Characterizing Improper Input Validation Vulnerabilities of Mobile Crowdsourcing Services

Sojhal Ismail Khan
sojhalis@usc.edu
University of Southern California
USA

Dominika Woszczyk
d.woszczyk19@imperial.ac.uk
Imperial College London
UK

Chengzeng You
chengzeng.you19@imperial.ac.uk
Imperial College London
UK

Soteris Demetriou
s.demetriou@imperial.ac.uk
Imperial College London
UK

Muhammad Naveed
mnaveed@usc.edu
University of Southern California
USA

ABSTRACT

Mobile crowdsourcing services (MCS), enable fast and economical data acquisition at scale and find applications in a variety of domains. Prior work has shown that Foursquare and Waze (a location-based and a navigation MCS) are vulnerable to different kinds of data poisoning attacks. Such attacks can be upsetting and even dangerous especially when they are used to inject improper inputs to mislead users. However, to date, there is no comprehensive study on the extent of improper input validation (IIV) vulnerabilities and the feasibility of their exploits in MCSs across domains. In this work, we leverage the fact that MCS interface with their participants through mobile apps to design tools and new methodologies embodied in an end-to-end feedback-driven analysis framework which we use to study 10 popular and previously unexplored services in five different domains. Using our framework we send tens of thousands of API requests with automatically generated input values to characterize their IIV attack surface. Alarmingly, we found that most of them (8/10) suffer from grave IIV vulnerabilities which allow an adversary to launch data poisoning attacks at scale: 7400 spoofed API requests were successful in faking online posts for robberies, gunshots, and other dangerous incidents, faking fitness activities with supernatural speeds and distances among many others. Lastly, we discuss easy to implement and deploy mitigation strategies which can greatly reduce the IIV attack surface and argue for their use as a necessary complementary measure working toward trustworthy mobile crowdsourcing services.

CCS CONCEPTS

• **Security and privacy → Mobile and wireless security**; **Web application security**.

KEYWORDS

crowdsourcing, real-time, data-poisoning, api fuzzing

ACSAC '21, December 6–10, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8579-4/21/12.
https://doi.org/10.1145/3485832.3485888

ACM Reference Format:
Sojhal Ismail Khan, Dominika Woszczyk, Chengzeng You, Soteris Demetriou, and Muhammad Naveed. 2021. Characterizing Improper Input Validation Vulnerabilities of Mobile Crowdsourcing Services. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3485832.3485888

1 INTRODUCTION

Mobile crowdsourcing services (MCSs) enable economical, rapid, and scalable data acquisition utilized for accurate information sharing for smart navigation and transportation ([4, 10]); health and fitness recommendations [3, 6, 7]; and price tracking [1, 13] among others. However, location-based (Foursquare, Facebook Places [42]) and navigation (Google Maps [45], Waze [52]) MCSs have been shown to be susceptible to ad-hoc data poisoning attacks. For example, recently an individual showed how a real-world navigation service can be fooled to make wrong predictions on traffic density, allowing an adversary to redirect traffic [45]. This experiment was performed manually by carrying 100 smartphones while walking up and down the target road. While this demonstrates both the feasibility and potential consequences of data poisoning attacks on MCS, the experiment is hard to replicate and systematically scale it to study fundamental issues enabling such exploits in other MCS. More systematic studies conducted by Polakis et al. [42] and Wang et al. [52], while sound, they are specific to the characteristics of the target MCS and thus fall short in providing generalizing insights on the vulnerabilities of MCSs.

We observe, that MCSs across application domains suffer from a common vulnerability, that of *improper input validation* (*IIV*) which can be exploited by an adversary to inject hazardous data or spread mis-information. To better understand the presence of IIV vulnerabilities in MCSs and to what extend they contribute to their exposure to improper input injection attacks, we conduct a systematic analysis on 10 high-profile, previously unexplored MCSs across 5 different application domains. To perform our analysis we had to overcome two main challenges: firstly, the closed source nature of MCSs does not allow for trivial examination of their input validation mechanisms; and secondly, testing a large number of input values for different input types is impractical. To overcome the first challenge we present a feedback-driven analysis framework suitable for black-box analysis of MCSs. The framework

leverages the observation that most MCSs interface with their participants through companion mobile apps and embodies a set of *input injection components* for interacting with the remote service. The injection components target three main avenues an adversary can exploit on companion mobile apps to inject improper inputs to the remote service: the sensor measurements used by the companion apps, their user interface inputs, and their network requests to the target service. The framework also uses *feedback monitoring* strategies for facilitating the evaluation of each input injection. To overcome the second challenge, we introduce and integrate in the end-to-end analysis framework a set of *input exploration strategies*. These generate values for various input types, supporting *range and constraint* and *semantic* input exploration.

We then design systematic experiments for all selected services and leverage our framework to characterize the extent of their exposure to improper input injections. Our analysis led to an array of alarming findings. We found that 8/10 services are vulnerable to such attacks. Some of them do not perform any kind of input validation, accepting values which span across the expected input's domain *range*. Some have some restrictions on the range but perform no *semantic* validation. Even for the ones that do take measures to verify inputs we show that they are bypassable. Our findings are alarming: we were able to fake running activity speeds equivalent to 10x the speed of a commercial jet aircraft and run distances equivalent to running around the Earth 2007 times (Section 4), reduce prices for commodity items down to 10% their value and up to double their value (Section 5), fake public bus rides (6), inject fake places of interest in the middle of the ocean (Section 7), and fake reports for robberies, gunshots and other dangerous incidents in safety services (Section 8). Demos of successful improper input injections can be found on our project's website [22]. To mitigate such issues, we discuss a set of backward-compatible and simple to implement input validation strategies which can reduce the attack surface of MCSs up to 99.58%. These can complement and increase the efficiency and effectiveness of existing countermeasures such as reputation schemes, UI hints and majority voting.

Contributions. Below we summarize our main contributions:

• *New Techniques.* We develop range and constraint, and semantic input exploration strategies for generating values for numeric inputs, GPS coordinates and social posts. We further design methods to simulate adversarial capabilities for spoofing network API requests, UI inputs and sensor inputs.

• *Framework for Analysis of IIVs in MCSs.* We introduce a feedback-driven framework which embodies input exploration and injection methods in tandem with feedback monitoring mechanisms to facilitate the analysis of IIV vulnerabilities in MCSs from the vantage point of their companion mobile apps.

• *New Findings.* We discovered and reported previously unknown vulnerabilities for 8 high-profile MCSs that can have grave consequences on their veracity and ensuing trustworthiness.

Ethical Considerations. Even though this study was classified as IRB–exempt, we take various measures in our experiments to mitigate risk of affecting users or services. These include monitoring services' activity and focusing on regions and times to minimize the exposure of erroneous values to real users; and deleting or reverting values to their original state immediately after we verify their approval by the service. See Appendix C for further details.

Responsible Disclosure. Affected services were contacted at least 3 months prior to the time of writing through their mobile app's developer email address. Transit, MapMyRun and Fitbit responded with an automated email confirming receipt, but there was no follow-up. Strava requested further details which we provided. We also submitted reports to bug bounty programs of Fitbit and MapMyRun on bugcrowd.com. Fitbit responded stating that the issue is not applicable for a reward because they couldn't identify a security impact for their customers.

2 BACKGROUND AND THREAT MODEL

Improper Input Validation. Mobile crowdsourcing services face the risk of data injection attacks. In such an attack, a malicious participant node injects an erroneous measurement in the global service aiming to force an error or deceive the users of the service. To achieve this, the adversary might target *improper input validation* (IIV) vulnerabilities. Validation can be *syntactic*, *range and constraint* or *semantic*. Lack of syntactic verification might cause crashes. For example, a service or its corresponding user-facing mobile application might expect the user to report a number. However, in light of insufficient input syntactic validation, an adversary might cause a service to crash by introducing a measurement of an unexpected input type. Fuzzing techniques are typically employed to uncover such reliability issues [26]. In this work, we focus mainly on range and constraint validation and semantic validation. Attacks leveraging these are harder to detect as at a first glance the reported values do not seem anomalous.

• *Range and Constraint validation.* This step ensures that the input domain range is minimized to accept values meaningful to the context of the service. For example, a service expecting GPS coordinates as float values with lack of range validation might be poisoned with non-existing coordinates (e.g. longitude value greater than 180.0°).

• *Semantic Validation.* Semantic validation is used to validate the meaning of the input. An adversary exploiting the lack of semantic validation in GPS inputs might introduce float values that do correspond to valid GPS coordinates range but for an implausible location of the given point of interest.

Threat Model. We consider an adversary ($\mathcal{A}$) with access to the mobile app of the target ubiquitous crowdsourcing service. $\mathcal{A}$ can observe the traffic generated between the app and the remote service either by passive eavesdropping or active man in the middle attacks. $\mathcal{A}$ can also reverse engineer and analyze the mobile app interfacing with the service. Thus the adversary is in knowledge of the communication protocol and can leverage it to try to inject fake data into the service. However, the adversary has no access to the remote service and can only treat it as a black box.

