CACTI: Captcha Avoidance via Client-side TEE Integration

Yoshimichi Nakatsuka*
UC Irvine
nakatsuy@uci.edu

Ercan Ozturk*
UC Irvine
ercano@uci.edu

Andrew Paverd†
Microsoft Research
andrew.paverd@microsoft.com

Gene Tsudik
UC Irvine
gene.tsudik@uci.edu

Abstract

Preventing abuse of web services by bots is an increasingly important problem, as abusive activities grow in both volume and variety. CAPTCHAs are the most common way for thwarting bot activities. However, they are often ineffective against bots and frustrating for humans. In addition, some recent CAPTCHA techniques diminish user privacy. Meanwhile, client-side Trusted Execution Environments (TEEs) are becoming increasingly widespread (notably, ARM TrustZone and Intel SGX), allowing establishment of trust in a small part (trust anchor or TCB) of client-side hardware. This prompts the question: can a TEE help reduce (or remove entirely) user burden of solving CAPTCHAs?

In this paper, we design CACTI: **C**APTCHA **A**voidance via **C**lient-side **T**EE **I**ntegration. Using client-side TEEs, CACTI allows legitimate clients to generate unforgeable *rate-proofs* demonstrating how frequently they have performed specific actions. These rate-proofs can be sent to web servers in lieu of solving CAPTCHAs. CACTI provides strong client privacy guarantees, since the information is only sent to the visited website and authenticated using a group signature scheme. Our evaluations show that overall latency of generating and verifying a CACTI rate-proof is less than 0.25 sec, while CACTI's bandwidth overhead is over 98% lower than that of current CAPTCHA systems.

1 Introduction

In the past two decades, as Web use became almost universal and abuse of Web services grew dramatically, there has been an increasing trend (and real need) to use security tools that help prevent abuse by automated means, i.e., so-called **bots**. The most popular mechanism is CAPTCHAs: Completely Automated Public Turing test to tell Computers and Humans Apart [58]. A CAPTCHA is essentially a puzzle, such as an object classification task (Figure 1a) or distorted text recognition (see Figure 1b), that aims to confound (or at least slow down) a bot, while being easily[1] solvable by a human user. CAPTCHAs are often used to protect sensitive actions, such as creating a new account or submitting a web form.

Although primarily intended to distinguish humans from bots, it has been shown that CAPTCHAs are not very effective at this task [50]. Many CAPTCHAs can be solved by algorithms (e.g., image recognition software) or outsourced to human-driven *CAPTCHA-farms*[2] to be solved on behalf of bots. Nevertheless, CAPTCHAs are still widely used to increase the adversary's costs (in terms of time and/or money) and reduce the *rate* at which bots can perform sensitive actions. For example, computer vision algorithms are computationally expensive, and outsourcing to CAPTCHA-farms costs money and takes time.

From the users' perspective, CAPTCHAs are generally unloved (if not outright hated), since they represent a barrier and an annoyance (a.k.a. Denial-of-Service) for legitimate users. Another major issue is that most CAPTCHAs are visual in nature, requiring sufficient ambient light and screen resolution, as well as good eyesight. Much less popular audio CAPTCHAs are notoriously poor, and require a quiet setting, decent-quality audio output facilities, as well as good hearing.

More recently, the reCAPTCHA approach has become popular. It aims to reduce user burden by having users click a checkbox (Figure 1c), while performing behavioral analysis of the user's browser interactions. Acknowledging that even this creates friction for users, the latest version ("invisible reCAPTCHA") does not require any user interaction. However, the reCAPTCHA approach is potentially detrimental to **user privacy** because it requires maintaining long-term state, e.g., in the form of Google-owned cookies. Cloudflare recently decided to move away from reCAPTCHA due to privacy concerns and changes in Google's business model [14].

Notably, all current CAPTCHA-like techniques are server-

*The first and second authors contributed equally to this work.

†Work partially done while visiting the University of California, Irvine, as a US-UK Fulbright Cyber Security Scholar.

[1] Exactly what it means to be "easily" solvable is subject to some debate.

[2] A CAPTCHA farm is usually sweatshop-like operation, where employees solve CAPTCHAs for a living.

side, i.e., they do not rely on any security features of, or make any trust assumptions about, the client platform. The purely server-side nature of CAPTCHAs was reasonable when client-side hardware security features were not widely available. However, this is rapidly changing with the increasing popularity of Trusted Execution Environments (TEEs) on a variety of computing platforms, e.g., TPM and Intel SGX for desktops/laptops and ARM TrustZone for smartphones and even smaller devices. Thus, it is now realistic to consider abuse prevention methods that include client-side components. For example, if a TEE has a trusted path to some form of user interface, such as a mouse, keyboard, or touchscreen, this *trusted User Interface (UI)* could securely confirm user presence. Although this feature is still unavailable on most platforms, it is emerging through features like Android's Protected Confirmation [33]. This approach's main advantages are minimized user burden (e.g., just a mouse click) and increased security, since it would be impossible for software to forge this action. Admittedly however, this approach can be defeated by adversarial hardware e.g., a programmable USB peripheral that pretends to be a mouse or keyboard.

However, since the majority of consumer devices do not currently have a trusted UI, it would be highly desirable to reduce the need for CAPTCHAs using only existing TEE functionality. As discussed above, the main goal of modern CAPTCHAs is to increase adversarial costs and reduce the *rate* at which they can perform sensitive actions. Therefore, if legitimate users had a way to prove that their rate of performing sensitive actions is below some threshold, a website could decide to allow these users to proceed without solving a CAPTCHA. If a user can not provide such a proof, the website could simply fall back to using CAPTCHAs. Though this would not fully prevent bots, it would not give them any advantage compared to the current arrangement of using CAPTCHAs.

Motivated by the above discussion, this paper presents CACTI, a flexible mechanism for allowing legitimate users to prove to websites that they are not acting in an abusive manner. By leveraging widespread and increasing availability of client-side TEEs, CACTI allows users to produce *rate-proofs*, which can be presented to websites in lieu of solving CAPTCHAs. A rate-proof is a simple assertion that:

1. The rate at which a user has performed some action is below a certain threshold, and
2. The user's time-based counter for this action has been incremented.

When serving a webpage, the server selects a *threshold* value and sends it to the client. If the client can produce a rate-proof for the given threshold, the server allows the action to proceed without showing a CAPTCHA. Otherwise, the server presents a CAPTCHA, as before. In essence, CACTI can be seen as a type of "express checkout" for legitimate users.

One of the guiding principles and goals of CACTI is user privacy – it reveals only the minimum amount of information and sends this directly to the visited website. Another principle is that the mechanism should not mandate any specific security policy for websites. Websites can define their own security policies e.g., by specifying thresholds for rate-proofs. Finally, CACTI should be configurable to operate without any user interaction, in order to make it accessible to all users, including those with sight or hearing disabilities.

Although chiefly motivated by the shortcomings of CAPTCHAs, we believe that the general approach of client-side (TEE-based) rate-proofs, can also be used in other common web scenarios. For example, news websites could allow users to read a limited number of articles for free per month, without relying on client side cookies (which can be cleared) or forcing users to log-in (which is detrimental to privacy). Online petition websites could check that users have not signed multiple times, without requiring users to provide their email addresses, which is once again, detrimental to privacy. We therefore believe that our TEE-based rate-proof concept is a versatile and useful web security primitive.

Anticipated contributions of this work are:

1. We introduce the concept of a *rate-proof*, a versatile web security primitive that allows legitimate users to securely prove that their rate of performing sensitive actions falls below a server-defined threshold.
2. We use the rate-proof as the basis for a concrete client-server protocol that allows legitimate users to present rate-proofs in lieu of solving CAPTCHAs.
3. We provide a proof-of-concept implementation of CACTI, over Intel SGX, realized as a Google Chrome browser extension.
4. We present a comprehensive evaluation of security, latency, and deployability of CACTI.

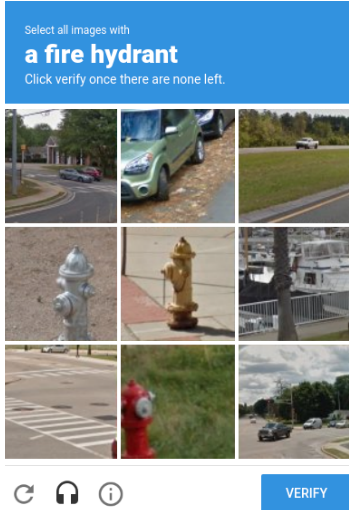
Organization: Section 2 provides background information, and Section 3 defines our threat model and security requirements. Next, Section 4 presents our overall design and highlights the main challenges in realizing this. Then, Section 5 explains our proof of concept implementation and discusses how CACTI overcomes the design challenges, followed by Section 6 which presents our evaluation of the security, performance, and deployability of CACTI. Section 7 discusses further optimizations and deployment considerations, and Section 8 summarizes related work.

2 Background

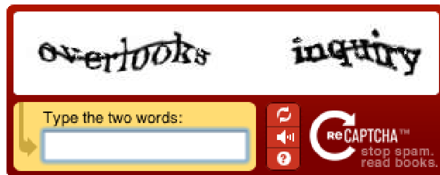
2.1 Trusted Execution Environments

A Trusted Execution Environment (TEE) is a primitive that protects confidentiality and integrity of security-sensitive code and data from untrusted code. A typical TEE provides the following features:

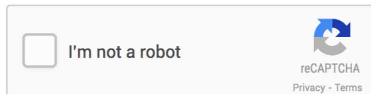
Isolated execution. The principal function of a TEE is to provide an execution environment that is isolated from all other software on the platform, including privileged system software, such as the OS, hypervisor, or BIOS. Specifically,



(a) Image-based object recognition reCAPTCHA [18]



(b) Image-based text recognition reCAPTCHA [18]



(c) Behavior-based reCAPTCHA [18]

Figure 1: Examples of CAPTCHAs

data inside the TEE can only be accessed by the code running inside the TEE. The code inside the TEE provides well-defined entry points (e.g., call gates), which are enforced by the TEE.

Remote attestation. Remote attestation provides a remote party with strong assurances about the TEE and the code running therein. Specifically, the TEE (i.e., the *prover*) creates a cryptographic assertion that: (1) demonstrates that it is a genuine TEE, and (2) unambiguously describes the code running in the TEE. The remote party (i.e., the *verifier*) can use this to decide whether to trust the TEE, and then to bootstrap a secure communication channel with the TEE.

