PADVA: A Blockchain-based TLS Notary Service

Pawel Szalachowski
SUTD, Singapore

***Abstract*—The TLS protocol is a de facto standard of secure client-server communication on the Internet. Unfortunately, the public-key infrastructure (PKI) deployed by TLS is a *weakest-link* system introducing hundreds of *links* (i.e., trusted entities). Consequently, an adversary compromising a single trusted entity can impersonate any website. Notary systems, based on multi-path probing, were early and promising proposals to detect and prevent such attacks. Unfortunately, despite their benefits, they are not widely deployed, mainly due to their long-standing unresolved problems. In this paper, we present Persistent and Accountable Domain Validation (PADVA), which is a next-generation blockchain-based TLS notary service. PADVA keeps notaries auditable and accountable, introduces service-level agreements and mechanisms to enforce them, relaxes availability requirements for notaries, and works with the legacy TLS ecosystem. We implemented and evaluated PADVA, and our experiments indicate its efficiency and deployability.**

I. Introduction

The security of TLS connections strongly depends on the authenticity of public keys. In the TLS PKI [11], a public key is authenticated by a trusted certification authority (CA), whose task is to verify a binding between an identity and a public key, and subsequently, issue a certificate asserting it. Unfortunately, this process does not provide a high-security level. The identity verification is usually conducted over the Internet basing on the *trust on first use* model [6], moreover, it is a one-time operation (i.e., per certificate issuance). Thus, an adversary impersonating a domain, even for just a moment, can obtain a valid certificate for this domain. The TLS PKI is also a weakest-link system and compromising a single CA (out of hundreds) can result in a successful impersonation attack [1].

One of the first approaches to mitigate such attacks were notary systems [33], [22]. The main idea behind them is to introduce a new trusted party, known as a notary, that provides TLS clients with its view of the contacted server's public key. Hence, a client comparing its and notary's view can get better guarantees that the key is legitimate. Notary systems are based on multi-path probing and assume that attacks are usually short-lived or/and scoped to a network topology fragment. Although they inspired the research community, they did not receive enough traction to be widely deployed. They are often critiqued as they introduce privacy issues (i.e., TLS clients reveal servers they contact), increase the latency of TLS connections (response from a notary is blocking), and are required to provide high availability (otherwise client queries timeout or introduce significant latency) [5], [25]. Notaries are also trusted, not transparent, and difficult to audit.

This research was supported by ST Electronics and National Research Foundation (NRF), Prime Minister's Office Singapore, under Corporate Laboratory @ University Scheme (Programme Title: STEE Infosec - SUTD Corporate Laboratory).

An earlier and extended version of this paper is available as a preprint [31].

In this paper, we present PADVA, a blockchain-based TLS notary service, where notaries persistently validate public keys of domains in an auditable and accountable manner. By leveraging properties of a blockchain platform, PADVA achieves transparency, provides a framework for service-level agreement (SLA) enforcement, and relaxes availability requirements. PADVA is compatible with the legacy TLS infrastructure and can be deployed as today. Our implementation and evaluation indicate efficiency and deployability of PADVA.

II. Background and Preliminaries

Transport Layer Security The TLS protocol is designed to provide secure communication in the client-server model. It is widely deployed and its most prominent use is to secure the HTTP protocol (HTTPS). In such a setting, only a (web) server is authenticated and the authentication is based on X.509 certificates [11]. We focus on TLS 1.2 which is the most popular TLS version [18]. The TLS connection establishment starts with the TLS handshake protocol whose goal is to securely negotiate a shared symmetric key between communicating parties. There are many variants of the TLS handshake and we present the version based on the ephemeral elliptic-curve Diffie-Hellman (DH) protocol [7] as this version is recommended (due to its security benefits) and used by default by modern implementations [29], [18].

The handshake starts with `ClientHello` and `ServerHello`, where the parties indicate their supported cryptographic primitives, their random nonces, and other connection parameters. Next, the server sends its certificate and the `ServerKeyExchange` message. This message contains DH parameters required to conduct an ephemeral DH key exchange and a signature that protects these parameters. The signature is computed using the private key corresponding to the public key of the server's certificate. After the ServerHello phase is finished, the client verifies the signature and sends its DH contribution. Next, the client and server compute the shared secret and signal that the following communication will be encrypted.

Certificates in the TLS protocol are used to verify end entities. In most cases, like HTTPS, only the server is authenticated (i.e., the client does not have a certificate), and usually, the server is identified by its domain name. Certificates are issued by a trusted CA that is obligated to validate the ownership of a public key before the certificate issuance. In the current TLS ecosystem, the dominant way of the validation is based on a domain name ownership. To get a domain-validated certificate, an entity has to prove to a CA that it controls the domain name for which a certificate is requested. CAs automate this process, and usually, the validation is conducted via DNS, HTTP, or e-mail [6].