3 ANALYSIS FRAMEWORK

3.1 Overview

To characterize the IIV attack surface of MCSs we need a systematic way of exploring the ability of $\mathcal{A}$ to launch successful attacks exploiting the lack of input validation in MCS. This is by no means a trivial endeavor. One could perform improper input injection

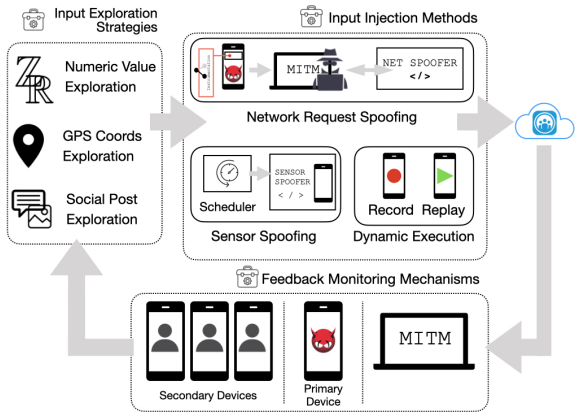


Figure 1: IIV Analysis Framework.

attempts by manually interacting with the services' companion mobile apps, but this would be both tiresome and impractical for a large number of test inputs. To make things worse, we do not have white box access to the MCSs and hence we cannot trivially determine if a service performs input validation.

To overcome these challenges, we present an analysis framework which can aid with the characterization of the IIV attack surface of MCSs (Figure 1). While it still requires some manual effort such as identifying the right APIs (for spoofing network requests) or the right UI elements (for dynamic execution method), our framework can significantly speed up the analysis process compared to a completely manual analysis because it supports range and constraint, and semantic *input exploration strategies*, which drive a set of *input injection methods*. These methods are designed to quickly identify the attack surface of a service by fast repeated injections to the service compared to a completely manual attacker who has to navigate the whole application UI by hand for each injected value. Moreover, since we only have blackbox access to the services, we need a way to verify the success of an injection attempt. To address this, we introduce in our framework *feedback monitoring mechanisms*. Next we elaborate on each of the framework's main components.

3.2 Input Exploration Strategies.

Our framework can leverage a wide range of input exploration strategies. These provide the inputs to be used by the injection methods. These strategies are also dynamically informed through feedback monitoring. For our analyses we devised and implemented three types of strategies supporting range and constraint, and semantic analysis: *numeric value exploration* strategies, a *GPS coordinates exploration* strategies and a *social post generation strategies*.

Numeric Value Exploration (NVE) Strategies. Exploring numeric values, (these can correspond to distance, speed, prices, etc.) is tedious when performed manually, and in some cases temporally prohibitive. For example, the range of 32-bit integer values is $-(2^{31}-1) \leq x \leq 2^{31}-1$. A brute force approach is also impractical. Firstly, most services, to deal with denial of service attacks and to reduce the load on the server-side, rate limit the requests made by clients. For example *Strava* (see Section 4) only allows 50 `POST` requests per account per day. Some of them (see Section 5) even blacklist offending clients. Secondly, some experiments (see Section 8) require dynamic execution and interaction with the target apk. A single injection experiment on *Transit* (see Section 6) takes 15 minutes. Thus, performing all the trials is not efficient. To address this we devise a simple yet efficient strategy to explore the domain range of numeric (integer and float) value injections.

NVE begins with a geometric growth approach (doubling) starting from an initial value aiming to identify the first value that results in attack failure. It then switches to a linear strategy starting from the last successful injection until another unsuccessful injection is encountered. More formally, during geometric growth, each value is calculated as $x_i = x_{i-1} \times 2$, where x_i is the value to be tried at time interval i. Then, during the linear growth, each value is calculated as $x_i = x_{i-1} \pm s$, where s is the step size and $(\pm)$ dictates the direction of exploration. The value of s is set to the minimum positive value for integers (i.e. 1) and for floats is set according to semantics. For example, for prices, it is set to represent 1 cent or (0.01). This approach has clear benefits. For example, in a scenario where the positive integer injection success boundary is at 140, linear exploration requires 140 injections. NVE's hybrid approach requires 20 (an 85.7% percentage reduction): 8 steps to find the first failure value $256 = 2^8$; and another 12 linear steps $(140 - (2^{8-1}))$.

GPS Coordinates Exploration (CE) Strategies. Geo-sensitive services crowdsource GPS coordinates for a point or event of interest (PoI). GPS coordinates can be derived from a pair of angular measurements, known as longitude and latitude. Longitude represents the east-west geo-position of a PoI on the surface of Earth. Latitude represents the north-south geo-position of a PoI on the surface of Earth. Both angular measurements can be measured in `degrees`; longitudes range from $-180°$ to $+180°$; latitudes range from $-90°$ to $+90°$. CE encompasses four approaches to explore the geo-location ranges for a reported PoI.

• *CE-O: Out of range.* CE-O uses *numeric value exploration* to explore longitude and latitude values outside of their expected range. It uses each of the 4 boundaries of longitude and latitude as initial values and explores in the opposite direction of the expected range. CE-O is configured with $s = 1$ and explores degrees as integers.

• *CE-Long: Longitude exploration.* CE-Long fixes the latitude and explores the longitude range $(-180° \leq x \leq +180°)$, increasing linearly in the positive and negative integer direction.

• *CE-Lat: Latitude exploration.* CE-Lat fixes the longitude and explores the latitude range $(-90° \leq x \leq +90°)$, increasing linearly in the positive and negative integer direction .

• *CE-2D: Lat/Long Range exploration.* CE-2D also uses linear exploration but this time to create PoIs over the whole 2D range of longitudes (-180 to 180) and latitudes (-90 to 90). CE-2D uses a step of size $s = 5$. Note that 2D does not need to use the hybrid geometric-linear NVE approach since it is temporally feasible to explore the entire 2D range using fixed increments.

• *CE-Prec: Precision exploration.* This is used to explore the adversary's capabilities at the precision of the fractional part of the longitude/latitude float values. First, we trigger search APIs on the target services to acquire existing PoIs and from those we identify the maximum number of decimal places the service returns for reporting the location of its PoIs. To explore how close two PoIs can be introduced, CE-Prec chooses a starting point (e.g. (1.0, 1.0))

and explores all combinations of values (0–9) for the rightmost decimal place for longitude and latitude for a total of 100 injections. If a failure follows a successful injection (too close to the previous injection), we reduce the number of decimal points and repeat.

Social Post Generation Strategies. Some services allow their participants to share semantically meaningful information (see Section 8). This information is communicated in the form of posts which can support both text and images. To better understand the extend of the effectiveness of the semantic validation employed by such services we devise three fake post generation strategies. These are designed to explore a target service's level of semantic validation: no validation; general natural language understanding; natural understanding relevant to the service.

• *Random Sentence Generation (RSG).* The first strategy aims to explore whether the service accepts any text input without any semantic validation. To verify that, we generate sentences made of words randomly sampled from the English dictionary. The sentence length can be determined by computing the average number of words in genuine posts collected from the target service.

• *Sentence Generation with Pre-trained GPT-2 (SGP).* The second strategy tests posts of semantically similar topics as the extracted posts and a category label under which they are published or organised on the service. To do so, SGP leverages the pre-trained model GPT-2, a state-of-the-art transformer-based model [43] to generate the text. The model takes keywords as prompts that it will then complete given its language model. The keywords are selected as the most common words in a set of genuine posts for each category. Those keywords also become the title of the posts. For the ones that get rejected by the service, SGP uses them again but this time it augments the text with an image. In the first attempt, SGP uses an image irrelevant to all categories. It then uses the rejected posts a second time but this time it augments the generated text with a semantically relevant image. To select relevant images, SGP extracts the most significant entity from our fake text using Google Cloud Natural Language API [20] and searches for an image with the entity as a keyword using "Google Images Download" [18]. It then selects the first search result as the image to use in the post. Lastly, SGP uses text+image-accepted posts again but this time augmented with an image that is irrelevant to all categories.

• *Sentence Generation with Adapted GPT-2 (SGA).* The third strategy tests more relevant posts generated by fine-tuning the model [19] on the set of collected genuine posts. The prompts given to the model now become the category of posts that are to be generated. Similarly with the SGP approach, SGA repeats the generated posts with and without the irrelevant and relevant image.

3.3 Input Injection Methods

MCS companion apps, collect data through *sensors* and from their users inputting data in the apps' *UI*. This data is then communicated to the remote services through *Internet-exposed APIs*. We implement three methods each targeting in aiding the IIV analysis through each of those interfaces: the *Sensor Spoofing Method* facilitates sensor input analysis; the *Dynamic Execution Method* facilitates UI input analysis; and the *Network Request Spoofing Method* which facilitates the analysis of inputs directly through the network.

