

## Cloudfront

Cloudfront is AWS's CDN service. We chose this strategy to reduce latency since edge locations of Cloudfront may now cache requests and deliver them quicker. During application use, the client requests either static or dynamic content. When static data is requested, Cloudfront retrieves it from the Static-Content S3 Bucket. When requesting dynamic content, the Cloudfront distribution requests the appropriate API from the API Gateway (see section 4.3) and returns the data from the business logic.

## 4.5 Performance

This section will explore the performance of the CAPTCHA module. Specifically, with a focus on how it impacts RTB operations since, as established, performance in terms of time is an essential criterion for assessing any RTB system. In our particular case, however, we may be more forgiving since our advertisement is not crucial to the user's experience because it only appears in very restricted circumstances as a CAPTCHA. For instance, a popular use would be to display it throughout the sign-up process. The CAPTCHA test might be readily loaded while the user is completing the corresponding form. As it does not impact RTB operations, the performance of the Captcha challenge generation (methods Add CAPTCHA and Generate CAPTCHA) will be discussed in a separate chapter (see section 5.3).

Specifically, two different functionalities must be explored: retrieving the challenge (Get CAPTCHA function) and validating the submitted answer (Check Solution function). The AWS Cloudwatch dashboard allows for simple monitoring of our services and performance. Figure 4.6 displays the performance of the whole CAPTCHA retrieval mechanism (RTB included).

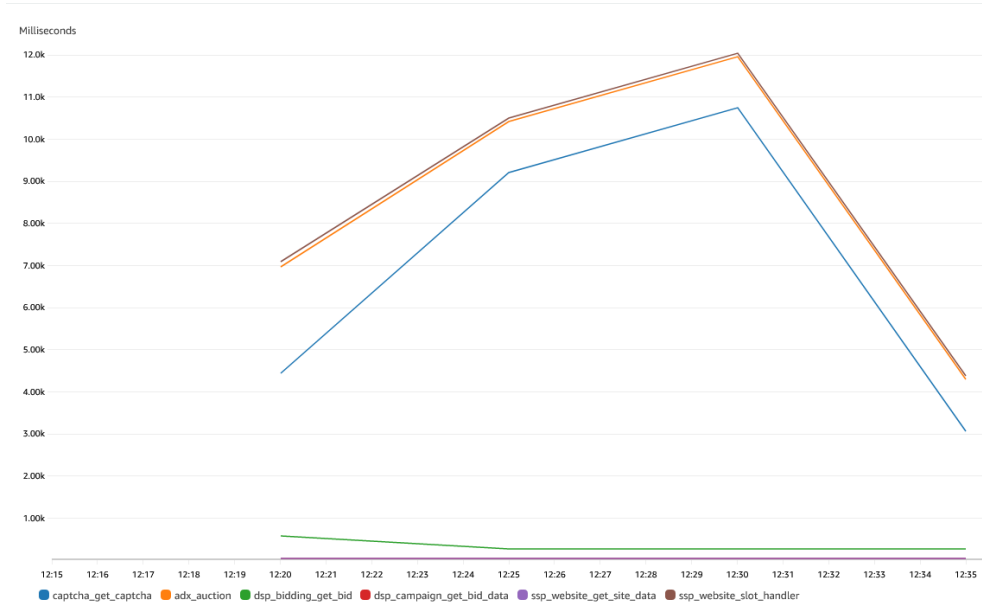


Figure 4.6: Get CAPTCHA Duration Monitoring

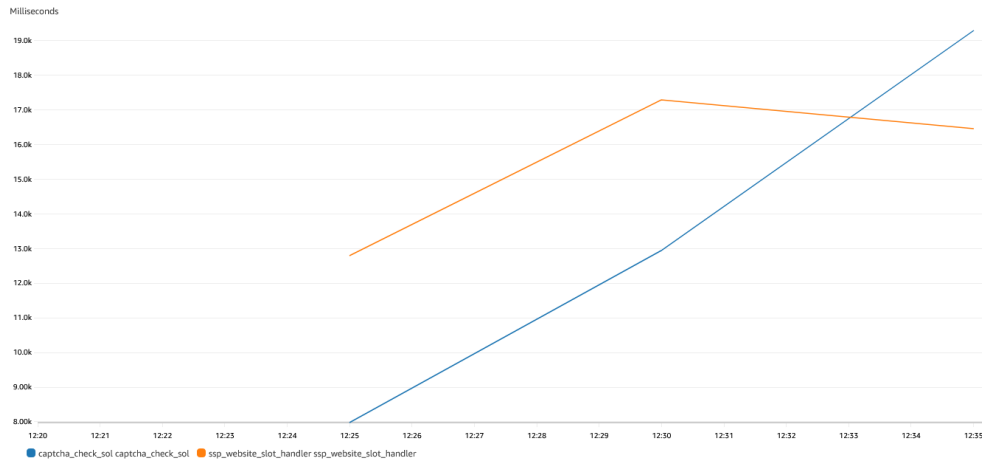


Figure 4.7: Check Solution Duration Monitoring

As described in the dedicated chapter of the complementary thesis [7] concerned with RTB performance, the bottleneck of the whole process is the *captcha\_get\_captcha* function. , as seen in Figure 4.6. The **captcha-db-cs** DynamoDB table is the ultimate source of this problem. Regarding the Check Solution function, as it is executed alongside Get CAPTCHA and queries the same database, it exhibits a similar behavior.

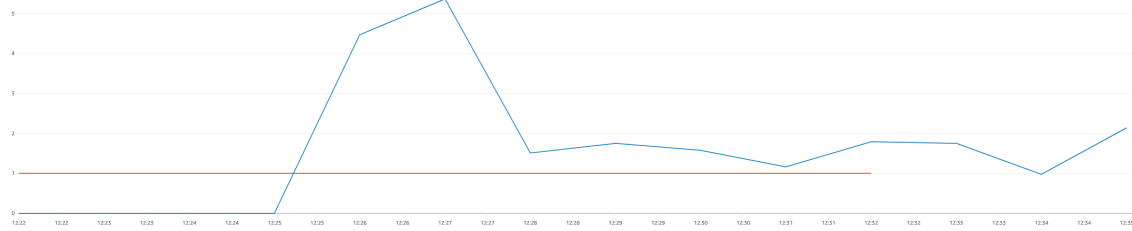


Figure 4.8: Read Capacity Unit Monitoring

During the testing, the Read Capacity Unit, a measure based on the number of reads per second and the size of the read items, was exceeded. Since we are working on the free tier, our read capacity units restriction is quite low. This is likely the reason for our timeouts, since DynamoDB’s policy prohibits it from responding to invocations that exceed our plan’s threshold.