Data sealing. Data sealing allows the code running inside the TEE to encrypt data such that it can be securely stored outside the TEE. This is typically implemented by providing the TEE with a symmetric *sealing key*, which can be used to encrypt/decrypt the data. In current TEEs, sealing keys are platform-specific, meaning that data can only be unsealed on the same platform on which it was sealed.

Hardware monotonic counters. A well known attack against sealed data is *rollback*, where the attacker replaces the sealed data with an older version.Mitigating this requires at least some amount of rollback-protected storage, typically realized as a hardware monotonic counter. When sealing, the counter can be incremented and the latest value is included in the sealed data. When unsealing, the TEE checks that the included value matches the current hardware counter value. Since hardware counters themselves require rollback-protected storage, TEEs typically only have a small number of counters.

One prominent TEE example is *Intel Software Guard Extensions (SGX)* [24,43,48]. SGX is a hardware-enforced TEE available on Intel CPUs from the Skylake microarchitecture onwards. SGX allows applications to create isolated environments, called *enclaves*, running in the application's virtual address space. A special region in physical memory is reserved for enclaves, called the Enclave Page Cache (EPC). The EPC can hold up to 128MB of code and data, shared between all running enclaves. When enclave data leaves the CPU boundary, it is transparently encrypted and integrity-protected by CPU's Memory Encryption Engine (MEE) to defend against physical bus snooping/tampering attacks. Since enclaves run in the application's virtual address space, enclave code can access all the memory of its host application, even that outside the enclave. Enclave code can only be called via predefined function calls, called `ECALL`s.

Every enclave has an enclave identity (`MRENCLAVE`), which is a cryptographic hash of the code that has been loaded into the enclave during initialization, and various other configuration details. Each enclave binary must be signed by the developer, and the hash of the developer's public key is stored as the enclave's signer identity (`MRSIGNER`).

SGX provides two types of attestation: local and remote. Local attestation allows two enclaves running on the same platform to confirm each other's identity and communicate securely, even though this communication goes via the untrusted OS. SGX uses local attestation to build remote attestation. Specifically, an application enclave performs local attestation with an Intel-provided *quoting enclave*, which holds a group private key provisioned by Intel. The quoting enclave verifies the local attestation and creates a signed *quote*, which includes the application enclave's and signer's identities, as well as user-defined data provided by the application enclave. This quote is sent to the remote verifier, which, in turn, uses the Intel Attestation Service (IAS) to verify it. Since the attestation uses a group signature scheme, the verifier cannot determine whether two quotes were generated by the same platform.

In SGX, data can be sealed in one of two modes, based on: (1) the enclave's identity, such that only the same type of enclave can unseal it, or (2) the signer identity, such that any enclave signed by the same developer (running on the same platform) can unseal it. SGX provides hardware monotonic counters and allows each enclave to use up to 256 counters at a time.

2.2 Group Signatures

A group signature scheme aims to prevent the verifier from determining the group member which generated the signature. Each group member is assigned a group private key under a single group public key. In case a group member needs to be revoked, a special entity called *group manager* can open the signature. A group signature scheme is composed of five algorithms [26]:

- **Setup:** Given a security parameter, an efficient algorithm outputs a group public key and a master secret for the group manager.
- **Join:** A user interacts with the group manager to receive a group private key and a membership certificate.
- **Sign:** Using the group public key, group private key, membership certificate, and a message m, a group member generates a group signature of m.
- **Verify:** Using the group public key, an entity verifies a group signature.
- **Open:** Given a message, a putative signature on the message, the group public key and the master secret, the group manager determines the identity of the signer.

A secure group signature scheme satisfies the following properties [26]:

- **Correctness:** Signatures generated with any member's group private key must be verifiable by the group public key.
- **Unforgeability:** Only an entity that holds a group private key can generate signatures.
- **Anonymity:** Given a group signature, it must be computationally hard for anyone (except the group manager) to identify the signer.
- **Unlinkability:** Given two signatures, it must be computationally hard to determine whether these were signed by the same group member.
- **Exculpability:** Neither a group member nor the group manager can generate signatures on behalf of other group members.
- **Traceability:** The group manager can determine the identity of a group member that generated a particular signature.
- **Coalition-resistance:** Group members cannot collude to create a signature that cannot be linked to one of the group members by the group manager.

Enhanced Privacy ID (EPID) [30] is a group signature scheme used by remote attestation of Intel SGX enclaves. It satisfies the above properties whilst providing additional privacy-preserving revocation mechanisms to revoke compromised or misbehaving group members. Specifically, EPID's *signature-based revocation* protocol does not "Open" signatures but rather uses a signature produced by the revoked member to notify other entities that this particular member has been revoked.

3 System & Threat Models

The ecosystem that we consider includes three types of principals/players: (1) servers, (2) clients, and (3) TEEs. There are multitudes of these three principal types. The number of clients is the same as that of TEEs, and each client houses exactly one TEE. Even though a TEE is assumed to be physically within a client, we consider it to be separate security entity. Note that a human user can, of course, operate or own multiple clients, although there is clearly a limit and more clients implies higher costs for the user.

We assume that all TEEs are trusted: honest, benign and insubvertible. We consider all side-channel and physical attacks against TEEs to be out of scope of this work and assume that all algorithms and cryptographic primitives implemented within TEEs are impervious to such attacks. We also consider cuckoo attacks, whereby a malicious client utilizes multiple (possibly malware infected) machines with genuine TEEs, to be out of scope, since clients and their TEEs are not considered to be strongly bound. We refer to [62] and [36] as far as means for countering such attacks. We assume that servers have a means to authenticate and attest TEEs, possibly with the help of the TEE manufacturer.

All clients and servers are untrusted, i.e., they may act maliciously. The goal of a malicious client is to avoid CAPTCHAs, while a malicious server either aims to inconvenience a client (via DoS) or violate client's privacy. For example, a malicious server can try to learn the client's identity or link multiple visits by the same client. Also, multiple servers may collude in an attempt to track clients.

Our threat model yields the following requirements for the anticipated system:

- **Unforgeability:** Clients cannot forge or modify CACTI rate-proofs.
- **Client privacy:** A server (or a group thereof) cannot link rate-proofs to the clients that generated them.

We also pose the following non-security goals:

- **Latency:** User-perceived latency should be minimized.
- **Data transfer:** The amount of data transfer between client and server should be minimized.
- **Deployability:** The system should be deployable on current off-the-shelf client and server hardware.

4 CACTI Design & Challenges

This section discusses the overall design of CACTI and justifies our design choices.

4.1 Conceptual Design

Rate-proofs. The central concept underpinning our design is the *rate-proof* (*RP*). Conceptually, the idea is as follows: Assuming that a client has an idealized TEE, the TEE stores

one or more named sorted lists of timestamps in its rollback-protected secure memory. To create a rate-proof for a specific list, the TEE is given the name of the list, a *threshold* (*Th*), and a new timestamp (t). The threshold is expressed as a starting time (t_s) and a count (k). This can be interpreted as: *"no more than k timestamps since t_s"*. The TEE checks that the specified list contains k or fewer timestamps with values greater than or equal to t_s. If so, it checks if the new timestamp t is greater than the latest timestamp in the list. If both checks succeed, the TEE pre-pends t to the list and produces a signed statement confirming that the named list is below the specified threshold and the new timestamp has been added. If either check fails, no changes are made to the list and no proof is produced. Note that the rate-proof does not disclose the number of timestamps in the list.

Furthermore, each list can also be associated with a public key. In this case, requests for rate-proofs must be accompanied by a signature over the request that can be verified with the associated public key. This allows the system to enforce a *same-origin* policy for specific lists – proofs over such lists can only be requested by the same entity that created them. Note that this does not provide any binding to the *identity* of the entity holding the private key, as doing so would necessitate the TEE to check identities against a global public key infrastructure (PKI) and we prefer for CACTI not to require it.

Rate-proofs differ from rate *limits* because the user is allowed to perform the action any number of times. However, once the rate exceeds the specified threshold, the user will no longer be able to produce rate-proofs. The client can always decide to *not* use its TEE; this covers clients who do not have TEEs or those whose rates exceeded the threshold. On the other hand, if the server does not yet support CACTI, the client does not store any timestamps, or perform any additional computation.

CAPTCHA-avoidance. In today's CAPTCHA-protected services, the typical interaction between the client (C) and server (S) proceeds as follows:

1. C requests access to a service on S.
2. S returns a CAPTCHA for C to solve.
3. C submits the solution to S.
4. If the solution is verified, S allows C access to the service.

Although modern approaches, e.g., reCAPTCHA, might include additional steps (e.g., communicating with third-party services), these can be abstracted into the above pattern.

Our CAPTCHA-avoidance protocol keeps the same interaction sequence, while substituting steps 2 and 3 with rate-proofs. Specifically, in step 2, the server sends a threshold rate and the current timestamp. In step 3, instead of solving a CAPTCHA, the client generates a rate-proof with the specified threshold and timestamp, and submits it to the server. The server has two types of lists:

- **Server-specific:** The server requests a rate-proof over its own list. The name of the list could be the server's URL, and the request may be signed by the server. This determines the rate at which the client visits this specific server.
- **Global:** The server requests a rate-proof over a global list, with a well-known name, e.g. CACTI-GLOBAL. This yields the rate at which the client visits all servers that use the global list.

The main idea of CAPTCHA avoidance is that a legitimate client should be able to prove that its rate is below the server-defined threshold. In other words, the server should have sufficient confidence that the client is not acting in an abusive manner (where the threshold of between abusive and non-abusive behaviors is set by the server). Servers can select their own thresholds according to their own security requirements. A given server can vary the threshold across different actions or even across different users or user groups, e.g., lower thresholds for suspected higher-risk users. If a client cannot produce a rate-proof, or is unwilling to do so, the server simply reverts to the current approach of showing a CAPTCHA. CACTI essentially provides a *fast-pass* for legitimate users.

The original CAPTCHA paper [58] suggested that CAPTCHAs could be used in the following scenarios:

1. **Online polls:** to prevent bots from voting,
2. **Free email services:** to prevent bots from registering for thousands of accounts,
3. **Search engine bots:** to preclude or inhibit indexing of websites by bots,
4. **Worms and spam:** to ensure that emails are sent by humans,
5. **Preventing dictionary attacks.** to limit the number of password attempts.

