Fall 9-26-2013

Trust-but-Verify: Guaranteeing the Integrity of User-generated Content in Online Applications

Akshay Dua
Portland State University

Trust-but-Verify: Guaranteeing the Integrity of User-generated Content in Online Applications

by

Akshay Dua

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Nirupama Bulusu, Chair
Wu-chang, Feng
Wu-chi, Feng
Tom Shrimpton
Miguel Andres Figliozzi

Portland State University
2013

Abstract

Online applications that are open to participation lack reliable methods to establish the integrity of user-generated information. Users may unknowingly own compromised devices, or intentionally publish forged information. In these scenarios, applications need some way to determine the "correctness" of autonomously generated information. Towards that end, this thesis presents a "trust-but-verify" approach that enables open online applications to independently verify the information generated by each participant. In addition to enabling independent verification, our framework allows an application to verify less information from more trustworthy users and verify more information from less trustworthy ones. Thus, an application can trade-off performance for more integrity, or vice versa. We apply the trust-but-verify approach to three different classes of online applications and show how it can enable 1) high-integrity, privacy-preserving, crowd-sourced sensing 2) non-intrusive cheat detection in online games, and 3) effective spam prevention in online messaging applications.

To An-cheng Huang and Candy Yiu
for inspiring me to begin this journey.

Acknowledgments

I would like to thank my advisers Dr. Nirupama Bulusu and Dr. Wu-chang Feng for guiding me through the challenges of doing research, for teaching me to ask the right questions, for helping me formulate problems, for proof-reading what I wrote, and for pointing me in the right direction whenever I got stuck. I would also like to thank James Binkley and Dr. Suresh Singh for advising me on the problem I addressed during my Research Proficiency Examination. I owe a great deal to Dr. Wu-chi Feng and Dr. Tom Shrimpton for all their advice and help in making this dissertation possible and also for serving on my committee. Thank you Dr. Miguel Andres Figliozzi for agreeing to be on my committee and for all the valuable advice.

This journey would be more difficult and much less fun without my fellow graduate students Candy Yiu, Emerson Murphy-Hill, Thanh Dang, John Kassebaum, Phillip Sitbon, Francis Chang, Chuan-Kai Lin, Tom Harke, Ed Kaiser, Kevin Dyer, Scott Brittel, and Jeremy Steinhauer. Together we shared many enjoyable lunches and dinners, thought provoking conversations, Foosball games, and excellent research advice. I also greatly appreciate the implementation and research help I received from my colleagues and co-authors Tien Le, Sam Moffat, Fletcher Hazlehurst, Danny Aley, Thai Bui, and Nhan Huynh.

I cannot help but thank the staff at Portland State University, especially Rene Remillard, Beth Phelps, Sara Smith, Paula Harris, Roxanne Treece, and Krys Sarreal for always rescuing me from unintentionally breaking the rules.

Special credit goes to my wife Candy Yiu for all her patience and encouragement. I also would not have taken the plunge without the support of my parents, uncle, and aunt. Moreover, my parents are the reason this dissertation finished at the time that it did. They have my eternal gratitude for always being there.

Finally, this dissertation could not have been possible without the financial support I received from NSF grant 0747442.

Table of Contents

List of Tables

List of Figures

Chapter 1

THE PROBLEM

Online applications that encourage open participation remain vulnerable to spurious information. As examples, consider the following scenarios: (i) A crowd-sourced sensing application like Waze [143] collects real-time GPS data from its users for map-building and traffic estimation but has no way to determine if the published coordinates were indeed generated by a user physically present at that location. Malicious users can report fake incidents to gain points and an elevated status. They can also discourage others from taking a certain route by easily creating multiple emulated devices and publishing locations in a way that indicates congestion on that route, (ii) Malicious players of online games can access hidden game client memory and learn peer secrets that inform their next move. These moves would not normally occur without illegal access to hidden game client memory and are a result of cheating, (iii) online messaging applications like Email, Twitter, or Facebook still receive copious amounts of spam from software robots that hijack user accounts [51, 80, 56].

This dissertation aims to provide online applications with the tools to distinguish between "genuine" and "fabricated" content. Of interest, is the scenario where users generate and send the content to the application, which in turn, determines the integrity of that content before processing it any further. Specifically, this dissertation develops data integrity verification methods that (i) enable high-integrity privacy-preserving crowd-sourced sensing applications, (ii) facilitate cheat

detection in online games, and (iii) effectively combat spam in online messaging applications.

1.1 BACKGROUND AND MOTIVATION

In this section, we discuss existing approaches that allow an application to verify the integrity of user-generated content and explain why a new approach is needed. Quite often, applications employ no specific integrity checking techniques. Thus, the burden of validating published content is left to the end-user. For example, the popular social network, Facebook [2], only provides guidelines on how users can identify fake accounts [1]; the open, collaborative, and online encyclopedia, Wikipedia [3], only provides guidelines on how readers can identify reliable sources so that they can determine the validity of the published information for themselves [4]. When integrity verification methods have been used by applications, they have traditionally been based on reputation ratings [48, 66, 107], anomaly detection [29, 26, 104, 78], or user-device monitoring [84, 112].

Reputation Ratings Reputation rating algorithms are used mainly by applications whose success depends on trust relationships between users. For example, the popularity of the online marketplace, EBay [41], is attributed to its use of seller and buyer ratings. The problem, however, is that users may not have any existing trust relationships to begin with. Thus, the goal of a reputation rating algorithm is to create and maintain these trust relationships.

A reputation rating algorithm defines, initializes, and updates a trust metric or "reputation" for each user of the application. The fundamental assumption is that a user's past reputation is a good predictor of future behavior, implying that a user with a better reputation rating can be trusted more. The reputation of a given user is updated based on what other users perceive that user's reputation to be. Thus, each user assigns a reputation rating to every other user it interacts

with. Then, multiple ratings for the same user are combined to determine an overall reputation rating. How this rating is represented depends on the particular reputation rating algorithm. For example, EBay [41] represents them as discrete values, whereas Ganeriwal et al. [48] and Jsang et al. [66] represent them using the Beta distribution.

An important factor in the success of a reputation rating algorithm is how the overall rating for a particular user is calculated [76]. The overall rating for a given user is a combination of ratings assigned by other users. There is also the issue of combining the ratings one user assigns to another user across multiple interactions. Several approaches have been proposed, the simplest being the average [76].

Unfortunately, reputation ratings are not directly applicable in the scenario under consideration where users are generating and sending content to an application, but not necessarily interacting with each other.

Anomaly Detection Anomaly detection is used mainly by applications that collect numerical data from multiple sources, such as from Wireless Sensor Networks (WSNs) [29, 26, 104, 78]. The basic assumption is that clusters of sources produce "similar looking" data (e.g. locality). The goal then is to identify data samples that "look different" from what is considered "normal" for the cluster. This requires creating a model for normal data and checking if incoming live data adheres to that model.

A popular approach for building the model is Principal Component Analysis (PCA) [65]. Given a set of multi-variate data samples, PCA's goal is to reduce the number of variables required to represent the data while still capturing most of the variation in the data. The way PCA does this is by removing correlated variables since they carry very little variance information and leave behind the uncorrelated ones. This essentially reduces the number of dimensions required to represent the data. Later, a live sample of data can be projected onto the reduced dimensions

and checked if the projected point lies too far from the projections of the normal data samples.

Unlike reputation rating algorithms, anomaly detection is directly applicable in the crowd-sourced sensing scenario considered in Chapters 3 and 4 where the sensors are personal mobile devices carried by people. However, as indicated earlier in the Waze [143] example, anomaly detection solutions can be duped by a participant that emulates multiple colluding participants (called "Sybils") [37]. Additionally, if the data sources are few, sparsely distributed, or constantly in motion as in crowd-sourced sensing, anomaly detection may fail to be accurate [29].

Device Monitoring Device monitoring tries to ensure that the software users employ to generate and send data to the application is running as expected. This requires a monitoring service running on the user's device that is in some way trusted by the application.

Device monitoring approaches have been used to detect cheating in online games [112, 69]. Warden [112, 84] is one such monitor used by the popular game World of Warcraft [19]. Warden tries to find cheat code with known signatures on a player's machine by scanning the memory of all running processes. Due to the use of signature-based detection via widespread scanning, Warden has suffered from false positives, an inability to detect new cheats, performance issues, and has been labeled "spyware" by the Electronic Frontier Foundation [81].

As a solution, a more conservative scanner, called Fides [69], was developed by Kaiser et al. In addition to restricting scanning only to the game client, Fides looked for anomalies rather than particular cheat signatures. This made Fides more effective against new attacks whose signatures were not yet know. To detect anomalies, Fides profiled the game client to establish a model for normal execution. The profile was then used on the game server to partially emulate the client and validate memory signatures sent by the scanner. If the client-side memory

signatures did not match the "normal" for a given stage in the game, then a client was assumed to be cheating. However, since the scanner ran on the client — a machine in control of the cheater — it could be tampered with in ways that avoided cheat detection. Other challenges included the complexity of accurate game client profiling and emulation, determining the set of scanned memory locations that when checked, would reliably detect unknown cheats, and earning a player's trust that the scanning will indeed be restricted to game client memory.

1.2 RESEARCH CHALLENGES

The research challenge is to develop integrity verification methods for user-generated content that address the drawbacks discussed in the previous section, while satisfying the constraints of the individual applications. Specifically, this dissertation develops integrity verification mechanisms that enable high-integrity privacy-preserving crowd-sourced sensing, non-intrusive cheat detection in online games, and effective spam prevention in online messaging applications.

1.2.1 High-integrity Crowd-sourced Sensing

Crowd-sourced sensing systems allow people to voluntarily contribute sensed information (e.g. location, images, audio) using personal sensing devices like smartphones [43]. The volunteers collecting the data are called *data sources* and the online portal publishing the data received from volunteers is called the *data consumer*. In this scenario, the data consumer would like to determine if the sensory data submitted by volunteers accurately represents the phenomenon being sensed. This is challenging because data sources have autonomous control of their sensing devices and could therefore modify its firmware to publish fabricated information instead of what is actually sensed. Sources could also publish false sensor data using emulated devices rather than physical sensors. Thus, we must find a way

to convince the consumer that the sensed data originated from correctly functioning physical sensors that actually witnessed the phenomenon. This problem is addressed in Chapter 3.

1.2.2 High-integrity Privacy-preserving Crowd-sourced Sensing

Another important issue in crowd-sourced sensing is location privacy. In general, crowd-sourced sensory data is supplemented with a corresponding location that indicates where that data was sensed. The problem is that the data consumer could potentially track a source using the published locations. Thus, we must find a way to guarantee the integrity of published locations without revealing those locations to the data consumer. This problem is addressed in Chapter 4.

1.2.3 Non-intrusive Cheat Detection in Online Games

A lot of hidden information is present in client programs of most existing online multi-player games (e.g. unexplored map regions, opponent resource information). Although a lot of this information is not necessary to render the player's current view of the game, it is necessary to render any *new* views. Having local access to this information results in efficient game rendering. The problem, is that the unnecessary hidden information can be used to cheat. For example, a player could use a "map hack" to uncover a hidden mine field that would otherwise only be visible with special powers. Unfortunately, if this hidden information is not locally present, but instead received *on-demand* from the server, then the game rendering slows down significantly [75]. What is needed is a more efficient method to detect moves that are a result of this kind of cheating while also being an alternative to existing solutions that perform intrusive client-device scanning [112]. This problem is addressed in Chapter 5.

1.2.4 Effective Spam Prevention in Online Messaging Applications

A lot of effort has been devoted to building accurate spam filters that can hide spam from the user [59, 121, 130, 11], but little is done to effectively prevent it. The *proof-of-work* approach has been proposed as an effective spam prevention technique [40, 15, 149, 44, 68, 64, 77].

Proof-of-work systems can be used by online messaging applications to impose a cost per online transaction. This cost is in terms of computational work, also called a *puzzle*, that a client must correctly execute before being allowed to send a message. The assumption is that solutions of puzzles are hard to compute and easy to check [40, 39, 64], and that more malicious users (e.g. spammers) are issued "harder" puzzles [74, 77]. Puzzles can be computed by the browser behind-the-scenes and require no user involvement. However, the work done in computing puzzle solutions is most often wasted because the solution provides no other value than to aid in checking that the puzzle was correctly executed. The Reusable Proof-of-Work (RePoW) approach [64] proposes using puzzles whose solutions can be used for a greater purpose (e.g. protein folding [95]).

Unfortunately, there are very few known reusable proof-of-work puzzles whose solutions are easy to check [64, 17]. What is needed is a proof-of-work system that enables issuing generic computational tasks (e.g. protein folding [95]) whose result can be checked more efficiently than recomputing the task itself. This problem is addressed in Chapter 6.

1.3 SOLUTION OVERVIEW

We address the research challenges discussed in the previous section by employing a "trust-but-verify" approach that enables applications to independently verify the integrity of data published by each user, and do so as often as necessary. In the case of crowd-sourced sensing applications, the published data is a location and

associated sensory information; in the case of online games, it is the player's next move; and in the spam prevention scenario, it is the solution of a puzzle.

The idea behind the trust-but-verify approach is to first, identify or define the data generation functions executed by users, clients, or data sources to create and publish data, and then, enable the application or data consumer to verify that those data generation functions were correctly executed. The challenge is in building verification methods that satisfy the constraints of individual applications. For example, we will see in Chapter 4 that privacy-preserving crowd-sourced sensing applications must verify the integrity of data source locations without being provided with those locations.

Unlike anomaly detection (Section 1.2), the trust-but-verify approach enables applications to independently verify the integrity of data published by each user. Unlike device monitoring (Section 1.2), the trust-but-verify approach does not require intrusive scanners to ensure that generation functions are correctly executed. It also enables efficient integrity checking by allowing the application to verify less information from a participant it trusts more, and verify more information from a participant it trusts less. The details of the trust-but-verify approach have been presented in Chapter 2.

Applying, the trust-but-verify principles to guarantee the integrity of user-generated content in crowd-sourced sensing, online gaming, spam prevention systems have resulted in the following contributions:

- In Chapter 3 we present the design, implementation, and evaluation of the *Trusted Sensing Peripheral (TSP)*: a sensing device with an embedded trusted-third-party that attests to the validity of the generated sensory information.

- In Chapter 4 we present the design, implementation, and evaluation of *LocationProof*, a protocol that enables TSP-based crowd-sourced location sensors to prove the integrity of locations without revealing those locations.

- In Chapter 5, we present the design, implementation, and evaluation of *SpotCheck*, a system for detecting information exposure cheats in online games.

- In Chapter 6, we present the design, implementation, and evaluation of *MetaCAPTCHA*, an effective spam prevention system for online messaging applications based on the reusable-proof-of-work approach. MetaCAPTCHA allows the "work" issued to a user's device be any generic computation.

1.4 THESIS STATEMENT

The trust-but-verify approach allows online applications that encourage open participation to

- independently determine the integrity of data published by each participant

- obtain the desired level of integrity guarantee

- non-intrusively establish data integrity

Chapter 2

THE TRUST-BUT-VERIFY APPROACH

As shown in Figure 2.1, the *trust-but-verify* approach is applicable to a system in which a set of *data sources* $S = s_1, ..., s_n$ periodically send the content they generate to a *data consumer* C. The sources are usually user devices and the consumer is usually the application interested in the data produced by the sources. At any given time instant t, a source i uses input $x_{i,t}$ to generate content $y_{i,t} = f(x_{i,t})$. The input $x_{i,t}$ may be provided by the user or obtained directly from the environment: for example, in crowd-sourced location sensing applications, sources publish coordinates that are generated using GPS satellite signals present in the environment. The generated content $y_{i,t}$ may further be aggregated locally by the source and globally by an *aggregator* A before finally reaching a consumer C. At a given time t, local aggregation involves combining past and current content $\{y_{i,t_p} : t_p \leq t\}$ while global aggregation combines content $\{y_{u,t} : s_u \in S\}$ generated by multiple sources.

Ultimately, the consumer C would like to determine with some confidence that the received *aggregate* z_t is correct. Here, correctness implies that each input $x_{i,t}$ was processed in order using only the functions $f(\cdot)$, $f_1(\cdot)$, and $f_2(\cdot)$; together called the data *generation* functions. The goal then, is to develop a data *verification* function that allows C to determine if the data generation functions were faithfully executed over the inputs.

If the data sources and aggregators are trustworthy, there would be no need for a verification function; C could simply assume that z_t is correct. However, C

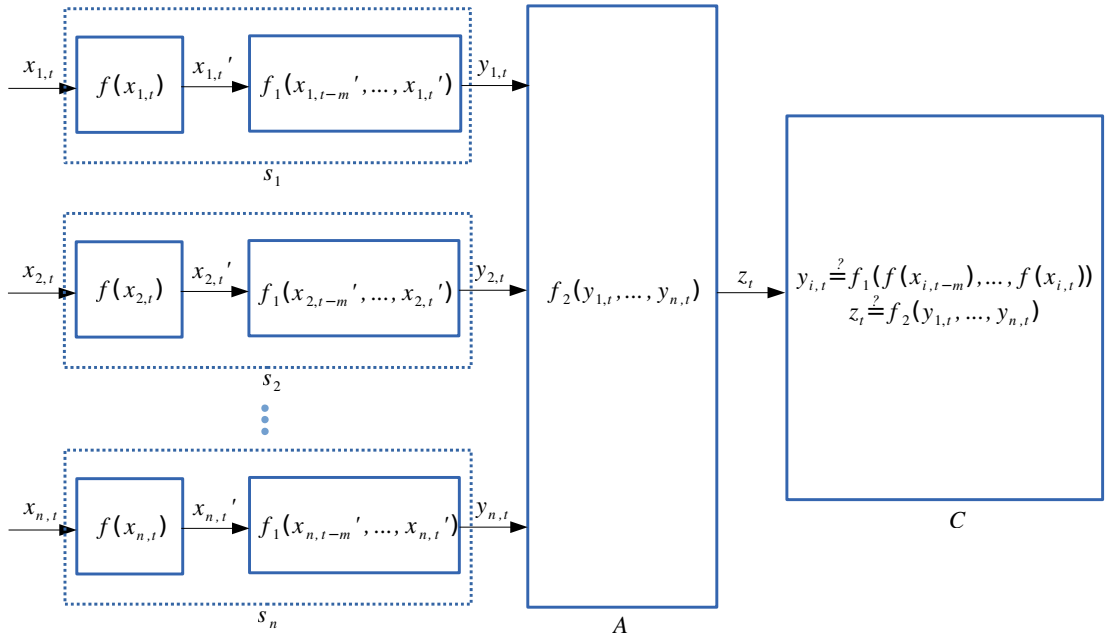


Figure 2.1: The trust-but-verify system model. Data sources $S = s_1, ..., s_n$ are generating and publishing data to a consumer C. The generated data may optionally be aggregated locally with past data or globally, by an aggregator A, with data from other sources. The goal of the trust-but-verify approach is to provide C with a way to determine if the inputs to the sources were indeed processed in order by the functions $f(\cdot)$, $f_1(\cdot)$, and $f_2(\cdot)$ (together called the data generation functions).

may not always have a way to establish a basis of trust [7]: for example, in crowdsourced sensing applications, people volunteer to collect and publish data using personally owned mobile devices. In this scenario, the consumer has no way of knowing if those mobile devices are faithfully processing inputs sensed from their immediate environment. The goal of this work is to provide the data consumer C with tools to verify the correctness of received aggregate z_t, thereby allowing C to cultivate trust in the sources and aggregators that contributed to computing z_t. More specifically, upon receiving z_t, C may challenge the sources and aggregators to prove its correctness. The sources and aggregators then collectively send back a

set of responses R_t that allows C to verify the correctness of z_t. A trivial solution is for each source i to provide C with input $x_{i,t}$, i.e. $R_t = \{x_{i,t} : s_i \in S\}$. C can then independently compute a value z using the generation functions $f(\cdot)$, $f_1(\cdot)$, and $f_2(\cdot)$ over respective subsets of R_t and check if it matches z_t. The problem with this approach is that it is not generally applicable. Consider again, our crowd-sourced location-sensing example: here, the sources have no way of sending the analog GPS satellite signals they sensed in their immediate environment. Thus, the set of responses R_t and the verification method must be tailored to each application. Notice that the trust-but-verify approach does not attempt to establish the authenticity of input $x_{i,t}$, it attempts only to determine if $x_{i,t}$ is faithfully processed at each intermediate step.

Now, assuming that a verification procedure has been defined, the data consumer may choose to check the integrity of aggregate z_t during only a subset of time instances $\{t : t_{begin} \leq t \leq t_{curr}\}$ where t_{begin} represents the time data collection began and t_{curr} represents the current time. How often a consumer checks the integrity of an aggregate is called the consumer's *verification strategy*. Picking a strategy will depend on a trade-off between how much unchecked data the consumer can tolerate and the amount of computational resources it can devote to integrity checking. The advantage of the trust-but-verify approach is that a consumer *can* pick a strategy.

Once a consumer picks a strategy, it can estimate the amount of time before a fabricated aggregate is detected using the discrete negative binomial distribution. The discrete negative binomial distribution $\mathrm{NB}(r, p)$ describes the probability of the number of "successes" in a sequence of binomial trials before r "failures" occur. The success and failure probabilities in any given trial are p and $1-p$ respectively. If we assume that received data z_t is being corrupted with probability q and that the consumer is checking the integrity of z_t with probability p, then the probability of detecting a corrupt z_t during time instant t is pq. Now, in the context of the

negative binomial distribution, "success" can be represented by the event, "not detecting a corrupt z_t", and "failure" can be represented by the event, "detecting a corrupt z_t". Thus, the success and failure probabilities in any time instant t are $1 - pq$ and pq respectively. Also, since we are interested in the number of "successes" before the *first* "failure", the parameter $r = 1$. Given these distribution parameters, the expected number of time instants before the first corrupt z_t is detected can be written as:

$$\frac{1 - pq}{pq}$$

Figure 2.2 shows the expected number of time instants before detecting a corrupt z_t given $q \in Q$, $Q = \{0.2, 0.3, 0.5, 0.7\}$, and $.01 \leq p \leq .99$. We can see that when checking 20% of the time ($p = 0.2$), the consumer is expected to detect a corrupt z_t within 7 to 24 time instants given that $q \in Q$.

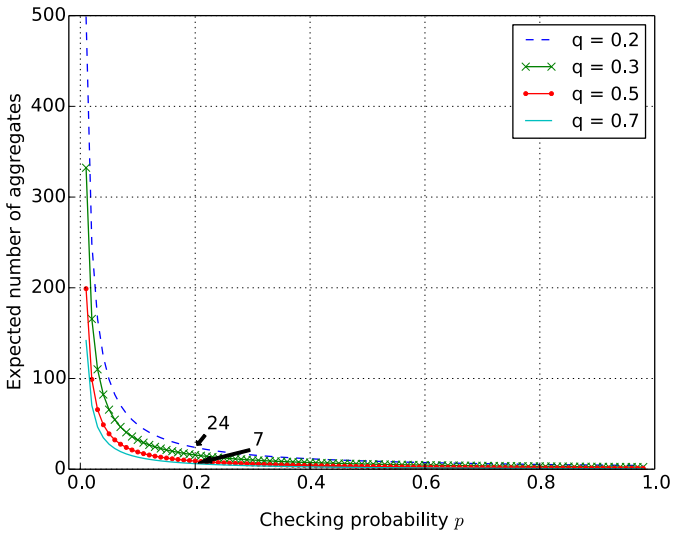


Figure 2.2: Expected number of time instances before a corrupt z_t is detected for the first time. Here, the z_t is being corrupted with probabilities $q \in Q$, $Q = \{0.2, 0.3, 0.5, 0.7\}$, and checked with probabilities $.01 \leq p \leq .99$ respectively.

In the following chapters, we explain how the trust-but-verify approach can be

used to determine the correctness of data received from sources and aggregators in different applications.

Chapter 3

HIGH-INTEGRITY CROWD-SOURCED SENSING

Crowd-sourced sensing systems allow people to voluntarily contribute sensed information (e.g. location, images, audio) to Internet portals like SensorMap [93] using a personal sensing device [43]. These systems have the potential to provide unprecedented insight into our local environment because data is collected everywhere people are. For example, Paulos et al. [96] conducted a field study in Accra, Ghana where they retrofitted seven taxis with pollution sensors that constantly collected air quality information en route. The proliferation of increasingly connected, programmable, and sensor-rich mobile platforms like smartphones constantly fuels the growth of crowd-sourced sensing applications. Noteworthy examples include traffic, health, and safety monitoring; citizen science initiatives; and personal environmental impact reporting [31, 131, 106, 96]

This use of existing mobile infrastructure makes crowd-sourced sensing cheaper to deploy than dedicated sensor networks. Data collected from such systems can improve understanding of local environments, help shape public policy or facilitate new scientific research. However, government and researchers alike will be reluctant to use the data if they cannot trust its integrity.

This lack of trust would be unfounded if sensed data were infeasible to fabricate, but that is not the case. Consider a crowd-sourced traffic navigation system like Waze [143]. Participants of Waze download a client application on their smart phones that activates the phone's internal GPS receiver, and periodically reports their GPS coordinates to Waze. With sufficient user participation, Waze can provide optimal routes to destinations based on the real-time traffic information

learned from participants. A malicious participant could easily fool Waze by modifying the client application to publish false hazards in order to score points. He could launch multiple emulated smart phones on resource-abundant hardware and submit random GPS coordinates. We classify the former as a *software attack* and the latter as a *Sybil attack* [37]. The aforementioned attacks are possible mainly because the sensing devices (e.g. smart phones) are under autonomous control of the participants. Although our example highlights the GPS sensor, software and Sybil attacks are applicable to any of the phone's sensors.

3.1 APPLYING THE TRUST-BUT-VERIFY APPROACH

We address software and Sybil attacks by enabling crowd-sourced sensing applications to establish the integrity of the sensory information received from the participants. Sensory information has integrity if it was captured by appropriate physical sensors that actually witnessed the phenomenon. This chapter establishes this fact by applying the trust-but-verify approach. First, we must identify the *data* to verify: in this case, it is the output of the sensors. Next, the *generation functions* for that data must be identified: in crowd-sourced sensing applications, these are the functions in the device driver and sensing platform that together collect data from the environment. The final step is to develop a *verification function* that allows the application to determine that the generation functions were faithfully executed. The approach we use here, is to build a trustworthy sensing platform that can attest to the integrity of the software running on it. The attestation is like a signature, that when verified by a remote party, guarantees that the platform was running a certain unmodified version of software. Trust in the platform, in turn, induces trust in the faithful execution of the generation functions. Sections 3.3 – 3.6 describe how to build this trustworthy sensing platform.

Applications may also require that trustworthy sensing platforms aggregate their data locally before publishing it. In this case, the aggregation function is an

additional generation function whose faithful execution must be guaranteed by the trust-but-verify approach. We describe how this is done in Section 4.6.