However, this issue is associated with the academic scope of the project, not its design or implementation. In a real-world scenario, the answer would be to adopt a pay-per-use charging policy to accommodate all requests. The Figure 4.8 produces the graph illustrating the threshold and read capacity units used.



## 5. CAPTCHA Generation

The goal of this chapter is to explore the issues with the CAPTCHaStar algorithm in our setting and how it was adapted to better fit our problem. The original algorithm was developed and tested on binary images (only white and black pixels); as such, it only has to select white pixels and create stars from them (details below). However, in our scenario, we need to handle more complex images (i.e., logos) that are not natively binary images.

### 5.1 CAPTCHaStar Algorithm

This section will explain the basic functioning of the CAPTCHaStar algorithm as explained in [1].

The user will be prompted by the CAPTCHA with a series of small white squares that are randomly positioned inside a larger black square. A single white square will henceforth be referred to as a *star*, and the squared black space will be referred to as the *drawable space*. Within the drawable space, the location of each star moves around dynamically in response to the coordinates of the cursor at any given moment. Given a challenge, we define as the *state* a snapshot of the location of the stars in the drawable space, relative to a certain cursor position. The user is given the task of rearranging the positions of the stars on the screen by moving the cursor until he is able to identify a particular shape.

Using a sampling method, the system decomposes the chosen image into many stars. The system determines the movement pattern that will be followed by each star, this is done in such a way that it will be possible for the stars to cluster together and take on the appearance of the sampled picture, which happens in a secret position. That particular position is what is referred to as a solution to the problem.

To make solving the CAPTCHA more difficult for a computer, a certain amount of stars that are not part of the solution and move randomly, are added. These stars are called *noisy stars*.

#### Algorithm in Detail

The following section will delve further into the specifics of the algorithm's actual implementation.

During the generating phase of a challenge, several parameters are used to tune usability and security, including the following:

- **Noise:** the proportion of noisy stars added to the scheme, relative to the total number of stars;
- **Sensitivity:** the relation between the amount of cursor displacement (in pixels) and each star's movement;
- **PicSize:** the number of pixels representing the maximum value between width and height in the sampled image;
- **Rotation:** boolean value indicating whether the image is rotated by a random amount of degrees.

The process of generating a CAPTCHaStar challenge can be split into three major steps:

1. **Preprocessing:** The image is shrunk or grown in proportion to the value that is given in for the PicSize parameter. When the rotation parameter is turned on in CAPTCHaStar, the image is rotated by a random number of degrees. The algorithm converts the image to black-and-white at this point (i.e., binarization);
2. **Picture decomposition:** The picture is first broken up into tiles that are 5 by 5 pixels, and then the sampling method counts the number of black pixels that are contained within each tile. When one of the following conditions is met by a tile, it results in the creation of an original star:
  - if the tile is completely covered in black pixels (that is, it has 5x5 squares full of black pixels), then our system generates a unique star and positions it in the middle of the tile;
  - if the tile has a number of black pixels ranging from 9 to 24, our system generates a unique star and positions it in a location that is off-center on the tile, closer to the area where the majority of the tile's black pixels are located. This ensures that the tile's overall appearance remains consistent;

Our algorithm sets the final shape, which is made up of stars, anywhere inside the drawable space at an arbitrary coordinate (such that all the original stars lie inside).

3. **Trajectory computation:** The pair of coordinates is what we mean when we refer to the solution  $sol$  for the problem  $(sol_x, sol_y)$ . The system produces  $sol_x$  and  $sol_y$  at random, with their values falling somewhere in the interval  $[5, 295]$ . For each initial star  $i$  our system also defines  $(P_x, P_y)$  as the

coordinates of the star's position when the cursor is in coordinates  $P_x, P_y$  ( $sol_x, sol_y$ ). Our system produces four coefficients at random for each star  $i$ . These coefficients relate the coordinates of the star to the coordinates of the cursor. CAPTCHaStar generates the noisy stars in a similar way, but their coordinates  $(P_x, P_y)$  have random values.

## 5.2 Custom Logo Binarization

Since in our context we are working with the specific picture domain of logos, we require an ad-hoc binarization method capable of producing the binary image that yields the best possible CAPTCHaStar challenge given a logo image. The method employed in the original work is unsuitable for our purposes because it only applies to images that are already black and white. A simplistic approach, like applying a threshold binarization algorithm, does not always produce optimal results. To make sure that the logo in the CAPTCHaStar challenge is recognizable, as much detail as possible must be preserved from the original logo; standard binarization algorithms are often unable to perform in such a way in our domain.



Figure 5.1: Original Image

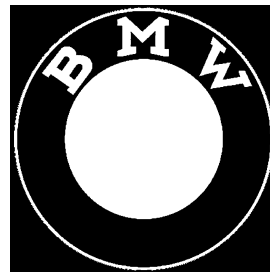


Figure 5.2: Result after basic binarization

For these reasons, a custom binarization algorithm based on the Otsu binarization approach [4] was created. The technique is based on the idea of applying the original Otsu's method to many Regions of Interest (ROIs), and then merging them to preserve as many features as possible.