As discussed in Section 1, it is unrealistic to assume that CAPTCHAs cannot be solved by bots (e.g., using computer vision algorithms) or outsourced to CAPTCHA farms. Therefore, we argue that all current uses of CAPTCHAs are actually intended to slow down attackers or increase their costs. In the list above, scenarios 2 and 5 directly call for rate-limiting, while scenarios 1, 3, and 4 can be made less profitable for attackers if sufficiently rate-limited. Therefore, CACTI can be used in all these scenarios.

In addition to CAPTCHAs, modern websites use a variety of abuse-prevention systems (e.g., filtering based on client IP address or cookies). We envision CACTI being used alongside such mechanisms. Websites could dynamically adjust their CACTI rate-proof thresholds based on information from these other mechanisms. We are aware that rate-proofs are a versatile primitive that could be used to fight abusive activity in other ways, or even enable new use-cases. However, in this paper, we focus on the important problem of reducing the user burden of CAPTCHAs.

4.2 Design Challenges

In order to realize the conceptual design outlined above, we identify the following key challenges:

TEE attestation. In current TEEs, the process of remote attestation is not standardized. For example, in SGX, a verifier must first register with Intel Attestation Service (IAS) before it can verify TEE quotes. Other types of TEEs would have different processes. It is unrealistic to expect every web server to establish relationships with such services from all manufacturers in order to verify attestation results. Therefore, web servers cannot directly verify the attestation, but still need to ascertain that the client is running a genuine TEE.

TEE memory limitations. TEEs typically have a small amount of secure memory. For example, if the memory of an SGX enclave exceeds the size of the EPC (usually 128 MB), the CPU has to swap pages out of the EPC. This is a very expensive operation, since these pages must be encrypted and integrity protected. Therefore, CACTI should minimize the required amount of enclave memory, since other enclaves may be running on the same platform.

Limited number of monotonic counters. TEEs typically have a limited number of hardware monotonic counters, e.g., SGX allows at most 256 per enclave. Also, the number of counter increments can be limited, e.g., in SGX the limit is 100 in a single epoch [8] – a platform power cycle, or a 24 hour period. This is a challenge because hardware monotonic counters are critical for achieving rollback-protected storage. Recall that CACTI requires rollback-protected storage for all timestamps, to prevent malicious clients from rolling-back the timestamp lists and falsifying rate-proofs. Furthermore, this storage must be updated every time a new timestamp is added, i.e., for each successful rate-proof.

TEE entry/exit overhead. Invoking TEE functionality typically incurs some overhead. For example, whenever an execution thread enters/exits an SGX enclave, the CPU has to perform various checks and procedures (e.g., clearing registers) to ensure that enclave data does not leak. Identifying and minimizing the number of TEE entries/exits, whilst maintaining functionality, can be challenging.

4.3 Realizing CACTI Design

We now present a detailed design that addresses aforementioned design challenges. We describe its implementation in Section 5.

4.3.1 Communication protocol

The web server must be able to determine that a supplied rate-proof was produced by a genuine TEE. Typically, this would be done using remote attestation, where the TEE proves that it is running CACTI code. If the TEE provides privacy-preserving attestation (e.g., the EPID protocol used in SGX remote attestation), this would also fulfill our requirement for client privacy, since websites would not be able to link rate-proofs to specific TEEs.

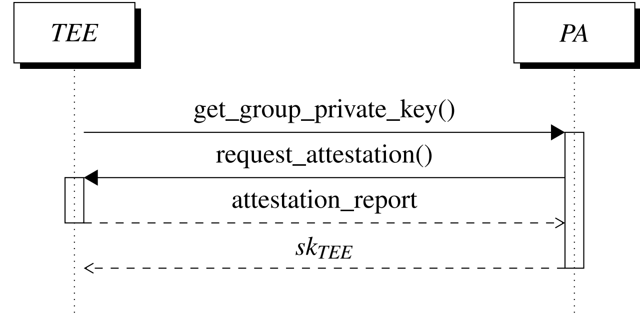


Figure 2: CACTI provisioning protocol. The interaction between the Provisioning Authority (*PA*) and the client's TEE takes place over a secure connection, using the client to pass the encrypted messages. After verifying the attestation report (and any other required information), the *PA* provisions the TEE with a group private key (sk_{TEE}).

However, as described above, current TEE remote attestation is not designed to be verified by anonymous third parties. Furthermore, as CACTI is not limited to any particular TEE type, websites would need to understand attestation results from multiple TEE vendors, potentially using different protocols. Finally, some types of TEEs might not support privacy-preserving remote attestation, which would undermine our requirement for client privacy.

To overcome this challenge, we introduce a separate *Provisioning Authority* (*PA*) in order to unify various processes for attesting CACTI TEEs. Fundamentally, the *PA* is responsible for verifying TEE attestation (possibly via the TEE vendor) and establishing a privacy-preserving mechanism through which websites can also establish trust in the TEE. Specifically, the *PA* protects user privacy by using the EPID group signature scheme. The *PA* plays the role of the EPID *issuer*, and – optionally – the *revocation manager* [30]. During the provisioning phase (as shown in Figure 2), the *PA* verifies the attestation from the client's TEE and then runs the EPID `join` protocol with the client's TEE in order to provision the TEE with a group private key sk_{TEE}. The *PA* certifies and publishes the group public key pk_G. The *PA* may optionally require the client to prove their identity (e.g., by signing into an account) – this is a business decision and different *PA*s may take different approaches. After provisioning, the *PA* is unable to link signatures to any specific client thanks to the properties of the underling BBS+ signature scheme and signature-based revocation used in EPID [30]. We analyze security implications of malicious *PA*s in Section 6.1, and discuss the use of other group signature schemes in Section 7.2. There can be multiple *PA*s and websites can decide which *PA*s to trust. If a TEE is provisioned by an unsupported *PA*, the website would fall back to using CAPTCHAs.

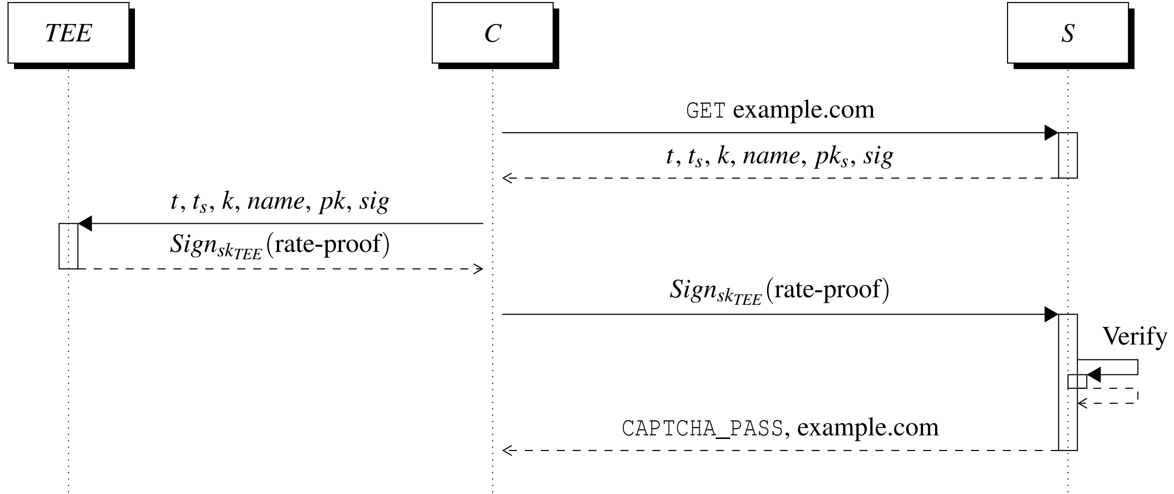


Figure 3: CACTI CAPTCHA-avoidance protocol. The client (C) requests a resource from the web server (S). In response, the server provides a timestamp for the current event (t), a threshold consisting of a starting time (t_s) and a count (k), and the *name* of the list. Optionally, the server also provides a signature (sig) over the request and the public key (pk_s) with which the signature can be verified. The client passes this information to its TEE in order to produce a rate-proof, signed by a group private key (sk_{TEE}), which can be verified by the server.

Once the TEE has been provisioned, the client can begin to use CACTI when visiting supported websites, as shown in Figure 3. Specifically, when serving a page, the server includes the following information: a timestamp t, a threshold Th (including start time t_s and count k), the name of the list (or CACTI-GLOBAL for the global list), and (optionally) a public key and signature for rates that enforce a same-origin policy. The client uses this information to request a rate-proof from their TEE. If the client's rate is indeed below the threshold, the TEE produces the rate-proof, signed with its group private key. The client then sends this to the server in lieu of solving a CAPTCHA.

4.3.2 TEE Design

To realize the conceptual design above, the client's TEE would ideally store all timestamps indefinitely in integrity-protected and rollback-protected memory. However, as discussed above, current TEEs fall short of this idealized representation, since they have limited integrity-protected memory and a limited number of hardware counters for rollback protection. To overcome this challenge, we store all data outside the TEE, e.g., in a standard database. To prevent dishonest clients from modifying this data, we use a combination of hash chains and Merkle Hash Trees (MHTs) to achieve integrity and rollback-protection.

Hash chains of timestamps. To protect integrity of stored timestamps, we compute a hash chain over each list of timestamps, as shown in Figure 4. Thus the TEE only needs to provide integrity and rollback-protected storage for the most recent hash in each hash chain. For efficiency, we store intermediate value of the hash chain along with each timestamp outside the TEE.

MHT of lists. Although it would be possible for the TEE to seal the most recent hash of each list individually, the lists may be updated independently, so the TEE would need separate hardware monotonic counters to provide rollback protection for each list. In a real-world deployment, the number of lists is likely to exceed the number of available hardware counters, e.g., 256 counters per enclave in SGX. To overcome this challenge, we combine the lists into a Merkle Hash Tree (MHT). As shown in Figure 5, each leaf of the MHT is a hash of the list information (list name and public key) and the most recent hash in the list's hash chain. With this arrangement, the TEE only needs to provide integrity and rollback-protected storage for the MHT root R, which can be achieved using sealing and a single hardware monotonic counter.