3.2 CONTRIBUTIONS

This chapter presents a trusted-hardware based sensing platform to combat the above attacks, enabling crowd-sourced collection of high-integrity *raw* or *aggregate* data. Its contributions are:

- *The design and implementation of a Trusted Sensing Peripheral (TSP).* The TSP is a sensing platform that consists of a Trusted Platform Module (TPM) [134] and hardware sensors. Our goal is to make the TSP economically infeasible to emulate or modify without being detected. The TPM facilitates this by attesting to the authenticity and integrity of the TSP as well as the sensor data. Successfully verified attestations prove to a data receiver, such as an online portal, that the data is indeed authentic and was not altered, either by the TSP or during transit. Experiments show that the TSP is very energy efficient: it has a projected battery life of over *80 days* when collecting, attesting, and transmitting a temperature sample every 30 seconds.

- *STAP: an efficient, end-to-end, high-integrity, data aggregation protocol for crowd-sourced sensing.* The TSP publishes sensor data to an online portal via the participant's mobile device acting as a forwarding proxy. To reduce the energy expended in transmitting raw data, and the processing overhead at the portal, the mobile proxy could aggregate the TSP's raw data before forwarding it. But, a malicious proxy could fabricate the aggregates instead of actually computing them from the TSP's raw trusted data. The Secure Tasking and Aggregation Protocol (STAP) modeled on a bit-commitment scheme [27, 16], uses a pseudo-random challenge-response mechanism that allows a portal to detect a lying proxy. By randomly challenging only *20%*

of the aggregates, the portal can detect a lying proxy within the first *six* false aggregates received.

- *A high-integrity platform that augments legacy mobile devices with trustworthy sensing.* The Trusted Sensing Peripheral (TSP) enables legacy untrustworthy mobile devices to produce trusted data. Section 3.5.2 discusses challenges to making contemporary sensing-capable phones trustworthy: namely the absence of proper hardware support, distrust in trusted hardware, and growing exposure to remote attacks. Besides trustworthy sensing, the TSP provides greater sensing modalities than commodity mobile devices.

3.3 SYSTEM AND THREAT MODEL

As Figure 3.1 shows, the TSP collects sensor data, has the TPM sign it, and forwards it to the mobile proxy, which either aggregates the raw data or forwards it as is. Adversaries may compromise either the TSP, or the mobile proxy.

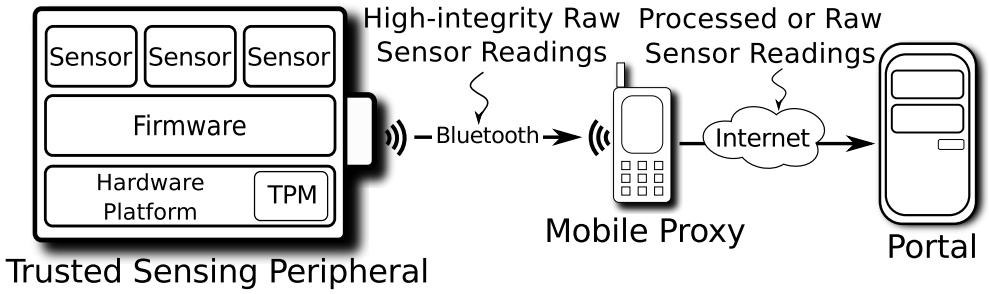


Figure 3.1: The TSP publishes data via the participant's mobile device, which can then process the data, or forward it as is.

In the context of the trust-but-verify approach, the TSP is the data source, the mobile proxy is the local aggregator, and the portal is the data consumer. The generation functions include those on the TSP that perform the sensing (e.g. functions in the sensing device drivers) and any data aggregation functions executed by the mobile proxy.

3.3.1 Threats to the TSP

Our goal is to prevent or detect software and Sybil attacks against the TSP, to ensure that data reported to the portal was not modified in unintended ways by the TSP software, and did not originate from emulated, simulated, or fake devices. We considered the following specific threats:

- *Software modification.* This refers to any modification of the TSP firmware. It can happen remotely or with physical access to the TSP. An adversary that gains control of the firmware can modify the included device drivers to alter sensor data or the time when it is published.

- *Masquerading.* The adversary can have a device that either pretends to be a legitimate TSP, or creates multiple emulated or simulated TSPs, each called a Sybil. The Sybil attack [37] will be successful if the system cannot detect or prevent the creation of a Sybil.

- *Communication Integrity Compromise.* A man-in-the-middle can inject, modify, drop, or replay messages in transit between the TSP and the portal. A successful attack on communication integrity can result in the publication of false, corrupt or stale data.

3.3.2 Threats to the Mobile Proxy

The participant's mobile device serves either as a *forwarding proxy* for raw data originating from the TSP, or as a *data aggregating proxy* when processing the raw data before sending it to the portal. A malicious participant may modify its mobile proxy to compromise the integrity of published data. The proxy is not trusted by the portal or the TSP in any role and poses the following threats to data integrity:

- *Communications compromise.* The threats posed by a compromised forwarding proxy are the same as threats to communication integrity between the

TSP and the portal. Addressing communication integrity threats simultaneously addresses threats from a malicious proxy.

- *Malicious data aggregation.* When a compromised mobile proxy functions as a data aggregating proxy, it can fake the aggregate values, drop them, or inject any new ones. In this scenario, the portal must be able to detect and reject data from the malicious proxy.

3.3.3 Threats not Addressed

Although we address some of the most challenging threats to data integrity in crowd-sourced sensing, the following threats, although important, have not been addressed:

- *Availability.* A remote adversary may force the TSP or proxy to continuously receive messages, depleting their battery [122]. Or the mobile proxy may drop all communications between the TSP and the portal. Such forms of Denial of Service (DoS) attacks are currently not addressed.

- *Confidentiality.* We primarily address data integrity in crowd-sourced sensing, as such, we rely on other components to provide confidentiality. For example, the channel between the TSP and the mobile proxy (see Figure 3.1) can be secured using Bluetooth's security features.

- *Physical Data Poisoning.* A participating adversary may alter the very phenomenon being sensed. For example, he could collude with others to drive slowly, thus, depicting congestion. It is important to note that the TSP *necessitates* physical tampering of the phenomenon to publish fabricated data, as opposed to simple software modification to do the same. We believe this greatly *raises the bar* for data poisoning attacks.

- *Hardware attacks on the TPM.* Since the TPM is the root of trust for our approach, we assume that it is tamper-proof as claimed, not compromised, and functioning as expected. This is a reasonable assumption given that simple yet effective hardware attacks on the TPM, like the timing and reset attacks have already been addressed [13, 133]. Furthermore, the most recent attack involving extraction of an obsolete TPM's "burned-in" private key took specialized skills, around six months, and costly equipment worth about 200,000 USD [129]!

3.4 THE TRUSTED PLATFORM MODULE

As we discuss in Section 3.5, the TPM is the root of trust for data authenticity and platform integrity assurance provided by the TSP. Trust in the TPM stems from internationally recognized common criteria standards [127]. The TPM is a tamper-proof secure co-processor that enables trusted computing principles on commodity computing platforms. It is housed on the host platform and provides tamper-proof storage for cryptographic keying material, and platform configuration information. Additionally, it can digitally sign and report the securely stored configuration information, thus indirectly attesting to the integrity of the platform. Further, since the digital signatures are computed using keys protected from extraction, any signed information from the TPM can be considered authentic once the signature is verified.

A TPM is associated with several credentials, each containing information regarding the TPM or its associated platform, and digitally signed by the entity issuing the credentials. References to the various TPM credentials (in the form of message digests) along with the public portion of the TPM's RSA signing key (AIK_{pub}: Attestation Identity Key) are included in a final *attestation identity credential* that is then signed by a trusted certificate authority. Once presented to remote entities, the successful verification of the attestation identity credential

proves that the specified platform indeed contains a certified TPM and that any digital signatures performed by that TPM (using AIK_{priv}) can be verified using the included public signing key. Since the corresponding private key is securely created, stored, and protected from extraction by the TPM itself, the TPM's signature *cannot be repudiated*.

The TPM also provides *load-time* platform integrity verification via its *platform attestation* feature. However, *run-time* changes to the platform cannot be directly captured by the TPM. Section 3.5.2 discusses the design of our Trusted Sensing Peripheral (TSP) that inherently resists run-time software attacks. Thus, trust in the run-time state along with trust in the initial platform configuration transitively induces trust in the load-time state of the platform, allowing the TSP to use the TPM solely as a signing authority.

3.5 DESIGN

In this section, we describe the Trusted Sensing Peripheral (TSP) and the Secure Tasking and data Aggregation Protocol (STAP).

3.5.1 Design Assumptions

We assume that the online portal has the TPM's attestation identity credentials that contain its public Attestation Identity Key AIK_{pub}, used to verify the TPM's signature. For STAP, we assume a secure transmission channel (e.g. the Secure Sockets Library (SSL)) between the proxy and the portal, and that the portal has a strong pseudo-random number generator. Our protocol's security depends on the assumption that a malicious proxy cannot reliably predict the sequence of random numbers generated by the portal.

3.5.2 Design Rationale

Designing a high-integrity sensing platform is not trivial. Especially, when the platform can potentially be in the physical possession of an adversary. This, by-definition, is the case with crowd-sourced sensing: participants carry mobile platforms with built-in sensors that upload data to an online portal. Although most participants may be honest, one can not ignore that malicious participants may poison the data integrity.

At the least, the portal must be able to *detect* sensory data tampering, or the sensing platform must be able to *prevent* it entirely. Since participation in crowd-sourced sensing is voluntary, people may register at any time with private heterogeneous sensing devices. Thus, the system must be able to detect fake devices or prevent their participation.

Cryptographic techniques that detect data tampering across a communication channel (e.g. message digests) work only if the communicating parties have a vested interest in protecting the transmitted data. However, in crowd-sourced sensing, one of the parties, namely the data producer, could very well be the adversary, making it very hard for the portal to discover if fabricated data was being protected in the first place.

The portal could compare multiple data values produced in the same region and reject the outliers. However, as discussed later in Section 3.9, this method may be effective at detecting outliers, but is counter to the objective of an adversary who actually wants to avoid detection. Consider again, the example of an adversary that emulates multiple distinct mobile devices and publishes GPS data to make a quiet neighborhood street appear congested. This situation will hardly appear to the system as outlying activity. Furthermore, since the portal cannot distinguish between real and emulated platforms, this sort of software collusion attack using multiple Sybil (or fake) identities will go undetected. Other issues with this approach are the need for redundant data sources in any region, and different outlier

detection mechanisms depending on the type of data being collected.

Consequently, our design takes the prevention approach. Here, the sensing platform itself prevents any modification to the sensed data. We use a trusted third-party housed on the platform to vouch for the integrity of all the software running on it. If the portal cannot verify the presence of a trusted third-party on the remote sensing platform, it can choose not to trust the data emanating from it. Since the presence of the trusted third-party cannot be cloned or faked, it is impossible for a sensing platform to appear trustworthy when its not. Our trusted third-party housed on the sensing platform is the TPM. It forms the root of trust for data and platform integrity assurances provided by the associated sensing platform.

Recall that the sensing platform in crowd-sourced sensing systems is usually the participant's mobile device, which is assumed to already have existing sensing capabilities (e.g. GPS, accelerometer, microphone). However, our approach was to build a separate "closed box" high-integrity sensing platform we now call the *Trusted Sensing Peripheral (TSP)*. Several factors motivated this approach:

- As Garfinkel et al. stated in their work on a trusted virtualization platform called Terra [50], "The security benefits of starting from scratch on a "closed box" special-purpose platform can be significant". They believed, like us, that an opaque special-purpose platform can more easily protect the integrity of its contents.

- Providing "closed box" semantics on most commodity mobile devices is not possible due to the lack of required hardware support and effective protection of software from run-time attacks. Terra [50] provides load-time application integrity guarantees using the TPM, and strong isolation by running it inside a virtual machine managed by a Trusted Virtual Machine Monitor. However, we are not aware of any existing commodity mobile devices equipped with

TPM chips. Further, Terra does not protect software from run-time attacks by malicious device drivers that have access to the DMA controller. Another system proposed by McCune et al. [82], called Flicker, provides a secure execution environment for a *portion* of an application's logic. Besides the need for a TPM, this approach also requires additional hardware support (like the $SKINIT$ instruction found on certain AMD processors [8], or the $GETSEC+SENTER$ instructions on certain Intel processors [63]) to setup the secure execution environment. Flicker provides protection from DMA attacks, and strong isolation by disabling all interrupts and debugging support. However, regardless of hardware support, certain practical limitations remain. Flicker requires that the OS and all associated tasks be suspended during the time the secure execution environment is active. Given the high overhead of the $SKINIT$ instruction and the required TPM operation on a desktop (912.6 msec on a 2.2 GHz AMD Athlon 64-bit Dual Core processor), it is safe to assume, for now at least, that the situation will be worse on a mobile platform with users frustratingly noticing their mobile device freeze periodically to allow secure sensing activities to take place.

- Existing mobile devices with sensing capabilities (e.g. smart phones) are as exposed and vulnerable as today's desktops. Our always-on Internet-connected smart phones are already at risk of being usurped by active botnets [72]. A "closed box" sensing platform, such as one without a connection to the Internet, will make remote attacks on sensory data integrity more challenging.

- Decoupling trustworthy sensing from trusted computing will make trustworthy sensing less invasive and encourage its adoption. Features provided by trusted computing, like TPM-based platform attestation, have been the topic of much debate. Some like Ross Anderson and Bruce Schneier have been

vocal critics and consider it to be an invasive technology that encourages software monopolies, facilitates DRM, and compromises privacy [23, 109]. For example, since it is necessary to verify the integrity of all software components on a platform to provide integrity assurances about a single one [110], the verifier would effectively know all about the platform's OS and application configuration. Thus, the TPM's platform attestation feature may inadvertently pose a threat to privacy.

- A separate trustworthy sensing device, like our TSP, could expand sensing capabilities. Mobile devices, like smart phones are equipped with only those sensors that help their functionality (e.g. GPS, microphone, accelerometer). However, some crowd-sourced sensing applications may require more specialized sensors, such as smog, dust, or chemical pollutant sensors to measure air quality. Furthermore, even mobile devices without any sensing capabilities could help produce trustworthy sensor data.

3.5.3 TSP Architecture

Figure 3.1 shows the TSP architecture, consisting of a TPM-capable hardware platform with attached sensors. A Bluetooth module allows the TSP to communicate with an array of mobile devices supporting the technology.

We achieve "closed-box" semantics on the TSP by building it using a special-purpose Modified Harvard-architecture sensing platform, and permanently disabling features that may compromise its software integrity. It is widely known, that run-time attacks exploiting memory-related vulnerabilities with the intent of modifying program instructions, have little or no chance of success on such platforms [47]. Such platforms provide strong physical isolation between executable instructions in program memory, and information in data memory. The program memory on such devices is read-only, and the program counter is not allowed to

refer to addresses in data memory. Consequently, program instructions can neither be changed, nor be executed from data memory at run-time. All usual methods to reprogram the TSP — physical or remote — are disabled permanently. Normal communication methods, like the radio, are also disabled. The only connection between the TSP and the outside world is via a Bluetooth channel. To enable genuine upgrades, we use the existing over-the-air self-reprogramming mechanism supported by the TSP. However, instead of receiving the firmware image over the radio channel, we receive it over Bluetooth. These protections are sufficient to prevent the TSP from a recent and only known permanent code injection attack on a Modified Harvard-architecture platform [47].

Given the above protections in place, a physical or remote adversary will not be able to change the pre-programmed firmware on the TSP. Consequently, the on-board TPM need only attest to a firmware version identifier on the device when challenged by a remote entity. Any data sensed by the TSP is signed by the TPM to guarantee its authenticity and integrity. Trust in the data is induced via trust in the platform integrity.

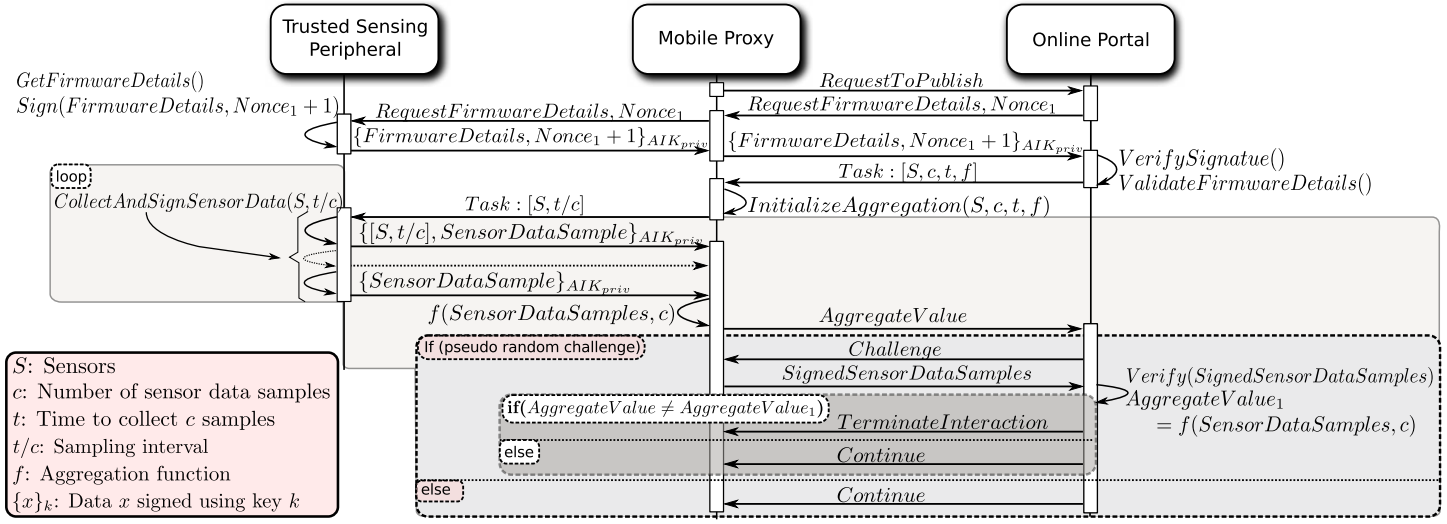


Figure 3.2: Secure Tasking and Data Aggregation Protocol (STAP)

3.5.4 Secure Tasking and Aggregation

Secure tasking involves verifying that the given platform has a TPM (see Section 3.4), then challenging it to produce a TPM attestation of the platform's firmware, then verifying the attestation is authentic and correctly represents the TSP firmware (e.g. comparing the respective message digests), and finally scheduling the TSP to collect, sign, and send data from a subset of its sensors at regular intervals.

To save energy required for data transmissions and to help reduce processing overhead at the portal, the data producer's mobile device (the mobile proxy) may choose to aggregate the raw data from the TSP before forwarding it to the portal. For example, instead of sending raw GPS coordinates from the TSP every 30 seconds, the mobile proxy may send a coarser region covered by those coordinates every five minutes.

We have developed STAP (Figure 3.2), a protocol that allows the portal to detect if the untrusted mobile proxy is fabricating the aggregates, as opposed to computing them using the TSP's signed raw data. To detect a lying proxy, the portal must know the aggregation function in advance. This function is therefore the generation and verification function in the trust-but-verify approach. Then, during the course of receiving data, the portal pseudo-randomly requests the mobile proxy for the latest window of signed raw data used to compute the just received aggregate (the proxy needs to buffer this data). When the portal receives the raw data from the mobile proxy, it verifies that the TSP signed them, then recomputes the aggregate using the aggregation function, and finally compares the computed aggregate with the one already sent. If the comparison fails, the portal can choose to reject further transmissions from this lying proxy. This is similar to the concept of bit-commitment introduced by Chaum et al. [27] where one commits to a value before it is checked for correctness.

The pseudo-random challenge forces the mobile proxy to guess when it is safe

to fabricate an aggregate. Eventually, a lying proxy will guess wrong and get caught. Lying more only causes the proxy to be detected faster and lying less only delays that outcome. The following equation represents the expected number of aggregates $E(n)$ a portal accepts, when challenging aggregates with probability q, before detecting a proxy lying with probability p.

$$E(n) = \sum_{n=1}^{\infty} n \times (1 - pq)^{n-1} \times pq \tag{3.1}$$

We evaluate STAP in Section 5.9 and show that the experimental results closely match the analytical one above.

3.6 IMPLEMENTATION

In this section, we discuss our TSP prototype and the implementation of STAP.

3.6.1 Trusted Sensing Peripheral (TSP)

We use the secFleck [60] as the foundation of the TSP. The secFleck is the portion of the TSP consisting of the Atmel TPM chip (based on v1.2 of the Trusted Computing Group specification [132]), and the Fleck sensor board (see Figure 3.3).

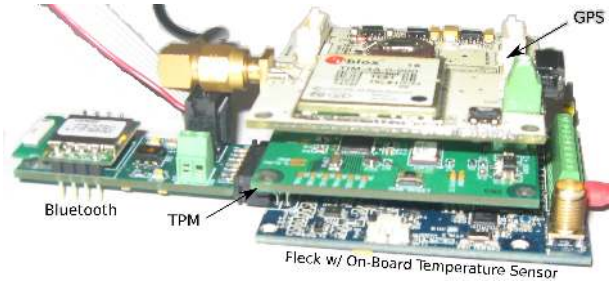


Figure 3.3: Trusted Sensing Peripheral with a Bluetooth and GPS module.

The Fleck is a sensing platform with 8 KB of memory and an 8 MHz Atmega micro controller [117]. It houses the TPM module, and is extensible with various

sensors. The TSP firmware, including the FleckOS, sensor device drivers, and our application, runs on the Fleck hardware.

Figure 3.3 shows the TSP. Attached to it, is a Parani-ESD Bluetooth module. The mobile proxy application communicates with the TSP using a local, intermediary Python relay service called `foslisten`. `foslisten` communicates locally using TCP sockets, while with the TSP, it uses a serial-over-Bluetooth channel (Figure 3.4).

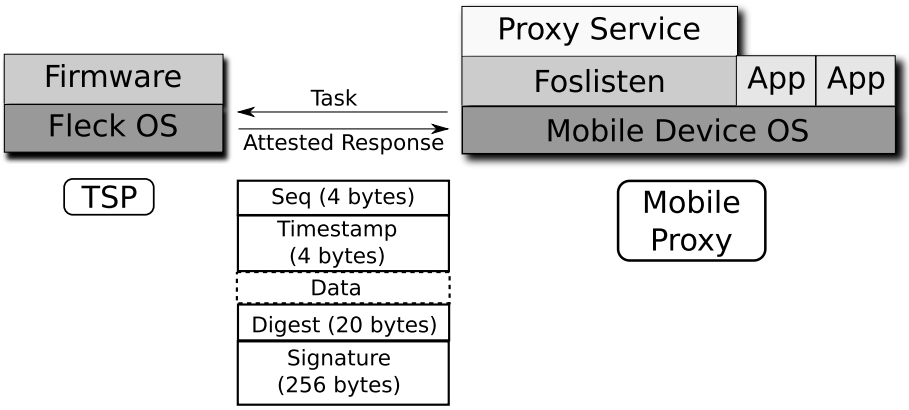


Figure 3.4: Data producers on the mobile proxy communicate with the TSP via a Python relay service called 'foslisten'.

3.6.2 Online Portal

The online portal is responsible for tasking the TSP, verifying the TSP's platform integrity and the integrity of any data received from it, requesting the mobile proxy to aggregate data if necessary, and pseudo-randomly challenging the mobile proxy to prove its trustworthiness when aggregated data is received. The portal runs as a standard multi-threaded TCP server on a 2.2 GHz Intel(R) Core(TM)2 Duo platform with 1.9 GB of memory running Linux kernel version 2.6.32.10.

3.6.3 Mobile Proxy

The service running on the mobile proxy is responsible for receiving a task from the portal, in turn tasking the TSP via the Bluetooth channel using our custom RPC protocol, retrieving data from the TSP, and forwarding that data to the portal. The mobile proxy is a Nokia N800 tablet with 128 MB of memory running Linux kernel version 2.6.21 on an ARMv6 processor. Although we use the N800, a widely popular tablet, our work is applicable to any smart phone or tablet device.

3.6.4 Secure Tasking and Aggregation

The details of STAP are shown in Figure 3.2. Here, we limit our discussion to how the TSP is tasked. The portal first sends a task description $[S, c, t, f]$ to the mobile proxy, where, S is the set of sensors to collect data from, c is the number of data samples to collect, and t is the time (in seconds) within which to collect the c samples (sampling interval is thus t/c seconds). S is expressed using a 16-bit mask, allowing sixteen sensors to be tasked simultaneously with the same sampling interval. Each set of data samples are reported in an attested response message signed using AIK_{priv} (see Figure 3.4), and can be verified by the portal using AIK_{pub}. A mobile proxy modifying the task description will be detected when the portal verifies the one echoed back by the TSP in the first attested response.

3.7 EVALUATION

We first evaluate the performance of the TSP and then evaluate STAP in Section 3.7.2.

3.7.1 TSP Performance

We analyze the performance of the TSP in terms of *time* and *energy* required to perform and transmit data attestations. Then, we discuss other costs involved in

building the TSP: code size, memory usage, and monetary cost. The TSP runs on three 1.5 V batteries with a 2500 mAh capacity each and is configured with two sensors: an on-board temperature sensor, and an attached GPS sensor (Figure 3.3).

Timing Measurements

The TSP is made to repeatedly perform and return forty attestations, each over a set of temperature data samples varying in size from 2 to 92 bytes at 10 byte intervals. Average time spent performing attestations, along with the 95% confidence interval, is shown in Table 3.1. The small confidence interval due to the dominant RSA signature operation that is always performed over a fixed size SHA-1 digest of the data. Table 3.1 also shows the average transmission time of the signed data samples. As we can see, the TSP can only sample, attest, and report data at intervals greater than ≈ 2 seconds. This lower limit on sampling and reporting is more than acceptable for existing crowd-sourced sensing systems, most of which, require data samples less frequently than that [62, 106, 42, 9].

Task	Compute Time (sec)	Transmit Time (sec)
Single Attestation	1.72 ($\pm$ 0.01)	0.3 ($\pm$ 0.1)

Table 3.1: Average time required to perform and transmit an attestation.

Energy Measurements

We measured the TSP's current draw in different energy states using an oscilloscope with an internal resistance of $10M\Omega$ across a 1Ω resistor placed in series with the TSP. Table 3.2 shows the average current drawn by the TSP while it was idle, performing attestations, and performing transmissions.

Energy State	Current Draw
Idle	80 μA
Attesting	50 mA
Transmitting	42 mA

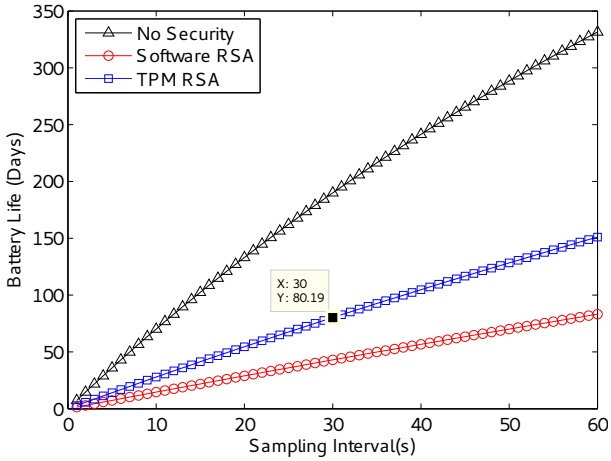
Table 3.2: Average current drawn in various TSP energy states.