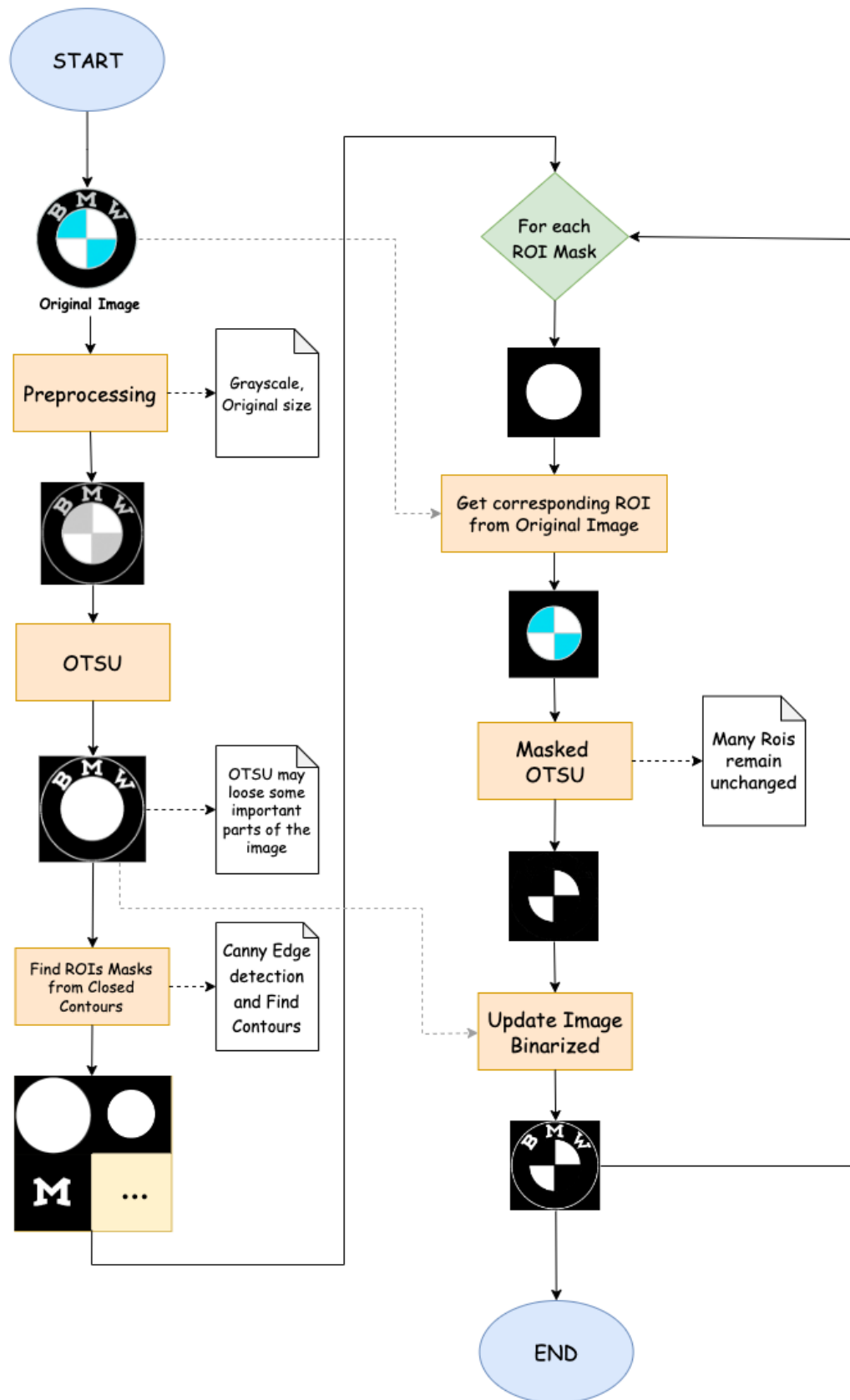


Figure 5.3: Custom Binarization Flow Chart



The following section, will describe each step and its implementation details. The algorithm was developed in Python 3 using several functions from the OpenCV library.

### Preprocessing

In this step, the image is converted into grey-scale with the help of the corresponding OpenCV functionalities. There is also an option for resizing the image. However, a smaller image results in a loss of details, which is exactly what is to be avoided using this algorithm. Thus, the image is kept in its original shape.

### First Otsu

The next step is to apply a binarization algorithm to the image. In particular, Otsu's method is applied. Otsu's method is a variance-based approach for determining the threshold value at which the weighted variance within the foreground and background pixels is minimized. The key concept here is to go through all potential threshold values and estimate the sum of the distances between pixels in the background and those in the foreground. Then, identify the threshold with the smallest variation.

$$total\_var = background\_var + foreground\_var$$

In contrast to other Otsu implementations, also the idea of a mask to restrict binarization to a subset of pixels (Region Of Interest, or ROI) is introduced. In subsequent steps, the method can be applied selectively to the image sub-regions that would be lost if the normal technique were applied to the entire image. Therefore, the initial step is to determine if the mask is to be applied, and if so, to use only the selected pixels. Then, the image's 256-bit histogram is computed, to produce an easily accessible representation of the occurrences of each pixel value [0,255]. Next, the algorithm iterates for each potential threshold value (from 0 to 255). Using each threshold, the pixels are split into foreground (*pixel value*  $\geq$  *threshold*) and background (*pixel value*  $<$  *threshold*); the previously calculated histogram is then used to determine the variance of the foreground and background pixels. The chosen threshold is the one that minimizes the sum of the variances in the foreground and background. Using this value, the picture (or ROI) is then thresholded.

### Find ROI Masks & Get Corresponding ROI

Now, the ROI of the original image must be determined (not the one binarized in the previous step) where the masked Otsu will be applied. This was accomplished by performing multiple OpenCV functionalities.

After obtaining the main edges using the Canny Edge Detection technique, the principal contours are identified. However, only the *closed contours* are of interest (shapes that may be lost during binarization); these are contours that have children in the contours hierarchy returned by the dedicated function. Given these closed contours, it is possible to apply a function that fills the appropriate polygon (which would not be possible with open contours) and then derive a mask from the result. Using each mask, finally extract the respective ROI from the original image.

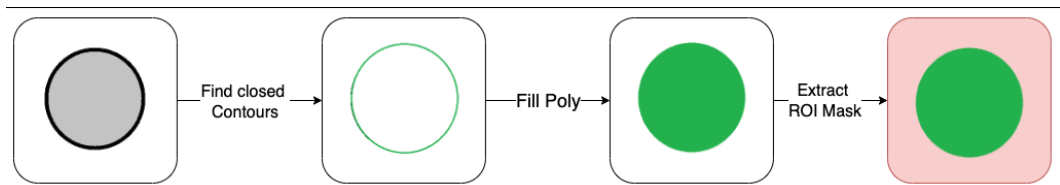


Figure 5.4: Find ROI Mask Pipeline

### Masked Otsu & Update Image Binarized

For each ROI extracted from the previous step, apply the masked Otsu algorithm only to the appropriate pixels. Finally, overwrite the binarized ROI with the original image.