4.3.3 Producing a Rate-Proof

The TEE first needs to verify the integrity of its externally-stored data structures (i.e., hash chains and MHT described above), and if successful, update these with the new timestamp and produce the rate-proof, as follows:

1. TEE inputs. The client supplies its TEE with the list information and all timestamps in the list that are greater than or equal to the server-defined start time t_s. The client also supplies the largest timestamp that is smaller than t_s, which we denote $t_{s-\delta}$, and the intermediate value of the hash chain up to, but not including, $t_{s-\delta}$. The client supplies the sealed

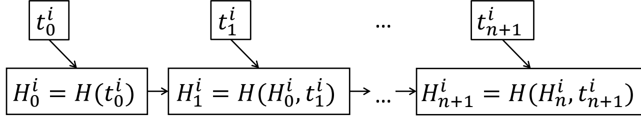


Figure 4: Hash chain of timestamps t_j^i for list i. $H()$ is a cryptographic hash function.

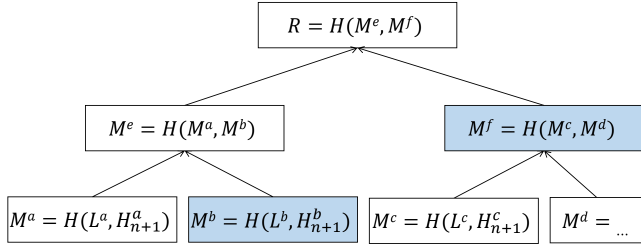


Figure 5: Merkle Hash Tree over lists $a...d$. Each leaf is a hash of the list information L^i (list name and public key) and the most recent hash of the list's hash chain H_{n+1}^i. $H()$ is a cryptographic hash function, R is the root of the MHT, and the nodes in blue illustrate the inclusion proof path for list b.

MHT root and intermediate hashes required to verify that the list is in the MHT.

2. Hash chain checks. The TEE first checks that $t_{s-\delta}$ is smaller than t_s and then recomputes the hash chain over included timestamps in order to reach the most recent value. During this process, it counts the number of included timestamps and checks that this is less than the value k specified in the threshold. The inclusion of one timestamp outside the requested range ($t_{s-\delta}$) ensures that the TEE has seen all timestamps within the range. This process requires $O(n)$ hashes, where n is the number of timestamps in the requested range.

3. MHT checks. The TEE then unseals the MHT root and uses the hardware counter to verify that it is the latest version. The TEE then checks that the list information and the calculated most recent hash value is indeed a leaf in the MHT. This process requires $O(log(s))$ hashes, where s is the number of lists. Including the list name in the MHT leaf ensures that the timestamps have not been substituted from another list. If the list has an associated public key, the TEE uses this to verify the signature on the server's request.

4. Starting a new list. If the rate-proof is requested over a new list (e.g., when the user firsts visits a website), the TEE must also verify that the list name does not appear in any MHT leaves. In this case, the client supplies the TEE with all list names and their most recent hash values. The TEE reconstructs the full MHT and checks that the new list name does not appear. This requires $O(s)$ string comparisons and hashes for s lists.

5. Updating a list. If the above verification steps are successful, the TEE checks that the new timestamp t supplied by the server exceeds the latest timestamp in the specified list. If so, the TEE adds t to the list and updates the MHT to obtain a new MHT root. The new root is sealed alongside the TEE's group private key. The TEE then produces a signed rate-proof, using its group private key. The rate-proof includes a hash of the original request provided by the server, thus confirming that the TEE checked the rate and added the server-supplied timestamp. The TEE returns the rate-proof to the client, along with the new sealed MHT root for the client to store. In the above design, the whole process of producing the rate-proof can be performed in a single call to the TEE, thus minimizing the overhead of entering/exiting the TEE.

4.3.4 Reducing Client-Side Storage

The number of timestamps stored by CACTI grows as the client visits more websites. However, in most use-cases, it is unlikely that the server will request rate-proofs going back beyond a certain point in time t_P.

To reduce client-side storage requirements, we provide a mechanism to *prune* a client's timestamp list by merging all timestamps prior to t_P. Specifically, the server can include t_P in any rate-proof request, and upon receiving this, the client's TEE counts and records how many timestamps are older than t_P. The old timestamps and associated intermediate hash values can then be deleted from the database. In other words, the system merges all timestamps prior to t_P into a single count value c_P. The TEE stores t_P and the count value in the database outside the TEE and protects their integrity by including both values in the list information that forms the MHT leaf. Pruning can be done repeatedly: when a new pruning request is received for $t_{P'} > t_P$, CACTI fetches and verifies all timestamps up to $t_{P'}$ and adds these to c_P to create $c_{P'}$. It then replaces t_P and c_P with $t_{P'}$ and $c_{P'}$ respectively.

This pruning mechanism does not reduce security of CACTI. If the server does request a rate-proof going back beyond t_P, CACTI will include the full count of timestamps stored alongside t_P. This is always greater than or equal to the actual number of timestamps; thus, there is no incentive for the server to abuse the pruning mechanism. Similarly, even if a malicious client could trigger this pruning (i.e., assuming the list is not associated to the server's public key), there is no incentive to do so because it would never decrease the number of timestamps included in rate-proofs.

Since the global list CACTI-`GLOBAL` is used by all websites, the client is always allowed to prune this list to reduce storage requirements. CACTI blocks servers from pruning CACTI-`GLOBAL` since this can be used as an attack vector to inflate the client rate by compressing all rates into one value – thus preventing use of CACTI on websites that utilize CACTI-`GLOBAL`. Thus, we expect pruning of CACTI-`GLOBAL` to be done automatically by the CACTI host application or browser extension.

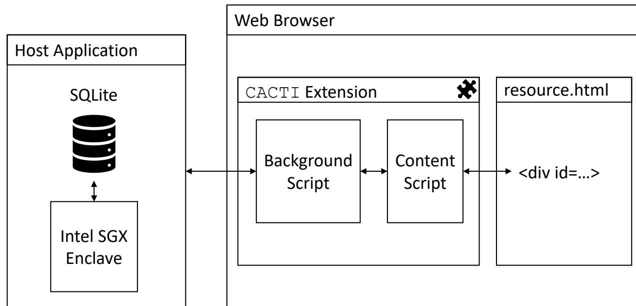


Figure 6: Overview of CACTI client-side components.

5 Implementation

We now describe the implementation of the CACTI design presented in the previous section. We focus on proof-of-concept implementations of: client-side browser extension, native host application, and CACTI TEE, as shown in Figure 6. Finally, we discuss how CACTI is integrated into websites.

5.1 Browser Extension

The browser extension serves as a bridge between the web server and our host application. We implemented a proof-of-concept browser extension for the Chrome browser (build 79.0.3945.130) [6]. Chrome extensions consist of two parts: a content script and a background script.

- **Content script:** scans the visited web page for an HTML `div` element with the id CACTI-`div`. If the page contains this, the content script parses the parameters it contains and sends them to the background script.
- **Background script:** we use Chrome Native Messaging to launch the host application binary when the browser is started and maintain an open port [20] to the host application until the browser is closed. The background script facilitates communication between the content script and the host application.

User notification. The browser extension is also responsible for notifying the user about requests to access CACTI. Notifications can include information, such as server's domain name, timestamp to be inserted, and threshold used to generate the rate-proof. By default, the background script notifies the user whenever a server requests to use CACTI, and waits for user confirmation before proceeding. This prevents malicious websites from abusing CACTI by adding multiple timestamps without user permission (for possible attacks, see Section 6.1). However, asking for user confirmation for every request could cause UI fatigue. Therefore, CACTI could allow the user to choose from the following options: (1) *Always ask* (the default), (2) *Ask only upon first visit to site*, (3) *Only ask for untrusted sites*, (4) *Only ask for more than x requests per site per time period*, and (5) *Never ask*. Advanced users can also modify our extension or code their own extension to enforce arbitrary policies for requesting user confirmation. The notification is displayed using Chrome's Notification API [3].

5.2 Host Application

The host application running on the client is responsible for: (1) creating the CACTI TEE, which we implement as an SGX enclave, and exposing its `ECALL` API to the browser extension; (2) storing (and forwarding) timestamps and additional integrity information for secure calculation of rate-proofs (to the enclave); and (3) returning the enclave's output to the browser extension.

The host application is implemented in C and uses Chrome Native Messaging [15] to communicate with the browser extension. Since Chrome Native Messaging only supports communication with JSON objects, the host application uses a JSON parser to extract parameters to the API calls. We used the JSMN JSON parser [12]. Moreover, the host application implements the Chrome Native Messaging protocol [2] and communicates with the browser extension using Standard I/O (`stdio`), since this is currently the only means to communicate between browser extensions and native applications.

The host application stores information in an SQLite database. This database has two tables: `LISTS` stores the list names and associated public keys, and `TIMESTAMPS` stores all timestamps and intermediate values of the hash chains. For each rate-proof request, the host application queries the database and provides the data to the enclave.

Since the timestamps are stored unencrypted, we use existing features of the SQLite database to retrieve only the necessary range of timestamps for a given list. Note that since data integrity is maintained through other mechanisms (i.e., hash chains and MHT), the mechanism used by the host application to store this data does not affect the security of the system. Alternative implementations could use different database types and/or other data storage approaches. Instead of hash chains and MHTs, it is possible to use a database managed by the enclave, e.g., EnclaveDB [54]. However, this would increase the amount of code running inside the enclave, thus bloating the trusted code base (TCB).

5.3 SGX Enclave

We implemented the TEE as an SGX enclave using the OpenEnclave SDK [16] v0.7.0. OpenEnclave was selected since it aims to unify the programming model across different types of TEEs. The process of requesting a rate-proof is implemented as a single `get_rate` `ECALL`. For timestamps, we use the UNIX time which denotes the number of seconds elapsed since the UNIX Epoch (midnight 1/1/1970) and is represented as a 4-byte signed integer. We use cryptographic functions from the `mbed` TLS library [13] included in OpenEnclave.