Using the timing and current draw measurements, we computed the energy consumed by the TSP while collecting, attesting and transmitting a 2 byte sample of data at various intervals. Figure 3.5(b) compares energy consumption of the TSP with the TPM, without any security, and when the TPM's operations are performed in software [60]. Hardware attestations end up being half as expensive as those performed in software, because, although they draw more current, they can be computed much faster. However, the TPM requires three times more energy than if there were no security at all.

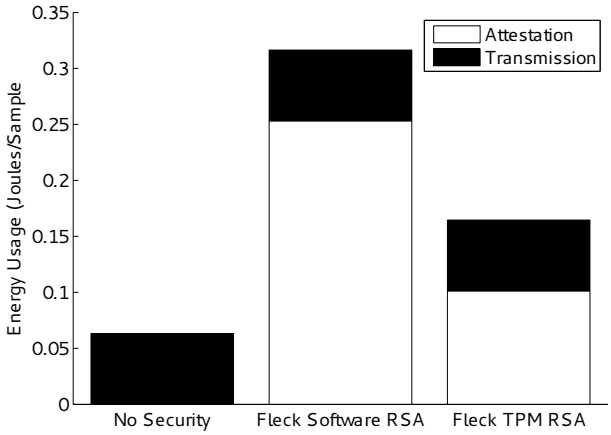
We also computed an estimate of the TSP's battery life. Figure 3.5(a) shows that when tasked with sampling, attesting, and transmitting a 2 byte sample every 30 seconds and remaining idle in-between, the TSP can achieve a battery life of over *80 days*. This is quite sufficient for crowd-sourced sensing, where participants will carry and eventually recharge these platforms. Also, notice that battery life is nearly double than when all security is in software.

Other Costs

Figure 3.6 compares compiled code sizes of various software components running on the TSP. The code size of our application is smaller than both the TPM library and the FleckOS. Memory (RAM) used by the firmware is approximately **4.2 KBytes**, which is a little over half the available memory (8 KBytes) on the current version of the Fleck. Monetary costs involved are currently high: the cost of the TSP hardware is approximately 300 USD. The TPM chip itself is inexpensive, about **6**



(a) TSP battery life.



(b) TSP energy usage.

Figure 3.5: Battery life and energy usage of the TSP.

USD when purchased in large quantities (≥ 1000).These monetary costs are higher due to the limited scale at which our TSP is currently produced; with economies of scale we believe that the cost could be brought down to lower than 50 USD.

3.7.2 Secure Tasking and Aggregation Protocol

We conducted experiments to answer the following questions about our protocol: 1) How many fabricated aggregate values are accepted by the portal before a lying mobile proxy is detected? 2) How long does it take for the portal to find a lying

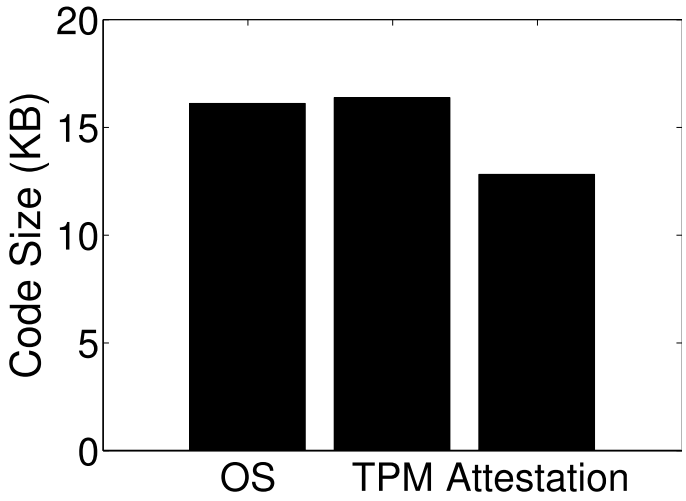
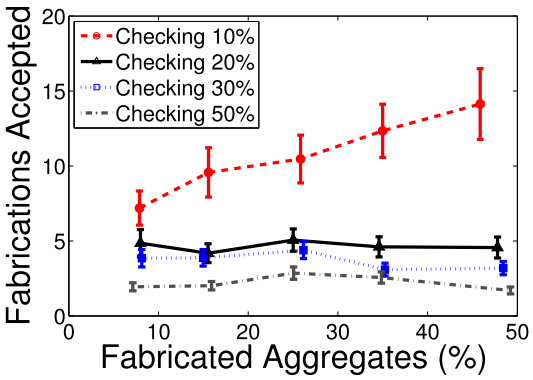


Figure 3.6: Compiled code size comparison of various software components on the TSP.

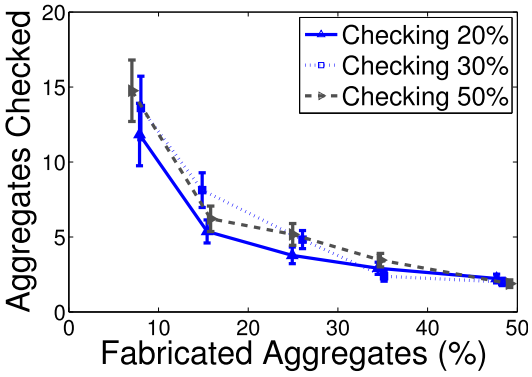
proxy? and, 3) What is the overhead of detection?

In each experiment, the mobile proxy is configured to randomly fabricate (or lie about) an aggregate with probability ranging between 1/10 to 1/2 (or 10% to 50% of aggregates). Here, fabrication involves adding a random number between 1 and 10 to the result of the aggregation function. The online portal then pseudo-randomly challenges the integrity of a received aggregate with the same probabilities. Aggregations are performed on every 10 two-byte samples of temperature data. The aggregation function f performed by the proxy is a *mean* of those samples. An experiment continues until the portal detects the first fabricated aggregate value. When necessary for clarity, we omit data points that don't add significantly to the information conveyed by the graphs.

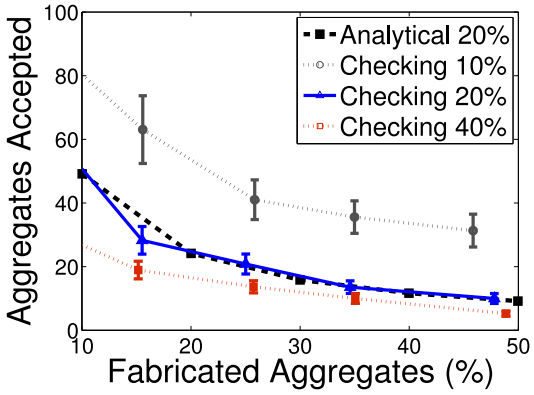
Figure 3.7(a) shows that the portal can quickly and efficiently detect a lying proxy. While pseudo-randomly challenging only 20% of the aggregates, the malicious proxy is detected within the first *six* fabrications received — no matter how much it lies. The advantage of challenging more than 20% of the aggregates is not significant, thus, this threshold reasonably trades off the integrity of aggregates



(a) Damage to data integrity



(b) Overhead of detection



(c) Aggregates accepted before detection

Figure 3.7: Performance of the secure aggregation protocol STAP

with the overhead of challenging.

Figure 3.7(c) shows how long it takes the portal to detect the malicious proxy in terms of the number of aggregates (false or otherwise) accepted before detection. Not surprisingly, the less a proxy lies the longer it takes to detect it. We also plotted Equation (5.1) with the portal's checking probability q set to 1/5 (or 20% of the aggregates). It can be seen that the analytical plot is within the confidence intervals of the experimental one. Notice also, the large drop in aggregates accepted when challenging 20%, rather than 10% of them.

Figure 3.7(b) shows the detection overhead in terms of the number of challenges issued before detecting a lying proxy. Surprisingly, the overhead largely depends

only on how often the proxy lies. Figure 3.7(c) provides an intuitive explanation: although the number of aggregates accepted before detection varies significantly, the number of challenges issued does not. This result works in favor of the portal, which can now minimize the number of challenges based on factors like reducing the number of fabrications accepted.

3.8 THREAT ANALYSIS

We now revisit our threat model and describe how our system addresses those threats to the TSP and mobile proxy.

3.8.1 Threats to the TSP

A majority of the threats are addressed as a consequence of the TPM's special capabilities, namely, a *sealed private key* and the property of being *infeasible to replicate*.

- *Software modification.* Such attacks are mitigated by building the TSP using a modified harvard-architecture based platform that provides strict isolation between program and data memory, and permanently disabling all methods that can possibly alter the TSP's firmware.

- *Masquerading.* The TPM's sealed private key ties the identity of the TSP (represented by that key) with the hardware platform. Since the TPM cannot be cloned, and its private key cannot be extracted, the adversary has no way to masquerade as a TSP.

- *Communications Compromise.* The portal can detect modification or injection of data because an adversary without the TSP cannot fabricate its signatures. Replay attacks can be detected because each data sample generated by the TSP has an incremental sequence number.

3.8.2 Threats to the Mobile Proxy

The portal uses STAP to detect a mobile proxy fabricating aggregates. The key idea is that a mobile proxy *commits* to the aggregate value by the very act of sending it to the portal. The portal then pseudo-randomly verifies the integrity of that commitment. A lying mobile proxy may be able to publish some fabricated aggregates (see Figure 3.7(a)), but will eventually get caught.

3.9 RELATED WORK

Existing research on crowd-sourced sensing does not address the data integrity problem the way we define it, namely, *"How can a data portal — receiving data from sensors not under its control — trust that the data is a true representation of the real-world phenomenon being sensed?"*.

Research on traditional sensing, however, does suggest some approaches that might be adapted to solving the above problem. For example, a reputation-based framework implemented at the portal could identify and ignore data from sources with a low reputation [48]. A source is assigned a low reputation as long as it generates outlying data samples when compared to others in its neighborhood. The reputation is actually assigned by neighboring sensors that also sense similar data. Even though this network architecture may not be practical for highly mobile crowd sensors, it also does not provide protection from Sybil attacks, where an adversary could create any number of virtual sources and use them to artificially raise a given source's reputation. A reputation based system will also be more effective in detecting faulty sources than malicious ones, reason being, malicious sources will try to *avoid* detection by publishing fabricated, but not necessarily outlying data [48]. Among other issues, reputation based frameworks require redundant sources of data to detect outliers, and are specific to the type of data being collected.

Another approach, also from traditional sensing, involves filtering fabricated

or anomalous data that is injected into the sensor network [150]. Data that at least $t + 1$ local sensors don't agree on, or data coming from unauthentic sensors is considered fabricated. The system is thus resilient to adversaries that have compromised at most t sensor nodes. Since each sensor in the network can be authenticated, the sensor network is resilient to Sybil attacks. However, the assumption is that each sensor node already shares a key with the authenticator (the base station). This assumption might be realistic for traditional sensor networks, but is not realistic for crowd-sourced sensing systems where the data portals and data producers (participants from the crowd) are different autonomous entities. It would be unreasonable to expect that every potential data portal has a preexisting secret with every potential data producer. The Sybil attack, could thus be launched by a data producer *during the time* a shared secret is established with a new data portal. Another disadvantage is that software attacks, where data is modified after being sensed, can be detected only when less than $t + 1$ nodes have been compromised.

An orthogonal approach, as opposed to those discussed above, and closer to what is presented in this chapter, is for data portals to use systems like Pioneer or SWATT (SoftWare-based ATTestation) to externally verify the code executing on a remote data producer's platform [113, 114]. With assurance in the producer's sensing platform, the portal will be inclined to trust data from it. Unlike our trusted-hardware approach, both SWATT and Pioneer are software-based systems that challenge the remote platform to compute a digest of its memory contents. The digest is computed by traversing the memory in a pseudo-random fashion as determined by the random challenge. The challenger is expected to know the memory contents of the remote platform beforehand and can therefore compute the correct digest independently for verification later in the protocol. A remote platform could fool the challenger by separately storing the original contents of the modified portions of memory, allowing it to still provide the correct digest.

However, since the traversal is pseudo-random, the malicious platform cannot know before hand the order in which the memory will be checked. As a result, the attacker will have to check each memory access to see if the addresses matches one of the modified ones. This extra check will cause the digest computation operation to take longer than expected, causing the challenger to become suspicious. Notice though, that both SWATT and Pioneer cannot prevent a Sybil attack from an adversary that has abundant computational resources. The attacker could *simulate* the remote platform on more powerful hardware while still meeting the expected response time. Since the response time of the remote platform is an estimate based on the hardware configuration and the expected communication latency, SWATT and Pioneer only provide a *probabilistic guarantee* of the remote platform's integrity. Estimating response times may itself be a daunting task.

Our protocol is inspired by an approach in traditional sensing called Secure Information Aggregation (SIA) [103]. However, unlike our approach, SIA assumes that the sensors are trustworthy and does not provide an implementation of the protocols. Our framework also makes it easy to incorporate other SIA algorithms that are more suitable for the respective aggregation function.

3.10 DISCUSSION

We recognize the use of the TPM cannot protect against all failure modes. For example, sensor measurements may be inherently corrupt, sensors maybe damaged, or a sensor's environment may be doctored. It is also likely that not every user has a Trusted Sensing Peripheral. In this case, the crowd-sourced sensing system could still collect data from untrusted sensing platforms, but then use the data from trusted peripherals to calibrate or validate the untrusted data. We plan to address a solution along these lines in future work.

The types of aggregation functions addressed in this chapter, work on discrete

windows of data. Thus, calculating metrics like the median would require a different aggregation protocol. Nonetheless, such a protocol could easily be integrated within our current framework.

The TSP may duplicate some functionality of in-built smart phone sensors such as audio and GPS, however, it also enables the addition of a wide range of new sensors such as CO, CO_2, humidity, temperature, seismic, and medical sensors. For example, carbon monoxide studies in Ghana, Accra used an additional device attached to the phone [97]. Furthermore, the TSP can be connected to a legacy mobile device, as such it is possible to deploy a trustworthy crowd-sourced sensing application without the cooperation of smart phone OEMs such as Apple or Nokia.

3.11 CONCLUSION

This chapter presented the design and implementation of a Trusted Sensing Peripheral (TSP) to provide data authenticity and integrity for crowd-sourced sensing systems. The TSP has built-in sensors, and a Trusted Platform Module (TPM) that helps it resist *software* and *Sybil* attacks to its platform. The TPM attests to the integrity of published data at the source, and this attestation is verified at a remote server. The TSP is energy-efficient, with a battery life of over *80 days* when collecting a temperature sample every 30 seconds.

Our secure data aggregation protocol STAP allows an untrusted intermediate mobile proxy to aggregate the TSP's raw signed sensor readings, while allowing the data portal to detect a malicious proxy that fabricates those aggregates. The portal can detect a lying proxy with very little overhead, and can do so within the first *six* fabricated aggregates received.

Chapter 4

HIGH-INTEGRITY PRIVACY-PRESERVING CROWD-SOURCED SENSING

Integrity of the received data and the privacy of the data sources remain first order concerns for crowd-sourced sensing systems. Unfortunately, people will be reluctant to volunteer sensitive information (e.g. location, health statistics) if they cannot trust the system to protect their privacy. Conversely, if the volunteered information is first modified to protect privacy, then the system will be reluctant to trust that modification without proof of its integrity.

Consequently, integrity and privacy compete with each other. If the collected data has been previously transformed to preserve privacy (e.g. mixed, aggregated), then a data consumer cannot determine the transformation's integrity unless the raw data used as input is presented as well. However, if the raw data is presented, then the privacy of the data sources gets compromised. This chapter describes how the trust-but-verify approach can be used to provide *simultaneous* data integrity and privacy guarantees without significantly compromising either.

4.1 APPLYING THE TRUST-BUT-VERIFY APPROACH

In the previous chapter, we described a verification function that enabled crowd-sourced sensing applications to determine the integrity of data received from individual sources. Recall however, that the information collected by the trusted sensing peripheral was sent to the consumer as-is, implying that the data sources had no privacy.

In this chapter, we develop a verification function that in addition to guaranteeing data integrity preserves privacy as well. This is done by involving a trusted intermediary that aggregates the data from multiple volunteers in a way that preserves privacy. In this case, the *generation function* is the aggregation performed by this intermediary and the *data* to verify is the output of this aggregation function. The *verification function*, then, must allow the consumer to check the integrity of this aggregate without needing the original data used to compute it. The key idea is to use a homomorphic commitment scheme [98] that allows the consumer to compute the same aggregation function over commitments to the original data. The result is then checked against a commitment to the initially received aggregate. A volunteer's data remains private because the respective commitments do not reveal any information about the data.

4.2 CONTRIBUTIONS

This chapter describes *LocationProof*, a protocol that enables a crowd-sourced sensing application to establish the integrity of location-based data while preserving the privacy of those who collected that data. The focus is mainly on location-based information since location privacy has become a fundamental concern in crowd-sourced sensing applications [43, 61, 49, 71]. It has been shown that anonymizing an individual's location traces are not enough to preserve privacy since they can be used to infer where a person lives and works [73, 148, 70, 146]. Moreover, Montjoye et al. [32] showed that among half a million anonymized traces at *cell-tower* granularity, just four spatio-temporal points were enough to uniquely identify the traces of 95% of individuals.

4.3 SYSTEM MODEL

The system model assumes a set of trusted data sources that collect and forward sensory data to an *aggregator*, which aggregates the received data, and finally forwards the result to a data consumer. The goal is to assure the data consumer that the proxy indeed computed the expected aggregation function using data from expected sources (integrity) without providing the consumer with that data (privacy).

Much of the existing work on crowd-sourced sensing, with a focus on integrity and privacy, adheres to this model. Examples include PoolView [49], which introduces the personal privacy firewall to perturb a user's raw data before publishing it to an aggregation service; DietSense [106], which provides private storage to a user where she can edit the images collected from her phone before sharing it further, AnonySense [71], which uses a trusted server to mix data from at least k clients to provide k-anonymity; and our earlier work on the design and implementation of a Trusted Sensing Peripheral (TSP) that produces and publishes trustworthy sensory information to a data portal via the user's personal mobile device [38].

Figure 4.1 describes the system model more formally. The crowd-sourced sensing system involves a set of sources $S = \{s_1, s_2, ..., s_n\}$ contributing data to an aggregator A, which aggregates the received data and forwards it to a consumer C. The data for a source i includes a tuple (x_{ij}, y_{ij}, d_{ij}) consisting of the source's location coordinate (x, y) and associated sensory data d (e.g. temperature) during interval j. The aggregation function computed by A is the mean $(\bar{x_j}, \bar{y_j}, \bar{d_j})$ of respective tuple elements contributed by groups of at least k users in each interval j (note that $n \geq k$). The sources are assumed to be contributing data as part of a campaign that defines the anonymity parameter k, the data collection intervals j, the type of data d to collect, and a rectangular region $R = [(x_{min}, y_{min}), (x_{max}, y_{max})]$

to collect the data in. The goal is to enable C to validate the result of the aggregation that A is performing, without learning the data that was aggregated.

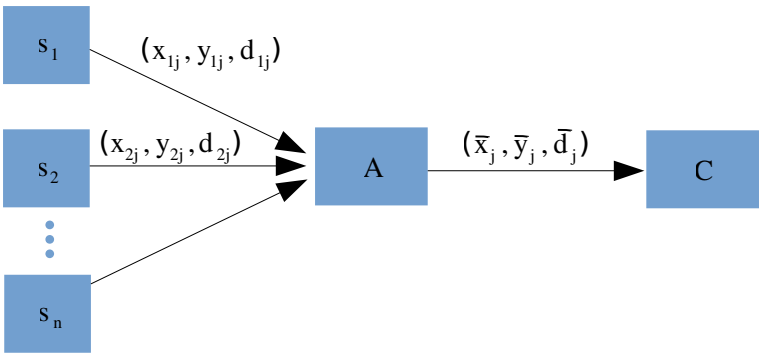


Figure 4.1: System model for privacy-preserving high-integrity crowd-sourced sensing

4.4 BACKGROUND

The privacy LocationProof provides depends on several fundamental concepts: k-anonymity, the *Hilbert Cloak* [52], the Pedersen homomorphic commitment scheme [98], and the Trusted Sensing Peripheral (Chapter 3).

4.4.1 k-anonymity

k-anonymity is a concept first introduced by Samarati and Sweeny [111] to protect the privacy of individuals when disclosing person-specific information (e.g. medical data for research purposes). Such information is assumed to be released in the form of a table where each row contains the data pertaining to one individual. The basic idea is that any person's record must be made indistinguishable from the records of at least $k-1$ others by suppressing or generalization the values of individual fields in the released data set. Thus, given only the released data, the probability that a given record belongs to a particular individual is at most

$1/k$. Gruteser and Grunwald [57] applied this idea towards anonymous usage of location-based services. The goal is to ensure that information requested for any region contains at least k requestors. Such a region is called a *location cloak* and it mitigates the severity of *inference attacks*: performing reverse-location lookups on an individual's location traces to re-identify them. Since k-anonymity ensures that any trace contains at least k people, a reverse-lookup reveals information that could correspond to any of the k individuals.

4.4.2 Hilbert Cloak: k-anonymous Location Cloaking

There are many strategies for computing a location cloak in a way that preserves k-anonymity. Consider our system model, where data sources are periodically collecting location-based information in a region R. In this scenario, we could achieve k-anonymity by computing a cloak that includes all users in the region as long as there are more than k of them. However, this only gives us one aggregated data point per interval for the entire region R regardless of its size. For more fine-grained data, it is desirable to create multiple location cloaks, each of which, are large enough to preserve k-anonymity. Gkoulalas et al. [52] provide an exhaustive analysis of the various strategies that can be used. One of the more efficient and secure algorithms is the *Hilbert Cloak* by Kalnis et al. [70]. The *Hilbert Cloak* translates a two-dimensional location coordinate into a one-dimensional distance on the Hilbert space filling curve. It has been shown that points close-by in 2D space are also close together on the Hilbert curve with high probability [88].

One can get an intuitive understanding of this result from Figure 4.2, which shows how points on the Hilbert curve represent points in 2D-space. Once the aggregator A has converted each location into a point on the Hilbert curve, it can sort those points and group them sequentially into buckets containing at least k sources each. Then, the region occupied by each group of k or more sources represents the group's location cloak.

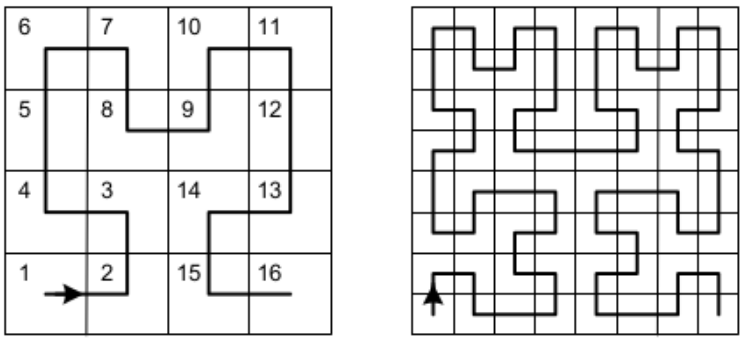


Figure 4.2: The Hilbert curve (left: 4×4, right: 8×8). Source: Kalnis et al. 2007 [70]

4.4.3 Homomorphic Commitment Scheme

As depicted by the system model in Section 4.3, the aggregator A computes the mean location (representing the cloak) for each group of k data sources and sends it to consumer C. C may then request A to prove that the mean was computed correctly using only the locations from sources in S. This work describes a protocol that allows A to do just that but without ever revealing the sensitive data source locations that were used as input. The protocol employs Pedersen's homomorphic commitment scheme [98] as a fundamental building block.

A commitment is a function of a value that does not reveal anything about that value. For example, $E(v, l)$ is a commitment to the value v using key l. The commitment can later be verified when the actual value v and key l are revealed. Verification involves computing the commitment function over the revealed value and comparing with the previously received commitment. Pedersen's commitment scheme has one other property: it is *additively homomorphic*, that is, it allows computing over commitments to values v_1 and v_2 using an operation $\odot$ such that $E(v_1, l_1) \odot E(v_2, l_2) = E(v_1 + v_2, l_1 + l_2)$.

In Pedersen's scheme:

$$E(v, l) = g^v h^l$$

and operation $\odot$ is multiplication. Here, g, h are publicly known elements of a group G_q and v, l are elements of $\mathbb{Z}_q = \{0, 1, ..., q-1\}$. Given two large primes p and q where q divides $p-1$, the group G_q is a subgroup of $\mathbb{Z}_p$ such that

$$a \in G_q \iff a^q = 1 \text{ and } a \in \mathbb{Z}_p$$

Notice that any element of G_q (except 1) generates the group and thus, for any two elements $a, b \in G_q$ the discrete logarithm $log_b(a)$ is defined. The assumption is that given $a, b \in G_q$, finding $log_b(a)$ is computationally intractable.

4.4.4 Trusted Sensing Peripheral (TSP)

Pedersen's additive homomorphic commitment scheme allows C to check if A is correctly aggregating the data. However, this still does not address the problem of how C can determine that the location-based information collected by the data sources accurately reflects their environment. Malicious sources could easily fabricate the location-based data without C knowing. In this work, we assume that either the sources in S are implicitly trusted, or use TSPs to perform the sensing. When TSPs are used, the sources could prove the integrity of the platform using the simple method described in Section 4.6. This way C would know that the data came from a trustworthy source, and along with A's proof that it was processed correctly. We will see in Section 4.5 that to preserve privacy a TSP must prove the integrity of its platform while remaining anonymous. It can do so using an anonymous signature scheme called Direct Anonymous Attestation [22] supported by its Trusted Platform Module [134] (the embedded secure co-processor that forms the root-of-trust for the TSP).

4.5 THREAT MODEL

Our protocol is designed to prevent a malicious C from compromising a source's location privacy, and prevent a malicious A from either injecting fabricated data or computing an aggregation function other than the mean. Since data consumer C has a vested interest in the integrity of the received information, it is assumed to be an adversary of privacy, but not of integrity. Thus, given any locally available protocol state information and the aggregated data from A, we would like to prevent C from recovering a source's raw data, or hone in on its location. C may narrow down a source's location using side-channel information it may already know about a particular source. Consider the *intersection attack* [70, 34] where C knows that a particular source s_i participates in data collection and that s_i is in some general area A at time t. Now, if at the same time, C discovers another area A' where s_i is, then the intersection of A and A' reveals a smaller area containing s_i. These attacks succeed even if areas A and A' preserve k-anonymity. The root cause is a lack of strong initial anonymity: notice that the consumer C is able to link two physical regions because they are associated with some identifying information about source s_i first revealed by the application itself. Such identifying information could include user IDs used during a registration phase, or IP addresses that remain the same across multiple data submissions. Thus, keeping a source anonymous during all steps of the LocationProof protocol is critical to preserving data source privacy.

The privacy proxy A is considered to be an adversary of integrity, but not of privacy. A is not considered an adversary of privacy because we assume that A is either controlled by the sources [49] or is implicitly trusted by them to protect their privacy [85, 71]. However, now, C would like to know that the data it is interested in, is being processed correctly. Thus, from C's perspective A is considered an adversary of integrity. An implication of the above adversarial model is that A

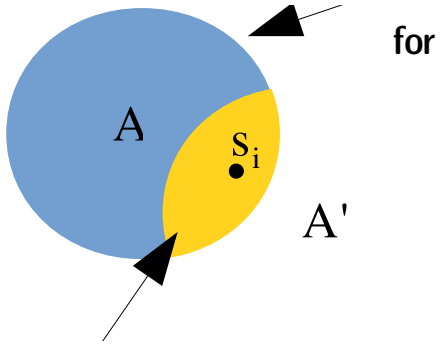


Figure 4.3: The intersection attack: given some previous estimate o
location and any new estimate, the intersection of the two reveals a
of s_i's location.

and C do not collude in any way.