Spoofing Network Requests. For some apps, we need to analyze their network traffic to extract target requests that can be spoofed for injecting data to the MCS or to monitor its response. In all cases, the communication between the app and the service is encrypted, hence common tools such as Wireshark will not help. To address this we use a man-in-the-middle (MITM) proxy and install its CA certificate on a mobile device. Normally this would have had been enough to enable observing the target apps traffic in plaintext but some apps are using mitigation strategies against MITM attempts and in particular a technique called certificate pinning. This configures the client app to accept connections only with the legitimate server. Thus the app under analysis would reject the proxy certificate. To mitigate this, we use dynamic instrumentation to target and overload the SSL context initialization function (`SSLContext.init`) of the target app at runtime so that it uses the proxy's certificate and effectively bypassing the app's certificate pinning. Our current implementation uses an extended version of the Android Frida framework [15] to facilitate the dynamic instrumentation. With this setup, we can now run the target app and monitor the network requests it makes and the service's responses. All networking information is logged. This allowed us to reverse engineer the network-exposed APIs of the target services, which are filtered to select the ones to be targeted for injection and response monitoring. Lastly, we developed a network request spoofer, which spoofs network requests to the given APIs, emulating the mobile device, target app, and a user of the service.

Dynamic Execution Method. In some cases, injection experiments require dynamic execution of the apps. Performing this manually for a large number of input trials might be prohibitively cumbersome. To address this we use a dynamic execution (DEM) method powered by record and replay tools to facilitate UI navigation and interaction. In particular, we employ tools that can monitor unique IDs of Android app UI elements the user interacts with during recording. When an app's layout is dynamically rendered, the DEM recorder parses the hierarchy tree of the layout and with the help of the analyst identifies the IDs corresponding to the UI elements of interest. Note that it is not mandatory for UI elements to have IDs or even if they do they are not necessarily unique. To address this, the DEM recorder also records auxiliary information related to the UI element of interest such as its position within the hierarchy tree, the element's class, and any textual information (e.g. strings associated with text labels of the element) semantically describing the element. During replay, a DEM replay component leverages the android debug bridge to install and launch the app under analysis on an emulator or real device. It then uses the recorded UI elements' information and their logical order of execution to generate UI interaction commands sent through `adb` to the app. Like before, if there is a conflict of IDs or absence of them, the DEM replay looks for semantic hints using regular expressions or exact matching on strings, class names, and position within the hierarchy tree to determine which element to emulate an interaction with. Our current implementation of the tools employed in the DEM method use extended versions of the Android UIAutomator [5] during recording and the Appium framework [14] during replay.

Spoofing Sensors. In other cases, we have to spoof sensor measurements. For example, the transportation service *Transit*, tracks

bus rides, during which it continuously reports the GPS coordinates of the host device. Manual attacks [45] are possible, yet not scalable. To address this, we develop a sensor spoofing module that can be configured to provide fake GPS measurements to a target device. The module uses the Genymotion shell connected to execute commands on a Genymotion Android device emulator [9]. We implement a *scheduler* which uses the command *gps setstatus enabled* to enable GPS readings from the shell. Then it invokes the sensor spoofing module which uses the *gps setlatitude* and *gps setlongitude* commands to update the GPS coordinates of the emulator according to a time series of GPS values provided by the scheduler. The scheduler generates the time series based on different speeds of movement we want to target (see Section 6).

3.4 Feedback Monitoring.

Since we only have blackbox access to the services, we need a way to verify the success of an injection attempt. To address this, our framework uses *feedback monitoring mechanisms*. We leverage the fact that most of these services need to provide real-time feedback to their participants through their mobile apps. Thus, on every input injection attempt, we take feedback from the service through (a) secondary spoofed network requests, (b) through the UI of the primary injection device—adversary's simulated vantage point, or (c) through secondary observer devices registered to the service with a different account—safely simulating victim participants. When feasible, feedback is observed manually. Otherwise, it is facilitated by our *DEM recorder* and our MITM tools.

3.5 Analysis Methodology

To better understand the extend of IIV vulnerabilities we focus our analysis on selected high-profile MCSs spanning various application domains. We regard a MCS to be high-profile if it has a wide userbase and/or it is developed by mature developers/companies. We choose such MCSs since if they exhibit any IIV vulnerabilities these would have the potential to affect a large number of users. At the same time, we expect such MCSs to have a higher responsibility and the resources to deploy security measures.

To identify representative cases, we searched for relevant apps using crowdsourcing related keywords on online search engines. We augment the results by crawling 3259 top free apps from all Google Play categories. This yielded a total of 3295 relevant apps. Two researchers then manually and independently rated each app as either "mobile crowdsourcing" (n=112) app or "other", resolving conflicts with a discussion. Subsequently, the relevant apps were categorized based on their application domain. We identified five domains: Fitness Activity Services, Pricing Services, Transportation Services, Location-Based Services and Safety Services. Then, we cherrypicked at least one representative and previously unexplored high-profile case for our analysis, resulting in a list of 10 apps (Table 6 in Appendix B). For each of the cases, we leverage our framework to characterize their IIV attack surface. Sections 4, 5, 6, 7 and 8 present our characterization experiments and results on Fitness Activity Services, Pricing Services, Transportation Services, Location-Based Services and Safety Services respectively.

4 FITNESS ACTIVITY SERVICES

Several MCSs collect fitness information from their participants for better health and wellness insights. For example, Strava Labs [8] leverages the large user base of Strava, a service that tracks exercise activities. Moreover, the information collected is used to enable social features such as competing with other participants on local and global challenges. Winners are incentivized with small motivating rewards. Here we explore how an adversary can poison the data collected by such popular services, namely, *Strava* (Section 4), *Fitbit* (Appx. A) and *Map My Run* (Appx. B) and fake a number of activities, including running, swimming, and cycling, with superhuman performance in terms of distance covered and speed achieved, allowing them to win challenges and rewards. All experiments below are launched through *spoofed networked requests*.

Strava Overview. *Strava* [7] allows its users to report a number of physical activities, such as running, cycling, and swimming. Its Android app is installed by more than 10,000,000 users. For each of the activities supported, users can report the date and time duration of the activity and distance covered among others. Using a fake account, we were able to fake a running activity covering 50,000 km in 3.5 hours which corresponds to a speed of 14,285 km/h . This constitutes a 98.4% increase on the world record for the fastest aircraft—7,200km/h . To better understand the extent of these attacks on Strava we design a set of systematic experiments.

Experiment Design. Using our MITM proxy testbed (see Section 3) we observed that the Strava app uses a `POST` request to submit a new activity to the remote service. The request is bundled with the activity data in JSON format which can include the activity date, duration, distance covered, and a description. We run our experiments by spoofing network requests from a fake account ID which is created when we create a fake athlete's profile. In each trial we use the *numeric value exploration strategy* (3.2) to check the range of successful injection attacks in terms of *distance* values and *duration* values. Both values are integers. We select the initial value in the exploration to be 0. To study the effect of different types of activities in the input validation we repeat the experiment for 3 types of activities: running, cycling, and swimming.

To detect whether an injection is successful, we leverage another network API. We observe that *Strava* responds to a `GET` request, with the stored statistics of a specific athlete. Thus, issuing this request using our fake athlete's account ID, allow us to check whether the previous injected value was accepted and stored in our profile by the service. In doing the experiments we found that *Strava* only accepts 50 posts by an athlete per day, irrespective of the activity posted. After that, it ignores all posts. To overcome this, we spread our experiments across multiple days.

Results. Firstly we observe that all three exercise types share the same input domain range boundaries namely (0 to 31,622,400 sec. and 0 to 50,000,000 meters)— at least they do not accept negative values. Unequivocally, these boundaries for duration and distance allow for implausible values. The maximum boundary for the duration (31,622,400 s) corresponds to a run activity which "took" 8784 hours or 1 year to complete. Considering that the equatorial circumference of the Earth is 40,075 km, a maximum distance of 50,000 km would correspond to running around the Earth 1.25 times. In terms

of the maximum distance an athlete can accumulate, we found this to have *no input range restrictions* as *Strava* accepts 4,294,967,295 meters, which is equivalent to the maximum value an unsigned 32-bit integer can have (2^{32}).

An adversary can select values from this range that look plausible but still unequivocally perform better than the top athletes to fake activities and finish first on any challenge leaderboard and claim the rewards (which have monetary value sometimes) and fame that come with it. Moreover, health insurance companies increasingly leverage individuals' health and fitness information to decide the risk score and thus the premium to charge potential and existing clients. IIV vulnerabilities can be exploited to artificially inflate physical activities to manipulate those predictions and thus incur costly damages to insurance companies.

Other Fitness Services. We conducted experiments to study the resilience of other popular fitness services, specifically *Fitbit* [6] and *Run with Map My Run* [3]. For *Fitbit*, we found that activity distance can be added from 1km to 1609.344km . For activity duration, we observe that there is no input data type positive range restriction as we could inject duration from 1 second to 2, 147, 483, 647 (which is the maximum positive value for a 32-bit signed binary integer, or $2^{31} - 1$). For *Run with Map My Run* alarmingly we found that any input validation happens on the client-side while the values are stored non validated (we injected arbitrarily large duration values) on the server. Due to space limitations, we defer further details to Appendix A.