Specifically, we use SHA-256 for all hashes and ECDSA for all digital signatures. For EPID signatures, we use Intel EPID SDK (v7.0.1) [5] with the performance-optimized version of Intel Integrated Performance Primitives (IPP) Cryptography library [9]. We use a formally-verified and platform-optimized MHT implementation from EverCrypt [55]. As an optimization, if the MHT is sufficiently small, we can cache fully inside the enclave. When a request for a rate-proof is received, the enclave recalculates the timestamp hash chain and then directly compares the most recent value to the corresponding leaf in the cached MHT, as described in Section 4.3.3.

OpenEnclave currently does not support SGX hardware monotonic counters, so we could not include these in the proof-of-concept implementation. However, a production implementation can easily include hardware counter functionality. Although our implementation uses SGX, CACTI can be realized on any suitable TEE. For example, OpenEnclave is currently being updated to support ARM TrustZone. When this version is released, we plan to port the current implementation to TrustZone, with minimal expected modifications.

5.4 Website Integration

Integrating CACTI into a website involves two aspects: sending the rate-proof request to the client, and verifying the response. The server generates the rate-proof request (see Section 4.3.1) and encodes it as `data-*` attributes in the `CACTI-div` HTML `div`. The server also includes the URL to which the generated rate-proofs should be sent. The browser extension determines whether the website supports CACTI by looking for the `CACTI-div` element. The server implements an HTTP endpoint for receiving and verifying rate-proofs . If the verification succeeds, this endpoint notifies the website and the user is granted access.

Integrating CACTI into a website is thus very similar to using existing CAPTCHA systems. For example, reCAPTCHA adds the `g-recaptcha` HTML `div` to the page, and implements various endpoints for receiving and verifying the responses [19]. We evaluate server-side overhead of CACTI, in terms of both processing and data transfer requirements, in Section 6.

6 Evaluation

We now present and discuss the evaluation of CACTI. We start with a security analysis, based on the threat model and requirements defined in Section 3. Next, we evaluate performance of CACTI in terms of latency and bandwidth. Finally, we discuss CACTI deployability issues.

6.1 Security Evaluation

Data integrity & rollback attacks. Since timestamps are stored outside the enclave, a malicious host application can try to modify this data, or roll it back to an earlier version. If successful, this might trick the enclave into producing falsified rate-proofs. However, if any timestamp is modified outside the enclave, this would be detected because the most recent value of the hash chain would not match the corresponding MHT leaf. Assuming a suitable collision-resistant cryptographic hash function, it is infeasible for the malicious host to find alternative hash values matching the MHT root. Similarly, a rollback attack against the MHT is detected by comparing the included counter with the hardware monotonic counter.

Timestamp omission attacks. A malicious application can try to provide the enclave with only a subset of the timestamps for a given request, e.g., to pretend to be below the threshold rate. Specifically, the host could try to omit one or more timestamps at the start, in the middle, and/or at the end, of the range. If timestamps are omitted at the start, the enclave detects this when it checks that the first timestamp supplied by the host is *prior to* the start time of request t_s. If timestamps are omitted in the middle (or at the end) of the range, the most recent hash value will not match the value in the MHT leaf.

List substitution attacks. A malicious client might attempt to use a timestamp hash chain from a different list, or claim that the requested list does not exist. The former is prevented by including list information (list name and public key) in the MHT leaf. If there is a mismatch between the name and the timestamp chain, the resulting leaf would not exist in the MHT. For the latter, when the host calls the enclave's `get_rate` function for a new list, the enclave checks the names of all lists in the MHT to ensure that the new list name does not already exist.

TEE reset attacks. A malicious client might attempt to delete all stored data, including the sealed MHT root, in order to reset the TEE. Since the group private key received from the provisioning authority is sealed together with the MHT root, it is impossible to delete one and not the other. Deleting the group private key would force the TEE to be re-provisioned by the provisioning authority, which may apply its own rate-limiting policies on how often a given client can be re-provisioned.

CACTI **Farms.** Similar to CAPTCHA farms, a multitude of devices with TEE capabilities could be employed to satisfy rate thresholds set by servers. However, this would be infeasible because: *(1)* CACTI enclaves would stop producing rate-proofs after reaching server thresholds and would thus require a TEE reset and CACTI re-provisioning – which is a natural rate limit; *(2)* the cost of purchasing a device would be significantly higher than CAPTCHA solving costs. For example, currently the cheapest service charges \$1.8 for solving 1,000 reCAPTCHAs [1][3], while a low-end bare-bones CPU with SGX support alone costs ≈ \$70 [11], in addition to the maintenance and running costs.

[3] See a comparison of CAPTCHA solving services [22]

CACTI **Botnets.** An adversary might try to build a CACTI botnet consisting of compromised devices with suitable TEEs in order to bypass CAPTCHAs at scale, similarly to a CACTI farm. However, if the compromised devices are not yet running CACTI, the adversary would have to provision them using a suitable *PA*, which could be made arbitrarily costly and time-consuming. Alternatively, if the compromised devices are already running CACTI, the adversary gains little advantage because the legitimate users will likely have been using CACTI to create their own rate-proofs. Furthermore, the legitimate user would probably notice any overuse/abuse of their system due to quickly exceeding the thresholds.

Client-side malware. A more subtle variant of the reset attack can occur if malware on the client's own system corrupts or deletes TEE data. This is a type of denial-of-service (DoS) attack against the client. However, defending against such DoS attacks is beyond the scope of this work, since this type of malware would have many other avenues for causing DoS, e.g., deleting critical files.

Other DoS attacks. A malicious server might try to mount a DoS attack against an unsuspecting client by inserting a timestamp for a future time. If successful, the client would be unable to insert new timestamps and create rate-proofs for any other servers, since the enclave would reject these timestamps as being in the past. This attack can be mitigated if the client's browser extension and/or host application simply check that the server-provided timestamp is not in the future.

Client tracking. A malicious server (or group of servers) might attempt to track clients by sending multiple requests for rate-proofs with different thresholds in order to learn the precise number of timestamps stored by the client. A successful attack of this type could potentially reduce the client's anonymity set to only those clients with the same rate. However, this attack is easy to detect by monitoring the thresholds sent by the server. A more complicated attack targeting a specific client is to send an excessive number of successful rate-proof requests in order to increase the client's rate. The goal is to reduce the size of the target's anonymity set. This attack is also easy to detect or prevent by simply rate-limiting the number of increments accepted from a particular server. Note that the window of opportunity for this targeted attack is limited to a single session, because malicious servers cannot reliably re-identify the user across multiple sessions (since this is what the attack is trying to achieve). The above attacks cannot be improved even if multiple servers collude.

Rogue *PA*s. A malicious *PA* might try to compromise or diminish client privacy. However, this is prevented by CACTI's use of the EPID protocol [30]. Specifically, due to the BBS+ signature scheme [27] during EPID key issuance, clients' private keys are never revealed to *PA*s. Also, EPID's signature-based revocation mechanism does not require member private keys to be revealed. Instead, signers generate zero-knowledge proofs showing that they are not on the revocation list. Therefore, client privacy does not depend on any *PA* business practices, e.g., log deletion or identifier blinding.

Each website has full discretion to decide which *PA*s it trusts; if a server does not trust the PA who issued the member private key to the TEE, it can simply fall back to CAPTCHAs. This provides no advantage to attackers, and websites can be as conservative as they like. If higher levels of assurance are required, *PA*s can execute within TEEs and provide attestation of correct behavior; we defer the implementation of this optional feature to future work.

Overall, we claim that CACTI meets all security requirements defined in Section 3 and significantly increases the adversary's cost to perform DoS attacks. Specifically, the **Unforgeability** requirement is satisfied since it is impossible for the host to perform rollback, timestamp exclusion and list substitution attacks. **Client privacy** is achieved because the rate-proof does not reveal the actual number of timestamps included, and is signed using a group signature scheme.

6.2 Latency Evaluation

We conducted all latency experiments on an Intel NUC Kit NUC7PJYH [10] with an Intel Pentium Silver J5005 Processor (4M Cache, up to 2.80 GHz); 4 GB DDR4-2400 1.2V SO-DIMM Memory; running Ubuntu 16.04 with the Linux 4.15.0-76-generic kernel Intel SGX DCAP Linux 1.4 drivers.

Recall that the host application is responsible for initializing the enclave, fetching data necessary for enclave functionality, performing `ECALL`s, and finally updating states according to enclave output. Therefore, we consider the latency in the following four key phases in the host application:

- *Init-Enclave*: Host retrieves the appropriate data from the database and calls `init_mt` `ECALL` that initializes the MHT within the enclave.[4]
- *Pre-Enclave*: Host retrieves the required hashes and timestamps from the database.
- *In-Enclave*: Host calls the `get_rate` `ECALL`. This phase concludes when the `ECALL` returns.
- *Post-Enclave*: Host updates/inserts the data it received from the enclave into the database.

We investigated the latency impact by varying (1) the number of timestamps in the rate-proof (Section 6.2.1), and (2) the number of lists in the database (Section 6.2.2). We evaluated the end-to-end latency in Section 6.2.4. Unless otherwise specified, each measurement is the average of 10 runs.

Note: The ECDSA and EPID signature operations are, by far, the dominant contributors to latency. However, they represent a fixed latency overhead that does not vary with the number of timestamps or servers. Therefore, for clarity's sake, figures in the following sections do not include these operations. We analyze them separately in Section 6.2.3.

[4] Init-Enclave is done only when the enclave starts.

6.2.1 Varying Number of Timestamps in Query

We measured the effect of varying the number of timestamps included in the query, while holding the number of lists constant. As shown in Figure 7, query latency increases linearly with the number of timestamps included in the query. The most notable increase is in the in-enclave phase, since this involves calculating a longer hash chain. However, even with 10,000 timestamps in a query, the total latency only reaches ~40 milliseconds (excluding signature operations).

6.2.2 Varying Number of Lists

Next, we varied the number of lists while holding the number of timestamps fixed at one per list. We considered two separate scenarios: adding a new list and updating an existing list.

Adding a new list. As shown in Figure 8, the latency for the pre-enclave phase is lower compared to Figure 7. This is because we optimize the host to skip the expensive `TIMESTAMPS` table look up operation if the host knows that this is a new list. The in-enclave phase increases as the number of lists increases due to the string comparison operations performed by the enclave to prevent list substitution attacks. However, this phase can be optimized by sorting the server names inside the enclave during initial MHT construction. The post-enclave latency is due to the cost of adding entries to the `TIMESTAMPS` table. Figure 8 assumes the enclave has already been initialized (see Figure 9 for the corresponding init-enclave phase).