We do not consider threats from eavesdropping adversaries since
work security protocols, like TLS, can easily be used to combat ther
not consider situations in which A, C, and S do not interact since
that neither side suppresses a response expected by the other. Furth
S is trusted; either implicitly or by the use of a Trusted Sensing Per
scribed in Chapter 3, and that its origin is anonymized when commu
C via a mix network like Tor [35]. Sources must also be capable
anonymous signatures like those described in group signature schem
use Direct Anonymous Attestation [22] when TSPs are employed for

4.6 THE PROTOCOL

LocationProof enables clients to send a response that allows C to determine if the means were computed correctly, while not revealing anything about the inputs used to compute them. More specifically, each source s_i sends *homomorphic commitments* to functions of its inputs. Then, C computes commitments to the initially received means, and checks if they equal to the means computed using commitments from the data sources. For simplicity, Section 4.6.2 first explains the protocol using only k data sources, implying that data from all k sources is aggregated in each interval. Later, in Section 4.6.3, we elaborate on how the protocol can be modified to provide integrity guarantees for data collected from $n \geq k$ sources.

4.6.1 Requirements

The following is required from the LocationProof protocol:

1. The data consumer is never provided with the raw data (x_{ij}, y_{ij}, d_{ij}). This is the basic privacy requirement.

2. The data consumer should not have any identifying information about the data sources. Otherwise, the consumer could execute an intersection attack to narrow down the location of a source.

3. The data consumer should be able to determine:

   - that the inputs used to compute the means were provided by sources in S
   - that indeed the means as opposed to any other functions were computed on those inputs.

s_i	**A**	**C**
interval j:		
sense x_{ij}		
$s_i \rightarrow A$: x_{ij}		
$b = x_{ij}$	$x_j = \sum_{1 \leq i \leq k} x_{ij}$	
	$A \rightarrow C$: x_j	
		$\bar{x}_j = x_j / k$
interval $j+1$:		
sense $x_{i(j+1)}$		
$s_i \rightarrow A$: $x_{i(j+1)}$		
$b = x_{i(j+1)}$	$x_{j+1} = \sum_{1 \leq i \leq k} x_{i(j+1)}$	
...	...	...

Table 4.1: Normal operation: sources in S collect data and send it to A. A aggregates the data and forwards the result to C. Please note that protocol steps below occur at a later time than those above. The notation $e_1 \rightarrow e_2 : m$ implies that entity e_1 sends message m to entity e_2.

4.6.2 With k Data Sources

We assume in this section that the number of data sources $n = k$. This allows us to defer the discussion of how n users are split up into groups of k for preserving k-anonymity. Also, we focus on verifying the integrity of one element, say x_{ij}, from the tuple of data collected by each source. The same verification process must be repeated for each tuple element.

Table 4.1 describes the "normal operation" of a data source s_i, the privacy proxy A, and data consumer C. Here, normal operation represents the activities of all entities in intervals where C does not challenge A to prove the integrity of the aggregated data. After collecting data, a source s_i buffers that data for one

additional interval. As shown in Table 4.2, this is required because the sources learn about C's challenge for data sent in interval j during interval $j+1$. Thus, the sources must be prepared to prove the integrity of data they collected in the last interval. On receiving each source's data, A sums them all and sends the total to C. C can then use the sum to compute the mean $\bar{x}_j$. The reason that A sends the sum (an integer) rather than the mean (possibly non-integer) is to avoid passing around fractional numbers: numbers that cannot be committed to under the Pedersen scheme [98] used in our integrity verification algorithm.

We now discuss the privacy-preserving data verification function at the heart of LocationProof. Let's say data consumer C challenges A during some interval j. In turn, A informs all the sources to prepare a response for data collected in interval j after it receives their data for interval $j+1$. As shown in Table 4.2, each source responds to A with a random number l_{ij}, a commitment $E(x_{ij}, l_{ij})$, and an anonymous signature σ_{ij} over the commitment $E(x_{ij}, l_{ij})$ to x_{ij}.

The random number l_{ij} serves as a *key* to the commitment scheme for source s_i during interval j. Also, the anonymous signature σ_{ij} is obtained by computing $Sign(gpk, gsk[i], E(x_{ij}, l_{ij}))$. Here, *Sign* represents the signature computation function of a group signature scheme [21, 22], gpk the group's public key, and $gsk[i]$ the corresponding secret key for source i. A group signature scheme preserves the anonymity of signer by not revealing the specific group member that created the signature. A signature can be verified using $Verify(gpk, E(x_{ij}, l_{ij}), \sigma_{ij})$. If verification succeeds, then the verifier learns that *someone* in the group created the signature, but not who in particular.

A then sums up the keys and sends the result l_j along with all the commitments and signatures to C. Here, l_j represents C's key to the commitment scheme. C then verifies all the signatures, computes a commitment $a_1 = E(x_j, l_j)$ to the sum x_j, a commitment $a_2 = E(x_{1j}, l_{1j}) \odot \ldots \odot E(x_{kj}, l_{kj})$ using the commitments from each source, and checks if $a_1 \stackrel{?}{=} a_2$. If this check fails or any of the signatures can't

be validated, then the consumer rejects the data x_j.

s_i	**A**	**C**
interval j:		
...	...	...
		$\bar{x}_j = x_j/k$
		$C \rightarrow A$: prove $\bar{x}_j$ is correct
interval $j+1$:		
...	...	...
$s_i \rightarrow A$: $x_{i(j+1)}$		
	$A \rightarrow s_i$: prepare response	
pick $l_{ij} \in \mathbb{Z}_q$		
$x_{ij} = b$		
$\sigma_{ij} = Sign(gpk, gsk[i], E(x_{ij}, l_{ij}))$		
$s_i \rightarrow A$: $l_{ij}, E(x_{ij}, l_{ij}), \sigma_{ij}$		
$b = x_{i(j+1)}$	$l_j = \sum_{1 \leq i \leq k} l_{ij}$	
	$A \rightarrow C$: $l_j, \{E(x_{ij}, l_{ij}), \sigma_{ij} : 1 \leq i \leq k\}$	
	$x_{j+1} = \sum_{1 \leq i \leq k} x_{i(j+1)}$	$a_1 = E(x_j, l_j)$
	$A \rightarrow C$: x_{j+1}	$a_2 = \odot_{i=1}^{k} E(x_{1j}, l_{1j})$
		$a_3 = \wedge_{i=1}^{k}$ *Verify*$(gpk, E(x_{ij}, l_{ij}), \sigma_{ij})$
		if $a_1 \neq a_2 \vee \neg a_3$ **reject** $\bar{x}_j$

Table 4.2: Protocol steps executed by each entity after C challenges A to prove the integrity of data received in some interval j. As before, time increases from top to bottom.

4.6.3 With $n \geq k$ Sources

When the number of data sources $n \geq k$, the privacy proxy A needs to determine which users to group together. Once the sources have been grouped, the LocationProof protocol described in Section 4.6.2 must be repeated for each group. A could always group all sources together, but then, there would only be one data point $(\bar{x}_j, \bar{y}_j, \bar{d}_j)$ during each interval j for the data collection region R regardless of how large it was. For more fine-grained location-based information, A needs to cluster as few close-by users as possible. More specifically, to preserve k-anonymity, A must place at least k users in each group. Gkoulalas et al. [52] provide an exhaustive overview of algorithms that cluster close-by data sources into groups of k or more. The clustering protocol assumed by LocationProof is HilbertCloak [52], and we have described it in Section 4.4. Eventually, when C challenges the integrity of data received in some interval j for a given k-group, the LocationProof protocol for that group continues as described in Section 4.6.2.

4.7 IMPLEMENTATION

We have simulated each of the components of LocationProof in Java. Data sources are assumed to be participating in a data collection campaign to gather temperature data in an area around downtown Portland, OR. The region R where sources simulate data collection is shown in Figure 4.4. In each interval, sources pick a random temperature at a random location in region R and send it to the aggregator. When C challenges the integrity of an aggregate, sources send the homomorphic commitments to the data they collected in the corresponding interval.

The Pedersen homomorphic commitment scheme [98] was implemented using the Java Qilin library [89]. Since the commitment scheme only works with integers, a latitude x in region R was converted to an integer offset from the base using the

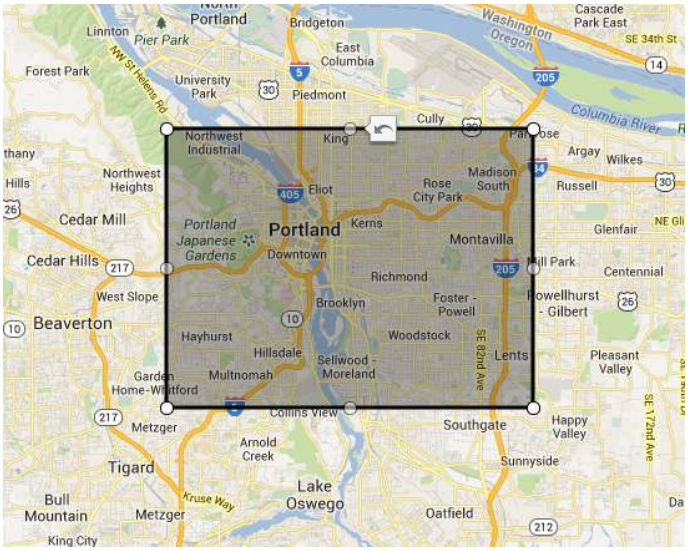


Figure 4.4: Data collection campaign simulated for a region around downtown Portland, OR. $R = [(-122.752°, 45.455°), (-122.556°, 45.560°)]$

following formula:

$$(x - x_{min}) \times 10^6$$

A longitude in the region R was converted similarly. Both latitudes and longitudes were represented using 6 digits of precision in our implementation.

4.8 SECURITY ANALYSIS

As discussed in Section 4.5, LocationProof must be capable of preventing a malicious C from compromising the privacy of data sources using the *intersection* or *inference* attacks. LocationProof prevents these attacks by ensuring that sources remain anonymous from C's perspective at all times. There is no registration phase and all data submissions happen via aggregator A. Furthermore, when sending the response to C's challenge, each source only sends commitments to their locations rather than the locations themselves. The signatures on the responses are also anonymous. Thus, C never obtains any linkable identifying information from the

application about sources across submissions.

Another threat LocationProof considers is the one to data integrity from aggregator A. LocationProof prevents A from injecting inputs to the aggregation function, or computing a different aggregation function altogether. This is done by enabling C, the party most interested in the integrity of the aggregate, to challenge the result of the aggregation function. Using the homomorphic commitment scheme, the data sources help C ensure that A is computing the right function using the right inputs. C has the choice of checking every aggregate for correctness or some fraction of them. However, when checking only a fraction, C may miss some fabricated aggregates before eventually detecting a malicious aggregator. We analyze how long this might take in Section 4.9.

4.9 EVALUATION

In this section, we evaluate LocationProof to determine, (i) *detection time*: how long it takes a consumer C to detect a malicious A, and (ii) *source overhead*: the computational burden LocationProof places on a source.

4.9.1 Detection Time

If C challenges the integrity of aggregate x_t received in each interval t, then it can detect a malicious A the first time it *lies*, i.e. injects fabricated data or computes an aggregation function other than the sum. However, in the interest of efficiency C could choose to randomly challenge A with a probability p during any given interval t. The random challenge forces the aggregator to guess when it is safe to fabricate x_t. Eventually, a lying aggregator will guess wrong and get caught. Lying more only causes the aggregator to be detected faster, and lying less only delays that outcome. The number of intervals before an aggregator lying with probability q is detected can be modeled using the negative binomial distribution.

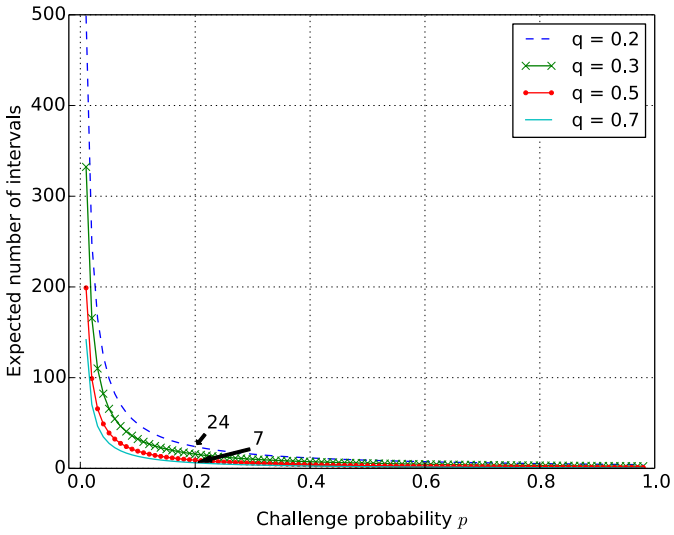


Figure 4.5: Expected number of intervals before aggregator A lying with probability q is detected by consumer C challenging with probability p.

The discrete negative binomial distribution $\mathrm{NB}(r, p)$ describes the probability of the number of "successes" in a sequence of binomial trials before r "failures" occur. The success and failure probabilities are p and $1 - p$ respectively. In the LocationProof protocol, "success" is represented by the event, "not detecting a lying proxy", and "failure" is represented by the event, "detecting the lying proxy". Thus, the success and failure probabilities in any given interval are $1 - pq$ and pq respectively. Also, since we are interested in the number of intervals before a lying aggregator is caught for the first time, the parameter $r = 1$.

Now, the expected number of successes for a negative binomial distribution with the above parameters represents the expected number of intervals before a lying aggregator is caught for the first time, and this can be written as

$$\frac{1 - pq}{pq}$$

Figure 4.5 shows the expected number of intervals before aggregator A lying with probabilities $q = \{0.2, 0.3, 0.5, 0.7\}$ is detected by consumer C challenging with probabilities $.01 \leq p \leq .99$. We can see that while challenging 20% of

the time, the consumer is expected to detect a lying aggregator within 7 to 24 intervals for the aforementioned set of lying probabilities q. However, challenging only a fraction of intervals also implies that the application must be resilient to some amount of fabricated data. A consumer C can thus choose its challenge probability p depending on how mission critical the application is.

4.9.2 Source Overhead

In this section, we measure the time and energy required by the sources to compute commitments. These measurements reflect the computational overhead *LocationProof* places on data sources. Recall that LocationProof requires sources to create commitments of their location coordinates and corresponding data in response to C's challenge. Table 4.3 shows the overhead experienced by an Android Nexus 4 platform (Version 4.3; quad-core 1.5 GHz processor; 2 GB RAM) and the Trusted Sensing Peripheral (Chapter 3).

	Android	**TSP**
Current	144 mA	55 mA
Time	2.325 msec ($\pm$1.26)	3.44 sec ($\pm$0.03)
Energy Consumption	1.2 mJ	5 Joules

Table 4.3: Computational cost of computing commitments for a tuple of data (x_{ij}, y_{ij}, d_{ij}) on a standard Android smart-phone and the Trusted Sensing Peripheral (TSP). Note the 95% confidence intervals indicated next to timing measurements.

In the case of Trusted Sensing Peripherals (TSP), the time and energy measurements have been extrapolated from those of software RSA encryption on the TSP (Figure 3.5(b)). An RSA encryption requires one modular exponentiation as opposed to the two required to create a commitment. Thus, computing a commitment will require approximately twice as much energy as an RSA encryption

assuming the cost of the product operation is relatively insignificant[1]. Thus, the energy consumed will be around $2 \times 0.25 = 5$ Joules per commitment and since three commitments are computed per challenge, the energy consumption will be 6 times more than software RSA. Consequently, the battery life will be $1/6^{th}$ of the lifetime achieved with software RSA. Figure 3.5(a) depicts the battery life of the TSP when performing RSA encryptions in software. Performing one encryption every 30 seconds, the TSP achieves a battery life of ≈ 80 days. If C were to challenge the data in each such interval, implying $p = 1$, the battery life of the TSP would reduce to approximately $1/6 \times 80 = 13.33$ days. However, with a challenge probability $p = 0.2$, the sources would have to compute commitments only $1/5^{th}$ of the time, improving the battery life to ≈ 66 days.

4.10 RELATED WORK

Much of the previous security oriented work in crowd-sourced sensing has focused *either* on data integrity, or privacy, but not both. PoolView [49] enables community statistics to be computed using perturbed private data, but trusts its users to honestly perturb that data. PriSense [116] employs data slicing and mixing to provide privacy while still supporting a wide variety of aggregation functions. However, it is assumed the functions themselves are honestly computed. Our work on Trusted Sensing Peripherals [38] supports high-integrity aggregation, but does not provide privacy guarantees.

VPriv [100] seeks to offer integrity and privacy for computing functions (e.g. total toll) over paths driven by individual vehicles. However, VPriv offers location privacy by keeping the uploaded location tuples anonymous. Anonymity, in turn, is provided using pseudonyms. Unfortunately, it has been shown that pseudonyms don't protect against inference attacks [73] since the uploaded tuples contain the

[1]The current draw remains the same during modular exponentiations, but since there are two per commitment, the entire computation takes twice as long

raw locations. Additionally, VPriv does not support aggregating data from multiple sources.

PrivStats [101] uses a combination of homomorphic encryption [94] and trusted intermediaries to provide integrity guarantees to the data consumer while protecting the privacy of data sources. The sources upload location-based information that they encrypt using additive homomorphic encryption. The consumer then computes over these encrypted tuples and sends the result back to a trusted intermediary (can be one of the data sources). The trusted intermediary then decrypts and returns the final result. Although, the PrivStats data publication model is similar to LocationProof, it does not support collecting actual locations, just location-based information (e.g. vehicle speed) from pre-determined locations. Additionally, PrivStats relies on the separation of duties to preserve privacy. More specifically, malicious clients colluding with the consumer can compromise the privacy of other users. This is because the consumer gets encrypted samples of data, the decryption keys for which, belong to the users.

Unlike PrivStats [101], Rastogi et al. [105] and Shi et al. [115] protect a data source's location privacy while actually collecting locations as part of the data. The fundamental tools are still homomorphic encryption [94] and a scheme similar to the Pedersen commitment scheme [98] used in LocationProof respectively. However, they are query-based data collection systems, implying that the application must first know the identities of the users it needs data from. Since sources are not anonymous to the system, their locations are vulnerable to intersection attacks (see Section 4.10). Another important issue is that there is no way to know which general area the location-based information will come from: again, since these systems are query-based and the data source locations are private, the only solution is to randomly query subsets of users and only then can the consumer learn the region where data was collected from. Thus, unlike LocationProof, these systems provide no control over the granularity of the collected location-based information.

4.11 LIMITATIONS

Currently, LocationProof can only be used to verify a location cloak represented by the mean of locations it contains. A more useful cloak should depict the extent of the cloaked region as well. For example, in addition to the mean, aggregator A could also compute and send the standard deviation to C. Since the standard deviation is also a mean (of distances from the mean), LocationProof can be used to prove its integrity [25]. However, it is yet to be determined if giving the consumer C both the standard deviation and the mean compromises a source's location privacy. Consider, for example, a brute-force attack where C tries all possible combinations of k locations in R that satisfy both the mean and the standard deviation. If the set of location combinations is small, then C may be able to learn the precise locations of the k sources in the cloak. We hope to explore this attack further in future work.

4.12 CONCLUSION

Crowd-sourced sensing has a bright future, but both the integrity of the collected data and the privacy of data sources are always at risk. Without integrity assurances, data consumers like the government or researchers will be reluctant to use the data, and without privacy assurances, people will be reluctant to contribute the data.

We have proposed a solution based on Pedersen's homomorphic-commitment scheme that *simultaneously* addresses the conflicting problems of integrity and privacy. This solution allows an intermediary to convince a data consumer that it is accurately performing a privacy-preserving transformation with inputs from trusted sources, without providing those inputs to the consumer. Sources can be trusted when, for example, they provide verifiable attestations to the integrity of sensed data with the aid of integrated trusted platform modules [38].

Chapter 5

NON-INTRUSIVE CHEAT DETECTION IN ONLINE GAMES

A lot of hidden information is present in client programs of most existing online multi-player games (e.g. unexplored map regions, opponent resource information). This hidden information is necessary to locally render a player's view of the game, thus, avoiding the overhead of retrieving it from the server before each move. However, the same hidden information can be exploited by cheaters to gain an unfair advantage over other players. Cheating has been shown to degrade the playing experience for honest players and eventually cause loss of revenue for game developers [144]. For example, in the real-time strategy game *Age of Empires*, a "map hack" allows the cheater to uncover map areas by modifying game client memory rather than actually exploring them. Similarly, inspecting game client memory can also reveal an opponent's secret resources or activities normally hidden from the player. In the words of Matthew Pritchard [102], "this cheating method was the next best thing to looking over the shoulders of his [the cheater's] opponents." Such cheats have been classified under the term *information exposure* [145, 102] and have traditionally been identified using stealth memory and process scanning programs deployed on a client's gaming device. Warden [112, 84] is one such scanning software used by the popular game World of Warcraft [19]. Warden has suffered from false positives, performance issues, and has been labeled "spyware" by the Electronic Frontier Foundation.

A less invasive method for preventing information exposure cheats involves minimizing the hidden state information in the game client. This is done by having the server load a client's game state information *on-demand* [75]. For example, the

server could gradually expose only those portions of the game map a player can 'see'. Unfortunately, this method introduces game rendering delays because clients now need to retrieve state information from the server after each move. Further, the computational overhead of managing, disseminating, and maintaining consistency across views hinders server scalability [137, 83, 75]. As a result, servers opt for the simpler *eager loading* mechanism, in which, move updates from one player are sent to every other player as they happen. This way, clients always have the game rendering information available locally and the server does not have to manage each player's game state as in on-demand loading. The drawback is that game clients are forced to secretly store sensitive game-state information belonging to other players — regardless of whether its needed to render the current game view. Thus, methods like eager-loading, although efficient, create the threat of information exposure cheats. What is needed, is a more efficient method to detect or prevent information exposure cheats.

5.1 APPLYING THE TRUST-BUT-VERIFY APPROACH

As discussed above, on-demand loading involves game servers accepting a player's move, computing a game view based on the new move, and returning the view so that it can be rendered by the game client. It has been shown that computing this view remotely results in significant game rendering delays. In this chapter, we use the trust-but-verify approach to offload some of that computation to the client. More specifically, after each move, a client computes and submits a view descriptor which also represents the *data* to verify. The computation of this view descriptor represents the *generation function*. The server then validates the descriptor and returns a response containing the contents of the view (e.g. other players, moving obstructions). Validation involves checking if the descriptor is consistent with the player's last move and the current global game state. This validation represents the *verification function* in the trust-but-verify model.

5.2 CONTRIBUTIONS

This chapter presents a system called *SpotCheck* that provides a more balanced defense against information exposure cheats in client–server online games. Like on-demand loading, SpotCheck eliminates the need for hidden information in the game client; and like eager loading it doesn't require the server to track a client's view. The lack of hidden information obviates the need for invasive cheat detection software like Warden [112]. However, information exposure cheats can still occur by sending illegal view descriptors to the server. The advantage with SpotCheck is that the overhead of checking the view descriptor can be traded for improved game performance. For example, instead of checking every view descriptor, the server can randomly check only a fraction of them. Although this introduces opportunities for cheating, repeat offenders will eventually be identified. The general system model in which SpotCheck is applicable is discussed in Section 5.3. SpotCheck's design is presented in Sections 5.4 – 5.8 and an evaluation of its efficiency and effectiveness in the context of a simple real-time strategy game called *Explorer* is described in Section 5.9.

5.3 SYSTEM MODEL

SpotCheck is applicable in online map-based games with a client–server architecture. More specifically, as Figure 5.1 depicts, SpotCheck enables the game client to compute a view for a player's character (in the game) based on the player's input move. The idea is that as long as the view contains only those portions of the map the player's character is "supposed" to see in that stage of the game, then the player could not be cheating. For example, if a character's view is obstructed by a hill, but the view sent to the server includes portions of the map behind the hill, then the player must be cheating. The server can check for such fabricated views to detect malicious players.

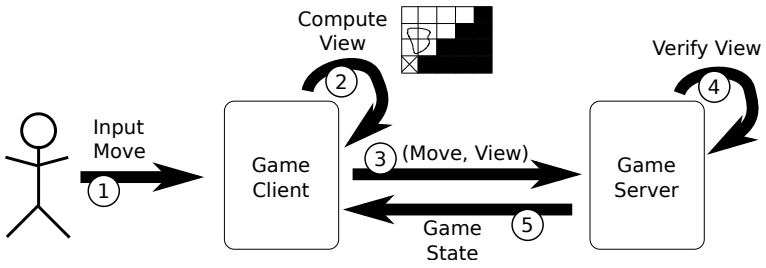


Figure 5.1: After the user enters her move, the game client computes a description of the view (e.g. newly visible map regions). The server trusts the client to compute this view, but occasionally checks if the view descriptor was computed correctly.

In the trust-but-verify context (see Figure 2.1), the game client is the data source, the server is the data consumer, and the generation function is the one that computes the view descriptor. Note that since a user's every move is sent from the game client to the server, no intermediate aggregators are necessary.

5.4 MOTIVATION AND RELATED WORK

Our goal is to develop an effective and scalable solution for detecting information exposure cheats in client–server online games. Existing solutions are either effective and not scalable, or vice versa. Popular games like World of WarCraft [19] and StarCraft [18] use an anti-cheating software called Warden [112], which runs on the client, scanning periodically for signatures of common cheats [125]. Tools like Warden are not always effective for two reasons. First, they cannot detect new information exposure cheats. For example, earlier map hacks were easy to detect because they were implemented by patching the game client, but new map hacks are harder to detect because they are external to the game and work by reading game memory and exposing the map areas via overlays [12]. Second, the scanning performed by such tools is widely considered to be a risk to personal privacy. The Electronic Frontier Foundation has even labeled Warden as "spyware" [81]. Another solution, called *on-demand loading* [75], proposes a change to the

server's game state dissemination strategy that consequently prevents all forms of information exposure cheats. In this method, the server disseminates state information required to render the game only when clients need it. Thus, on-demand loading eliminates any hidden information in the game client, leaving no useful information to expose. Webb et al. [144] have called it the "most effective solution" against this form of cheating. Unfortunately, the price of on-demand loading is excessive server CPU overhead because the server must now track each client's view of the game. To avoid precisely this overhead, existing games like World of Warcraft prefer *eager-loading* [75]. Eager-loading involves sending state updates of one player to every other player. This eliminates the burden of tracking every player's view on the server and allows any client to independently render the game. However, now, each player has state information belonging to all other players whether or not it is needed to render their current view of the game. As a result, the extraneous information is kept hidden in the game client waiting to be exploited by those who know how to bypass cheat detection tools like Warden.