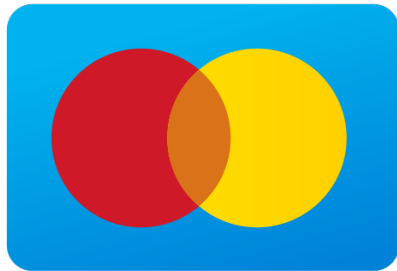


Figure 5.5: Original Image

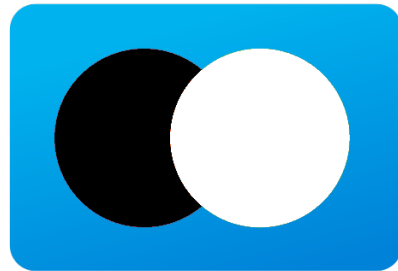


Figure 5.6: After Otsu on ROI

## 5.3 Results & Considerations

### Binarization

Figure 5.7 shows some images where the advantages of using the developed algorithm are clear.

Figure 5.7: Results

| Original logo   | Otsu binarization   | Ad-hoc binarization   |
|---|---|---|
|    |    |    |
|    |    |    |
|   |   |   |
|  |  |  |
|  |  |  |
|  |  |  |

As shown, the normal binarization algorithm causes much of the details of the image to be lost, which, in some cases, might make the logo unrecognizable in a CAPTCHaStar challenge. However, the custom algorithm is able to identify important ROIs of the logo, find its optimal threshold, and keep it in the result.

Clearly, this algorithm is suited for our specific scenario and does not work better than Otsu's method in many other instances. Also, it appears that our

algorithm is much more sensitive to image quality. When the resolution of the image is not optimal, various artifacts are created and some detail can be lost; this can be attributed to the fact that the edge detection algorithm is not able to perform at its best in such conditions. However, this is not an issue for our purpose since those details that get lost are more often than not very small and not recognizable from the final challenge.



Figure 5.8: Otsu's method



Figure 5.9: Masked Otsu on low resolution image



Figure 5.10: Masked Otsu on high resolution image

Another kind of problem occurs when the logo presents a color gradient in sections of the image that should be part of the same shape, as shown in Figure 5.11. This represents the limit of adopting an approach solely based on color thresholding.

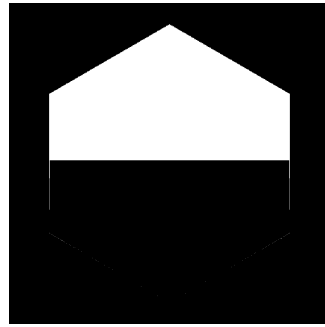


Figure 5.11: Instance where a shape presents a color gradient within

The most extreme case in which this strategy fails is shown by pictures that are defined nearly entirely by color combinations and have few shapes.

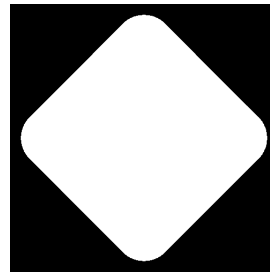


Figure 5.12: Instance where a logo is mostly characterized by colors and not shapes

## CAPTCHaStar Challenges

Using the custom binarization algorithm, the resulting CAPTCHaStar challenges are able to work effectively with logos without losing their identity and meaning. As stated before, even if images with non-optimal resolution are used and, as a result, some details are lost, the overall shape of the logos is almost always faithful to the original.































| Original Image  | CAPTCHaStar   | Original Image  | CAPTCHaStar  | Original Image  | CAPTCHaStar   |
|---|---|---|--|---|---|
|    |    |    |    |    |    |
|    |    |    |    |    |    |
|    |   |   |   |   |   |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

Figure 5.13: Examples of CAPTCHaStar challenges

There are still some concerns with the generation of CAPTCHaStar challenges, despite the fact that the results are fairly good. The most significant relates to how the original method samples pixels during the image decomposition phase. At this stage, the algorithm determines whether white or black pixels constitute the foreground. This decision is based on the number of pixels of each color; the color with fewer pixels is chosen as the foreground. However, this strategy is not always appropriate since the distinction between foreground and background is not always clear. Many logos, for instance, have their own backgrounds within the picture, as seen in Figure 5.14, or have a background with fewer pixels than the

foreground, as depicted in Figure 5.15. This often results in challenges that do not depict the actual picture or do so in an inefficient manner.



Figure 5.14: Instance where the selected foreground alone is not able to correctly convey the ad



Figure 5.15: Instance where the background pixels are used as stars

## Performance

This section will explore the execution time performance of the CAPTCHA challenge generation. All tests were conducted on a computer equipped with an AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx, a 2.10 GHz processor, and 8.0 GB of RAM. These experiments were not conducted on the AWS platform because I wanted to concentrate on the performance of the algorithm itself, which may have been affected by a number of factors had they been conducted on the cloud.

Table 5.1 displays some of the most illustrative findings collected from testing with pictures of varying sizes and from different domains. Note that the performance without addressing the binarization times may vary from the original paper[1]; this is due to the fact that the algorithm was recreated in Python to ease system integration and was slightly optimized.