5 PRICING SERVICES

Pricing crowdsourcing services, allow their participants to report the exact price value of an asset. Tampering with these prices can lead to an unfair competition where customers are driven away from target stores or directed toward particular stores. In this section we elaborate on a practical attack we launched against a popular pricing crowdsourcing service (*Basket Savings*). The attack is launched through *spoofed network requests*.

Manual Analysis Description and Objectives. *Basket Savings* [13] is an example of a money-saving service which crowdsources prices of grocery items on superstores. Its Android mobile app was downloaded more than 100,000 times. The app allows users to add a price (float value) for an item only if they are within a GPS-determined geo-location close to the target store and by scanning the bar-code of the target product or their purchase receipt. Moreover, the *Basket Savings* service blocks devices by IP in case they detect suspicious activity like a user making malicious price changes on the app. Nonetheless, we found that one can bypass this by leveraging a feature on the app which allows users to manually input and submit prices through the app's user interface. We suspect that this feature was added to increase crowdsourcing opportunities, for example by supporting price reports when the user has left the store. We verified that an adversary can use an emulator device to inject both lower and higher than the real price values for selected target products. For example, we verified that we could increase the price of one gallon of milk by 129% of its usual price (from $3.49 to $8). As with the case of Transit, we monitor feedback to verify the results by accessing the values from secondary passive devices registered as users of the service. This, can be leveraged by an adversary to launch an unfair competition attack where customers are driven away from target stores or directed toward particular stores by manipulating the advertised prices for popular products.

Table 1: Basket: Trader Joe's & Amazon Prime(*)

Product	Value	Min	Max	*Value	*Min	*Max
Apples	0.49	0.05	2.0	1.58	0.16	4.0
Bananas	0.19	0.09	2.0	0.55	0.06	2.0
Strawberries	0.99	0.09	2.0	5.0	2.21	8.3
Eggs	1.99	0.2	4.0	2.12	0.21	6.0
Chicken Breasts	2.69	0.27	6.0	3.25	0.33	8.0
Organic whole Milk	3.49	0.35	8.0	3.76	0.38	8.0

Table 2: Basket: Milk on Trader Joe's

Product	Gallons	Value	Min	Max
Whole Milk 1	0.5	1.29	0.13	4.0
Whole Milk 2	0.5	2.29	0.23	6.0
Organic Whole Milk 1	0.5	2.99	0.30	6.0
Organic Whole Milk 2	1	5.69	1.71	10.58
Homogenized Whole Milk	1	5.99	1.80	6.59

Experiment Design. The case above demonstrates that *Basket Savings* is vulnerable to improper input injections. Next, we design a set of experiments to characterize the IIV attack surface of the service. Prices in Basket are reported as float values. To find the input domain range accepted by the service—and thus the range of the adversary—we perform price injection attacks on the user interface of its mobile app, aiming to identify the boundaries (minimum and maximum values) the adversary can inject. Trying all possible values for an input type can be very time-consuming. In this case, a float data type has a range $-3.4E+38 \leq 3.4E+38$. Therefore to find the accepted input range in a more efficient manner, we follow the numeric value exploration strategy outlined in Subsection 3.2. We choose the initial value to be the current value of a target product. We also choose the step size to be equivalent to 1 cent ($s = 0.01$).

To perform the injections, we leveraged our MITM-proxy testbed again to obtain the API call responsible for injecting a new price value for any given store. We observed that the API call uses the new price, product and store-id along with a longitude, latitude pair and a timestamp to submit a new price value for the given product at the given store to Basket's system. We wrote a script that could replay this API call to Basket's server and used it manually for injecting different prices for any product according to the aforementioned numeric exploration approach.

To detect the success or failure of an injection trial, we used another API call discovered through our MITM-proxy testbed. Using a secondary passive device we manually verify the injected value was visible for other participants and has replaced the prior price for the given item on a particular store. We used this injection and observation approach to verify the *minimum and maximum value* possible for "bananas" and "strawberries" at the two stores. Later, we discovered another API call that had these maximum and minimum allowed price values i.e. the acceptable price range embedded in its response. We used this request to verify and obtain the price ranges quoted in the results for this app.

To examine the effect of *location*, we repeat this experiment for two stores (*Amazon Pantry* and a *Trader Joe's* store in Los Angeles, CA). To examine the effect of *product type*, for each store we try manipulating prices for 6 different products: apples, bananas strawberries, eggs, chicken breast, and organic while milk. Lastly, to examine the variation within product types we select 5 different kinds of organic milk.

Results. Table 1 summarizes the range of successfully injected prices for Trader Joe's and Amazon Prime. We observe that the minimum value allowed for both stores is mostly the 10% of the current value and the maximum value allowed for any product mostly seems to be $2 * \lceil currentValue \rceil$). These boundaries are clearly not realistic. Note that only bananas on Trader Joe's and strawberries on Amazon show exception to these rules in the results. Table 2 shows the successfully injected prices for 5 different kinds of organic milk on Trader Joe's (these are different from the one listed on Table 1). We observe that the minimum allowed price was 10% of the shown price for milk under $5 and roughly 30% of the shown value for milk over $5. The same rules as above were followed for 3 out of 5 kinds of milk for the maximum price.

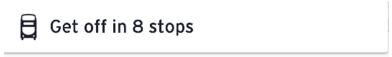
Other pricing services. For our analysis we also selected *GasBuddy* (Section 3). *GasBuddy* crowdsources real-time gas stations' fuel prices through their app. Like *GoogleMaps*, it uses client certificate pinning so we couldn't decrypt and reverse engineer the API calls used by the app. Another challenge was the variable nature of *GasBuddy*'s user interface where random pop-up screens (e.g. ads) would cause the app to crash or frequently end up at unintended screens, which currently our DEM module cannot handle.

6 TRANSPORTATION SERVICES

Transit [10] is a public transportation service for 175 cities. Its Android app enjoys over 5,000,000 installations. By crowdsourcing every passenger's real-time *sensory data* including positioning information and speed of movement, Transit can help its users plan a trip and support them in their travel by predicting the expected arrival time of the next subway or bus. However, we found that it is possible to fool the Transit service to accept fake measurements. To demonstrate this, we performed an experiment using 3 Android device emulators. The first device (E_α) acted as the adversarial device aiming to fool the service, while we used the other two as observer devices (E_{o1} and E_{o2}) for verifying the result of the attack on the service on other users' devices. We installed Transit on all three emulators and used the app to plan a bus trip from point A to point B in London, UK. E_{o1} and E_{o2} started their trip 3 bus stops later than E_α. E_α was driven by our sensor spoofer, which was automatically feeding fake GPS updates to the device, simulating a movement along the target bus route with a steady speed (12km/h). In this manner, Transit was instantly fooled to accept that E_α is actively riding the target bus, which we could verify as the Transit app rendered a new bus icon moving at E_α speed and direction, on the screen of all three devices. We were also successful in manipulating the expected bus arrival time for E_{o1} and E_{o2}, by faking the speed of movement of E_α to be 575mph which is equivalent to the average speed of a commercial plane. All experiments were performed at an off-peak time in a rural area to avoid affecting real users. Competing transit agencies can use these attacks to deter customers from using another transit service.

Experiment Design and Results. Next we design a set of experiments to better understand the range and predictability of values an adversary can leverage for manipulating bus routes. We leverage our DEM method (see Subsection 3.3) to dynamically install and execute the Transit app on Genymotion non-root emulators and move to a target screen for enabling active route navigation in the app. Then we use our sensor spoofing method to fake sensor GPS values to the victim app. All our experiments are performed targeting the same bus route in a rural area. We configure the attacker's device starting location to be 18km earlier on the bus route than the victim device's location (also on the bus route). The scheduler is configured to emulate the *speed of movement* of the adversarial device by generating GPS timeseries values corresponding to different frequencies. An injection attempt at a specific speed of movement is considered successful when it can affect the expectation of the bus arrival time on the observer device.

• *Linear value exploration.* We first use linear numeric exploration (see Subsection 3.2) to generate speed values from 0 to 1000 km/h with a step size of 10 km/h ($s = 10$). We found that 97/100 (97%) fake movements succeeded in fooling *Transit* that the adversarial device is actively riding a fake bus. We determine success by tracking visual hints on the UI of the app on the adversary's device: *Transit* shows a textual description to its user when waiting for a bus ("Waiting for the bus"). When the user is riding a bus this textual hint changes to "Get off in [number of] stops" as shown in Figure 2(a). Using our DEM module (see Section 2) we can track the target UI element with the textual hint to verify the success or failure of the experiment. However, we observe that even when the adversary succeeds to create a fake bus, this is not always reflected on other users' devices, especially when moving at high speeds. We verify this by randomly selecting 20 values of speeds and repeat the experiments for each of them, this time also monitoring the observer device. We can confirm that the fake bus also appears on the observer devices by looking for the bus icon with a happy rider face (see Figure 2(b)). The exact icon can be extracted beforehand as it is stored in the victim app's `res/drawable` folders which we access by decompiling the app using an Android reverse engineering tool (apktool [12]). This then used during the experiments with the `MatchTemplate()` function of the *openCV2* library [2] to search for that icon within a grayscale version of screenshots of the UI of the victim's device. Using this approach, we verify that 17/20 (85%) successful bus fakes also appear on the victim device.