SpotCheck proposes a middle ground: trust the client to compute a description of its own view, but enable the server to verify the description's integrity. This approach does not require the server to track client views, instead it only requires the server to maintain global game state consistency and verify view descriptors. As long as the verification process is cheaper than tracking a client's view, our approach will be more scalable than on-demand loading. Additionally, like on-demand loading, game state information is disseminated to clients only as needed. Thus, there is no hidden information in the game client that cheaters can exploit.

There is, however, another approach that could be used to prevent information exposure cheats. Monch et al. [87] propose a technique that perpetually obfuscates all game state information stored in the client. The solution involves using decoding and encoding functions before accessing or writing any game state information. Also, since obfuscation does not provide cryptographic secrecy, the hiding

functions are changed periodically via a secret channel. Thus, forcing an adversary to constantly reverse engineer the hiding functions.

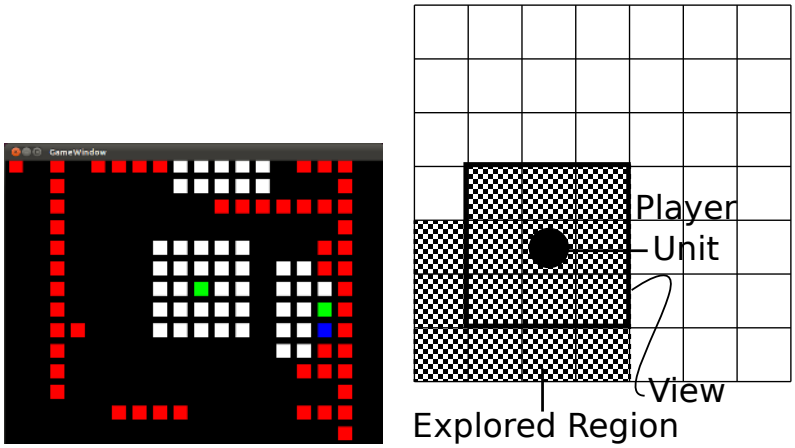
If hiding functions are efficient and can indeed be surreptitiously changed faster than the time required to reverse engineer them, then the above solution may be effective. Unfortunately, the efficiency of the solution, the feasibility of constantly generating and secretly transmitting strong hiding functions, and in general the feasibility of developing tamper-proof software is still unclear [144].

5.5 EXPLORER

We analyze SpotCheck in the context of a simple real-time strategy game we call *Explorer*. The game consists of players exploring a 2D terrain consisting of walls and other obstructions (Figure 5.2(a)). Multiple players can play the game and each player can have multiple units exploring the terrain. The map of the terrain is composed of grids and each grid can contain either the terrain itself, part of a wall, a player's unit, or an obstruction. Additionally, unexplored regions of the terrain and those that are outside a unit's vision are kept hidden from the player (Figure 5.2(b)). At any given time, a unit's vision consists of the contents of a 5×5 grid around its current position. The only exception is that units cannot see through walls, thus, parts of the terrain that belong inside the vision but are blocked by walls will still be hidden. Player units can move one grid at a time in vertical or horizontal directions uncovering contents of unexplored regions of the terrain. In its current state, the game defines no objectives, player unit resources, or conditions for victory.

Formally though, SpotCheck is applicable to games consisting of the following high-level components:

- 1 to n players
- 1 to m units per player



(a) Screenshot (b) Schematic

Figure 5.2: Explorer overview

- Global game state S_t: the state of the game on the server after move t, where t is a global move counter across all players. The global game state for Explorer includes location and type information about terrain, obstructions, and player units.

- Game map M: a set of $e \times e$ square-shaped cell locations. We assume a square-shaped map and cell for simplicity, but SpotCheck need not be limited by the shape of the map or the cell.

- View descriptor v_t^i: the set of cell locations visible to all units of player i after move t.

- View V_t^i: view descriptor v_t^i along with game state information associated with cells in the descriptor.

- Explored Region E_t^i: game state information pertaining to the region of the map explored by all the units of player i after move t.

- State request U_t^i: sent to the server by player i after move t. The state request consists of the view descriptor v_t^i ,unit identifier and the new location of the

player's unit.

- State update D_t^i: sent from the server to the client after move t. The state update may contain a player's current view if generated in response to a state request, or it may contain changes to the player's current view as a result of moves by other players.

The game progresses as players make their moves and send corresponding state requests to the server. On receipt, the server verifies the integrity of the state request by randomly performing either a heuristic-based check, or a more expensive full check. State requests for illegal cell locations could escape detection by heuristic-based checks but will eventually be detected by the full check. We evaluate this further in Section 5.9. Once verification succeeds, the server sends back a state update. The game client then uses this state update to render a player's current view of the game. Intermittently, move updates from other players cause state updates to be sent to affected players. However, these updates only include changes to a player's current view.

Our game, Explorer, can be configured to disseminate state information using the three different strategies discussed earlier: on-demand loading, eager loading, and SpotCheck's strategy. The state request in on-demand and eager loading consists only of the player's new unit location, where as for SpotCheck, the state request contains the view descriptor as well. The state update in on-demand loading and SpotCheck consists of the player's view, where as in eager loading, it consists of information in a state request from any of the n players.

5.6 CHEAT MODEL

SpotCheck addresses application-level *information exposure* cheats [144], but unlike eager loading where these cheats are executed by accessing game client memory, SpotCheck forces cheaters to send malformed state requests to the server. Recall

that players progress in the game by sending and receiving responses to state requests. These state requests contain the description of a player's view, which is verified by the server upon receipt. A cheater could conceivably construct a state request with an illegal view descriptor $v_t^{i\prime}$ that includes locations out of its actual view v_t^i. For example, by requesting state information for a cell across a wall, which by design blocks a unit's line of sight. If the server is unable to detect such malformed requests, then cheaters could potentially learn information about other players in any portion of the map.

In the context of SpotCheck, accessing game client memory is not considered an information exposure cheat. Mainly because SpotCheck eliminates the need for any hidden information in the game client. Note that hidden information is defined as any state information associated with map cells outside the explored region E_t^i. In the case where the player's view V_t^i is a proper subset of the explored region E_t^i, this definition does allow game state information $E_t^i - V_t^i$ in the client that is not displayed to the player. However, this information is not considered hidden because it was first displayed when the player last visited those cells and has not been updated since.

Like on-demand loading, SpotCheck prevents infrastructure-level information exposure cheats by design. These type of cheats involve using a network hub and another host to sniff one's own game traffic and change the way it is rendered on screen. For example, modifying the display driver on the consorting host to render the game world with transparent walls. In SpotCheck, since information about cells outside a player's view (e.g. behind walls) is never included in state updates, it becomes futile to mount this type of cheat.

There are other ways of accessing hidden information that are outside SpotCheck's scope: a cheater might spoof a state request pretending to be another player. This type of cheat would need to be addressed by introducing appropriate authentication mechanisms [144]; a cheater might snoop another player's communication channel

with the server. This type of cheat would need to be addressed by encrypting communications between the client and the server; a cheater might collude with other players to gain collective information; and lastly, a cheater might compromise the server to learn about global game state, which is normally not exposed to clients.

5.7 CHEAT DETECTION

The game server detects a cheat when it can't verify the integrity of a client's state request. State requests are sent to the server each time a client moves one of its units to a new location. The server then chooses to verify the state request with a probability p. The randomness ensures that a cheater does not know before hand if her state request is going to be checked or not. Without that knowledge, a cheater risks being detected and subsequently banned from the game. Since cheaters cannot progress in the game without sending state requests, they are forced to take their chances.

Notice that when $p = 1$, every state request is checked. The parameter p essentially enables SpotCheck to provide game servers the flexibility of balancing resources used for game state management against those used for cheat detection. This is unlike on-demand loading, where the server is forced to track every client's view. The trade off with SpotCheck is that a cheater may initially get away, but with more cheat attempts a malicious player will eventually be identified. Thus, assuming that a player cheats during a move with probability q, the expected number of moves in which the cheater will be detected is:

$$E(T) = \sum_{t=1}^{\infty} t \times (1 - pq)^{t-1} \times pq = \frac{1 - pq}{pq} \tag{5.1}$$

Where T is a discrete random variable representing the number of moves a cheater can make before getting caught. We compare the expected outcome with the experimental result in Section 5.9.

The server validates a state request by performing a *full check* of its contents. A state request contains the player unit's new location (move), unit identity, and a view descriptor. A move is considered valid if the unit advances to an empty and allowed map cell (currently, our game does not support multiple units in the same cell). A valid move results in an update to the global game state. Then, the updated game state information is used to construct the expected view descriptor for the player and compared to the one that was included in the request. Any discrepancies between the two is considered a cheat attempt. A state request that passes the full check is considered valid.

One disadvantage of probabilistically validating state requests is that easily preventable cheat attempts can occasionally escape detection. For example, a cheater may retrieve a snapshot of the global game state by constructing a view descriptor that includes all cell locations on the game map. This cheat can easily be prevented by validating the size of the view descriptor. To prevent such cheats, SpotCheck allows game servers to perform heuristic-based checks of the state request during those moves when the complete check described above is not performed.

We have currently implemented two heuristic-based checks of the state request. Each of these checks validate the range of legal values for individual components of the state request. The first heuristic, called *distance bound*, checks if the player's move is legal. So for example, in Explorer, units are not allowed to move more than one cell at a time. The second heuristic, called *vision size bound*, checks that the size of the view descriptor does not exceed the possible maximum for the player.

5.8 ARCHITECTURE

The components of our game are split across the client and server. At a high level, the game client accepts key stroke input and renders the game, whereas the server validates the inputs and sends back the information necessary to render the game.

5.8.1 Client Components

Game map. Stores the 2D game map. The map is composed of cells each of which contain a wall, a player unit, an obstruction, or the terrain itself. Information about the contents of the map is stored in the local game state.

Game state manager. All the information about a player's explored region is stored in the local game state. The state manager runs every time a move is generated and sends the move along with the current view descriptor in a state request to the server. The server validates the state request and sends back a state update containing information required to render the player's view.

Rendering engine. Renders units, obstructions, walls, and terrain on visible portions of the map while blacking out the rest of the cells. After the state manager has received the necessary information required to render the player's view, it is passed on to the rendering engine, which then draws the view on the screen.

Input mechanism. Interprets key strokes as moves and forwards them to the game state manager.

5.8.2 Server Components

Request validator. As shown in Figure 5.8.2, the validator receives a state request from the client and chooses to either validate it with a probability p or check it using simple heuristics (see Section 5.7). A state request that passes the checks is then forwarded to the state update generator.

State update generator. Uses information in the state request to update global game state. Then, information pertaining to the player's view is retrieved from the global game state and sent as a state update.

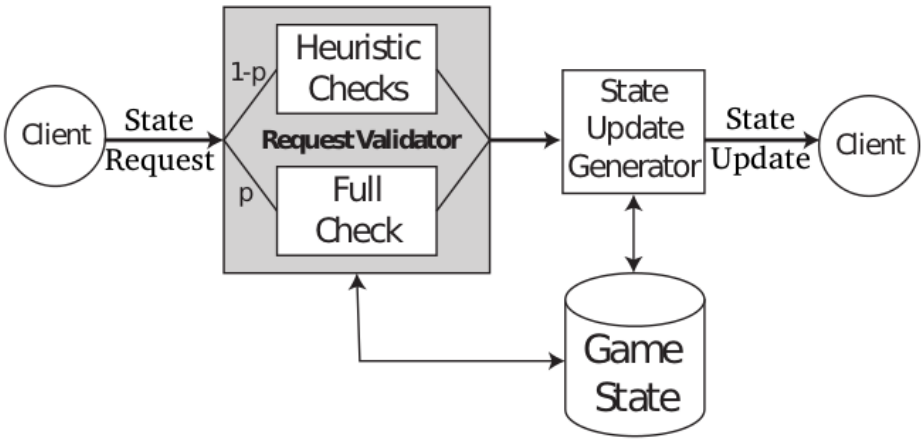


Figure 5.3: SpotCheck server-side architecture

It is worth mentioning that the state request validator need not run as part of the request processing pipeline (Figure 5.8.2). The heuristic checks could still be done serially while the full check would happen in the background. Any game state corruption discovered during the full check may need to be reconciled. Although we have not explicitly evaluated this alternative, we believe that having the validator run in parallel can reduce state update response time and lead to an improved gaming experience with minimal impact on cheat detection integrity or performance.

Although SpotCheck is designed to detect information exposure cheats, it does not preclude including methods that detect other classes of cheats. For example, in the future, we plan to augment SpotCheck with code injection and entanglement algorithms [87] that prevent bots/reflex enhancer cheats.

5.9 RESULTS

We will now evaluate SpotCheck against on-demand and eager loading while two players play a game of Explorer. The performance of eager loading forms our baseline since it provides no intrinsic protection against information exposure cheats

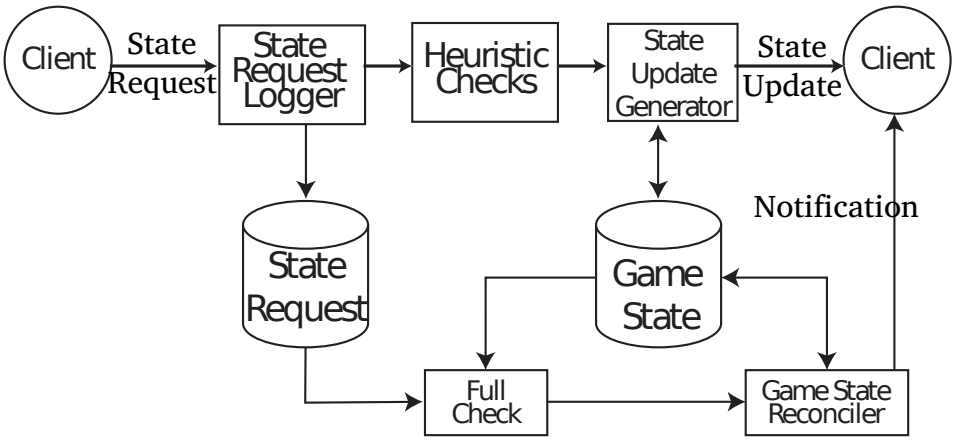


Figure 5.4: Alternative architecture: request validation in parallel

and is currently the most prevalent method for disseminating game state information. On-demand loading is very effective against protecting information exposure cheats, but is expensive for game servers to implement.

Our goal is to significantly reduce the overhead on the game server when compared to on-demand loading while providing similar levels of protection against information exposure cheats. Specifically, we will compare server CPU overhead, message sizes, and client game rendering latency of our approach with on-demand and eager loading. We will also evaluate the time it takes for SpotCheck to detect information exposure cheats.

5.9.1 Experimental Setup

The test system used to gather performance data is a Dell Latitude E6510 configured with an Intel Core i5-M580 CPU running at 2.67 GHz with 4 GB of main memory. Intel TurboBoost and SpeedStep were disabled through the system BIOS. The operating system is a 32-bit Ubuntu 11.04 with Linux kernel version 2.6.38.9-generic. All experiments were performed with the game client and server on the same machine.

5.9.2 Evaluation

Figure 5.5 plots server CPU overhead of the three schemes against units per player. Additionally, SpotCheck is plotted for the scenarios where 100% ($p = 1$), 25%, and 5% of the state requests are checked. The CPU overhead is measured as the total CPU time required by the server to process, check, and respond to a state request. Here, map size is held constant at 100×100 cells and the error lines indicate a 95% confidence interval over 200 runs per data point. Unless mentioned, the stated constants remain the same for all future plots.

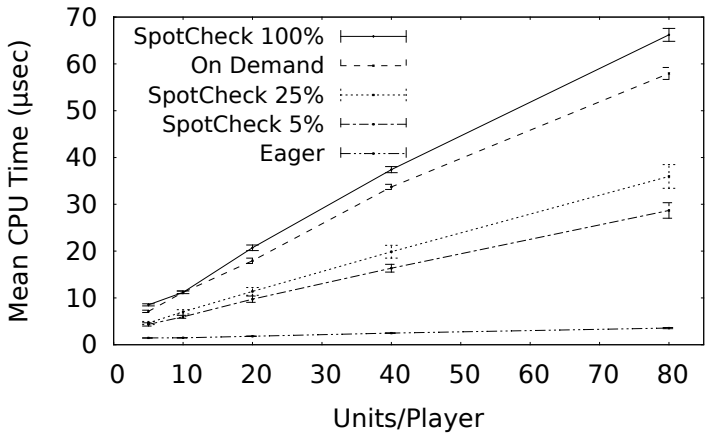


Figure 5.5: Server CPU overhead

The CPU overhead of SpotCheck and on-demand grows because more units increase the size of a player's view, which consequently, requires more processing on the server. In eager loading, the game client deals with views, thus the server's overhead remains nearly constant and less than on-demand or SpotCheck. SpotCheck 100% incurs more overhead than on-demand loading because checking every state request is more expensive than tracking a player's view. This is not surprising because view tracking updates a stored view descriptor on the server with every state request, where as validation involves the more expensive operation of computing a new one from scratch each time (see Section 5.7). Notice

though that spot checking 5% of the requests requires nearly half the overhead of on-demand, where as checking 25% requires slightly more.

Figure 5.6 plots expected (Equation 5.1) and observed number of moves SpotCheck requires to detect an information exposure cheat versus various values of checking (p) and cheating (q) frequencies. A move where no cheating takes place is referred to as an "honest move", while the opposite is referred to as a "cheat move". For this experiment, we perform an information exposure cheat by issuing an illegal state request, where the view descriptor contains a location not in the player's current view. We also ensure that the information exposure cheat can bypass all heuristic checks. Thus, the cheat can only be detected by a full check. We can see that even when spot checking only 5% of the state requests, players cheating 50% of the time are detected before their 25^{th} cheat attempt. If players cheat less (5%) and SpotCheck checks less (5%) then the number of moves required to detect a cheater increase significantly, but the number cheat moves remain small (≈ 15).

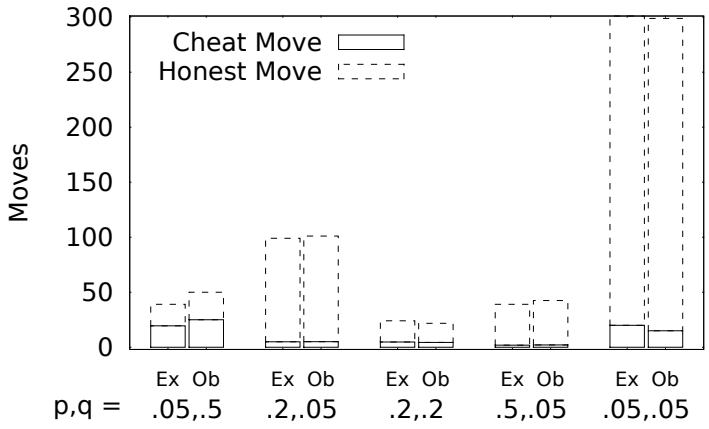


Figure 5.6: Expected (Ex) and Observed (Ob) number of moves before a cheat attempt is discovered. Number of cheats that escaped detection are also shown as a portion of total moves.

Figure 5.7 (top) plots the size of a state update against the percentage of occupied locations visible to a player on the map. The size of the state update

will significantly impact the server's outbound bandwidth requirements as players increase. Here, we focus on the request and update sizes for only one player with five units in the game. Additionally, 120 randomly chosen map locations are occupied by obstructions. In Explorer, the state update contains five bytes of information per location. We can see that under SpotCheck and on-demand, the size of the state update depends on the occupied locations visible to the player. Under eager loading, the server is oblivious to a player's view, thus it must send information about all the locations in question, forcing the update size to be larger than (or equal to) that of SpotCheck and on-demand.

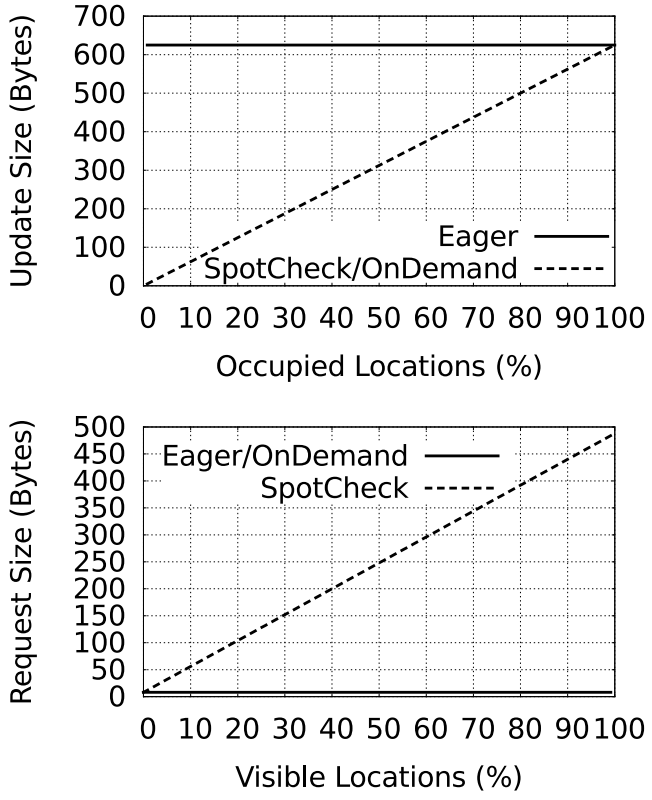


Figure 5.7: State update and request message sizes

Figure 5.7 (bottom) plots the size of the state request against the percentage of locations visible to a player on the map. Since the state request in eager and

on-demand loading does not contain a view descriptor, its size remains smaller than (or equal to) that of SpotCheck. Note, however, that Explorer uses the most naive method of encoding view descriptors: four bytes per visible location. Using better encoding schemes should result in much smaller request sizes.

Figure 5.8 plots client game rendering latency for SpotCheck ($p = 0.05$), on-demand, and eager loading. Rendering latency is measured as total CPU time between sending a state request and rendering the contents of the corresponding state update on screen. Keep in mind that the client and server are on the same machine hence the round-trip-time is fairly small. We can see that rendering latency increases with units per player. This is mainly because more resources are required to render the larger view of all units. SpotCheck and on-demand induce more latency than eager loading because of the additional time the server spends verifying the state request and tracking views respectively. Notice, however, that the latency incurred by SpotCheck is consistently less than on-demand even though SpotCheck requires more game processing on the client. This is because the server is more efficient under SpotCheck than under on-demand when responding to state requests.

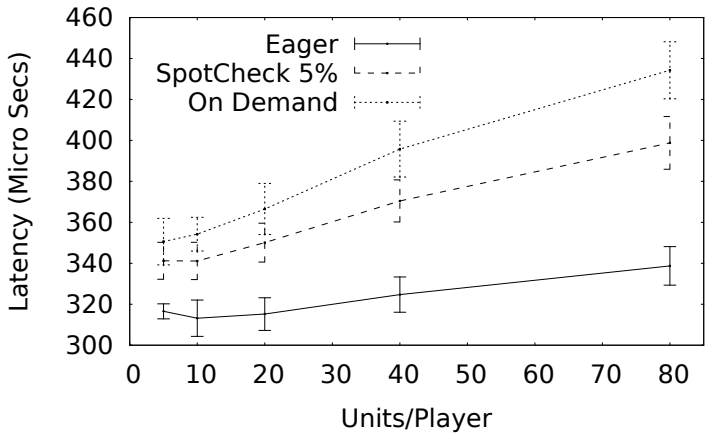


Figure 5.8: Client game rendering latency for SpotCheck, on-demand, and eager loading

5.10 CONCLUSION AND FUTURE WORK

We presented SpotCheck, an efficient defense against information exposure cheats. SpotCheck strives to be the middle ground between on-demand loading, which prevents this class of cheats entirely, but is expensive for the game server to implement; and eager loading, which provides no protection from cheats, but is prevalent in online games today due to its performance benefits. The key idea behind SpotCheck is to allow the clients to track and request contents of their game view, but randomly sample and validate their requests. SpotCheck allows game servers to change the sampling frequency, thus trading off CPU overhead for increased cheat protection or vice versa. We have shown that SpotCheck can reduce the server CPU overhead by as much as half when compared to on-demand loading, while still being an effective defense against information exposure cheats. In the future, we plan to incorporate SpotCheck into a real-world game and evaluate its performs with a much larger number of players.

Chapter 6

EFFECTIVE SPAM PREVENTION IN ONLINE MESSAGING APPLICATIONS

Internet spammers are relentless. Although, email spam is reducing ($\approx$ 70.5% in Jan 2012 from 92.2% in Aug 2010), the spam on social network sites is edging up [51]. Approximately 4 million Facebook users receive spam from around 600,000 new or hijacked accounts each day [51, 80]. What's worse is the success rates of social spam: in Jan 2010, 0.13% of all spam URLs on twitter were visited by around 1.6 million unsuspecting users [56]. This "clickthrough" rate is almost two orders of magnitude larger than for email spam. Spammers cost businesses $20.5 billion annually in decreased productivity and technical expenses, and this cost is projected to rise to $198 billion in the next four years [119]. There are two prevalent methods to prevent this deluge of spam: *CAPTCHA* and *proof-of-work*; both methods have benefits and drawbacks [59].

A CAPTCHA can effectively protect an online transaction so long as there aren't OCR algorithms that can automatically "solve" or "break" it [124]. Once a class of CAPTCHAs is broken, the corresponding application becomes defenseless against spam bots. CAPTCHAs are also prone to outsourcing attacks where humans are employed to solve CAPTCHAs *en masse*. A major cause of success for these attacks is that CAPTCHAs don't provide a way to change the cost of solving them [90, 91]. Additionally, the usability burden imposed by CAPTCHAs [147] limits their use to only protecting infrequent transactions like creating accounts. This leaves frequent transactions, like message posting, open to abuse. Attackers exploit this loophole by hijacking accounts and using them to send spam.

The *Proof-of-work* approach does not have CAPTCHA's usability issues and can therefore be used in frequent transactions. This is because the assigned "work" can be done automatically by the user's web browser without user intervention. Additionally, this paradigm enables an application to price a transaction by varying the amount of work that needs to be done as payment. The idea is to issue more "work" to suspected spammers than to honest users, thus, effectively reducing incoming spam.

The disadvantage of a proof-of-work approach is that the "work" done by users is essentially wasted. The Reusable Proof-of-Work (RePoW) approach [64] proposes using puzzles whose solutions can be used for a greater purpose (e.g. protein folding [95]). The problem is that puzzles must be hard to solve and easy to check if they are to be a practical spam prevention solution. Unfortunately, there are very few known reusable proof-of-work puzzles whose solutions are easy to check [64, 17]. What is needed is a proof-of-work system that enables issuing generic puzzles (e.g. protein folding [95]) whose result can be checked more efficiently than recomputing the puzzle itself.

6.1 APPLYING THE TRUST-BUT-VERIFY APPROACH

In this scenario, the puzzles are the *generation functions* and the result of solving the puzzle is the *data* to verify. The goal then, is to develop a *verification function* that allows a puzzle issuer to check the solution of a puzzle without having to recompute it. This is done by bootstrapping the puzzle issuer with a set of *known* puzzles — puzzles whose solutions are already known. Then, clients are issued a sequence of known and *unknown* puzzles. Once the clients execute the puzzles and return the solutions, the receiver verifies them by checking the solutions of only the known puzzles. As long as clients cannot distinguish between known and unknown puzzles, they will have no guaranteed way to pass verification. The details and

analysis of the verification process is presented in Section 6.5.3. Before the trust-but-verify approach can be applied, however, we need to build the infrastructure that allows issuing generic puzzles and accurately identifies malicious users so that they can be issued more difficult puzzles.