| Row | Dimension  | Stars | ROIs | Overall Time | Overall Time Excluding Binarization | Binarization Time | Binarization/Overall Time (%) |
|-----|------------|-------|------|--------------|-------------------------------------|-------------------|-------------------------------|
| 1   | 2048x2048  | 591   | 0    | 0.137        | 0.022                               | 0.115             | 83.79                         |
| 2   | 1023x1251  | 1570  | 0    | 0.21         | 0.046                               | 0.168             | 78.56                         |
| 3   | 500x500    | 657   | 3    | 0.233        | 0.022                               | 0.211             | 90.69                         |
| 4   | 2500x2500  | 427   | 3    | 0.453        | 0.023                               | 0.431             | 94.99                         |
| 5   | 340x474    | 504   | 5    | 0.439        | 0.022                               | 0.41              | 94.93                         |
| 6   | 1750x2300  | 657   | 8    | 0.38         | 0.037                               | 0.343             | 90.27                         |
| 7   | 6424x10000 | 516   | 11   | 1.608        | 0.029                               | 1.579             | 98.19                         |
| 8   | 1200x1200  | 1424  | 17   | 1.037        | 0.079                               | 0.95              | 92.40                         |
| 9   | 500x500    | 699   | 22   | 0.597        | 0.035                               | 0.562             | 94.12                         |
| 10  | 2400x2400  | 902   | 35   | 2.47         | 0.054                               | 2.419             | 97.82                         |
| 11  | 500x500    | 965   | 40   | 0.76         | 0.035                               | 0.729             | 95.40                         |
| 12  | 1080x1000  | 868   | 42   | 1.72         | 0.037                               | 1.69              | 97.84                         |
| 13  | 722x750    | 261   | 43   | 1.79         | 0.019                               | 1.77              | 98.91                         |
| 14  | 600x600    | 864   | 54   | 1.437        | 0.035                               | 1.402             | 97.56                         |
| 15  | 720x1280   | 1106  | 61   | 2.607        | 0.045                               | 2.562             | 98.27                         |
| 16  | 1390x1300  | 1786  | 69   | 2.830        | 0.095                               | 2.730             | 96.63                         |
| 17  | 967x1094   | 1326  | 86   | 2.525        | 0.079                               | 2.447             | 96.89                         |
| 18  | 500x500    | 478   | 89   | 2.078        | 0.035                               | 2.042             | 98.29                         |
| 19  | 500x500    | 672   | 112  | 2.30         | 0.033                               | 2.27              | 98.57                         |
| 20  | 1200x1200  | 484   | 258  | 5.037        | 0.02                                | 5.017             | 99.60                         |

Table 5.1: Results

Various patterns may be discovered by analyzing the data presented above. As shown by the last column, the custom binarization preprocessing described in section 5.2 has the greatest influence on the whole process (90 percent of the entire length of generation). Since the binarization is strongly correlated with the number of identified closed contours (ROIs) in the original image, this component has a substantial impact on the whole process.

Furthermore, rows 1 and 2 reveal that, in terms of duration, the number of stars is the most significant parameter in the absence of ROIs. Alternatively, with the same number of ROIs, the dimensions seem to have a greater influence than the number of stars (rows 3-4 and 12-13). In the most extreme situation, when the dimensions are almost out of range (rows 6 and 7), they seem to be a crucial determinant of performance. This behavior is not always explicit and may be very sensitive to the nature of the picture itself.



## 6. Security Assessment

This chapter will cover the evaluation of the system’s CAPTCHA resiliency to several types of attack, regarding both the original work[1] and our platform.

### 6.1 Sponsored CAPTCHaStar Resiliency

The original work provides a comprehensive analysis of CAPTCHaStar’s resiliency to conventional attacks. While the findings of the assessment of traditional attacks are encouraging, the outcomes of attacks employing ad-hoc heuristics and machine learning are more difficult to pinpoint.

This section will examine how the assessed attacks apply to our particular case.

- **Indirect Attack:** It is an attack that exploits information available on the client side. As for the original, an indirect attack is not feasible since the front-end lacks any knowledge about the solution;
- **Exhaustion of Database:** It describes a situation in which an attacker gathers information on all current CAPTCHAs. We cannot choose random pictures, unlike the original work, since we are constrained by the RTB and auction winner. Currently, we are susceptible to this kind of attack. Nonetheless, several potential solutions are addressed in section 7.2;
- **Leak of Database:** It is an attack that takes advantage of some leaked information from the CAPTCHAs database. A potential way to fix this problem is to hash the solution in the database;
- **Random Choice:** It is a kind of attack in which the attacker guesses the answer. As in the original work, CAPTCHaStar also accepts the neighborhood of the solution as a valid response (according to the value of the usability tolerance parameter). However, the likelihood of a random guess succeeding is around 0.09 percent with a usability tolerance of 5 (currently used);
- **Pure Relay Attack:** It refers to an attack consisting of the transmission of an image expressing a challenge to a third party who might solve it. CAPTCHaStar is immune to this approach since the finding of the answer involves continual contact with the CAPTCHA. Because of this, a single screenshot transmitted to a third party is insufficient to execute a relay attack;

- **Stream Relay Attack:** It is similar to the Pure Relay, but a constant transmission of data regarding the challenge is employed. Unfortunately, this is still the most successful approach against CAPTCHAs (including our proposal);
- **Ad-hoc Heuristics:** It refers to all of the approaches specifically designed to attack CAPTCHaStar challenges. The authors [1] tested this kind of attack using a technique of their own (as explained in section 1.1). They concluded that the efficacy of this approach can be successfully reduced by increasing the number of noisy stars in the challenge while maintaining acceptable usability values;
- **Machine Learning:** It encompasses all of the attacks that rely on a machine learning model to solve the challenges. Even if the authors tested the resiliency with some ad-hoc models (see section 1.1), I wanted to create and test a model specifically designed to deal with sponsored CAPTCHAs.

## 6.2 Ad-hoc Adversarial Attack

The main idea behind this attack is rather simple: to train a model to recognize whether a given sponsored CAPTCHaStar challenge is a solution or not. Then, given a challenge, it automatically retrieves all the states (or a subset of them, depending on the usability tolerance parameter) and feeds them to the model. The best candidate solution is the one for which the model estimates the highest likelihood of being a solution.

### Attack Model

This section will address the context of the attack simulation and the constraints of an eventual attacker. As stated before, the malicious user's objective is to exploit a pretrained model to estimate the probability of each state being solution. As such, there is no upper bound to the amount of labeled states that can be used to train such model; of course, it is always preferable, from the point of view of an attacker, to use as less data as possible. Moreover, the attacker can extract each state in form of an image without any constraints as it would be reasonably easy to do in a real-life scenario (e.g., take a snapshot for each cursor position). I am also assuming that the attacker has some prior knowledge regarding the CAPTCHaStar algorithm and the value of some parameters, in particular the tolerance parameter and the constraint on the coordinates of the possible solution (see chapter 5). Finally, the parameters used for the CAPTCHaStar challenge generation are the ones used in the original work[1].

## Dataset

The first step was to build a dataset for training the model. To do this, a collection of initial logo pictures was required. However, I was unable to locate any such datasets, so I downloaded such pictures from the Icons8 [6] website, which provides a vast selection of free icons and logos. I specifically gathered 400 pictures from there (500x500). I generated 5 challenges for each of them using the existing settings and the binarization technique (section 5.2). Finally, five images representing states deemed valid as solutions and five images representing states not considered valid as solutions were gathered for each challenge. The final database had 10,000 positive pictures and the same number of negative images, for a total of 20,000 samples.