(a) Text element hint on adversarial device's UI.

(b) Icon hint on victim device's UI.

Figure 2: Transit mobile app UI hints.

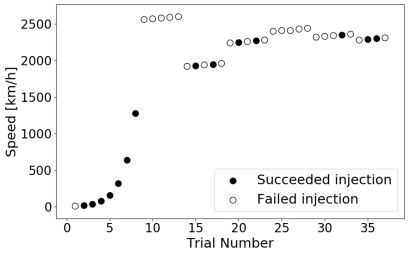


Figure 3: Transit: Faking Buses with Supersonic Speeds.

• *Supersonic Speeds.* In this experiment, we aim to find whether supersonic speeds (greater than the speed of sound—1235km/h) are possible. To explore this we use a variation of the numeric value exploration strategy. However, when a failure is encountered during geometric growth, before reverting back to the last successful value, we linearly ($s = 10$) try the next 4 values, and only if all 4 fail we finish exploring higher values. This is needed to deal with uncertainties at high speeds where we found the behavior of the service can be unpredictable. Specifically, our algorithm proceeds as follows: to find the highest acceptable speed S_h, the attack emulator starts at the speed 10 km/h and keeps doubling this value until the first failed attack speed S_i is observed on the victim phone. The failed attack is further confirmed with 4 more adversarial speeds : $S_i + 10$, $S_i + 20$, $S_i + 30$, $S_i + 40$. If more than 4 attacks fail out of 5 experiments, we regard S_i as the first failed attack speed and S_{i-1} as the last successful attack speed. Therefore, S_h is within the range $[S_{i-1}, S_i]$. Given $S_j = \frac{S_{i-1}+S_i}{2}$, range $[S_{i-1}, S_i]$ is divided into two ranges: $[S_{i-1}, S_j]$ and $[S_j, S_i]$. If S_j is able to attack the victim phone, S_h is within $[S_j, S_i]$, otherwise, $[S_{i-1}, S_j]$. By keeping on dividing the range, S_h is finally confirmed. As shown in Figure 3, we managed to succeed with supersonic speeds of up to 2350 km/h. We repeated the attack at this speed 10 times. We found that 3/10 (30%) of the times the injection at 2350 km/h influences the victim.

Other transportation services. For our analysis we also selected *GoogleMaps* (Section 3). Even though its susceptibility to general data poisoning attacks was established manually [45], we could not apply our framework to scale up the analysis. In particular, we could not decrypt its APIs calls due to the usage of not only server but also client certificates which our framework cannot currently bypass. However, our framework can be expanded with a farm of real phones and emulators to support the analysis of such cases.

7 LOCATION-BASED SERVICES

Services such as *Police Detector* and *ToiFi (Toilet Finder)*, crowdsource points of interests (PoIs). An adversary targeting such services might create or remove PoIs to their own benefit. For example, a fake police radar can deceive an individual into using another route; a fake toilet might be used to lure potential victims to deserted locales. Using *spoofed networked requests* we were able to verify and analyze IIV vulnerabilities for both *ToiFi* and *Police Detector*. Due to space limitations, we present our analysis on *ToiFi (Toilet)* in Appendix A.

Police Detector Overview. *Police detector (Speed Camera Radar)* [17] uses crowdsourcing to help users make intelligent decisions while driving. Its Android app was installed more than 5,000,000 times. Its users can report the location of police speed detectors, road repairs and road accidents. As with *ToiFi (Toilet Finder)* we were successful in faking all PoIs: fake police speed radars, fake road accidents and fake road repairs. As before, to better understand the susceptibility of the service to these attacks, we perform experiments using the *GPS coordinates exploration* strategies (see Subsection 3.2).

Experiment Design and Results. Using our MITM proxy setup, we reverse engineer the API of the service and identified an API call used to add a speed detector and a second API which allows searching for speed detectors. The second is useful for observing the success or failure of the injection. We successfully spoofed the service's mobile app's network requests using our framework. We found that values outside the expected longitude and latitude range (*CE-O*) are rejected. However, for the latitude and longitude experiments (*CE-Long* and *CE-Lat*), 48/180 (26.6%) and 55/360 (15.3%) of the injections succeeded respectively. Consecutive injections are blocked after a fixed number of requests (see red points on Figure 4), which led us to hypothesize that the points were not rejected based on semantic validation but instead were rejected due to a rate limit on the number of points a registered user can submit. To overcome this, we generate a pool of fake user fingerprints which we rotate through when performing the injections. We noticed that the user-id for a participant depends on the Android-id of her phone and relaunching the app after changing the Android-id of the emulator phone creates two requests with two different but corresponding ids for registration and retrieval of PoIs on the map. So, we wrote two scripts to automate this process, one of them employed `adb` to keep relaunching the app and the other script interacted with our MITM proxy to scan the requests and responses and extract the (registration-id, user-id) pairs out of them so that we can use them later for our experiments. We follow this setup to generate a pool of 86 fake users and conduct the *CE-2D* experiment where we do injections for the whole 2D range of latitudes and longitudes. Furthermore, we add a delay of 10 seconds between successive injections and repeat the whole sequence of requests that happen on launching the app for each injection, instead of just calling the injection API call. With this setup, we were able to perform successful injections for the entire 2D range of points as described in *CE-2D* (see Figure 4 (green points)). The only exceptions where the injection failed was when either the longitude or latitude was 0 or when the value of latitude was exactly on the boundaries -90 or 90. For *CE-Prec*, we saw that POIs can be inserted with a precision of up to 5 decimal places but no two POIs can be closer than 0.002 on either longitude or the latitude scale, which happens to be equivalent to around 222 meters in distance. To prevent any harm to the service or its users we identified another API request which we employ with an input value of 0 in the request body, to remove the added PoIs after each successful injection.

8 SAFETY SERVICES

A number of MCSs allow users to share safety-related information. One such service is *Neighbors by Ring* [16] (NbR), whose Android app has been downloaded over 1,000,000 times. This service allows its users to share four kinds of posts: crimes, safety-related events, lost pets, and unexpected activities. These can include text, images,

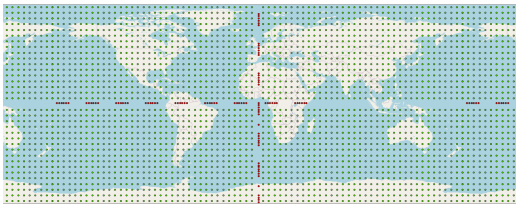


Figure 4: Police Detector: *CE-Long* & *CE-Lat* Successful Injections (•), *CE-2D* Successful Injections (•).

even video streams from security cameras, and share the reporting device's location. Fake reports can be used to spread chaos or defame a location or neighborhood. In a more sophisticated scenario, enemy states or organizations with political affiliations can deploy elaborate propaganda schemes.

Manual Attack. We submitted manually–constructed sample textual posts controlling for both the location and the semantics of the text. We found that NbR does not verify the location but does verify the semantics of the textual report. For example, our attempt to inject "dangerous cat spotted" was rejected, but our attempt to inject "dangerous dog spotted" was successful. Additionally, posts that were too vague about the safety issue were rejected as well. We suspect that NbR uses a machine learning algorithm to determine whether posts are legitimate or not in order to tackle fake posts.

Experiment Design. To further explore the service's semantic validation, we submit posts generated using our post generation strategies: random sentence generation (RSG), sentence generation with pre-trained GPT-2 (SGP), and sentence generation with adapted GPT-2 (SGA), and report the posts' acceptance rate. To configure the strategies' parameters, we first use our DEM module to execute the app, interact with it, and extract already present but unseen posts. We collected a dataset of 1080 genuine posts which follow the format: <Category: $category_p$, Title: $title_p$, Description: $description_p$ >. Using the genuine posts descriptions we determine the average sentence length (= 30) words for the RSG strategy. We also identify the three most common words present in the first sentence of each post's description, by category. These are used as the keywords and titles of the fake posts in the SGP strategy. For the adaptive approach (SGA), we fine-tuned the text generation model. The model was trained for 1000 epochs and with a learning rate of 10^{-4} and a temperature of 0.7 for the generation.

To test the generated posts, we set up three Android devices and submit them via the UI of the target service's Android app, using our DEM module. Since the app requires a unique email address to create an account, we create temporary emails for each device. We set the devices' location in Death Valley, California at different spots such to minimize exposure to real users. Once a post is submitted, our script updates the page for up to 8 minutes until the post appears in the user's submissions. We noticed that the time to get the decision varies between 1 to 7 minutes. If the post appears, it is marked as accepted and the script deletes the post. If the post is not accepted, the respective email address receives a rejection email and the post never appears in the submissions. We also found that the service blocks accounts that post too often and set a limit of 8 rejected posts a day. Once a user is blocked, their submissions are ignored without any notification. To overcome this problem, we submit at most 3 posts an hour and set random delays in the range of 20 to 35 minutes between posts. We also monitor the emails manually to check that the posts are not ignored.