6.2 CONTRIBUTIONS

This chapter describes MetaCAPTCHA, an application-agnostic spam prevention service for the web. It can be used by applications like discussion forums, web-mail, and social web sites. MetaCAPTCHA's novelty stems from the following contributions:

- It seamlessly integrates the CAPTCHA and proof-of-work approaches while augmenting each: it can dynamically issue proof-of-work or CAPTCHA puzzles while ensuring that malicious users solve much "harder" puzzles — CAPTCHAs included — than honest users. More specifically, our results show that honest users were never issued a puzzle during 95% of their transactions.

- Puzzles are randomly picked and delivered within a generic solver that eventually executes those puzzles in the user's web browser. Thus, the solver code is metamorphic: changing randomly in each transaction. This *turns the reverse engineering problem around on the adversary* who must now attach a debugger to discover the solver's execution steps.

- It uses a Bayesian reputation system that can accurately predict a user's reputation score based on features configured by the web application. Since multiple web applications can be protected by MetaCAPTCHA, its reputation system provides global visibility on attacks across all those applications.

- It contains a modular puzzle library that can be configured with new classes

of CAPTCHAs or proof-of-work puzzles while allowing the removal of those classes that are known to be "broken". These puzzle library modifications can be made by the web application without *any change to its source code*. Furthermore, the variety of puzzles in the library ensures that breaking one class of puzzles won't compromise MetaCAPTCHA as a whole.

- Generic-puzzle solution verification. Proof-of-work approaches usually rely on the puzzle construction to provide a short-cut for checking the puzzle solution (otherwise verification would involve recomputing the costly puzzle). However, this severely limits the class of puzzles that can be issued. MetaCAPTCHA has the ability to issue and verify the output of generic computational tasks without having to redo those tasks. This facilitates the use of volunteer computing projects [20] as puzzles. For example, see Section 6.6.2 for the description of a puzzle that enables counting fish in the Bonneville Dam [136] in Oregon, USA.

- Web applications can easily install the MetaCAPTCHA API by making changes similar to those required by existing CAPTCHA implementations [53, 99].

6.3 BACKGROUND

MetaCAPTCHA dynamically issues CAPTCHA and proof-of-work puzzles. We now provide a brief background on each kind of puzzle.

6.3.1 CAPTCHA

CAPTCHA stands for "Completely Automated Public Turing-test to tell Computers and Humans Apart". CAPTCHAs usually consist of images containing squiggly characters that are easy for humans to read, but hard for programs to parse. If a respondent solves the CAPTCHA correctly, the implication is that the

respondent is most likely human. The idea is to allow humans to access the web application's services while deterring automated adversaries like bots. A popular implementation of the CAPTCHA is the reCAPTCHA [138]. Notice also, that the verification strategy (in the trust-but-verify parlance) is to check the solution to every CAPTCHA issued.

6.3.2 Proof-of-work

The proof-of-work approach was first proposed by Dwork and Naor [40] to combat email spam. The idea was to impose a per-email cost on senders, where, the cost was in terms of computational resources devoted by the sender to compute the pricing function. Once a sender proved that it correctly computed the pricing function, the server would send the email. As with CAPTCHAs, the verification strategy is to validate the result of every pricing function issued. Effectively, sending bulk spam would become "expensive" because computational resources are finite. The characteristics of such a pricing function f was then described as follows:

1. "moderately" easy to compute
2. not amenable to amortization: given any l values $m_1, \ldots, m_l$, the cost of computing $f(m_i)$ is similar to the cost of computing $f(m_j)$ where $i \neq j$. In other words, no amount of pre-processing should make it easier to compute f on any input.
3. Given x and y, it is easy to check if $y = f(x)$

An example of a pricing function is one that finds partial hash collisions [14]. A function $f_k : x \rightarrow y$ is said to compute a k-bit partial hash collision on string x, if given a hash function H, the first k bits of $H(x)$ are equal to the first k bits of $H(y)$. Notice that $f_k(\cdot)$ has all the properties required of a pricing function.

Although the proof-of-work approach seemed promising, Laurie and Clayton [74] demonstrated in 2004 that reducing spam to 1% of normal email would require

delaying each message — including one that an honest user sends — by ≈ 6 minutes; a high price to pay for innocent users. This delay was computed based on then current rates of spam, number of email users, and under the assumption that 1 million compromised machines were spewing spam. Since then, spam has increased by 18% to 74.2%, so we expect the aforementioned delay to be much larger now.

To reduce this delay, Liu and Camp [77] proposed basing puzzle difficulties on user reputation. The idea was that users with lower reputations would receive harder puzzles than those with higher reputations. Since easier puzzles would be much quicker to solve, honest users would experience a nominal delay when sending messages where as malicious users may be significantly delayed. Thus, with an accurate reputation system, the proof-of-work approach can be a practical, fair, and effective technique for combating spam.

6.4 SYSTEM MODEL

This section describes the system model in which MetaCAPTCHA is applicable. In general, interactive web applications where online transactions can be exploited by spammers, such as message forums, webmail, social applications, and event-ticket purchasing can employ MetaCAPTCHA for spam prevention. Heymann et al. [59] provide an exhaustive discussion on the common characteristics of such web applications.

An overview of the system model and high-level interactions between the Meta-CAPTCHA service, the web client, and the corresponding web application is shown in Figure 6.1. The interactions begin when a user attempts to perform an online transaction. The web application allows the transaction to proceed only when it has sufficient proof that the client completed the work it was assigned by Meta-CAPTCHA.

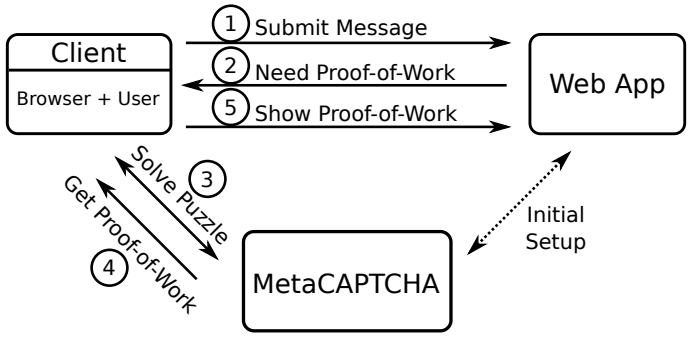


Figure 6.1: System model: user's browser must show proof-of-work before the web application accepts the user's message. The dotted line indicates initial setup performed by the web application to use the MetaCAPTCHA service.

The work is issued in the form of "puzzles". Puzzles can be *interactive*, *non-interactive*, or *hybrid*. Interactive puzzles are generally CAPTCHAs, whereas non-interactive puzzles are pricing functions as described in Section 6.3.2. A hybrid puzzle combines a CAPTCHA and a pricing function into one puzzle. Furthermore, puzzles can either be *known* or *unknown* depending on whether their solutions are already known or not.

As shown in Figure 6.1, we treat the *user* separate from the *browser* while collectively referring to them both as the *client*. This is due to the existence of interactive puzzles that need user interaction to solve, and non-interactive puzzles that are solved automatically by the browser.

In the trust-but-verify context, the client is the data source and the MetaCAPTCHA service is the consumer. The function being validated is the puzzle whose inputs are provided by MetaCAPTCHA. The goal is to ensure that the source correctly computes or "solves" the puzzles. Details on how puzzle solutions are verified are presented in Section 6.5.3.

6.5 COMMUNICATION PROTOCOL

This section discusses the MetaCAPTCHA communication protocol. For simplicity, we assume the scenario where a client is attempting to post a message. Note, however, that MetaCAPTCHA can protect more general web transactions like purchasing event tickets, creating accounts, etc.

A web client begins communicating with MetaCAPTCHA after being referred by the corresponding web application. In this case, the application will refer a client attempting to post a message to MetaCAPTCHA. The client will then need to obtain and solve a puzzle. The idea is that the web application will allow messages from only those clients that have successfully solved a puzzle issued by MetaCAPTCHA. The communication protocol for obtaining and solving a puzzle begins with authentication as explained in the next section.

6.5.1 Authentication

MetaCAPTCHA only issues puzzles to clients of participating web applications. This requires MetaCAPTCHA to authenticate two things, (i) the identity of the web application, and (ii) the client is an authorized user of the web application. MetaCAPTCHA provides each web application with an API key K during a registration phase. The web application must keep K secret as it will later be used for authenticating both the application itself and all its clients.

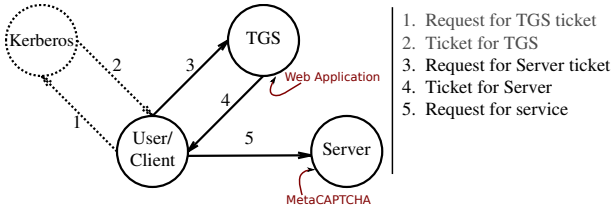


Figure 6.2: Kerberos authentication overview and how it relates to MetaCAPTCHA authentication. Figure adapted from Steiner et al. [123]

As implied by the system model in Section 6.4, a client is not given access to the services provided by the web application until it shows proof of a correctly solved puzzle. The only way to be issued a puzzle is to first show that the client is an authorized user of a registered web application. A client does so by presenting to MetaCAPTCHA a "server-ticket" issued by the web application. The authentication protocol used is modeled around Kerberos [123], wherein the web application acts as the Ticket-Granting-Server (TGS) for the MetaCAPTCHA service as shown in Figure 6.2 [123]. Notice that steps 1 and 2 of the Kerberos protocol — where a client authenticates itself to Kerberos — are not required because MetaCAPTCHA assumes that it will be replaced by the web application's existing authentication mechanism (e.g password).

After a client submits a message, the web application returns a server-ticket $S_1 = C||ID||\mathbf{HMAC}(K, C||ID)$ containing client-specific information C, a web application ID issued by MetaCAPTCHA during the registration phase, and a Hash-based Message Authentication Code (HMAC) for C created using the web application's secret key K (See Figure 6.3). The server-ticket S_1 is called the *puzzle-request ticket* and is sent by clients to MetaCAPTCHA for requesting puzzles.

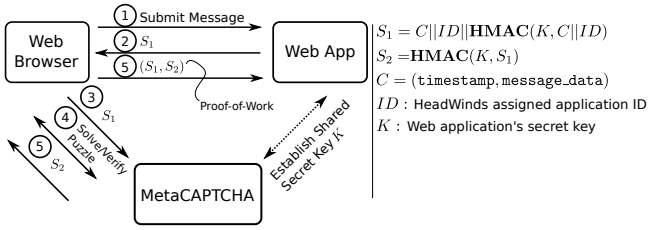


Figure 6.3: MetaCAPTCHA authentication and puzzle solution verification

When MetaCAPTCHA receives the puzzle-request-ticket S_1, it verifies that the client is indeed a user of a registered web application. MetaCAPTCHA performs this verification by checking the integrity of the HMAC included in the ticket. Notice that the correct HMAC can only be generated by a registered web application because it includes that application's unique API key.

Once the integrity of the HMAC is ascertained by MetaCAPTCHA, the client is issued a puzzle to solve. Details of client-specific information C are presented in Section 6.5.2.

6.5.2 Puzzle Delivery

MetaCAPTCHA only issues puzzles to authenticated clients as previously shown. The hardness of the issued puzzle depends on the client-specific information $C =$ (`timestamp, message_data`) sent by the client to MetaCAPTCHA during the authentication phase. Here, `timestamp` indicates when the message was created (this assumes the web application and MetaCAPTCHA are loosely time-synchronized); `message_data` contains the message text submitted by the client and any other information related to it. MetaCAPTCHA uses the information in C to compute a reputation score, which in turn is used to determine the puzzle *difficulty level*: the amount of time a user's browser must compute to provide sufficient proof-of-work to the web application. In MetaCAPTCHA, higher reputation scores imply more malicious clients. As a result, such clients are issued puzzles with increased difficulty levels. The details of how reputation scores are computed, and how puzzles of varying difficulty are generated are presented in Sections 6.6.1 and 6.6.2 respectively.

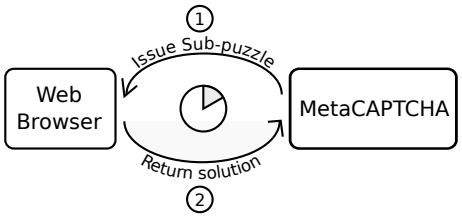


Figure 6.4: The user's web browser is continuously issued puzzles until it has spent enough time computing. This amount of time is called the difficulty-level; more malicious the client, the larger the puzzle difficulty-level.

It is important to note here that MetaCAPTCHA may issue multiple puzzles

during a single puzzle solving session. Puzzles are continuously issued until the client has computed for an amount of time similar to the estimated difficulty level. Pre-determining a difficulty level eliminates the usual incentive of solving puzzles faster. Furthermore, the estimated difficulty level is never directly revealed, thus the client cannot make the decision to stop solving based on the amount of work it needs to do. Figure 6.4 shows the puzzle solving protocol.

6.5.3 Puzzle Verification

Once the user's browser has solved all puzzles, it must send back the final solutions to MetaCAPTCHA. If the solutions are correct, MetaCAPTCHA will issue the client a *proof-of-work-ticket* $S_2 = T_s||T_e||HMAC(K, T_s||T_e||S_1)$, where T_s and T_e are the start and end time stamps of the puzzle solving session. The client must present this ticket to the web application (see Figure 6.3), which will then verify its integrity before allowing the client to complete posting the message. Additionally, if the difference between the current time and T_e is greater than some threshold t_{diff}, the client's proof-of-work ticket is considered to be expired and is subsequently rejected.

Verifying a puzzle solution usually involves a short-cut method described by the corresponding puzzle construction [108]. This ensures that the resources required to verify a puzzle solution is less than that required to compute or "solve" the puzzle. However, such short-cut methods may not always be available. Consider a volunteer computing project like BOINC [20]. BOINC enables people to contribute computational resources to scientific computing projects like protein folding [95, 128]. These projects divide the computational work among programs that can be executed simultaneously by multiple processors. Then, computers owned by BOINC volunteers retrieve these programs, execute them, and return the result to the respective project. Essentially, MetaCAPTCHA could treat each of these programs as a puzzle and issue them to clients. However, there are no readily

available short-cuts to verify the solutions of such puzzles.

MetaCAPTCHA employs the trust-but-verify approach to check puzzle solutions without having to recompute them. Here, the puzzle represents the generation function of the trust-but-verify approach. To build the verification function, MetaCAPTCHA must first be bootstrapped with a set of puzzles whose solutions are already known; let's call this the set of *known puzzles*. Then, during a puzzle solving session, MetaCAPTCHA issues a known puzzle with probability p or a *normal puzzle* otherwise (one whose solution is not known). As long as a client cannot distinguish between a normal puzzle and a known puzzle, it will have no choice but to faithfully compute the solutions to all puzzles.

A client that cheats, one that sends back random numbers instead of actually computing puzzle solutions, will eventually get caught. If we assume that a malicious client cheats with probability q when issued a puzzle, then the probability with which it will get caught is pq. The probability of the number of puzzles issued before a cheating client is caught for the first time can be modeled as the negative binomial distribution $\mathrm{NB}(1, 1-pq)$. Here, the first parameter is the number of cheat attempts before getting caught and the second parameter, $1-pq$, is the probability of not getting caught when issued a puzzle. Given these parameters, the number of puzzles expected to be issued before a cheating client is caught for the first time can be written as

$$\frac{1-pq}{pq}$$

Additionally, MetaCAPTCHA can add to its set of known puzzles by incorporating the solutions of normal puzzles submitted by honest clients. An honest client could be defined as one that gets the solutions to all known puzzles right. However, this does not mean that the client did not cheat at all, but that cheating was less likely. To see how likely, we quantify the probability that a cheater gets

the solutions to each of the randomly injected known puzzles right as:

$$\sum_{c=k}^{n} \binom{n}{c} (1-q)^c q^{n-c}$$

Here, n represents the total number of puzzles issued in a single puzzle solving session and k represents the number of known puzzles injected during that session. Recall, that a known puzzle is injected with probability p so we expect that $k = p \times n$. As indicated by the above formula, the cheater faithfully solves the k injected puzzles, but may or may not do the same for the rest of the puzzles.

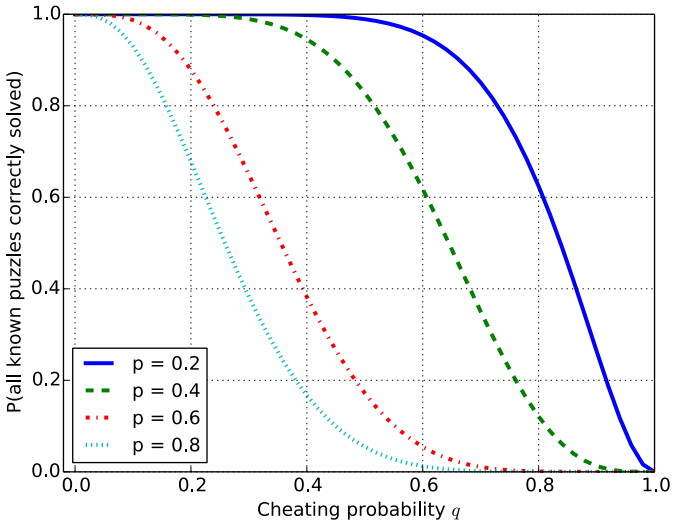


Figure 6.5: Probability that a cheating client correctly solves each of the randomly injected puzzles during a single puzzle solving session. The total number of puzzles issued is $n = 10$, the number of randomly injected puzzles is $k = p \times n$, and q is the probability with which a client cheats i.e. responds with a random number instead of actually solving the puzzle.

Figure 6.5 plots this probability for various values of p and q. We can see that when $p = 0.6$ and $q = 0.5$, the probability that a cheater gets all known puzzles right is less than 0.15. To further increase the confidence in the correctness of a puzzle solution incorporated in the above manner, MetaCAPTCHA could also

compare the solutions of multiple honest users. If a majority of honest clients get the solution to a particular normal puzzle correct, then the corresponding puzzle can be added to the known puzzle set. This approach is similar to the one used by reCAPTCHA [53] where two CAPTCHAs, one whose solution is known and the other not, are presented to the user.

6.6 SYSTEM COMPONENTS

In this section, we describe the main components of MetaCAPTCHA: the reputation service, the puzzle service, and the public API used by web applications and their clients to access the aforementioned services. Briefly, the interactions between these individual components begins when MetaCAPTCHA receives a client's message. This message is first handled by MetaCAPTCHA's reputation service, which determines the client's reputation score. The puzzle service then uses this score to generate and issue a puzzle of an appropriate difficulty. Figure 6.6 shows an overview of the various MetaCAPTCHA components and interactions, while the following sections describe each of them.

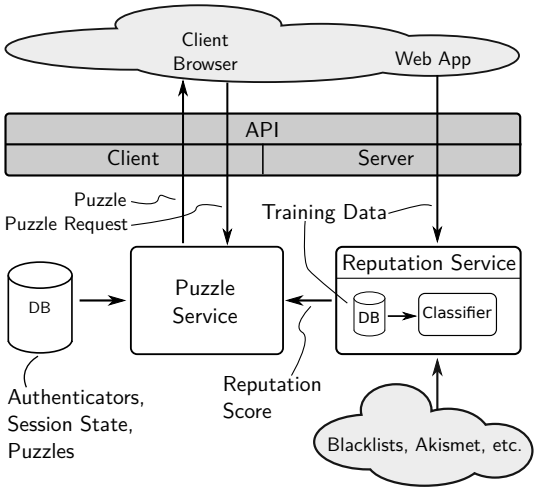


Figure 6.6: Design of MetaCAPTCHA

6.6.1 Reputation Service

Proof-of-work systems that do not assign more "work" to malicious clients than legitimate ones are easily circumvented [74]. Many existing systems do vary the amount of work, but fail to characterize maliciousness appropriately. For example, they base maliciousness on just one feature, such as system load [33, 67], a client's request rate [45, 46], contents of the request [149], or service demand [140, 141]. Without a defense-in-depth approach, it is unlikely that proof-of-work systems will be able to effectively deter automated adversaries. Additionally, if a reputation system intends to be widely deployed, it must be capable of adapting to the needs of individual applications [77]. For example, Twitter may associate low reputation with accounts that show aggressive following behavior [135], whereas Facebook may do the same for accounts with abnormally large amount of ignored friend requests [24]. MetaCAPTCHA's reputation service addresses these issues by allowing applications to easily configure the features that will determine a client's reputation score, and then use a Naive Bayes classifier to generically predict the score based on the values of the configured features.

The *reputation score* is the probability that a given message is spam as determined by a Naive Bayes classifier. A client's reputation score is calculated each time she posts a message to the web application and is dependent on the features of the message and the client that sent it. A *feature* is any metric with a finite set of values. For example, blacklist status of the message's source IP address, SpamAssasin score of the message, or number of times the poster was "thanked". Given such message features and any other client-related features provided by the web application, MetaCAPTCHA's reputation service can generate the client's reputation score.

An important characteristic of a reputation system is its ability to react to a client's changing reputation. For example, if a user's account is hijacked by a spammer, her account's reputation worsens; however, once the threat from the

hijacker is neutralized (say, by a password change) the reputation goes back to normal. Thus, a good reputation service must be capable of identifying these changes and assigning scores accordingly. MetaCAPTCHA's reputation service adapts to reputation changes by incorporating time-varying features in determining the reputation score. For example, relative account age, and relative number of positive votes a user's posts have received.

The reputation service is initialized by training the classifier using ground-truth feature values for messages that have already been posted. Training information about each message must include the values for each feature and its classification as spam or ham (not spam). The information about all messages is then fed to the classifier, which builds a probability model to determine how likely a given new message is spam. This likelihood or probability is called the reputation score and its value ranges from 0 to 1 with higher scores implying more malicious users.

6.6.2 Puzzle Service

The puzzle service is responsible for authenticating users, using the contents of their transaction (e.g. message, IP address) to obtain a reputation score from the reputation service, converting that score to a puzzle difficulty, and finally, issuing the user a puzzle of that difficulty. Note that the authentication protocol was described in Section 6.5.1. In the following sections, we discuss the remaining responsibilities of the puzzle service.

Reputation Score to Puzzle Difficulty

As mentioned before, puzzle difficulty is the amount of time a client must be kept busy solving a puzzle. Traditionally, most puzzles have been CPU bound, causing devices with different processing speeds to solve the same puzzle for different amounts of time. Abadi et al. highlighted this issue and proposed memory-bound functions since memory access speeds vary much less across devices [6]. Memory

bound puzzles, unfortunately, are expensive to create and verify [36, 30]. Furthermore, memory-bound puzzles limit the types of computation that can be performed: we envisage a future where puzzles are generic computations whose results are eventually reusable for solving larger problems like climate modeling, or curing cancer [128].

MetaCAPTCHA's approach is to first, determine the puzzle difficulty solely based on the reputation score — in units of time — and then, continuously issue puzzles until the client has computed for that pre-determined amount of time. The advantage of this approach is that it gets rid of an adversary's incentive to solve puzzles quicker (e.g by offloading, or parallelizing the computation). What is needed then, is a formula for converting the reputation score to a puzzle difficulty; the rest of this section derives such a formula.

Intuitively, the formula must ensure that puzzle difficulty is proportional to the reputation score since larger scores imply more malicious users. The remaining questions, then, are (i) how fast should the difficulty grow with respect to the reputation score?, and (ii) for any given reputation score, what should the difficulty value be to effectively reduce the amount of spam the web application receives?

The answer to Question (ii) is inspired by work on the impact of proof-of-work systems on reducing spam by Laurie and Clayton [74]. We begin by fixing the amount of spam reduction δ the web application seeks as a fraction of the total number of spam messages s_p it receives in time period t_p. We can then determine the maximum difficulty or amount of time t_{max} a spammer must be kept busy to reduce the spam to $s_p - \delta s_p$:

$$t_{max} = \frac{t_p}{s_p(1-\delta)}$$

Notice that if the desired spam reduction $\delta = 1$, the hardest puzzle a spammer may have to solve would be infinitely long; causing MetaCAPTCHA to forever wait for a solution! Since this is not feasible, the spam reduction fraction δ must

be judiciously chosen. Additionally, the tighter the choice of t_p for the same s_p, the more accurate t_{max} will be. For example, assume a forum receives its first spam message of the day at 8:00 am and last spam message at 5:00 pm. Then, choosing $t_p = 9$ hours as opposed to, say 24 hours, will lead to a more accurate value of t_{max}.

Now, the answer to question (i) depends on how accurately the reputation service can determine a user's reputation score; the more accurate it is, the less time honest clients should have to spend solving puzzles. We will see later that MetaCAPTCHA's reputation service aptly assigns $\approx 90\%$ of spammers a score over 0.95. Since the reputation service is fairly accurate, we must fashion a function that grows slowly until large reputation scores and then steeply afterwards. In the case of the reference web application used to evaluate MetaCAPTCHA (see Section 6.8.1), we empirically settled on the exponential function with a growth constant of 5. However, another web application could choose a different growth constant based on the shape of the curve desired.

Given the aforementioned growth function and maximum puzzle difficulty t_{max}, we can compute the corresponding maximum reputation score r_{max}:

$$t_{max} = e^{5r_{max}} - 1$$
$$\implies r_{max} = \frac{\ln(t_{max} + 1)}{5}$$

We can then normalize the user's current reputation score r with respect to r_{max} and subsequently calculate puzzle difficulty t:

$$t = e^{5r \cdot r_{max}} - 1$$
$$\implies t = (t_{max} + 1)^r - 1$$

Notice that when reputation score $r = 1$ (most malicious) the puzzle difficulty $t = t_{max}$. Furthermore, when $r = 0$ (most honest), $t = 0$. This implies that an honest user may not have to solve a puzzle at all, whereas a malicious user may have to solve the hardest one.

Issuing Puzzles

Once the puzzle difficulty t is determined, the puzzle service randomly generates a puzzle based on the list that is configured. The puzzle is then issued to the client who must solve it and return a solution. If the solution is returned in time $t' < t$, then a new puzzle is chosen and issued. This process is repeated until the client has computed for *at least* t amount of time. The idea behind issuing several puzzles is to ensure that no user can complete an online transaction unless they have computed for a length of time $\geq t$. An alternative is to first determine how long it takes to solve a puzzle, and then just generate and issue that puzzle. Unfortunately, the amount of time it takes to solve a puzzle varies on different platforms, and clients may get issued unfairly long or short puzzles [6, 39]. For this reason, MetaCAPTCHA first computes puzzle difficulty in units of time, and then has the client solve puzzles for that amount of time. Also, notice that the puzzle difficulty t is never revealed to the client, thus, there is no way to know how long the computation will last. This creates a disincentive for bots that would normally abandon puzzle computation altogether if t were known to be large beforehand. An additional advantage of issuing multiple puzzles is that each one can be randomly chosen, thus, preventing an attacker from being able to predict the puzzles she will be issued. This eliminates a critical advantage an adversary normally possesses: offline reverse engineering to find weaknesses. The next section discusses the various puzzle types supported by MetaCAPTCHA.