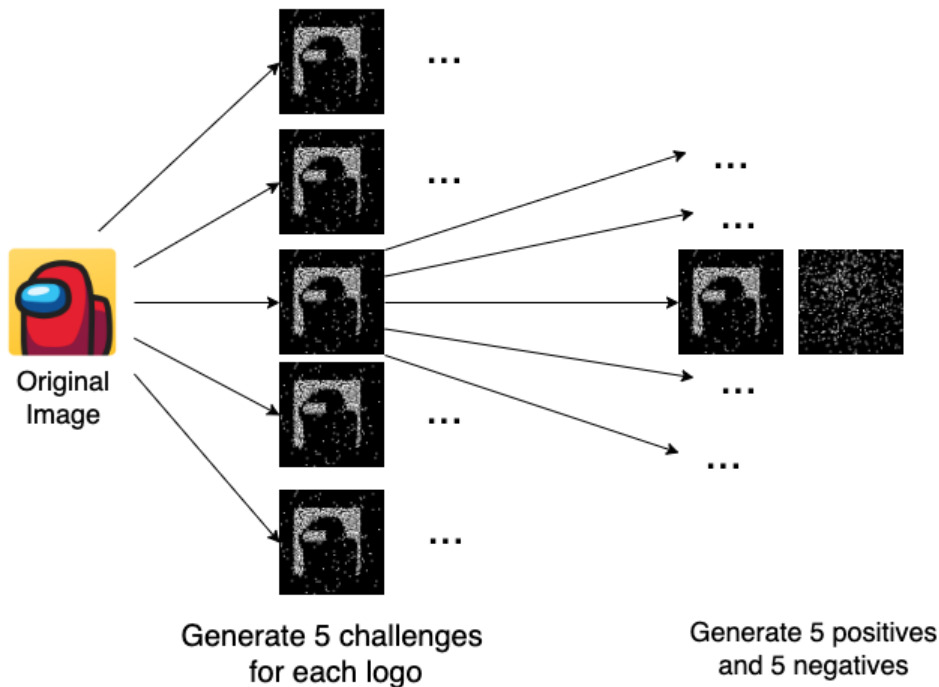


Figure 6.1: Dataset Generation

## Model & Training

This section addresses the training and validation of the model using the dataset covered in the preceding section. I used Python as the basic environment, PyTorch for building and training the model, and OpenCV for performing different picture transformations (I chose OpenCV instead of PyTorch for image transformations to minimize incompatibilities, see section 6.2).

Starting from the dataset as described before, I transformed the images in order to ease the learning process: the images were scaled to reach dimensions of 28x28 to minimize the complexity of the model, and for the same reason, the images were converted into grey scale. As a result, each sample of the dataset is formed by a 28x28 grey-scale image representing the state of a challenge and a label indicating whether the challenge is an acceptable solution or not. I ultimately employed a convolutional network as a prediction model after some testing. As shown in Figure 6.2, the network consists of two couples of convolutional (kernel size 5, stride 1, padding 2) and max pooling layers (kernel size 5) with 16 and 32 output channels. The features are then flattened and used as input for the final fully connected layers, which have two output features (*not solution* and *solution*).

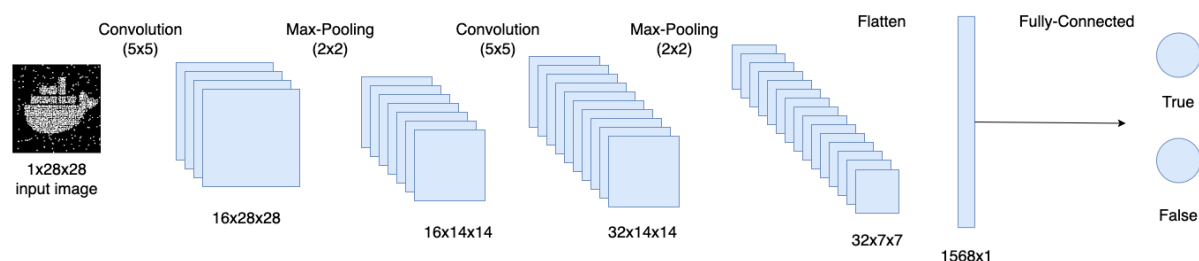


Figure 6.2: Model's Architecture

To assess the quality of the model, I divided the dataset into three distinct sets: train, test, and unseen test. The test set comprises only samples of logos that also occur in the train set, while the unseen test contains only samples of logos that have never appeared in such set. This was done to determine the model's ability to generalize outside of the training set.

After achieving almost optimum results on the whole test and unseen sets (0.99 accuracy), I attempted to reduce the amount of samples in the dataset while maintaining a high level of performance. This is because I wanted to determine the minimal number of samples an attacker would need to construct a model with adequate performance. After a series of tests, I was able to attain satisfactory results with a train set including 170 samples. The model was trained with the following parameters: 32 batch size, 0.001 learning rate, and 40 epochs. With so few images, performance may be erratic, but in general, the accuracy of both the test and unseen test exceeds 0.9, even with a 3800-sample unseen test. In conclusion, the model generalizes really well, and rather remarkable results may be produced with less than 200 images and less than one minute of training time.

## Attack Simulation

To evaluate the effectiveness of an attack based on the prediction model previously described in a real-life setting, I developed a prototype showing how an attacker would act if they had access to such a model. Since the overall accuracy of the attack depends directly on the model which was previously evaluated, the goal of this simulation was to test how feasible it is in real-life. In particular, I was interested in testing the performance in terms of elapsed time; keeping in mind that a human needs, on average, between 30 and 60 seconds to complete a challenge, the overall time needed by the attack should ideally be lower or equal.

Given a website displaying the CAPTCHaStar challenge, a collection of images representing a set of states are collected and fed into the model. The state with the highest probability of being the solution is chosen, and its relative coordinates are submitted as the solution. The attack simulation is a client-server application: the client-side consists of JavaScript code that collects the states, preprocesses them, and provides them to the server-side. The server then employs the predicting model described in the previous section and obtains the predictions for each state; the client-side then provides the coordinates associated with the state having the greatest probability of being the solution. The client-side leverages the *OpenCV.js* library to do the required transformations, while the server-side is developed using the Python-based Flask web framework. This decision is motivated by the fact that I need to utilize PyTorch in order to use the model, which can only be accessed through Python code.