Table 3: NbR: Fake Injection Success Rates.

Strategy	Image	Crime	Safety	Lost Pet	Unexpected Activity	Total
RSG	n/a	n/a	n/a	n/a	n/a	0/100
SGP	n/a	9/25	6/25	10/25	1/25	23/100
SGA	n/a	22/25	19/25	16/25	9/25	66/100
SGP	Irrelevant	9/25	6/25	10/25	1/25	23/100
SGA	Irrelevant	22/25	**20/25**	16/25	9/25	**67/100**
SGP	Relevant	9/25	6/25	**16/25**	**2/25**	**33/100**
SGA	Relevant	**23/25**	**20/25**	**25/25**	**12/25**	**80/100**

Results. Table 3 shows the number of accepted posts per category out of the total number of submitted fake posts for each of the generation strategies. Examples of successful injections are shown in Appendix B, Table 5. The results show that the app does not accept just any input text, as none of the random posts went through. Furthermore, some text-only posts generated with the *SGP* strategy are rejected even if they are of topics similar to the ones accepted by the service. Those posts were possibly too vague about safety issues and without a clearly defined structure. However, 23 out of 100 fake posts were indeed approved. Fake posts generated using the *SGA* strategy were more likely to get accepted as 66% of posts were able to replicate the format and or the necessary information required by the app. Moreover, the model generated crime related posts with almost 90% success rate. The "Unexpected Activity" category seems to be the hardest to imitate as it has the lowest acceptance for both strategies. When comparing fake and genuine posts, we found that "Unexpected Activity" posts often also include video footage captured with the Ring security camera.

Enriching posts with *irrelevant* images did not help except for one post that mentions a troop of policemen. We hypothesize that the blue color of the image could be mistakenly taken as their usually blue uniform. Nevertheless, this experiment show that there is semantic validation using the images. Indeed, posting the fake reports with *relevant* images improved the success rate for both the *SGP* and the *SGA* strategies and particularly for the category "Lost Pet". The rates also improve slightly for the other categories.

These results show that *Neighbors by Ring* does check for semantic soundness and also relevance to their categories. The app is also more lenient for "Lost Pet" posts, especially if given an image. However, for posts of type "Crime", "Safety" and "Unexpected Activity", the emphasis seems to be on the input text and the information it provides. Nevertheless, we show that using our fake post generation strategies, an adversary can generate multiple posts fulfilling these conditions and effectively perform successful injection attacks.

9 DISCUSSION ON COUNTERMEASURES

Majority voting, origin attestation, and reputation schemes can help alleviate improper input injections. However, majority voting depends on the availability of multiple sources of information at any given point in time which is not always true in services with real-time requirements. Origin validation approaches can limit an adversary's ability to scale up the attacks, but they are not effective

Table 4: Example countermeasures and the ensuing reduction in the affected attack surface. $e_1 = 0.2 * 350$ and $e_2 = 0.2 * 70$.

App Domain	App Type	Example Countermeasure	Function	Reduction
Strava	Fitness	Restrict running distance (d) to be at most the world record	$0 > d \leq 350m \pm e_1$	98.65%
Map My Run	Fitness	Restrict running distance (d) to be at most the world record	$0 > d \leq 350m \pm e_1$	99.58%
Fitbit	Fitness	Restrict running distance (d) to be at most the world record	$0 > d \leq 350m \pm e_1$	99.58%
Transit	Transportation	Enforce bus speed (v) according to highway code—70mph in UK Motorways	$0 > v \leq 70mph \pm e_2$	94.25%
Basket Savings	Pricing	Use auxiliary data sources to verify price	|aux_price - reported_price| < threshold	Varies
Police Detector	Location	Restrict distance ($d()$) between inserted location ($loc(i)$) and the nearest road segment to be within 10m	$d(loc(i), near(loc(i))) \leq 10m$.	99.89%
NbR	Safety	Use metrics based on user reputation	reputation(user) > threshold	Varies

when used in isolation. *Neighbors by Ring* uses an out-of-band channel for verifying a new user (i.e. email), *Police Detector* assigns user IDs based on unique Android IDs, and *Gas Buddy* and *Google Maps* use client certificates, but they are all bypassable either through technical means as we showed in our work or physical means [45]. On the other hand, reputation schemes have gained in popularity and found applications in other crowdsourcing domains (e.g. online reviews) but suffer from a cold start problem, which an adversary can leverage to inject only a few but high impact values before they are penalized. Input validation can be a great addition in our defense arsenal which can minimize the adversary's incentive during the cold start period but also throughout the lifetime of a participant account, while rendering the amount of reportings to be potentially checked more tractable. Such strategies are easy to implement, can be immediately deployed with a software update on the server side, and do not assume any capabilities on the participants' devices.

To demonstrate this, we show the resounding effect simple input validation functions can have (Table 4). For example, restricting running distances to the world record (350 miles) allowing for a 20% error—which is much higher than the 13m smartphone GPS empirical error [38]—it would constitute a 98.65%, 99.58% and 95.80% reduction of the maximum allowed value for *Strava*, *Map My Run* and *Fitbit* respectively. Constraining accepted bus speeds to the maximum speed limit allowed (in the UK buses speed limit is 70mph in motorways), allowing for a 20% error in estimation, it would constitute a 94.25% reduction in the speed allowed for *Transit*, which greatly restricts the effect an adversary can have on bus arrival times. Similarly, location-based services can geo-fence reports. To illustrate this we used the *CE-2D* strategy to generate 2701 values but filtered out any value that is not within a threshold distance from its nearest road segment. We found that only 3 and 33 of them would be accepted which constitutes a 99.89% and 98.78% reduction respectively. For pricing services, one can restrict inputs according to market price fluctuation. *Basket Savings* can leverage auxiliary data sources [32, 49] for this. Moreover, it could leverage majority voting or employ other verification mechanisms before displaying a value, since grocery item prices do not change frequently [41].

Even if such restrictions are present, a determined adversary can sometimes generate realistic inputs as we demonstrated in the case of *NbR*. Detecting fakes is an open and challenging problem. We used the GPT-2 Output Detector [46] which is specifically designed to detect inputs generated by the GPT-2 model. This resulted in a poor precision and recall of 55% and 53% respectively. While working toward improving our automated detection capability, visual hints about a reporting account's maturity and reputation can help users better judge the veracity of crowdsourced values. This is especially true when real-time distribution is paramount.

10 RELATED WORK

Prior work demonstrated data poisoning in crowdsourcing [25, 40, 51, 55, 56]. Our work focuses on services interfacing with their users through mobile apps. More related are [42, 53] but study only specific MCS domains. Polakis et al. [42] demonstrated that two location-based MCSs are vulnerable to fake check-ins. In contrast we observe a more general vulnerability of IIV and use our observation to design a broader study to characterize the exposure of MCS across domains to improper inputs. Proposed defenses include majority voting [37, 48], reputation systems [54, 58] or even trusted sensing [31]. However, these are not always applied in MCSs and even if they do their effect is limited when applied in isolation.

Related to our framework, prior works also perform analysis from the perspective of mobile apps but focus on IoT devices rather than MCSs [26, 27]. Others analyze the UI of mobile apps similar to our DEM approach [33, 34, 39, 59]. In contrast with those works, we do not aim to identify privacy leaks through the UI but instead automate navigation and value fuzzing. Zhao et al. [59] also focus on input validation but solely on the mobile app side rather than a remote web service, while others have studied how Android applications' network traffic can be intercepted [28, 30, 35, 47]. Furthermore, Zhao et al. [60] also reverse engineers API calls for Android apps but their work is mostly centered on data leakage vulnerabilities rather than discovering IIV vulnerabilities. We employ similar monitoring strategies which we augment with dynamic instrumentation for bypassing certificate pinning. Lastly, work is undergoing on fake text generation and detection [23, 29, 36, 43, 46, 56, 57]. Most of them are either not tested in practice or focus on social platforms rather than MCSs. In our work, we leverage such state of the art text generation strategies which we show how they can be combined with spoofed network requests embodied in end-to-end framework for analyzing real-world MCS services exposure to improper input injection attacks. Lastly, our framework bares similarities with model-based testing [24, 44, 44, 50] as it can be seen as a simplistic abstract model for MSCs behavior (the system under test). Building on this modeling is a promising future direction.

11 CONCLUSION

In this work, we developed a framework for analyzing improper input validation vulnerabilities of mobile crowdsourcing services. We successfully apply the framework on 8 high-profile services across 5 application domains and found that they are all severely exposed to improper input injection attacks. Our analysis showed that arbitrary inputs from fake accounts and devices are accepted as genuine, allowing an adversary to fake reports for robberies and gunshots in safety services, to fake fitness activities with supernatural performance, and to manipulate grocery items prices among others. Then, we discuss and showcase how simple to implement

and deploy input validation strategies can greatly reduce the IIV attack surface and complement existing defenses.

ACKNOWLEDGMENTS

This work was partially supported by the U.S. National Science Foundation (CNS-1956445).