Puzzle Types

Essentially, a *puzzle type* is a parameterized function. A puzzle-type with an instantiated set of parameters is called a *puzzle*. Puzzles that require human interaction to solve (e.g. CAPTCHA) are called *interactive* puzzles, while those that don't (e.g. proof-of-work), are called *non-interactive* puzzles. MetaCAPTCHA

additionally supports *hybrid* puzzles that have both an interactive and a proof-of-work component. The choice of which puzzle types to use depends on the web application's needs (See Figure 6.7).

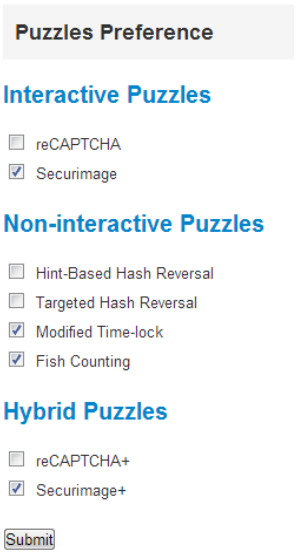


Figure 6.7: MetaCAPTCHA's puzzle configuration dashboard.

Thus, MetaCAPTCHA is quite flexible and can be easily configured to support new types of puzzles. In fact, MetaCAPTCHA is so named because it is a metamorphic puzzle issuing system and because it issues meta-puzzles rather than specific puzzles. It is metamorphic because the client-side puzzle solver code is non-deterministic. More specifically, the non-determinism results from issuing unpredictable puzzles within a generic puzzle solver — the meta-puzzle. Thus, the adversary has no way to know the client-side MetaCAPTCHA code beforehand. Furthermore, issuing a meta-puzzle ensures that finding a weakness in one puzzle type does not compromise MetaCAPTCHA as a whole. As mentioned earlier, a puzzle can additionally be *known* or *unknown*, depending on whether its solution already known or not. The following paragraphs discuss the currently supported puzzle types.

Hint-based Hash-Reversal (non-interactive) Hash-reversal puzzles force clients to reverse a given cryptographic hash of a random input, say x, with the k most significant bits erased. However, they lack fine-grained difficulty control because increasing k linearly, increases the solution search space exponentially. Hint-based hash-reversal puzzles address this drawback by providing an additional *hint*: the range of values to search.

Targeted Hash-Reversal (non-interactive) A targeted hash-reversal puzzle [45] with difficulty d forces a client compute an expected number d of hashes before finding the right answer.

Modified Time-Lock (non-interactive) Time-lock puzzles [108] are based on repeated squaring, a sequential process that forces the client to compute in a tight loop for an amount of time that can be precisely controlled. *Modified* time-lock puzzles on the other hand, retain most of the original properties of time-lock puzzles, but are faster to generate and verify [44].

Fish Counting (non-interactive) Requires clients to execute a machine learning algorithm that counts fishes in an underwater image of a section of Bonneville Dam [136]. The fish counting algorithm was adapted from an openly available algorithm that detects cats in an image [58]. Verifying the correctness of the solution involves issuing the same puzzle to multiple users and picking the majority answer as the correct one. This is similar to the verification mechanism currently used by reCAPTCHA [53]. The inclusion of this puzzle signifies a first step towards using MetaCAPTCHA as a platform for issuing proof-of-work puzzles whose solutions are useful in solving other problems.

CAPTCHA (interactive) A reCAPTCHA [138] or Securimage [99] CAPTCHA relayed to the client. MetaCAPTCHA only acts as a proxy for these CAPTCHA

services.

CAPTCHA+ (hybrid) A *CAPTCHA+* puzzle includes a reCAPTCHA or Securimage CAPTCHA along with a modified time-lock puzzle in the background. The advantage of combining the approaches, in this case, is that changing the difficulty of the time-lock puzzle changes the cost of solving the CAPTCHA. Consequently, hybrid puzzles could circumvent CAPTCHA outsourcing attacks [90, 91] since they enable the CAPTCHA solving cost to be changed.

6.6.3 Public API

The public API allows web applications and their clients to access MetaCAPTCHA services. A web application uses the server-side API to (i) register and maintain its account with MetaCAPTCHA, and (ii) as described in Section 6.5.1, use the account's unique API key to generate puzzle-request tickets for all its clients.

Web application clients implicitly use the client-side API during an online transaction, e.g. posting a message. More specifically, the API method calls are embedded in the client-side web page that accepts the online transaction (similar to a CAPTCHA setup). When the user "submits" the transaction, the client-side API is used to obtain a puzzle-request ticket from the web application and hand it over to the MetaCAPTCHA service. Once MetaCAPTCHA returns a puzzle, the client uses the solver — also a part of the client-side API — to compute and return the puzzle solution.

6.7 IMPLEMENTATION

MetaCAPTCHA has been deployed and has a public API that can be downloaded and used by interested parties. A Beta version of the MetaCAPTCHA service can be found at `http://www.metacaptcha.com/metacaptcha/`. We now discuss the implementation details of each MetaCAPTCHA component in Figure 6.6.

6.7.1 Reputation Service

The reputation service is implemented in PHP. Associated with this service is a NoSQL database, MongoDB [5], that stores the feature data necessary to determine user reputation. Initially, the web application provides the rows of feature data necessary to train a Naive Bayes classifier implemented in Java by the Weka [79] library. The trained classifier is then saved and used later when classifying new messages sent by the web application's users. Instead of using the binary classification — spam or ham — that normally is the output of a Naive Bayes classifier, the reputation service uses the probability distribution that the classifier determines in the step before performing the classification. That distribution provides the percentage likelihood that a given message is spam and this very likelihood represents MetaCAPTCHA's reputation score.

To enable the reputation service to compute accurate reputation scores, a web application can provide existing message and user data for training the classifier. In the case of our reference web application, a live discussion forum that employed MetaCAPTCHA's spam prevention services from September 1^{st} to October 19^{th} 2012, the classifier was given the following feature values for each existing message in the forum:

- Relative "Thanks" or "Likes": the proportion of positive votes received by the sender of the message.
- Language: the language the forum message was written in.
- Relative account age: the proportion of time an account has been alive with respect to the age of the forum.
- Relative post count: the proportion of total posts published by a given account.

- DShield "Attacks" attribute: number of packets, from the message's source to a distinct destination, that were blocked.
- GEOIP: an estimate of the distance between the message poster and forum server.
- Blacklist score: reputation score of the message source from Spamhaus [130]. The higher the score, the more malicious the source.
- Akismet score: Akismet [10] is a spam detection service that assigns a score of 1 to a message it thinks is spam and 0 otherwise.
- SA Score: the spam score as determined by the SpamAssassin [120] service running with only the Bayes plugin. The Bayes plugin uses a Naive Bayes classifier to determine the probability that the contents of a message resemble spam. SpamAssasin assigns a spam score between 1 and 5 to each message; 5 indicating that a message is most likely spam and 1 indicating that its not.
- isSpam: "yes" if forum moderator flagged the message as spam, "no" otherwise.

6.7.2 Puzzle Service

The puzzle service is implemented in PHP. Associated with this service is an instance of the mongoDB [5] that stores credentials necessary to authenticate a particular web application's clients, session details while the client is solving the issued puzzles, and the puzzles themselves. The authentication credentials include the web application's 96-bit API key and an application ID; the session details store the received server ticket (see Sections 6.5.1, 6.5.2) and the amount of time the client has spent solving puzzles. This amount of time when subtracted from the puzzle difficulty level determines if the client needs a new sub-puzzle or not (see Section 6.6.2); the puzzles themselves are stored in the database as members

of JSON (JavaScript Object Notation) objects and delivered in that format to clients. The parsing and execution of these JSON objects by client-side JavaScript engines can be thought of as "solving" a puzzle.

6.7.3 Client API

The client API includes JavaScript methods to request puzzles from MetaCAPTCHA, execute or "solve" them, and return the result of the execution. These methods must be embedded in an HTML form that accepts content from the users on behalf of the web application. As part of "submitting" that form, the client API will initiate the MetaCAPTCHA protocol to request a puzzle (see Section 6.5). The puzzle will be returned as a JSON object that the client must parse, evaluate, and then return the resulting value to MetaCAPTCHA.

The entire MetaCAPTCHA protocol occurs behind-the-scenes after a user clicks the "Submit" button. This behind-the-scenes behavior is enabled by the AJAX (Asynchronous JavaScript and XML) technique used to implement the MetaCAPTCHA communication protocol. Furthermore, puzzle execution is also pushed to the background by employing JavaScript worker threads [139] which are now supported in newer versions of most popular browsers.

6.7.4 Server API

The server API consists of $\approx$ 150 lines of PHP code and requires minor modifications to the web application for its default configuration. The modifications are similar to those required by existing CAPTCHA APIs like reCAPTCHA [138]. Web applications use the server API to receive a client's message, issue the corresponding server ticket necessary to request a puzzle from MetaCAPTCHA, and verify the proof-of-work presented by clients that have solved the issued puzzle (see Section 6.5).

6.8 RESULTS

We now evaluate MetaCAPTCHA and show that its defense-in-depth approach improves spammer identification, that this identification is accurate, and that it is an efficient spam prevention service.

6.8.1 Experimental Setup

The experimental setup used in our evaluation includes a MetaCAPTCHA server with a 2.4 GHz Intel Xeon quad-core processor running Red Hat Linux on a 2.6.18 kernel. A live discussion forum active from September 1^{st} to October 19^{th} 2012 employed MetaCAPTCHA as its spam prevention service. MetaCAPTCHA's effectiveness and performance has been evaluated in the context of this forum. At the time, the forum had 2282 messages from 485 users in 112 sub-forums containing 997 conversation threads. Upon registration, the forum provided most of this historical user and message data to help train MetaCAPTCHA's Naive Bayes classifier in identifying spam. Since the provided data was considered ground-truth, a part of it was used to train the classifier and the rest to evaluate it. The classification (spam or ham) was then compared with ground-truth to judge the classifier's effectiveness. The data consisted of values for all features described in Section 6.7.1 for each of 1442 messages posted to the forum. We now describe the experiments used to evaluate MetaCAPTCHA.

6.8.2 Defense-in-Depth

Defense-in-depth implies the use of multiple features to determine user reputation as opposed to only one or a few. Recall that a user's reputation score is the probability that the message she is posting is spam. This probability is determined by the Naive Bayes classifier. If the probability that a message is spam is higher

than the probability that it is not, the classifier tags the message as spam. Therefore, the better the classifier is at identifying spam, the better it is at identifying spammers and assigning appropriate reputation scores.

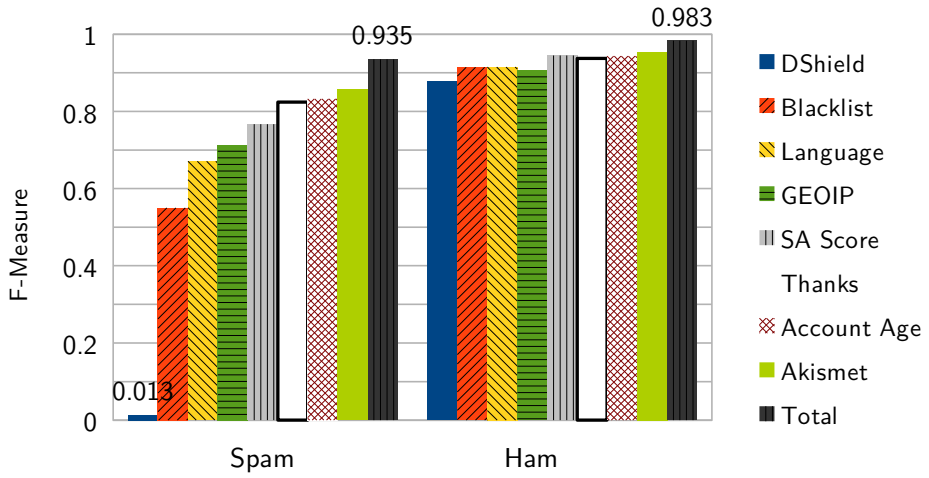
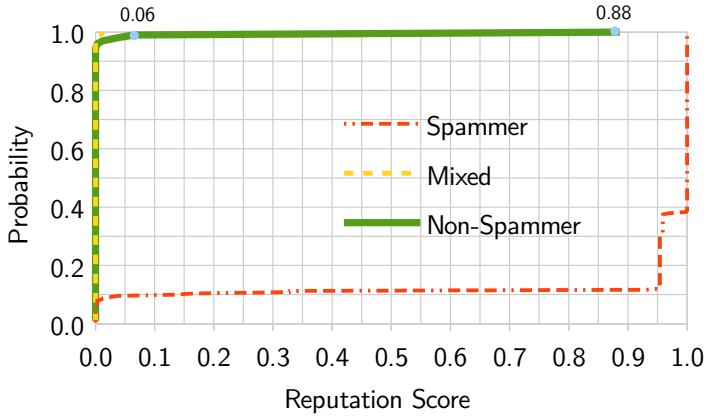


Figure 6.8: Defense-in-depth: using multiple features for spam classification is better than using one or a few. "Total" implies that all-of-the above features were used for training the classifier.

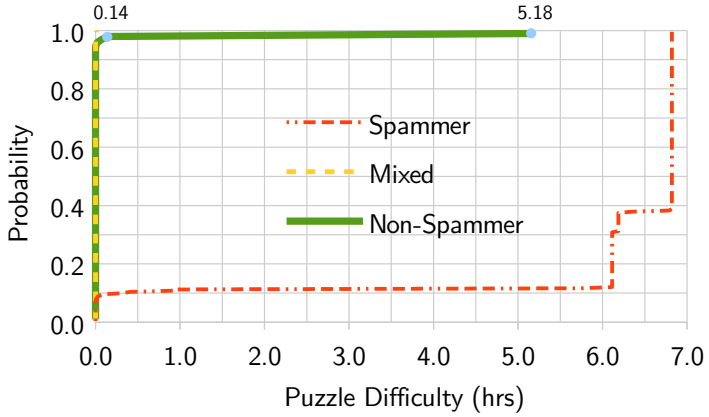
We evaluated the spam identification accuracy of the classifier by using standard machine learning techniques. The idea was to measure the classifier's *precision* and *recall*; *precision* is the fraction of messages that are actually spam (or ham) among those classified as spam (or ham); *recall* is the fraction of actual spam (or ham) that gets classified correctly. A commonly used combined metric is the harmonic mean of precision and recall, called the *F-measure*. Higher the F-measure, better the classifier is at identifying spam.

We used 10-fold cross-validation to train and test the classifier on feature data for 1442 messages. During each train-and-test run we limited the set of features that the classifier could use. More specifically, in all but the last run, the classifier was trained on one distinct feature. However, in the last run, it was trained on

all features together. The F-measure was then computed and plotted for each of the runs. We can see in Figure 6.8 that the classifier's F-measure is largest when using all features together than when using any single one.



(a) Reputation score accuracy



(b) Puzzle difficulty accuracy

Figure 6.9: Reputation Accuracy: CDF of reputation scores and puzzle difficulties assigned to spammers, non-spammers, and mixed users (those that sent at least 1 spam and 1 ham)

6.8.3 Reputation Accuracy

We evaluated the accuracy with which MetaCAPTCHA's reputation service distinguished between spammers and honest users. To do this, we first divided forum users into one of three categories, (i) *spammers*: those who sent only spam, (ii) *non-spammers*: those who sent no spam, and (ii) *mixed*: those who sent both spam and ham. Here, 'users' implies the senders of messages included in ground-truth information provided by the forum. After the categorization, there were 99 messages sent by non-spammers, 240 messages sent by spammers, and 151 messages sent by mixed users in the test set (34% of ground-truth data picked uniformly at random). We then fed these messages to MetaCAPTCHA's classifier and extracted the reputation scores from the output (note that reputation scores range from 0 to 1 and higher scores imply more malicious users). Finally, we plotted a CDF of reputation scores for each category of users.

Figure 6.9(a) shows that $\approx 90\%$ of *spammers* have reputation scores over 0.95, whereas $\approx 99\%$ of non-spammers got a reputation of 0.065 or less. Among the honest users, only one suffered the ill fate of being assigned a reputation of 0.88, whereas 94% were assigned a reputation of zero — implying that they did not solve a puzzle at all!

Interestingly, mixed users were treated largely as non-spammers. To understand why, we further analyzed messages sent by these users and realized that a majority of the users had posted vastly more ham than spam (see Figure 6.10). Thus, justifying their lower reputation scores.

Although reputation scores have accurately identified spammers from non-spammers, MetaCAPTCHA's success depends on issuing harder puzzles to more malicious users. This requires evaluating the function that converts reputation score to puzzle difficulty (see Section 6.6.2). We first computed the maximum puzzle difficulty $t_{max} = 6.82$ hrs based on time period $t_p = 1$ month, number of spam messages s_p seen in that month, and a spam reduction factor $\delta = 0.6$. We

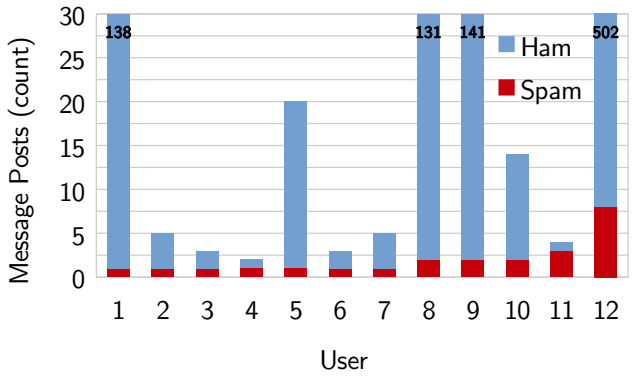
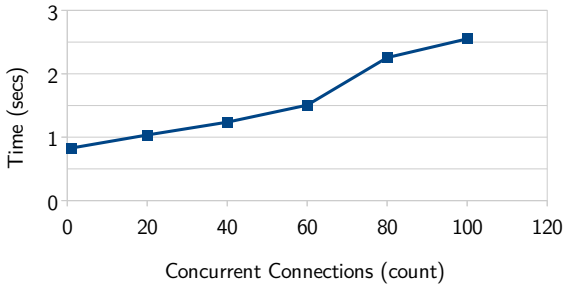


Figure 6.10: Distribution of spam and ham sent by mixed users. Mixed users sent very little spam (between 1 and 8) when compared to the total messages they posted. Note: columns that exceeded the y-scale have explicitly marked y-values

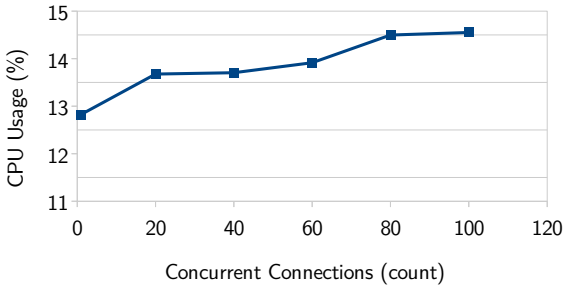
then plotted a CDF, shown in Figure 6.9(b), of the difficulty of puzzles issued to spammers, non-spammers, and mixed users for each message they sent. We can see that in this scenario, $\approx 90\%$ of spammers solved a puzzle over 6 hrs long, $\approx 5\%$ of non-spammers solved a puzzle between 7.2 secs and 8.4 minutes long, and $\approx 95\%$ of non-spammers solved *no puzzles at all*.

6.8.4 Performance

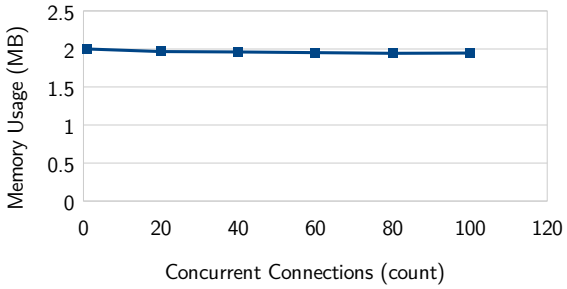
We evaluated the performance of MetaCAPTCHA in terms of CPU usage, memory consumption, and time spent in authenticating and issuing puzzles to users. We used Apache JMeter [126], a Java application that load-tests servers, to generate 1 - 100 concurrent puzzle requests incrementing each time by 20 to MetaCAPTCHA. Each test run (e.g 1, 20, 40, etc.) was repeated over a 100 times and the average measurement (e.g CPU usage) was plotted against the number of concurrent connections. The 95% confidence interval for the mean of each measurement was also calculated. However, the intervals might be too small to spot in the graphs.



(a) Time



(b) CPU usage



(c) Memory used

Figure 6.11: Performance overhead of MetaCAPTCHA when issuing the first puzzle.

Figures 6.11(a) - 6.11(c) show the amount of time, CPU, and memory consumed to authenticate the user, determine reputation score and puzzle difficulty, and generate the first puzzle. Although the time consumed was not prohibitive,

we were interested in determining where most of it was used. A more detailed analysis, shown in Figure 6.12, revealed that a majority of the time was spent in the reputation service when querying other remote services like Akismet [10], and blacklists like Spamhaus [130].

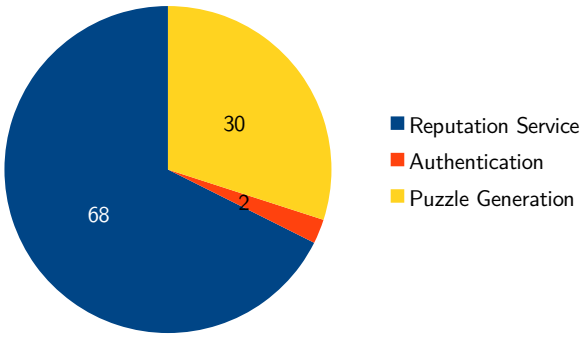
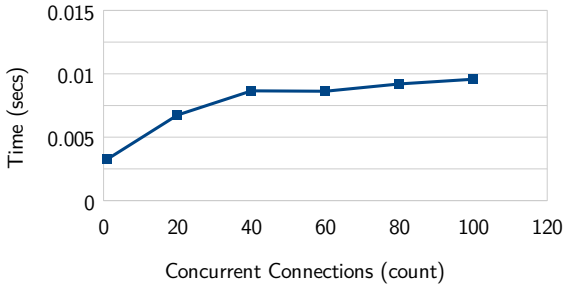


Figure 6.12: Breakdown (%) of the time spent in issuing the first puzzle. Notice that 68% of the time is spent in the reputation service due to all the remote queries that happen there.

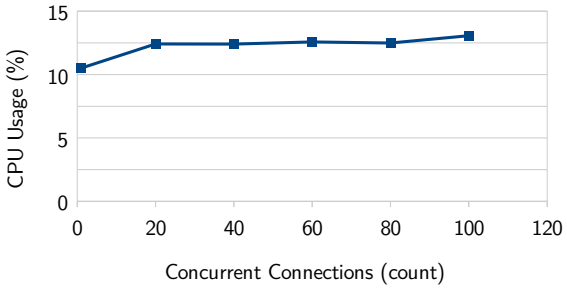
In the future, we hope to eliminate remote queries and mirror the applicable blacklists to significantly reduce the time required for issuing the first puzzle. Note that after the first puzzle, issuing subsequent ones only requires generating a new random puzzle (without the need for computing puzzle difficulty, or authenticating the user). Figures 6.13(a) - 6.13(c) depict the resources consumed while issuing subsequent puzzles.

6.9 SECURITY ANALYSIS

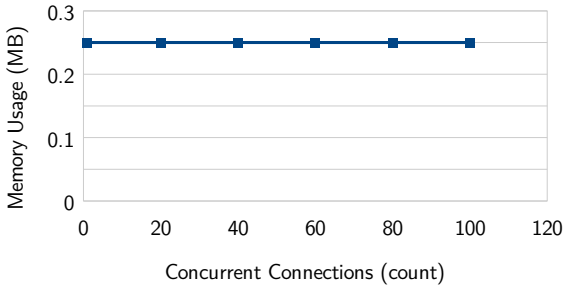
MetaCAPTCHA's goal is to address threats from automated adversaries like spam bots. The following paragraphs discuss those threats and how MetaCAPTCHA defends against them.



(a) Time



(b) CPU usage



(c) Memory used

Figure 6.13: MetaCAPTCHA performance overhead when issuing subsequent puzzles.

Bots may attempt to post spam in the web application. However, with MetaCAPTCHA protections in place, those attempts will result in a puzzle-request-ticket. Thus, preventing any efforts to directly post spam.

Bots may attempt to show proof-of-work without ever doing the work. However, they will be unable to forge a proof-of-work-ticket without the web application's secret API key.

Bots could present a proof-of-work-ticket for one message, but try to post another. However, since proof-of-work-tickets contain a digest of the original message, the ticket's verification will fail when associated with a new message.

Bots could replay old proof-of-work-tickets. However, as mentioned in Section 6.5.2, clients only have a small amount of time t_{diff} to submit the proof-of-work ticket. Thus, the same ticket cannot be replayed after t_{diff} time.

Bots could find short-cut methods to solve puzzles. However, since MetaCAPTCHA forces adversaries to solve puzzles for a pre-determined amount of time (the puzzle difficulty), solving a puzzle faster will only result in more puzzles to solve.

Bots may attempt to reuse the solutions of puzzles solved in the past. Recall, that puzzles are randomly selected and parameterized before being issued. Thus, bots will have to store an old puzzle in hopes of finding an exact match sometime in the future. We conjecture that this probability is negligible for the types of puzzles currently supported.

MetaCAPTCHA could be the target of a DoS attack where a flood of puzzle requests cause it to create state for an unsustainable number of puzzle solving sessions. However, effects of such attacks can be mitigated by using puzzle outsourcing techniques [142].

MetaCAPTCHA determines a user's reputation based on information related to messages posted by that user (e.g. contents, source IP). This may be a privacy concern for those who may trust the web application with their messages, but not MetaCAPTCHA. A possible solution to this problem is to eliminate privacy-sensitive features from being used for determining reputation. The drawback, however, would be reduced reputation score accuracy. Another way, would be

to empower the application to provide a local reputation score based on privacy-sensitive features. This local score could then be combined with the one determined remotely by MetaCAPTCHA to provide an accurate characterization of reputation. We hope to explore these avenues in future research.

Currently, MetaCAPTCHA issues puzzles without considering the platform it will be solved on. Thus, some clients may solve puzzles for longer than the amount of time determined by MetaCAPTCHA. One possibility is to use browser fingerprinting techniques [92] to determine a client's CPU speed and then issue puzzles accordingly. We also hope to address this direction of research in future work.

6.10 RELATED WORK

This section discusses relevant *spam prevention* schemes and how they relate to MetaCAPTCHA: the two prevalent ones are CAPTCHAs and proof-of-work [59].

CAPTCHAs come in many shapes and forms: textual CAPTCHAs require users to identify distorted letters [138, 99], visual CAPTCHAs require users to identify the content or characteristics of an image (e.g. orientation [54]), and audio CAPTCHAs usually require users to identify words in a noisy environment [118]. However, CAPTCHAs are not always fun to solve, so systems like Mollom [86] selectively issue them to only those users that appear to be posting spam. CAPTCHAs have also helped digitize books: words in scanned books that cannot be deciphered by character recognition programs are used as reCAPTCHAs [138]. MetaCAPTCHA can incorporate the above CAPTCHAs with the added benefit of a difficulty setting.