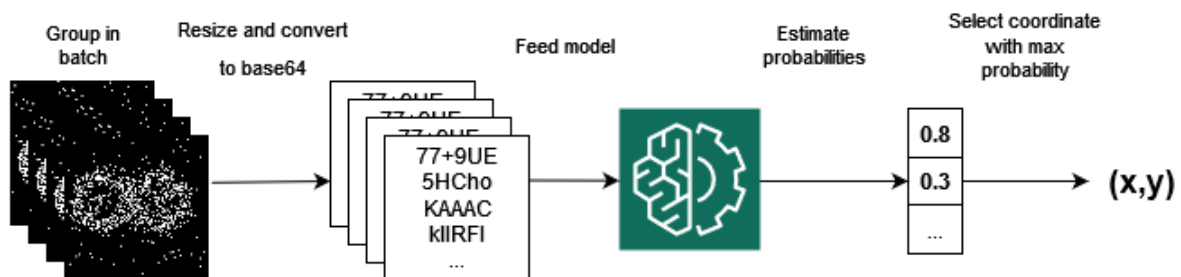


Figure 6.3: Attack Simulation Pipeline

The pipeline of the attack simulation is as follows:

- The images associated with the states are retrieved. Since the challenges have a tolerance on the position of the solution, it is not necessary to retrieve all of the states associated with every possible cursor position in the range [10,290]. So, I retrieve a state every 4 coordinates for both x and y axis;

- Each image is then resized into 28x28 (this operation is performed here in order to limit the size of the requests to the server-side) and converted into base64 format in order to be sent on the server-side;
- Since Flask has a limit on the size of the data that can be received by a single request, the images are grouped into batches of 700 and sent separately;
- For each batch received, the server-side, after performing some preprocessing transformation (e.g., converting into grey scale), feeds the batch to the model and identifies the state associated with the highest probability, and sends back to the client the coordinates and relative estimated probability;
- Once all the batches' results are obtained, the client-side only has to select the highest probability and submit the associated coordinates as a solution.

For the sake of simplicity, the images associated with each state are retrieved directly from the CAPTCHaStar code. However, it would be rather straightforward to do so using basic JavaScript code. The final thing to note is that the process could be sped up (i.e., by increasing the size limit of Flask requests), but I chose not to do so because, at the moment, the average execution time is 30 seconds, which is consistent with the time required by real users to complete a challenge according to [1].

In conclusion, I demonstrated that the CAPTCHaStar system, or at least this variant, has some significant security flaw. A malicious user can quite effectively attack the system after retrieving a modest number of images. In terms of time, the simulation is currently consistent with a human user and, if deemed required, might be improved with modest effort. That being said, the images must be balanced between positive and negative states and must be labeled. Possible solutions to this problem will be analyzed in section 7.2.

# 7. Conclusions

## 7.1 Obtained Results

Upon completion of this thesis work, I was able to accomplish many milestones. In conjunction with the complementary thesis [7], I designed a cloud architecture (chapter 3) that offers the following key functionalities: it delivers a sponsored CAPTCHA via an RTB system and provides advertisers and publishers with various utility capabilities. Moreover, the architecture, which is based on a microservices framework, is able to meet a number of requirements that we regarded essential, such as scalability, ease of maintenance, and adaptability to future modifications, particularly to the bidding strategy and auction.

I was responsible for the development of the APIs (section 4.3) and services directly connected to the CAPTCHA challenges (section 4.2): Captcha and Log, for this AWS-implemented architecture. These modules integrate into the RTB system all of the capabilities associated with the creation, selection, and resolution of challenges.

Particular emphasis was placed on the generation phase. Using a customized binarization method (section 5.2), the original CAPTCHA project was converted to operate in our domain. Using this approach, which is based on applying Otsu's method iteratively to each closed area of the picture, yields satisfactory results. However, considering the highly complicated and unique nature of the logo domain, the existing method may still be significantly improved.

Finally, in this study, I conducted an in-depth investigation of the solution's resilience against malevolent automated attacks (chapter 6). Once I evaluated the resiliency to more conventional methods, I created a custom automated attack based on Machine Learning using Convolutional Neural Networks (section 6.2) and tested it by simulating what a hostile user could try to do in order to automatically solve the challenge. Although this kind of attack requires a significant amount of effort to obtain the required data for training, the results reveal a weakness in this particular attack, indicating that the current compromise between security and usability must be revised.

## 7.2 Future Work

Several possible system enhancements have become apparent as a result of previous work. The first area for development is the binarization of logos used to produce challenges. As previously noted, the domain of logos is far more intricate than one may expect. Consequently, the existing strategy has flaws that need to be addressed in the future.

- Certain logos, especially those with an embedded background, present issues that diverge from a possible optimal solution. Currently, the CAPTCHA generating logic favors having fewer stars, picking the portion with fewer pixels as the foreground. However, this strategy is not ideal and should be reconsidered. Training an ad-hoc machine learning model to discriminate between background and foreground and then applying custom logic might be a feasible, albeit extremely expensive, option;
- Since I am using a color-based binarization approach, images with color gradients in the same region often result in sub-optimal binarizations. Applying preprocessing in order to "flatten" the colors in the pictures might be one option. As an alternative, a different binarization approach might be used in such circumstances;
- There are a few logos for which the binarization method is simply inappropriate. These images, however few in number, need a completely unique binarization approach. As an example, a method that, after identifying the picture's contours, puts the stars just on them rather than campionating the whole image.

The second area requiring future development relates to various aspects of security.

- Encrypting the database's solutions using one or more hashing algorithms would make the database more robust to potential leaks;
- Another concern that must be addressed is the possibility of database exhaustion. If an attacker can obtain knowledge of all the present challenges, he may be successful in overcoming them. This issue may be solved in a variety of ways, including serving each challenge just once and then generating a new one, regenerating all challenges according to certain parameters (e.g., time elapsed or challenges served), and potentially others;
- The final and most significant security concern is associated with the outcomes of the machine learning attack I created. As previously stated, the



most efficient solution to this issue is to reconsider the trade-off between security and usability. By decreasing the tolerance value, a hypothetical attacker would have to explore many more states than I did, hence raising the cost of an attack. Alternately, by increasing the noise, the learning process would become more complicated and would likely need a substantial amount more data. However, in our scenario (as with any CAPTCHA system), usability is of the upmost importance; as such, increasing the domain's complexity without sacrificing usability would be a preferable approach. For instance, the advertisements may consist of whole banners rather than just logos, which would make the deployment of this kind of attack far more challenging.