REFERENCES

[1] 2000. Gas Buddy. https://www.gasbuddy.com.
[2] 2000. OpenCV. https://opencv.org/.
[3] 2007. Map My Run. https://www.mapmyrun.com/.
[4] 2008. Google Maps. https://www.google.com/maps.
[5] 2008. UI Automator. https://developer.android.com/training/testing/ui-automator.html.
[6] 2009. Fitbit. https://www.fitbit.com.
[7] 2009. Strava. https://www.strava.com/.
[8] 2009. Strava Labs. https://labs.strava.com/.
[9] 2011. Genymotion Android Emulator. https://www.genymotion.com/.
[10] 2012. Transit. https://transitapp.com/.
[11] 2015. ToiFi. https://play.google.com/store/apps/details?id=com.apprevelations.indiantoiletfinder.
[12] 2016. Apktool. https://ibotpeaches.github.io/Apktool/.
[13] 2016. Basket. http://basket.com/.
[14] 2017. Appium. http://appium.io/.
[15] 2017. Frida. https://frida.re/.
[16] 2018. Neighbors App by Ring. https://store.ring.com/neighbors.
[17] 2018. Police Detector (Speed Camera Radar). https://play.google.com/store/apps/details?id=tat.example.ildar.seer&hl=en_GB.
[18] 2019. Google Images Download — Google Images Download documentation. https://google-images-download.readthedocs.io/en/latest/index.html.
[19] 2019. minimaxir/gpt-2-simple. https://github.com/minimaxir/gpt-2-simple.
[20] 2020. Google Cloud Natural Language. https://cloud.google.com/natural-language/.
[21] 2020. You VS the Year 2020 | MapMyFitness. https://www.mapmyrun.com/challenges/yvsty2020/register.
[22] 2021. Project Website. https://sites.google.com/view/data-poisoning-mcs.
[23] David Ifeoluwa Adelani, Haotian Mai, Fuming Fang, Huy H Nguyen, Junichi Yamagishi, and Isao Echizen. 2020. Generating sentiment-preserving fake online reviews using neural language models and their human-and machine-based detection. In *International Conference on Advanced Information Networking and Applications*. Springer, 1341–1354.
[24] Josip Bozic and Franz Wotawa. 2012. Model-based testing-from safety to security. In *Proceedings of the 9th Workshop on Systems Testing and Validation (STV'12)*. 9–16.
[25] Bogdan Carbunar and Rahul Potharaju. 2012. You unlocked the mt. everest badge on foursquare! countering location fraud in geosocial networks. In *2012 IEEE 9th International Conference on Mobile Ad-Hoc and Sensor Systems (MASS 2012)*. IEEE, 182–190.
[26] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing.. In *NDSS*.
[27] Soteris Demetriou, Nan Zhang, Yeonjoon Lee, XiaoFeng Wang, Carl A Gunter, Xiaoyong Zhou, and Michael Grace. 2017. HanGuard: SDN-driven protection of smart home WiFi devices from malicious mobile apps. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 122–133.
[28] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 50–61.
[29] Sebastian Gehrmann, Hendrik Strobelt, and Alexander M Rush. 2019. Gltr: Statistical detection and visualization of generated text. *arXiv preprint arXiv:1906.04043* (2019).
[30] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 38–49.
[31] Peter Gilbert, Landon P Cox, Jaeyeon Jung, and David Wetherall. 2010. Toward trustworthy mobile sensing. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*. 31–36.
[32] gov.uk. 2020. United Kingdom milk prices and composition of milk statistics notice (data for April 2020). https://www.gov.uk/government/publications/uk-milk-prices-and-composition-of-milk/united-kingdom-milk-prices-and-composition-of-milk-statistics-notice-data-for-june-2019.
[33] Yuyu He, Lei Zhang, Zhemin Yang, Yinzhi Cao, Keke Lian, Shuai Li, Wei Yang, Zhibo Zhang, Min Yang, Yuan Zhang, et al. 2020. TextExerciser: Feedback-driven Text Input Exercising for Android Applications. In *2020 IEEE Symposium on Security and Privacy*. IEEE.
[34] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. {SUPOR}: Precise and Scalable Sensitive User Input Detection for Android Apps. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 977–992.
[35] J. Hubbard, K. Weimer, and Y. Chen. 2014. A study of SSL Proxy attacks on Android and iOS mobile applications. In *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*. 86–91.
[36] Mika Juuti, Bo Sun, Tatsuya Mori, and N Asokan. 2018. Stay on-topic: Generating context-specific fake restaurant reviews. In *European Symposium on Research in Computer Security*. Springer, 132–151.
[37] Hongwei Li and Bin Yu. 2014. Error rate bounds and iterative weighted majority voting for crowdsourcing. *arXiv preprint arXiv:1411.4086* (2014).
[38] Krista Merry and Pete Bettinger. 2019. Smartphone GPS accuracy study in an urban environment. *PloS one* 14, 7 (2019).
[39] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. 2015. Uipicker: User-input privacy identification in mobile applications. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 993–1008.
[40] Victor Naroditskiy, Nicholas R Jennings, Pascal Van Hentenryck, and Manuel Cebrian. 2013. Crowdsourcing dilemma. *arXiv preprint arXiv:1304.3548* (2013).
[41] Martin Pesendorfer. 2002. Retail sales: A study of pricing behavior in supermarkets. *The Journal of Business* 75, 1 (2002), 33–66.
[42] Iasonas Polakis, Stamatis Volanis, Elias Athanasopoulos, and Evangelos P Markatos. 2013. The man who was there: validating check-ins in location-based services. In *Proceedings of the 29th Annual Computer Security Applications Conference*. 19–28.
[43] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI Blog* 1, 8 (2019), 9.
[44] Ina Schieferdecker. 2012. Model-Based Fuzz Testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 814–814.
[45] Weckert Simon. 2020. Google Maps Hacks. http://www.simonweckert.com/googlemapshacks.html.
[46] Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, and Jasmine Wang. 2019. Release strategies and the social impacts of language models. *arXiv preprint arXiv:1908.09203* (2019).
[47] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. 2014. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14*. Citeseer.
[48] Dapeng Tao, Jun Cheng, Zhengtao Yu, Kun Yue, and Lizhen Wang. 2018. Domain-weighted majority voting for crowdsourcing. *IEEE transactions on neural networks and learning systems* 30, 1 (2018), 163–174.
[49] usda.gov. 2019. Price Spreads from Farm to Consumer. https://www.ers.usda.gov/data-products/price-spreads-from-farm-to-consumer/.
[50] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. *Software testing, verification and reliability* 22, 5 (2012), 297–312.
[51] Gang Wang, Bolun Wang, Tianyi Wang, Ana Nika, Bingzhe Liu, Haitao Zheng, and Ben Y Zhao. 2015. Attacks and defenses in crowdsourced mapping services. *CoRR, abs/1508.00837* (2015).
[52] Gang Wang, Bolun Wang, Tianyi Wang, Ana Nika, Haitao Zheng, and Ben Y Zhao. 2016. Defending against sybil devices in crowdsourced mapping services. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. 179–191.
[53] Gang Wang, Bolun Wang, Tianyi Wang, Ana Nika, Haitao Zheng, and Ben Y Zhao. 2018. Ghost riders: Sybil attacks on crowdsourced mobile mapping services. *IEEE/ACM transactions on networking* 26, 3 (2018), 1123–1136.
[54] Kun Wang, Xin Qi, Lei Shu, Der-jiunn Deng, and Joel JPC Rodrigues. 2016. Toward trustworthy crowdsourcing in the social internet of things. *IEEE Wireless Communications* 23, 5 (2016), 30–36.
[55] Kan Yang, Kuan Zhang, Ju Ren, and Xuemin Shen. 2015. Security and privacy in mobile crowdsourcing networks: challenges and opportunities. *IEEE communications magazine* 53, 8 (2015), 75–81.
[56] Yuanshun Yao, Bimal Viswanath, Jenna Cryan, Haitao Zheng, and Ben Y Zhao. 2017. Automated crowdturfing attacks and defenses in online review systems. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1143–1158.
[57] Rowan Zellers, Ari Holtzman, Hannah Rashkin, Yonatan Bisk, Ali Farhadi, Franziska Roesner, and Yejin Choi. 2019. Defending against neural fake news. In *Advances in Neural Information Processing Systems*. 9051–9062.
[58] Rui Zhang, Jinxue Zhang, Yanchao Zhang, and Chi Zhang. 2013. Secure crowdsourcing-based cooperative pectrum sensing. In *2013 Proceedings IEEE INFOCOM*. IEEE, 2526–2534.

[59] Qingchuan Zhao, Chaoshun Zuo, Dolan-Gavitt Brendan, Giancarlo Pellegrino, and Zhiqiang Lin. 2020. Automatic Uncovering of Hidden Behaviors From Input Validation in Mobile Apps. In *2020 IEEE Symposium on Security and Privacy*. IEEE.
[60] Qingchuan Zhao, Chaoshun Zuo, Giancarlo Pellegrino, and Li Zhiqiang. 2019. Geo-locating Drivers: A Study of Sensitive Data Leakage in Ride-Hailing Services.. In *Annual Network and Distributed System Security symposium, February 2019 (NDSS 2019)*.