Proof-of-work systems that discourage spam include Hashcash, a system that requires senders to attach "postage" to e-mail [15]. The postage is a partial hash collision on a string derived from the recipient's email address. Another proof-of-work solution for throttling email spam was presented by Zhong et al. [149].

However, unlike Hashcash, their system based puzzle difficulty on the "spamminess" of the message. Feng et al. proposed kaPoW [44], a reputation-based proof-of-work system to discourage spam in webmail. There have also been proposals to put proof-of-work to good use. Jakobsson and Juels first suggested reusing the solutions to proof-of-work puzzles [64]. They described how a coin-minting operation could be broken up into several proof-of-work puzzles. CloudFlare [55] recently proposed using collaborative distributed computations like protein folding [128] as proof-of-work puzzles. MetaCAPTCHA incorporates the features of above proof-of-work systems while augmenting them with a generic puzzle issuing and verification mechanism along with a comprehensive reputation service. Furthermore, MetaCAPTCHA can be easily configured and used by generic web applications.

6.11 CONCLUSION AND FUTURE WORK

We presented MetaCAPTCHA, an application-agnostic spam prevention service for the web. MetaCAPTCHA seamlessly integrates the CAPTCHA and proof-of-work approaches while augmenting each: it can dynamically issue proof-of-work or CAPTCHA puzzles while ensuring that malicious users solve much "harder" puzzles than honest users. Web applications can configure MetaCAPTCHA to issue different classes of puzzles and even add new ones; regardless of whether there are short-cut methods to check their solutions. A configurable library of puzzles also ensures that weaknesses in one class of puzzles won't compromise MetaCAPTCHA as a whole. We evaluated MetaCAPTCHA in the context of a reference web application and showed that 95% of honest users hardly notice MetaCAPTCHA's presence, whereas the remaining 5% were required to solve very "easy" puzzles before accessing the application's services. In the future, we hope to improve MetaCAPTCHA's reputation system by incorporating new spam classification algorithms. Additionally, we would like to work towards transforming

MetaCAPTCHA into a proxy for volunteer computing projects like BOINC [20], so that useful computations like protein folding [128] can be issued as proof-of-work puzzles.

Chapter 7

CONCLUSION

Online applications that encourage open participation remain vulnerable to spurious information. This dissertation presented the *trust-but-verify* approach, a framework that enables applications to determine the integrity of the received information and thereby reject information that is spurious. The trust-but-verify approach enables applications to independently verify the integrity of data published by each user, and do so as often as necessary. This, in turn, enables the application to verify less information from a participant it trusts more, and verify more information from a participant it trusts less. How much of the received information is checked, depends on how tolerant the application is to spurious information and the resources it can devote to integrity checking. Thus, an application can trade-off performance for more integrity, or vice versa. The key idea behind the trust-but-verify approach is that it first identifies or defines data generation functions that data sources will use to create and publish information, and then, enables the respective application to verify that the data generation functions were faithfully executed. The challenge is in building verification methods that satisfy the constraints of individual applications. This dissertation described how the trust-but-verify approach can be used to enable high-integrity privacy-preserving crowd-sourced sensing, non-intrusive cheat detection in online games, and effective spam prevention in online messaging applications.

7.1 FUTURE DIRECTIONS

This dissertation developed verification procedures for information received in specific applications. However, a future area of research is to develop general verification procedures for entire classes of applications. Also, the trust-but-verify model currently assumes a single intermediary aggregator. In the future, it will be useful to evaluate the trust-but-verify approach in a model where there are multiple aggregators accepting data from multiple overlapping sources. Another interesting avenue for future research, is the feasibility of using the trust-but-verify approach in "big data" systems.

7.2 CONTRIBUTION SUMMARY

Table 7.1 highlights the contributions of the trust-but-verify approach in crowd-sourced sensing applications, online games, and spam prevention systems.

App. Domain	**Threats**	**State-of-art**	**Our Approach**
Crowd-sourced sensing	*Integrity:* masquerading, data fabrication; *Privacy:* location exposure	Reputation rankings: not resistant to masquerading (a.k.a Sybil) attacks, no privacy	Per-participant root-of-trust + Private data verification using homomorphic commitments
Online games	Information exposure cheats	Server-side view tracking: too expensive	Client tracks view, server checks view
Spam prevention systems	spam	spam filters: don't reduce spam; CAPTCHA: can be fooled, can't protect hijacked accounts	impose cost per message: force adversary to solve a puzzle whose result is useful to another application

Table 7.1: Contribution Summary

References

[1] 10-ways-to-detect-a-fake-facebook-account. https://www.facebook.com/notes/k-care-shop/10-ways-to-detect-a-fake-facebook-account/480043898294.

[2] Facebook. http://www.facebook.com/.

[3] Wikipedia. http://www.wikipedia.com/.

[4] Wikipedia:Identifying reliable sources - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Wikipedia:Identifying_reliable_sources.

[5] I. 10gen. Mongodb. http://www.mongodb.org/.

[6] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. *ACM Transactions on Internet Technology (TOIT)*, 5(2):299–327, 2005. Presents a set of memory-bound functions that can be used as client puzzles.

[7] A. Abdul-Rahman and S. Hailes. A distributed trust model. In *Proceedings of the 1997 workshop on New security paradigms*, pages 48–60. ACM, 1998.

[8] Advanced Micro Devices. SVM: AMD's Virtualization Technology. www.xen.org/files/xs0106_amd_virtualization.pdf.

[9] E. Agapie, G. Chen, D. Houston, E. Howard, J. Kim, M. Mun, A. Mondschein, S. Reddy, R. Rosario, J. Ryder, et al. Seeing Our Signals: Combining

location traces and web-based models for personal discovery. In *Proceedings of HotMobile*, pages 6–10. ACM, 2008.

[10] Akismet. Comment spam prevention for your blog. http://akismet.com/.

[11] Alex Chitu, Google Operating System, Unofficial news and tips about Google. How Gmail Blocks Spam. http://googlesystem.blogspot.com/2007/10/how-gmail-blocks-spam.html, Oct 2007.

[12] Artek72. [Undetected] SC2MapPro - An External Map Hack/Bot. http://www.blizzhackers.cc/viewtopic.php?f=220&t=473310.

[13] Atmel Corporation. The Atmel Trusted Platform Module. www.atmel.com/dyn/resources/prod_documents/doc5128.pdf.

[14] A. Back. Hashcash faq. http://www.hashcash.org/faq/.

[15] A. Back et al. Hashcash-a denial of service counter-measure. *URL: http://www. hashcash. org/papers/hashcash. pdf*, 2002.

[16] N. Baughman and B. Levine. Cheat-proof playout for centralized and distributed online games. In *IEEE INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings*, volume 1, 2001.

[17] Bitcoin. Bitcoin - P2P digital currency. http://bitcoin.org/.

[18] Blizzard Entertainment Inc. StarCraft II. http://us.battle.net/sc2/en/.

[19] Blizzard Entertainment Inc. World of Warcraft. http://us.battle.net/wow/en/.

[20] BOINC. BOINC. http://boinc.berkeley.edu/.

[21] D. Boneh, X. Boyen, and H. Shacham. Short Group Signatures. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 41–55, 2004.

[22] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145. ACM, 2004.

[23] Bruce Schneier. Palladium and the TCPA. `http://www.schneier.com/crypto-gram-0208.html#1`.

[24] Caroline Ghiossi. Explaining Facebook Spam Prevention Systems. `https://blog.facebook.com/blog.php?post=403200567130`, June 2010.

[25] C. Castelluccia, E. Mykletun, and G. Tsudik. Efficient aggregation of encrypted data in wireless sensor networks. In *Mobile and Ubiquitous Systems: Networking and Services, 2005. MobiQuitous 2005. The Second Annual International Conference on*, pages 109–117. IEEE, 2005.

[26] V. Chatzigiannakis and S. Papavassiliou. Diagnosing anomalies and identifying faulty nodes in sensor networks. *Sensors Journal, IEEE*, 7(5):637–645, 2007.

[27] D. Chaum, I. Damgård, and J. van de Graaf. Multiparty computations ensuring privacy of each partys input and correctness of the result. In *Advances in Cryptology*, pages 87–119. Springer, 1987.

[28] D. Chaum and E. Van Heyst. Group Signatures. *Berlin: Springer-Verlag*, 265, 1991.

[29] N. Chitradevi, V. Palanisamy, K. Baskaran, and U. Nisha. Outlier aware data aggregation in distributed wireless sensor network using robust principal component analysis. In *Computing Communication and Networking Technologies (ICCCNT), 2010 International Conference on*, pages 1–9, 2010.

[30] F. Coelho. Exponential memory-bound functions for proof of work protocols. Technical report, Research Report A-370, CRI, École des mines de Paris, 2005.

[31] S. Consolvo, D. McDonald, T. Toscos, M. Chen, J. Froehlich, B. Harrison, P. Klasnja, A. LaMarca, L. LeGrand, R. Libby, et al. Activity sensing in the wild: a field trial of ubifit garden. In *Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1797–1806. ACM, 2008.

[32] Y.-A. de Montjoye, C. A. Hidalgo, M. Verleysen, and V. D. Blondel. Unique in the crowd: The privacy bounds of human mobility. *Scientific reports*, 3, 2013.

[33] D. Dean and A. Stubblefield. Using client puzzles to protect tls. In *Proceedings of the 10th USENIX Security Symposium*, pages 13–17, 2001.

[34] R. Dewri. Location privacy and attacker knowledge: Who are we fighting against? 2011.

[35] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *USENIX Security*, pages 21–21. USENIX Association, Berkeley, CA, USA, 2004.

[36] S. Doshi, F. Monrose, and A. D. Rubin. Efficient memory bound puzzles using pattern databases. In *Applied Cryptography and Network Security*, pages 98–113. Springer, 2006.

[37] J. Douceur. The Sybil Attack. In *Proceedings of the IPTPS workshop*. Springer, 2002.

[38] A. Dua, N. Bulusu, W. Feng, and W. Hu. Towards Trustworthy Participatory

Sensing. In *HotSec'09: Proceedings of the 4th USENIX Workshop on Hot Topics in Security*. USENIX Association Berkeley, CA, USA, 2009.

[39] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. *Advances in Cryptology-Crypto 2003*, pages 426–444, 2003.

[40] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Advances in CryptologyCRYPTO92*, pages 139–147. Springer, 1993.

[41] EBay. Electronics, Cars, Fashion, Collectibles, Coupons and More Online Shopping — eBay. `http://www.ebay.com/`.

[42] S. Eisenman, E. Miluzzo, N. Lane, R. Peterson, G. Ahn, and A. Campbell. The BikeNet mobile sensing system for cyclist experience mapping. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 87–101. ACM, 2007.

[43] D. Estrin. Participatory sensing: applications and architecture [Internet Predictions. *IEEE Internet Computing*, 14(1):12–42, Jan. 2010.

[44] W. Feng and E. Kaiser. kapow webmail: Effective disincentives against spam. *Proc. of 7th CEAS*, 2010.

[45] W.-c. Feng and E. Kaiser. The case for public work. In *IEEE Global Internet Symposium, 2007*, pages 43–48. IEEE, 2007.

[46] W.-c. Feng, E. Kaiser, and A. Luu. Design and implementation of network puzzles. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 4, pages 2372–2382. IEEE, 2005.

[47] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26. ACM New York, NY, USA, 2008.

[48] S. Ganeriwal, L. Balzano, and M. Srivastava. Reputation-based framework for high integrity sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 4(3):15, 2008.

[49] R. Ganti, N. Pham, Y. Tsai, and T. Abdelzaher. PoolView: stream privacy for grassroots participatory sensing. In *Proceedings of ACM SenSys*, pages 281–294, Raleigh, North Carolina, 2008. ACM.

[50] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. *ACM SIGOPS Operating Systems Review*, 37(5):206, 2003.

[51] Geoffrey A. Fowler, Shayndi Raice, Amir Efrati. Facebook, Twitter battle 'social' spam. `http://www.theaustralian.com.au/business/wall-street-journal/facebook-twitter-battle-social-spam/story-fnay3ubk-1226237108998`, Jan 2012.

[52] A. Gkoulalas-Divanis, P. Kalnis, and V. S. Verykios. Providing k-anonymity in location based services. *ACM SIGKDD Explorations Newsletter*, 12(1):3–10, 2010.

[53] Google. recaptcha: Stop spam, read books. `http://www.google.com/recaptcha`.

[54] R. Gossweiler, M. Kamvar, and S. Baluja. What's up captcha?: a captcha

based on image orientation. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 841–850, New York, NY, USA, 2009. ACM.

[55] J. Graham-Cumming. Turning "i'm under attack" into "i'm doing some good". `http://blog.cloudflare.com/turning-im-under-attack-into-im-doing-some-go`, Aug 2012.

[56] C. Grier, K. Thomas, V. Paxson, and M. Zhang. @spam: the underground on 140 characters or less. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 27–37, New York, NY, USA, 2010. ACM.

[57] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 31–42. ACM, 2003.

[58] Heather Arthur. Face detection for cats in javascript. `https://github.com/harthur/kittydar`.

[59] P. Heymann, G. Koutrika, and H. Garcia-Molina. Fighting spam on social web sites: A survey of approaches and future challenges. *Internet Computing, IEEE*, 11(6):36–45, 2007.

[60] W. Hu, P. Corke, W. C. Shih, and L. Overs. secfleck: A public key technology platform for wireless sensor networks. In *Proceedings of EWSN*, pages 296–311, Cork, Ireland, 2009.

[61] K. L. Huang, S. S. Kanhere, and W. Hu. Towards privacy-sensitive participatory sensing. In *Percom'09: International Conference on Pervasive Computing and Communications*, pages 1–6. IEEE, 2009.

[62] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. Miu, E. Shih, H. Balakrishnan, and S. Madden. Cartel: a distributed mobile sensor computing system. In *Proceedings of ACM SenSys*, pages 125–138, Boulder, Colorado, 2006. ACM.

[63] Intel Corporation. Intel Trusted Execution Technology. `http://www.intel.com/technology/security/`.

[64] M. Jakobsson and A. Juels. Proofs of work and bread pudding protocols. In *Communications and Multimedia Security*, pages 258–272, 1999.

[65] I. Jolliffe. *Principal component analysis*. Wiley Online Library, 2005.

[66] A. Jsang and R. Ismail. The beta reputation system. In *Proceedings of the 15th bled electronic commerce conference*, pages 41–55, 2002.

[67] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. NDSS, 1999.

[68] E. Kaiser and W. Feng. Helping ticketmaster: Changing the economics of ticket robots with geographic proof-of-work. In *INFOCOM IEEE Conference on Computer Communications Workshops, 2010*, pages 1–6. IEEE, 2010.

[69] E. Kaiser, W. Feng, and T. Schluessler. Fides: Remote anomaly-based cheat detection using client emulation. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 269–279. ACM, 2009.

[70] P. Kalnis, G. Ghinita, K. Mouratidis, and D. Papadias. Preventing location-based identity inference in anonymous spatial queries. *Knowledge and Data Engineering, IEEE Transactions on*, 19(12):1719–1733, 2007.

[71] A. Kapadia, N. Triandopoulos, C. Cornelius, D. Peebles, and D. Kotz. AnonySense: Opportunistic and Privacy Preserving Context Collection. *LNCS*, 5013:280, 2008.

[72] Kelly Jackson Higgins for Dark Reading. Smartphone Weather App Builds A Mobile Botnet. http://www.darkreading.com/insiderthreat/security/client/showArticle.jhtml?articleID=223200001.

[73] J. Krumm. Inference Attacks on Location Tracks. In *Proceedings of the Fifth International Conference on Pervasive Computing (Pervasive)*, volume 4480, pages 127–143. Citeseer, 2007.

[74] B. Laurie and R. Clayton. Proof-of-work proves not to work. In *The Third Annual Workshop on Economics and Information Security*, 2004.

[75] K. Li, S. Ding, D. McCreary, and S. Webb. Analysis of state exposure control to prevent cheating in online games. *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video - NOSSDAV '04*, page 140, 2004.

[76] Z. Liang and W. Shi. Analysis of ratings on trust inference in open environments. *Performance Evaluation*, 65(2):99–128, 2008.

[77] D. Liu and L. Camp. Proof of work can work. In *Fifth Workshop on the Economics of Information Security*, 2006.

[78] M. Livani and M. Abadi. Distributed pca-based anomaly detection in wireless sensor networks. In *Internet Technology and Secured Transactions (ICITST), 2010 International Conference for*, pages 1–8, 2010.

[79] Machine Learning Group, University of Waikato. Weka 3 – data mining with open source machine learning software in java. http://www.cs.waikato.ac.nz/ml/weka/.

[80] Mark Risher. Social Spam and Abuse — Annual Trend Review. http://blog.impermium.com/2012/01/13/social-spam-and-abuse-the-year-in-review/, Jan 2012.

[81] Mark Ward. Warcraft game maker in spying row. http://news.bbc.co.uk/2/hi/technology/4385050.stm, Oct 2005.

[82] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of ACM SIGOPS/EuroSys*, pages 315–328, Glasgow, Scotland, 2008. ACM.

[83] S. Miller II. Beyond the hype of OnLive. http://www.jsonline.com/blogs/entertainment/41834997.html, Mar 2009.

[84] A. Modine. World of Warcraft spykit gets encrypted. http://www.theregister.co.uk/2007/11/15/world_of_warcraft_warden_encryption/, Nov 2007. The Register.

[85] M. Mokbel, C. Chow, and W. Aref. The new casper: query processing for location services without compromising privacy. In *Proceedings of the 32nd international conference on Very large data bases*, pages 763–774. VLDB Endowment, 2006.

[86] Mollom. How mollom works — mollom. http://mollom.com/how-mollom-works.

[87] C. Mönch, G. Grimen, and R. Midtstraum. Protecting online games against cheating. *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games - NetGames '06*, page 20, 2006.

[88] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *Knowledge and Data Engineering, IEEE Transactions on*, 13(1):124–141, 2001.

[89] T. Moran. The qilin crypto sdk. http://qilin.seas.harvard.edu/.

[90] M. Motoyama, K. Levchenko, C. Kanich, D. McCoy, G. M. Voelker, and S. Savage. Re: Captchas–understanding captcha-solving services in an economic context. In *USENIX Security Symposium*, volume 10, 2010.

[91] M. Motoyama, D. McCoy, K. Levchenko, S. Savage, and G. M. Voelker. Dirty jobs: The role of freelance labor in web service abuse. In *Proceedings of the 20th USENIX conference on Security*, pages 14–14. USENIX Association, 2011.

[92] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham. Fingerprinting information in javascript implementations. In *Proceedings of Web*, volume 2, 2011.

[93] S. Nath, J. Liu, J. Miller, F. Zhao, and A. Santanche. SensorMap: a web site for sensors world-wide. In *Proceedings of ACM SenSys*, pages 373–374. ACM Press New York, NY, USA, 2006.

[94] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in cryptologyEUROCRYPT99*, pages 223–238. Springer, 1999.

[95] S. U. Pande Lab. Folding@home. `http://folding.stanford.edu/`.

[96] E. Paulos, R. Honicky, and E. Goodman. Sensing atmosphere. In *Workshop on Sensing on Everyday Mobile Phones in Support of Participatory Research*. Citeseer, 2007.

[97] E. Paulos, I. Smith, and R. Honicky. Participatory urbanism. `http://www.urban-atmospheres.net/ParticipatoryUrbanism/index.html`.

[98] T. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in CryptologyCRYPTO91*, pages 129–140. Springer, 1992.

[99] D. Phillips. Securimage php captcha — free captcha script. http://www.phpcaptcha.org/.

[100] R. Popa, H. Balakrishnan, and A. Blumberg. VPriv: Protecting privacy in location-based vehicular services. In *Proceedings of the 18th Usenix Security Symposium*, 2009.

[101] R. Popa, A. Blumberg, H. Balakrishnan, and F. Li. Privacy and accountability for location-based aggregate statistics. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 653–666. ACM, 2011.

[102] M. Pritchard. How to Hurt the Hackers:The Scoop on Internet Cheating and How You Can Combat It. http://www.gamasutra.com/view/feature/3149/how_to_hurt_the_hackers_the_scoop_.php?page=3, July 2000. Gamasutra The Art & Business of Making Games.

[103] B. Przydatek, D. Song, and A. Perrig. SIA: Secure Information Aggregation in Sensor Networks. In *Proceedings of ACM SenSys*, pages 255–265. ACM New York, NY, USA, 2003.

[104] M. Rassam, A. Zainal, and M. Maarof. One-class principal component classifier for anomaly detection in wireless sensor network. In *Computational Aspects of Social Networks (CASoN), 2012 Fourth International Conference on*, pages 271–276, 2012.

[105] V. Rastogi and S. Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *Proceedings of the 2010 international conference on Management of data*, pages 735–746. ACM, 2010.

[106] S. Reddy, A. Parker, J. Hyman, J. Burke, D. Estrin, and M. Hansen. Image browsing, processing, and clustering for participatory sensing: lessons from

a DietSense prototype. In *ACM SenSys*, pages 13–17, Cork, Ireland, 2007. ACM.

[107] P. Resnick, K. Kuwabara, R. Zeckhauser, and E. Friedman. Reputation systems. *Communications of the ACM*, 43(12):45–48, 2000.

[108] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. 1996.

[109] Ross Anderson. 'Trusted Computing' Frequently Asked Questions. `http://www.cl.cam.ac.uk/~rja14/tcpa-faq.html`.

[110] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of USENIX Security*, pages 223–238, 2004.

[111] P. Samarati and L. Sweeney. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. Technical report, Technical report, SRI International, 1998.

[112] M. Schramm. Blizzard's new Warden, and our privacy. `http://wow.joystiq.com/2007/11/15/blizzards-new-warden-and-our-privacy/`, Nov 2007. WoW Insider.

[113] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. *Proceedings of ACM SIGOPS*, 39(5):1–16, 2005.

[114] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 272–282. Citeseer, 2004.

[115] E. Shi, T. H. Chan, E. Rieffel, R. Chow, and D. Song. Privacy-preserving aggregation of time-series data. In *Proceedings of NDSS*, volume 17, 2011.

[116] J. Shi, R. Zhang, Y. Liu, and Y. Zhang. PriSense: Privacy-Preserving Data Aggregation in People-Centric Urban Sensing Systems. In *IEEE INFOCOM*, 2010.

[117] P. Sikka, P. Corke, L. Overs, P. Valencia, and T. Wark. Fleck: A platform for real-world outdoor sensor networks. In *Intelligent Sensors, Sensor Networks and Information, 2007. ISSNIP 2007. 3rd International Conference on*, pages 709–714, 2007.

[118] Y. Soupionis and D. Gritzalis. Audio captcha: Existing solutions assessment and a new implementation for voip telephony. *Computers & Security*, 29(5):603–618, 2010.

[119] SPAM LAWS. Spam Statistics and Facts. `http://www.spamlaws.com/spam-stats.html`, 2011.

[120] SpamAssassin. The apache spamassassin project. `http://spamassassin.apache.org/`.

[121] spamcop.net. SpamCop.net: Beware of cheap imitations. `http://www.spamcop.net/`.

[122] F. Stajano and R. Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. *Lecture Notes in Computer Science*, 1796:172–182, 2000.

[123] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *USENIX conference proceedings*, volume 191, page 202, 1988.

[124] O. R. Team. List of weaknesses. http://ocr-research.org.ua/list.html.

[125] J. Thaler. WardenNet. http://www.ismods.com/warden/wardenfaq.php.

[126] The Apache Software Foundation. Apache jmeter. http://jmeter.apache.org/.

[127] The Common Criteria Recognition Agreement. CCRA - The Common Criteria Portal. http://www.commoncriteriaportal.org/.

[128] The Economist. Spreading the Load. http://www.economist.com/node/10202635, Dec 2007.

[129] The H. Security. Hacker extracts crypto key from TPM chip. http://www.h-online.com/security/news/item/Hacker-extracts-crypto-key-from-TPM-chip-927077.html, Feb 2010.

[130] The Spamhaus Project. About the Spamhaus Project. http://www.spamhaus.org/organization/index.lasso.

[131] A. Thiagarajan, L. Ravindranath, K. LaCurts, S. Madden, H. Balakrishnan, S. Toledo, and J. Eriksson. VTrack: accurate, energy-aware road traffic delay estimation using mobile phones. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 85–98. ACM, 2009.

[132] Trusted Computing Group. About TCG. http://www.trustedcomputinggroup.org/about_tcg.

[133] Trusted Computing Group. Platform Reset Attack Mitigation Specification, Version 1.0. http://www.trustedcomputinggroup.org/resources/pc_client_work_group_platform_reset_attack_mitigation_specification_version_10/.

[134] Trusted Computing Group. Trusted Platform Modue (TPM) Specifications. http://www.trustedcomputinggroup.org/developers/trusted_platform_module/specifications.

[135] Twitter Help Center. How to Report Spam on Twitter. http://support.twitter.com/articles/64986-how-to-report-spam-on-twitter.

[136] US Army Corps of Engineers. Bonneville lock and dam. http://www.nwp.usace.army.mil/Locations/ColumbiaRiver/Bonneville.aspx.

[137] J. Vilches. OnLive gets demoed, lag is a problem. http://www.techspot.com/news/37697-onlive-gets-demoed-lag-is-a-problem.html, Jan 2010.

[138] L. Von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum. recaptcha: Human-based character recognition via web security measures. *Science*, 321(5895):1465, 2008.

[139] w3schools.com. Html5 web workers. http://www.w3schools.com/html/html5_webworkers.asp.

[140] X. Wang and M. Reiter. Defending against denial-of-service attacks with puzzle auctions. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 78–92. IEEE, 2003.

[141] X. Wang and M. K. Reiter. Mitigating bandwidth-exhaustion attacks using congestion puzzles. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 257–267. ACM, 2004.

[142] B. Waters, A. Juels, J. A. Halderman, and E. W. Felten. New client puzzle outsourcing techniques for dos resistance. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 246–256, New York, NY, USA, 2004. ACM.

[143] Waze. Free GPS Navigation with Turn by Turn Directions. `http://www.waze.com/homepage/`.

[144] S. D. Webb and S. Soh. Cheating in networked computer games. *Proceedings of the 2nd international conference on Digital interactive media in entertainment and arts - DIMEA '07*, page 105, 2007.

[145] S. Webb, S. and Soh. A survey on network game cheats and P2P solutions. *Australian Journal of Intelligent Information*, 9(4):34–43, 2008.

[146] S. B. Wicker. The loss of location privacy in the cellular age. *Communications of the ACM*, 55(8):60–68, 2012.

[147] J. Yan and A. El Ahmad. Usability of captchas or usability issues in captcha design. In *Proceedings of the 4th symposium on Usable privacy and security*, pages 44–52. ACM, 2008.

[148] H. Zang and J. Bolot. Anonymization of location data does not work: A large-scale measurement study. In *Proceedings of the 17th annual international conference on Mobile computing and networking*, pages 145–156. ACM, 2011.

[149] Z. Zhong, K. Huang, and K. Li. Throttling outgoing spam for webmail services. In *Conference on Email and Anti-Spam*, 2005.

[150] S. Zhu, S. Setia, S. Jajodia, and P. Ning. An interleaved hop-by-hop authentication scheme for filtering of injected false data in sensor networks. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 259–271, 2004.