Updating an existing list. As shown in Figure 9, the latency of the init-enclave phase increases as the number of lists increases. This is expected, since the enclave reconstructs the MHT in this phase. The pre-enclave phase also increases slightly due to the database operations.

6.2.3 Signature Operation Latency

Evaluation results presented thus far have not included the ECDSA signature verification or EPID signature creation operations. Specifically, the server creates an ECDSA signature on the request, which the enclave verifies. The enclave creates an EPID group signature on the response, which the server verifies using the EPID group public key. The average latencies over 10 measurements for these four signature operations are shown in Figure 10. We can see that the EPID group signature generation operation is an order magnitude slower compared to the other cryptographic operations including EPID group signature verification. The latency of our enclave is thus dominated by the EPID signature generation operation.

6.2.4 End-to-End Latency

Table 1 shows the end-to-end latency (excluding network communication) from when the server begins generating a request until it has received and verified the response from the client. In both settings, the end-to-end latency is below 250 milliseconds. The latency will be lower if there are fewer lists or included timestamps. Compared to other types of CAPTCHAs, image-based CAPTCHAs take ~10 seconds to solve [31] and behavior-based reCAPTCHA takes ~400 milliseconds, although this might change depending on the client's network latency.

6.3 Bandwidth Evaluation

We measured the amount of additional data transferred over the network by different types of CAPTCHA techniques. Minimizing data transfer is critical for both servers and clients. We compared CACTI against image-based and behavior-based reCAPTCHA [18] (see Figure 1). The former asks clients (one or more times) to find and mark certain objects in a given image or images, whilst the latter requires clients to click a button. To isolate the data used by reCAPTCHA, we hosted a webpage with the minimal auto-rendering reCAPTCHA example [19]. We visited this webpage and recorded the traffic using the Chrome browser's debugging console.

Table 2 shows the additional data received and sent by the client to support each type of CAPTCHA. Image-based reCAPTCHA incurs the highest bandwidth overhead since it has to download images, often multiple times. Although not evaluated here, text-based CAPTCHAs also use images and would thus have a similar bandwidth overhead. Behavior-based reCAPTCHA downloads several client-side scripts. Both types of reCAPTCHA made several additional connections to Google servers. Overall, CACTI achieves at least a 97% reduction in client bandwidth overhead compared to contemporary reCAPTCHA solutions.

6.4 Server Load Evaluation

We analyzed the additional load imposed on the server by CACTI. Unfortunately, CAPTCHAs offered as services, such as reCAPTCHA [18] and hCAPTCHA [7], do not disclose their source code and we have no reliable way of estimating their server-side overhead. Therefore, we compared CACTI against two open-source CAPTCHA projects published on GitHub (both have more than 1,000 stars and been forked more than a hundred times):

dchest/captcha [17] (Figure 11a) generates image-based text recognition CAPTCHAs consisting of transformed digits with noise in the form of parabolic lines and additional clusters of points. It can also generate audio CAPTCHAs, which are pronunciations of digits with randomized speed and pitch and randomly-generated background noise.

produck/svg-captcha [21] (Figure 11b) generates similar image-based text recognition CAPTCHAs, as well as challenge-based CAPTCHAs consisting of simple algebraic operations on random integers. Noise is introduced by varying the text color and adding parabolic lines.

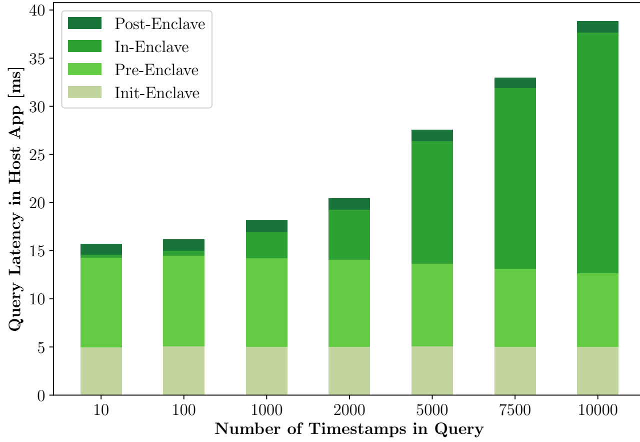


Figure 7: Latency of initializing the enclave and creating a rate-proof for different numbers of timestamps in the query (excluding signature operations).

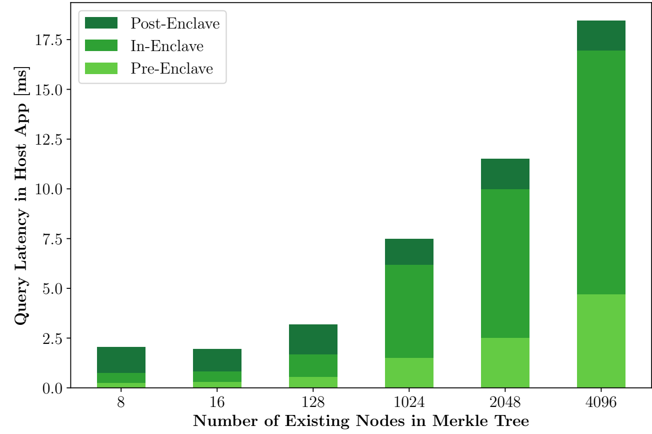


Figure 8: Latency of creating the first rate-proof in a new list for different numbers of existing lists (excluding enclave initialization and signature operations).

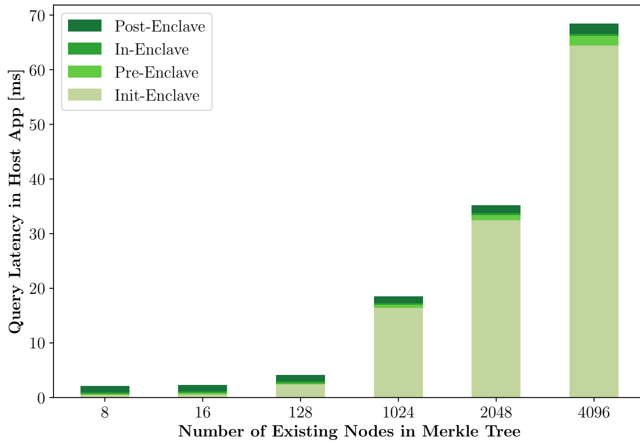


Figure 9: Latency of initializing the enclave and updating an existing list for different numbers of existing lists (excluding signature operations).

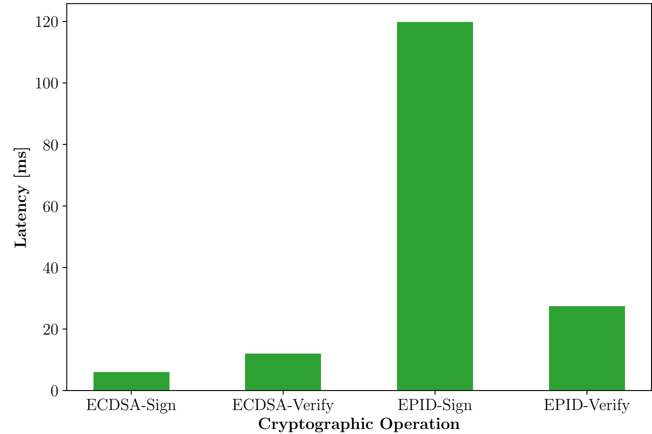


Figure 10: Microbenchmarks of signature operations. ECDSA signatures were created and verified using the `mbed` TLS library [13] and EPID signatures with the Intel EPID SDK [5].

(a) dchest/captcha image-based CAPTCHA [17].

(b) produck/svg-captcha image-based CAPTCHAs [21].

Figure 11: CAPTCHAs generated using open-source libraries.

Table 3 shows the time to generate different types of CAPTCHAs using the above libraries with typical configuration parameters (e.g., eight characters for text CAPTCHAs). Since CAPTCHA verification with these libraries is a simple string comparison, we assume this is negligible. CACTI's server-side processing is due almost entirely to the EPID signature verification operation. We expect that this time could be improved by using more optimized implementations of this cryptographic operation. Additionally, CACTI uses significantly less communication bandwidth than other approaches, which also reduces the server load (which is not captured in this measurement). Most importantly, the biggest gain of CACTI is on the user side; saving more than ~10 seconds per CAPTCHA for users.

Table 1: End-to-End Latency of CACTI for different numbers of timestamps and lists. The *Browser* column represents the latency of the browser extension marshalling data to and from the host application. The other columns are as described above.

	ECDSA-Sign	Browser	Pre-Enclave	In-Enclave	Post-Enclave	EPID-Verify	Total
10,000 timestamps in 1 list	6.3 ms	15.2 ms	7.7 ms	181.7 ms	1.0 ms	27.3 ms	239.2 ms
4,096 lists with 1 timestamp each	6.3 ms	15.2 ms	1.8 ms	157.4 ms	2.0 ms	27.3 ms	210.0 ms

Table 2: Additional data received and sent by the client for image-based and behavior-based reCAPTCHA, compared with CACTI.

	Received	Sent	Total
Image-based	140.05 kB	28.97 kB	169.02 kB
Behavior-based	54.38 kB	26.12 kB	80.50 kB
CACTI	0.82 kB	1.10 kB	1.92 kB

Table 3: Server-side processing time for generating a CAPTCHA and verifying the response.

Library	Type	Time
dchest/captcha	Audio	13.3 ms
	Image-based text	1.7 ms
produck/svg-captcha	Image-based text	2.2 ms
	Image-based math	1.4 ms
CACTI	Rate-proof	33.6 ms

6.5 Deployability Analysis

We analyze deployability of CACTI by considering changes required from both the server's and client's perspectives:

Server's perspective. The server will have to make the following changes: (1) create and maintain a new public/private key pair and obtain a certificate for the public key, (2) add an additional `div` to pages for which they wish to enable CACTI, (3) create and sign requests using the private key, and (4) add an HTTP endpoint to receive and verify EPID signatures. The server-side deployment could be further simplified by providing the request generation and signature operations as an integrated library.

Client's perspective. The client will have to make the following changes: (1) download and install the CACTI native software, and (2) download and install the browser extension. Although CACTI requires the client to have a suitable TEE, this is a realistic assumption given the large and increasing deployed base of devices with e.g., ARM TrustZone or Intel SGX TEEs.