A OTHER SERVICES

Fitbit [6] is a fitness service. Its Android app is installed more than 10,000,000 times. It allows its users to record walks, hikes and runs, share their achievements with others, participate in challenges and earn badges as rewards for their activities. An adversary faking activities can earn badges, win challenges and rewards. To demonstrate Fitbit's susceptibility to improper input injections we spoof network requests to the Fitbit service. We were successful in posting activities spoofing a real device and user account. To verify the injection, we use the response of the spoofed requests and we also verify manually on a real device in parallel for some randomly chosen requests. To characterize the IIV attack surface of Fitbit we follow the *NVE* approach (see Section 3), to study the successful injection boundaries for an activity's distance and duration. We repeat for three activities: walk, hike and run. We found that no negative or zero values for distance or duration are allowed. Distance can be added from 1km to 1609.344km (equivalent to 10,000 miles). For duration we observe that there is no input data type positive range restriction as we could inject durations from 1 second to 2,147,483,647 (which is the maximum positive value for a 32-bit signed binary integer, or $2^{31} - 1$).

Run with Map My Run [3] is a fitness tracking service whose Android app is also installed more than 10,000,000 times. Like other fitness services, it allows its users to track and share activities, participate in challenges, and earn rewards. Its challenges can be sponsored and result in real in-kind rewards. For example, the *You VS the Year 2020* challenge asks users to *"Cover 1,020KM in 2020 and be eligible for exclusive prizes from Under Armour"* [21]. Like before, we were able to reverse engineer the service's API and successfully spoof network requests to the service, which appear to come from a real device and user, and observe the result of the injection. To characterize the adversary's reach, we perform experiments on the distance and duration of a running activity using the *NVE* strategy.

We found that the min duration that can be added is 0. For the max value, we found that the remote service accepts an arbitrarily large value for the duration, but this is always rounded within the constraints of a single day. That is, if value x in seconds is stored on the server-side, the client always renders $x \mod 86400$. This is alarming because any input validation happens on the client-side while the values are stored non validated on the server. Thus, even though we do not know whether the stored values are sanitized or not on the server-side, this would require a lot of ad-hoc validation checks whenever the value is to be used, which is a bad prone to errors design. In terms of distance, we found that the min distance accepted is 0. However, the max allowed distance for an activity can be arbitrarily large. We were able to increase it up to 100051.4 miles, which is approximately 4 times the earth's perimeter.

ToiFi (Toilet Finder) [11] is a location-based MCS which allows users to find public restrooms (as PoIs) in time of need. It allows participants to add new or edit/remove previous PoIs. Its Android app was installed more than 50,000 times. We were successful in both manually adding and removing PoIs at arbitrary locations. To systematically characterize the attack surface of the service next we use the *GPS coordinates exploration* strategy to discover the range of coordinates that can be used to fake a public toilet. To conduct the experiments we identified—through our MITM proxy—the API the remote service exposes for receiving a request to add a PoI. We then carefully craft network requests triggering the API, by spoofing the mobile app and a service user. Success of the injection is determined by spoofing a second request targeting a different exposed API which allowed us to search for the presence of a PoI. We found that out of bound longitudes and latitude values (CE-O) are rejected. However, all trials on *CE-Long*, *CE-Lat* and *CE-2D* were successful allowing us to inject PoIs anywhere on the surface of the Earth. For *CE-Prec*, we saw that we can inject POIs up to a precision of 7 decimal places for both longitude and latitude. Furthermore, we noticed that there is no check preventing two points from being too close on the map as the first round of injections went fine as it is. Evidently, this can be exploited for fun, for example by injecting toilets in the middle of the desert or in the middle of the ocean. It can also be used for pranks as public toilets can be injected on a target individual's or enterprise's private premises; or even for harm, as people can be lured in dark alleys and isolated areas. We marked all PoIs added by our experiments as non existent by spoofing another API call we discovered so as to not harm the service or its users.

B OTHER DETAILS

Table 5 lists some representative examples of fake posts we generated with the SGP and SGA models, and accepted by *NbR*. Table 6 lists the number of successful injections per service and targeted interface performed throughout our experiments.

Table 5: Examples of accepted fake text posts by NbR.

Model	Category	Fake Title	Fake Description
SGP	Safety	Reports of Fire	Reports of Fire. A fire was reported at the Victoria Inn & Suites on 16th Febuary, 2011. The fire was extinguished with minimal damage to the building. There was no water damage.
SGP	Lost Pet	Lost My Pet	Lost My Pet??? My dog has been missing for a while. We have tried searching for him, but I can't find him. He was a good dog, but he was missing for a long time. I can't find him. Where is he? Is he in the house?
SGA	Unexpected Activity	Creeper	My neighbor is a creep. He hangs around our yard and keeps looking for food. Last week he came back after we left and stole a can of tomatoes.
SGA	Crime	Stolen Packages	My neighbors kids came in and stole some packages from the front porch. Kids about 12 and under. They were looking for something in a brown bag.

Table 6: Number of successful input injections.

Service	No. Installations	Domain	Input Injections	Interface
Strava	10,000,000+	Fitness Service	708	Web API
Fitbit	50,000,000+	Fitness Service	98	Web API
Map My Run	10,000,000+	Fitness Service	797	Web API
Basket Savings	100,000+	Pricing Service	156	Web API
ToiFi (Toilet Finder)	50,000+	Location-Based Service	2728	Web API
Police Detector	5,000,000+	Location-Based Service	2910	Web API
Transit	5,000,000+	Transportation Service	403	Sensor
Neighbors By Ring	1,000,000+	Safety Service	113	App UI
Google Maps	5,000,000,000+	Transportation Service	-	App UI, Sensor
Gas Buddy	10,000,000+	Pricing Service	-	Sensor

C ETHICAL CONSIDERATIONS

C.1 Details on IRB Approval

We applied for an IRB at the University of Southern California which can be verified by study-iD:IIR00003094. The following is an excerpt from the response we received: "Therefore, this study is considered Not Human Subjects Research* and is not subject to 45 CFR 46 regulations, including informed consent requirements or further IRB review.". Our study do not qualify for IRB approval because we do not collect data through interaction with humans and do not collect personally identifiable information.

Further details from the IRB response email are provided below:

> The University Park Institutional Review Board (UPIRB) designee reviewed the information you submitted pertaining to your study and has determined on 12/10/2019 that the project does not qualify as Human Subjects Research.
>
> From 45 CFR 46.102, The Federal Regulations on Human Subjects Research is as follows:
>
> Human Subject: A living individual about whom an investigator (whether professional or student) conducting research obtains data through intervention or interaction with the individual, or identifiable private information.
>
> Research: A systematic investigation, including research development, testing, and evaluation, designed to develop or contribute to generalizable knowledge.

Nonetheless, we integrated various measures in the design of our experiments' to protect users and apps from harm. Next we elaborate on some of those measures.

C.2 Basket Savings

The experiments were conducted after midnight when the grocery stores are closed. Item prices were reverted to their original value right after each experiment.

C.3 Fitbit

The added fake activities were deleted right after the experiments. The fake accounts did not add any real people as friends on the app to minimize interaction with actual users on the app and we did not participate in any real challenges to win rewards from a leaderboard.

C.4 Police Detector

The vast majority of the inserted POIs were positioned in the ocean or away from human populations, therefore it is unlikely that people were misled by them. We also deleted all POIs right after their insertion to ensure that fake POIs could only be visible for a couple of seconds.

C.5 Transit

Our experiments were performed targeting the same bus route (18km) in a rural area. To minimize influence, we performed our experiments during off-peak hours. *Transit* displays the number of users viewing one's contribution. In our experiments, we verified that we only affected our observer devices and no actual user was affected.

C.6 Neighbors by Ring

We set our location in Death Valley to limit the possible number of inhabitants and checked that there were no activity within the app at and around that location. We also removed the accepted posts within 8 min of being published. Furthermore, we only interacted with the algorithms and the service itself and collected data that is openly available and non-identifiable.

C.7 Strava

We did not participate in any real challenges to win real rewards from leaderboards. Furthermore, we did not add any friends on the app which means no other user got notified or could view our fake activities. The added fake accounts and fake activities were deleted right after the experiments.

C.8 Map My Run

We did not participate in any real challenges to win real rewards from leaderboards. Furthermore, we did not add any friends on the app which means no other user got notified or could view our fake activities. The added fake accounts and fake activities were deleted right after the experiments.

C.9 Toifi (Toilet Finder)

The majority of the inserted POIs were located in the ocean or away from human populations. All POIs were deleted the POIs right after inserting which means they were only visible for a couple of seconds at most.

C.10 Google Maps

The experiments were conducted within the university campus and after midnight to minimize the effect on users. For the experiments, we were physically present to ensure that there were no people present at the time and could abort the experiment otherwise.

C.11 Gas Buddy

We reverted the prices back to normal right after changing them, which means they were visible on the app for a couple of seconds. All experiments took place after midnight to further minimize potential exposure of users to the fake prices.