Checking for unusual cursor behavior during challenge resolution might be an alternate strategy for increasing resilience to this sort of attack. Since the attack must scan the majority of states, one approach would be to determine if the cursor follows a predetermined pattern or if it scans an excessive number of states after getting close to the actual solution. Implementing such controls would assure that the logic of any future assault would be substantially more complex and expensive.



## A. Code

---

```
1  def binarize(self, img: np.array):
2      img = self._img_preprocess(img)
3      img_bin = self._otsu(img)
4
5      # find ROIs
6      contours = self._get_closed_contours(img_bin)
7      rois = self._find_rois(img, img_bin, contours)
8
9      # overwrite binarized ROI on original binarized img
10     for roi_mask, contour in rois:
11         roi_bin = self._otsu(img, roi_mask)
12         img_bin[roi_mask] = roi_bin[roi_mask]
13
14     return img_bin
```

---

Listing A.1: Main Code of the Binarization Algorithm

---

```

1 def _otsu(self, img, mask=[]):
2     least_var = float('inf')
3     final_thr = None
4     image = img.copy()
5     is_gray = (len(img.shape) == 2)
6     if not is_gray:
7         image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
8
9     masked_image = image if len(mask)<=0 else image[mask]
10
11     # useful for retrieving occurrences
12     [hist, _] = np.histogram(masked_image, bins=256, range=(0, 255))
13     p_all = masked_image.size
14
15     # for each possible threshold
16     for t in range(256):
17         # number of pixels in each class
18         n_bg = np.sum(hist[:t])
19         n_fg = np.sum(hist[t:])
20
21         if n_bg < 1 or n_fg < 1: continue
22
23         # proportion of pixels in each class, sum up to 1
24         w_bg = n_bg/p_all
25         w_fg = n_fg/p_all
26         # get values in each class,
27         # obtained by the product of the value of the pixels by their occurrences
28         vals_bg = np.array(range(t)) * hist[:t]
29         vals_fg = np.array(range(t, 256)) * hist[t:]
30         mean_bg = np.sum(vals_bg)/n_bg
31         mean_fg = np.sum(vals_fg)/n_fg
32         var_bg = np.sum(np.power(vals_bg-mean_bg, 2))/n_bg
33         var_fg = np.sum(np.power(vals_fg-mean_fg, 2))/n_fg
34
35         #  $\sigma^2(t) = w_{bg}(t)\sigma^2_{bg}(t) + w_{fg}(t)\sigma^2_{fg}(t)$ 
36         total_var = w_bg * var_bg + w_fg * var_fg
37         if total_var < least_var:
38             least_var = total_var
39             final_thr = t
40
41     # set new colors based on obtained threshold
42     if final_thr:
43         image[image <= final_thr] = 0
44         image[image > final_thr] = 255
45
46     return image

```

---

Listing A.2: Masked Otsu Algorithm

---

```

1 async function solve(){
2     // hyperparameters
3     const step = 4;
4     const batch_size = 700;
5
6     // vars used while looking for solution
7     let batch = [];
8     let candidate_x, candidate_y, batch_max_prob;
9     let highest_prob = 0;
10    let sol_x = 0;
11    let sol_y = 0;
12
13    // get images of all states
14    for(x=10; x<290; x=x+step){
15        for(y=10; y<290; y=y+step){
16
17            // retrieve image from canvas
18            // getImageFromCanvas() returns the images resize 28x28
19            // and converted into base64 format
20            draw(x, y);
21            img = await getImageFromCanvas();
22
23            state = {'x':x, 'y':y, 'img':img};
24            batch.push(state);
25
26            // when the batch is full
27            // send data to backend and get predictions
28            if(batch.length >= batch_size){
29
30                // retrieve predictions from backend
31                res = await getPredictions(batch);
32                candidate_x = res['x'];
33                candidate_y = res['y'];
34                batch_max_prob = res['prob'];
35                batch = [];
36
37                // if we found a new state with higher prob,
38                // update solution data
39                if(batch_max_prob > highest_prob){
40                    highest_prob = batch_max_prob;
41                    sol_x = candidate_x;
42                    sol_y = candidate_y;
43                }
44            }
45        }
46    }
47    console.log("solution: " + sol_x + ", " + sol_y);
48    draw(sol_x, sol_y);
49 }

```

---

Listing A.3: Main code of the attack simulation



## 8. Acknowledgements

This project, as well as the entire journey, would not have been possible without the participation or support of many people. First and foremost, Professor Gabriele Tolomei and Mauro Conti, respectively Advisor and Co-Advisor of this thesis, proposed and made this work possible.

I also want to express my heartfelt gratitude to my colleague Dennis Cimatori, with whom I spent practically every second of my last years of academic life. Without him, none of this would have been as enjoyable. Of course, I'd like to thank my friends and family for their unwavering support throughout my years of study.

Last but not least, I want to thank my other half, Claudia, for being with me on this adventure every day and caring about me, my outcomes, and my progress in life almost as much as I do.





## 9. Bibliography

- [1] Mauro Conti, Claudio Guarisco, Riccardo Spolaor, *CAPTCHaStar! A Novel CAPTCHA Based on Interactive Shape Discovery*.
- [2] Jeff Yan, Ahmad Salah El Ahmad. *Usability of CAPTCHAs or usability issues in CAPTCHA design*
- [3] Mohit Agarwal, Dr. Gur Mauj Saran Srivastava. *Cloud Computing: A Paradigm Shift in the Way of Computing*.
- [4] Noboyuki Otsu, [https://en.wikipedia.org/wiki/Otsu%27s\\_method](https://en.wikipedia.org/wiki/Otsu%27s_method)
- [5] Open CV, <https://docs.opencv.org/4.x/>
- [6] Icons8, <https://icons8.com>
- [7] Dennis Cimatori, *Design and Implementation of a Sponsored CAPTCHA Platform: Real Time Bidding based on Click-Through Rate Prediction*