7 Discussion

7.1 *PA* Considerations

As discussed in Section 4.3, CACTI's use of a provisioning authority (*PA*) provides the basis for client privacy. CACTI does not prescribe the *PA*'s policies. For example, the *PA* has the choice of running the provisioning protocol (Figure 2) as a one-off operation (e.g., when installing CACTI) or on a regular basis, depending on its risk appetite. If there are attacks or exploits threatening the Intel SGX ecosystem (and consequently the security of group private keys), the *PA* can revoke all group member keys. This would force all enclaves in the group to re-register with the *PA*. A similar scenario applies if key-rotation is implemented on the *PA*, e.g., the master secret held by the *PA* is rotated periodically. This forces all enclaves to regularly contact the *PA* to obtain new group member keys. Frequent key-rotation introduces a heavier burden on the clients (although this can be automated), but provides better security.

7.2 EPID

Even though CACTI uses EPID group signatures to protect client privacy, CACTI is agnostic to the choice of the underlying signature scheme as long as it provides signer unlinkability and anonymity. We also considered other schemes, such as Direct Anonymous Attestation (DAA) [28], as used in the Trusted Platform Module (TPM). However, DAA is susceptible to various attacks [29, 45, 56] and, due to its design targeting low-end devices, suffers from performance problems. In contrast, EPID is used in current Intel SGX remote attestation and is thus a good fit for enclaves. Moreover, as mentioned in the previous section, the *PA* must revoke group member keys in the event of a compromise. EPID offers privacy-preserving *signature-based revocation*, wherein the issuer can revoke any key using only a signature generated by that key. Signature verifiers use signature revocation lists published by issuers to check whether the group member keys are revoked. Using this mechanism, CACTI provides *PA*s with revocation capabilities without allowing them to link keys to individual users. *PA*s can define their own revocation policies to maximize their reputation and trustworthiness.

7.3 Optimizations

7.3.1 Database Optimizations

As with most modern database management systems, SQLite supports creating indexes in database tables to reduce query times. Also, as discussed in Section 6, placing all timestamps for all servers in one table and conducting `JOIN` operations incurs performance overhead. An alternative is to use a separate table per list. However, we presented CACTI evaluation results without creating any indexes or separate timestamp tables in order to show the worst-case performance. Performance optimizations, such as changing the database layout, can be easily made by third parties, since they do not affect the security of CACTI.

7.3.2 System-level Optimizations

As a system-level optimization, CACTI can perform some processing steps in the background while waiting for the user to confirm the action. For example, while the browser extension is displaying the notification and waiting for user approval, the request can already be sent to the enclave to begin processing (e.g., loading and verifying the hash chain of timestamps and the MHT). The enclave creates the signed rate-proof but does not release it or update the hash chain until the user approves the action. This optimization reduces user-perceived latency to that of client-side post-enclave and server-side EPID verification processes, which is less than 14% of the end-to-end latency reported in Section 6.2.4.

7.3.3 Optimizing Pruning

Although it is possible to create another `ECALL` for pruning, this might incur additional enclave entry/exit overhead (see Section 4.2). Instead, pruning can be implemented within the `get_rate` `ECALL`. Since `get_rate` already updates the hash chain and MHT, the pruning can be performed at the same time, thus eliminating the need for an additional `ECALL` and hash chain and MHT update.

7.4 Deploying CACTI

7.4.1 Integration with CDNs and 3rd Party Providers

Although CACTI aims to reduce developer effort by choosing well-known primitives (e.g., SQLite and EPID), we do not expect all server operators to be experienced in implementing CACTI components. The server-side components of CACTI can be provided by Content Delivery Networks (CDNs) or other independent providers.

CDNs are widely used to reduce latency by serving web content to clients on behalf of the server operator. CDNs have already recognized the opportunity to provide abuse prevention services to their customers. For example, Cloudflare offers CAPTCHAs as a free rate-limiting service [4] to its customers [14]. CACTI could easily be adapted for use by CDNs, which would bring usability benefits across all websites served by the CDN.

In addition, independent CACTI providers could offer rate-proof services that are easy to integrate into websites – similar to how CAPTCHAs are currently offered by reCAPTCHA [18] or hCAPTCHA [7]. These services would implement the endpoints described in Section 5.4 and could be integrated into websites with minimal effort.

7.4.2 Website Operator Incentives

There are several incentives for website operators to support CACTI. Firstly, in terms of usability, CACTI can drastically improve user experience by allowing legitimate users to avoid having to solve CAPTCHAs. Secondly, in terms of privacy, some concerns have been raised about existing CAPTCHA services [14]. By design, CACTI rate-proofs cannot be linked to specific users or to other rate-proofs created by the same user. Thirdly, in terms of bandwidth usage, CACTI requires an order of magnitude less data transfer than other CAPTCHA systems.

User demand for privacy-preserving solutions that reduce the amount of time spent solving CAPTCHAs has led Cloudflare to offer *Privacy Pass* [35], a system designed to reduce the number of CAPTCHAs presented to legitimate users, especially while using VPNs or anonymity networks [23].

7.4.3 *PA* Operator Incentives

In CACTI, *PA*s are only involved when provisioning credentials to CACTI enclaves (i.e., not when the client produces a rate-proof). This is a relatively lightweight workload from a computational perspective. *PA*s could be run by various different organizations with different incentives, for example:

1. TEE hardware vendors wanting to increase the desirability of their hardware;
2. Online identity providers (e.g., Google, Facebook, Microsoft) who already provide federated login services;
3. For-profit businesses that charge fees and provide e.g., a higher level of assurance;
4. Non-profit organizations, similarly to the Let's Encrypt Certificate Authority service.

CACTI users can, and are encouraged to, register with multiple *PA*s and randomly select which private key to use for generating each rate-proof. This allows new *PA*s to join the CACTI ecosystem and ensures that clients have maximum choice of *PA* without the risk of vendor lock-in.

7.4.4 Client-side components

On the client-side, CACTI could be integrated into web browsers, and would thus work “out of the box” on platforms with a suitable TEE.

8 Related Work

CACTI is situated in the intersection of multiple fields of research, including DoS (or Distributed DoS (DDoS)) protection, human presence, and CAPTCHA improvements and alternatives. In this section, we discuss related work in each of these fields and their relevance to CACTI.

Network layer defenses. The main purpose of network layer DoS/DDoS protection mechanisms is to detect malicious network flows targeting the availability of the system. This is done by using filtering [47] or rate-limiting [32] (or a combination thereof) according to certain characteristics of a flow. We refer the reader to [52] for an in-depth survey of network-level defenses. Moreover, additional countermeasures can be employed depending on the properties of the system under attack (e.g., sensor-based networks [51], peer-to-peer networks [53] and virtual ad-hoc networks [44]).

Application layer defenses. Application layer measures for DoS/DDoS protection focus on separating human-originated traffic from bot-originated traffic. To this end, problems that are hard to solve by computers and (somewhat) easy to solve by humans comprise the basis of application layer solutions. As explained in Section 1, CAPTCHAs [58] are used extensively. Although developing more efficient CAPTCHAs is an active area of research [34, 41, 57, 59], research aiming to subvert CAPTCHAs is also prevalent [39, 40, 49, 61]. In addition to such automated attacks, CAPTCHAs suffer from inconsistency when solved by humans (e.g., perfect agreement when solved by three humans are 71% and 31% for image and audio CAPTCHAs, respectively [31]). [50] suggest that although CAPTCHAs succeed at telling humans and computers apart, by using CAPTCHA-solving services (operated by humans), with an acceptable cost, CAPTCHAs can be defeated. Moreover, apart from questions regarding their efficacy, one other concern about CAPTCHAs is their usability. Studies such as [31, 38] show that CAPTCHAs are not only difficult but also time-consuming for humans, with completion time of $\approx$10 seconds on average. While behavioral CAPTCHAs are available, they suffer from privacy issues. A prevalent example, reCAPTCHA [18], works by analyzing user behavioral data (which requires sharing this data with the CAPTCHA provider) and claims to work more efficiently if used on multiple pages. In contrast, CACTI can provide at least the equivalent of abuse-prevention as CAPTCHAs, whilst minimizing the burden on users and offering strong privacy guarantees.

Human presence detection. Human presence refers to determining whether specific actions were performed by a human. VButton [46] proposes a system design based on ARM's TrustZone [25]. Secure detection of human presence is achieved by setting the display and the touch input peripherals as secure peripherals which can only be controlled by the TEE while VButton UI is displayed. With a secure I/O mechanism in place, user actions can be authenticated to originate from VButton UI by a remote server using software attestation. Similarly, Not-a-Bot [42] designs a system based on TPMs by tagging each network request with an attestation assuring that the request has been performed not long after a keyboard or mouse input by the user. Unfortunately, Intel SGX does not support secure I/O and it is not currently possible to implement similar systems on devices with only Intel SGX support. SGXIO [60] proposes an architecture for creating secure paths to I/O devices from enclaves using a trusted stack which contains a hypervisor, I/O drivers and an enclave for trusted boot. In addition, an untrusted VM hosts secure applications. The communication between secure applications and drivers are encrypted using keys generated at the end of the local attestation process. Unfortunately, the implementation of this system is not yet available. Fidelius [37] protects user secrets from a compromised browser or OS by protecting the path from the input and output peripherals to the hardware enclave. Similar to SGXIO, this is a promising step towards general-purpose trusted UI. If trusted UI capabilities do become widely available on TEEs, these can complement our CACTI design (e.g., providing stronger assurance of human presence).

Privacy Pass. Privacy Pass [35] implements a browser extension to reduce the burden of CAPTCHAs for legitimate users when visiting websites served by Cloudflare. When a user solves a CAPTCHA, Cloudflare sends the user multiple anonymous cryptographic tokens, which the user can later "spend" to access Cloudflare-operated services without encountering additional CAPTCHAs Although Privacy Pass significantly benefits benign users, it could still be exploited by CAPTCHA farms. Additionally, Privacy Pass' is currently limited to Cloudflare users.

9 Conclusion & Future Work

CACTI is a novel approach for leveraging client-side TEEs to help legitimate clients avoid solving CAPTCHAs on the Web. The unforgeable yet privacy-preserving rate-proofs generated by the TEE provide strong assurance that the client is not behaving abusively. Our proof-of-concept implementation demonstrates that rate-proofs can be generated in less than 0.25 seconds on commodity hardware, and that CACTI reduces data transfer by more than 98% compared to existing CAPTCHA schemes. As for future work, we plan to employ optimization techniques discussed in Section 7, implement and evaluate CACTI on ARM TrustZone using OpenEnclave, and explore new types of web security applications that are enabled using client-side TEEs.

Acknowledgements

We thank the anonymous reviewers for their valuable comments on prior versions of this paper. The first author was

supported in part by The Nakajima Foundation. The work of UCI was supported in part by: NSF Award #:1840197, NSF Award # 1956393, NCAE-C CCR 2020 Award #: H98230-20-1-0345, as well as UCI VCR and School of ICS Seed Funding Awards. The third author was supported by a US-UK Fulbright Cyber Security Scholar Award.

References

[1] AntiCAPTCHA. https://anti-captcha.com/mainpage, [Online] Accessed: 2020-05-22.

[2] Chrome Native Messaging Protocol. https://developer.chrome.com/extensions/nativeMessaging#native-messaging-host-protocol, [Online] Accessed: 2020-02-09.

[3] Chrome Notifications. https://developer.chrome.com/apps/notifications, [Online] Accessed: 2020-02-14.

[4] Cloudflare Rate Limiting. https://www.cloudflare.com/rate-limiting/, [Online] Accessed: 2020-05-19.

[5] EPID SDK. https://github.com/Intel-EPID-SDK/epid-sdk, [Online] Accessed: 2020-02-14.

[6] Google Chrome. https://www.google.com/chrome/, [Online] Accessed: 2020-02-11.

[7] hCaptcha. https://www.hcaptcha.com/, [Online] Accessed: 2020-05-21.

[8] Intel Dynamic Application Loader Developer Guide: Monotonic Counters. https://software.intel.com/en-us/dal-developer-guide-features-monotonic-counters, [Online] Accessed: 2020-02-05.

[9] Intel Integrated Performance Primitives Cryptography. https://github.com/intel/ipp-crypto, [Online] Accessed: 2020-05-28.

[10] Intel NUC Kit NUC7PJYH. https://ark.intel.com/content/www/us/en/ark/products/126137/intel-nuc-kit-nuc7pjyh.html, [Online] Accessed: 2020-02-11.

[11] Intel Pentium Processor G4400. https://ark.intel.com/content/www/us/en/ark/products/88179/intel-pentium-processor-g4400-3m-cache-3-30-ghz.html, [Online] Accessed: 2020-05-19.

[12] JSMN JSON Parser. https://github.com/zserge/jsmn, [Online] Accessed: 2020-02-13.

[13] Mbed TLS. https://github.com/ARMmbed/mbedtls, [Online] Accessed: 2020-02-14.

[14] Moving from reCAPTCHA to hCaptcha. https://blog.cloudflare.com/moving-from-recaptcha-to-hcaptcha/, [Online] Accessed: 2020-05-19.

[15] Native Messaging. https://developer.chrome.com/extensions/nativeMessaging, [Online] Accessed: 2020-02-13.

[16] Open Enclave SDK. https://openenclave.io/sdk/, [Online] Accessed: 2020-02-14.

[17] Package captcha. https://github.com/dchest/captcha, [Online] Accessed: 2020-05-21.

[18] reCAPTCHA. https://www.google.com/recaptcha/intro/v3.html, [Online] Accessed: 2020-02-05.

[19] reCAPTCHA v2. https://developers.google.com/recaptcha/docs/display, [Online] Accessed: 2020-02-13.

[20] runtime.Port. https://developer.chrome.com/extensions/runtime#type-Port, [Online] Accessed: 2020-02-12.

[21] svg captcha. https://github.com/produck/svg-captcha, [Online] Accessed: 2020-05-21.

[22] Top 10 Captcha Solving Services Compared. https://prowebscraper.com/blog/top-10-captcha-solving-services-compared/, [Online] Accessed: 2020-05-22.

[23] Using Privacy Pass with Cloudflare. https://support.cloudflare.com/hc/en-us/articles/115001992652-Using-Privacy-Pass-with-Cloudflare, [Online] Accessed: 2020-06-01.

[24] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, page 7. ACM New York, NY, USA, 2013.

[25] ARM Holdings. ARM Security Technology, Building a Secure System using TrustZone Technology, 2009.

[26] G. Ateniese, J. Camenisch, M. Joye, and G. Tsudik. A Practical and Provably Secure Coalition-Resistant Group Signature Scheme. In M. Bellare, editor, *Advances in Cryptology — CRYPTO 2000*, pages 255–270, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[27] M. H. Au, W. Susilo, and Y. Mu. Constant-size dynamic k-TAA. In *International conference on security and cryptography for networks*, pages 111–125. Springer, 2006.

[28] 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, 2004.

[29] E. Brickell, L. Chen, and J. Li. A static diffie-hellman attack on several direct anonymous attestation schemes. In *International Conference on Trusted Systems*, pages 95–111. Springer, 2012.

[30] E. Brickell and J. Li. Enhanced Privacy ID: A Direct Anonymous Attestation Scheme with Enhanced Revocation Capabilities. In *Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society*, WPES '07, page 21–30, New York, NY, USA, 2007. Association for Computing Machinery.

[31] E. Bursztein, S. Bethard, C. Fabry, J. C. Mitchell, and D. Jurafsky. How good are humans at solving CAPTCHAs? A large scale evaluation. In *2010 IEEE symposium on security and privacy*, pages 399–413. IEEE, 2010.

[32] C.-M. Cheng, H. Kung, and K.-S. Tan. Use of spectral analysis in defense against DoS attacks. In *Global Telecommunications Conference, 2002. GLOBECOM'02. IEEE*, volume 3, pages 2143–2148. IEEE, 2002.

[33] J. Danisevskis. Android Protected Confirmation: Taking transaction security to the next level. https://developer.android.com/training/articles/security-android-protected-confirmation, [Online] Accessed: 2020-02-05.

[34] R. Datta, J. Li, and J. Z. Wang. IMAGINATION: a robust image-based CAPTCHA generation system. In *Proceedings of the 13th annual ACM international conference on Multimedia*, pages 331–334, 2005.

[35] A. Davidson, I. Goldberg, N. Sullivan, G. Tankersley, and F. Valsorda. Privacy pass: Bypassing internet challenges anonymously. *Proceedings on Privacy Enhancing Technologies*, 2018(3):164–180, 2018.

[36] X. Ding and G. Tsudik. Initializing trust in smart devices via presence attestation. *Computer Communications*, 131:35 – 38, 2018.

[37] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. Garcia, E. Gong, H. T. Nguyen, T. K. Sethi, V. Subbiah, M. Backes, G. Pellegrino, and D. Boneh. Fidelius: Protecting user secrets from compromised browsers. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 264–280, 2019.

[38] C. A. Fidas, A. G. Voyiatzis, and N. M. Avouris. On the necessity of user-friendly CAPTCHA. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2623–2626, 2011.

[39] H. Gao, W. Wang, and Y. Fan. Divide and conquer: an efficient attack on Yahoo! CAPTCHA. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 9–16. IEEE, 2012.

[40] P. Golle. Machine learning attacks against the Asirra CAPTCHA. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 535–542, 2008.

[41] 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*, pages 841–850, 2009.

[42] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-Bot: Improving Service Availability in the Face of Botnet Attacks. In *NSDI*, volume 9, pages 307–320, 2009.

[43] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA*, 11(10.1145):2487726–2488370, 2013.

[44] C. A. Kerrache, N. Lagraa, C. T. Calafate, and A. Lakas. TFDD: A trust-based framework for reliable data delivery and DoS defense in VANETs. *Vehicular Communications*, 9:254–267, 2017.

[45] A. Leung, L. Chen, and C. J. Mitchell. On a possible privacy flaw in direct anonymous attestation (DAA). In *International Conference on Trusted Computing*, pages 179–190. Springer, 2008.

[46] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan. Vbutton: Practical attestation of user-driven operations in mobile apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 28–40, 2018.

[47] X. Liu, X. Yang, and Y. Lu. To filter or to authorize: Network-layer DoS defense against multimillion-node botnets. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 195–206, 2008.

[48] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.

[49] G. Mori and J. Malik. Recognizing objects in adversarial clutter: Breaking a visual CAPTCHA. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, volume 1, pages I–I. IEEE, 2003.

[50] 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, page 3, 2010.

[51] X. Ouyang, B. Tian, Q. Li, J.-y. Zhang, Z.-M. Hu, and Y. Xin. A novel framework of defense system against DoS attacks in wireless sensor networks. In *2011 7th International Conference on Wireless Communications, Networking and Mobile Computing*, pages 1–5. IEEE, 2011.

[52] T. Peng, C. Leckie, and K. Ramamohanarao. Survey of network-based defense mechanisms countering the DoS and DDoS problems. *ACM Computing Surveys (CSUR)*, 39(1):3–es, 2007.

[53] P. Perlegos. *DoS defense in structured peer-to-peer networks*. Computer Science Division, University of California, 2004.

[54] C. Priebe, K. Vaswani, and M. Costa. EnclaveDB: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

[55] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. Wintersteiger, and S. Zanella-Beguelin. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. Cryptology ePrint Archive, Report 2019/757, 2019.

[56] C. Rudolph. Covert identity information in direct anonymous attestation (DAA). In *IFIP International Information Security Conference*, pages 443–448. Springer, 2007.

[57] M. Sanghavi and S. Doshi. Progressive captcha, Apr. 30 2009. US Patent App. 11/929,716.

[58] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems for Security. In E. Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, pages 294–311, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[59] J. Z. Wang, R. Datta, and J. Li. Image-based CAPTCHA generation system, Apr. 19 2011. US Patent 7,929,805.

[60] S. Weiser and M. Werner. SGXIO: Generic trusted I/O path for Intel SGX. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 261–268, 2017.

[61] J. Yan and A. S. El Ahmad. A Low-cost Attack on a Microsoft CAPTCHA. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 543–554, 2008.

[62] Z. Zhang, X. Ding, G. Tsudik, J. Cui, and Z. Li. Presence Attestation: The Missing Link in Dynamic Trust Bootstrapping. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 89–102, New York, NY, USA, 2017. Association for Computing Machinery.