Problem Definition Notary systems were motivated by the weak authentication of public keys in the TLS PKI. In most cases, CAs rely only on a domain validation conducted over an insecure channel and they validate the given domain only once, just before the certificate is issued. Thus, an occasional on-path (or even off-path [9]) adversary can get a valid certificate for a targeted domain and then use it to impersonate the domain to any client. Notary systems try to improve the security of the TLS connections, providing their views of domains' public keys. Notaries periodically check domains' public keys and measure their *key continuity* [15], such that notaries can inform clients about historical views and give them better guarantees about public keys they see. The effectiveness of notary systems is based on the following assumptions: *a*) impersonation attacks are usually short-lived, thus key continuity, that describes for how long a public key is being used by an entity, can be a practical measure to estimate how a given public key is "suspicious", *b*) multipath probing can detect various MitM attacks, as usually attacks are limited to a fragment of the network topology. Unfortunately, notary systems never received mainstream deployment, mainly due to their availability, privacy, and security issues. Motivated by lessons learned from deployments of notary (and other security) infrastructures, we list the desired properties that a successful notary system should hold.

Persistence: in contrast to today CAs, notaries should validate public keys persistently. It could increase the security level, as then an adversary has to have a permanent ability to impersonate a targeted domain. With such validation, it is possible to reliably implement key continuity.

Auditability: in the previous proposals notaries are trusted, which is circular as CA trust issues are addressed by other trusted parties. Some level of trust seems to be unavoidable as clients contacting a notary rely on its view. However, notaries should be auditable and accountable for their actions.

Privacy: previous notary systems were designed in the interactive client-server model, where a client queries a notary server for a specific view. That design gives notaries ability to learn which websites are visited by which clients, which is unacceptable from the privacy point of view.

Availability: another consequence of the interactive model is an increased latency of TLS connection establishments and strong availability requirements, as information from a notary has to be reliably sent to clients. Unfortunately, maintaining highly available front-end servers is a challenging task, and the currently existing security infrastructures have major problems with achieving it [20], [17].

Backward compatibility: a notary should be compatible with the legacy TLS ecosystem, such that it can provide benefits immediately when deployed. Ideally, a notary system should be orthogonal to the deployed TLS protocol and the TLS PKI, such that it can be deployed almost immediately without requiring flag days, servers' upgrades, protocol changes, or stakeholders' consensus.

System Model PADVA introduces the following parties. A *server* provides service via the TLS protocol (e.g., HTTPS). We assume that services (and servers consequently) are identified by domain names. A *notary* is an entity that offers a PADVA service. Notaries are obligated to monitor servers by validating their public keys. A *requester* is an entity interested in monitoring the server's public key(s). It can be a server's operator or any other entity. A requester orders the PADVA service from a notary.

We assume that notaries and requesters have access to a blockchain platform with smart contracts enabled. For simple description, we also assume that the platform has its cryptocurrency that is used by notaries and requesters. We describe PADVA using an open blockchain platform, however, PADVA can be adjusted to a private platform if needed. We do not require that clocks of servers are synchronized. However, we assume that the requester and notary can agree on at least one reference time source (e.g., it can be the monitored server).

Adversary Model As in previous notary systems [33], [22], we assume that impersonation (MitM) attacks have limited duration, or are limited to only a fragment of the network topology. Otherwise, a long-lived attack that encompasses the entire topology would be difficult to detect by any notary system (as the detection bases on finding inconsistent views). We assume that a notary can misbehave by reporting an invalid view or censoring requester audit queries. However, in such a case the misbehavior should be detected and the notary should be punished. We assume that the adversary cannot undermine the security properties of the underlying blockchain platform and cannot compromise the security of the cryptographic primitives used.

III. PADVA Overview

Design Choices The main observation behind the PADVA's design is that although the benefits of notary systems are attractive, their deployment is marginal due to deployment, operational, and design issues. Our first design choice is to position a *persistent and auditable key validation* as the main task of notaries. In PADVA, notaries constantly validate public keys of domains and measure key continuity. Notaries are similar to CAs, except they: *a*) validate public key of domains permanently (in short intervals), and *b*) are not authorized to issue certificates. PADVA aims to provide auditable and accountable key validation. Notaries in our system are not trusted parties and they have to get a proof that given domain used given key at a given time. These proofs called *validation views* (or just *views*), and other actions of notaries are auditable and verifiable by other entities (requesters, especially).

PADVA notaries offer *key validation as a service*. We argue that notaries should serve a demand-driven service, ideally with payment and SLA frameworks available. That could incentivize notaries to maintain a robust infrastructure. On the other hands, implementable SLAs would secure requesters from notaries violating mutual agreements.

We use a *blockchain as a publishing and contract enforcement platform* since its inherent properties, like transparency, consistency, and censorship resistance, make it a promising underlying technology for a notary system. Firstly, notaries publish the validation outcomes on the blockchain, thus anyone can read it *passively*. It increases transparency, auditability, availability, and preserves users' privacy. Secondly,

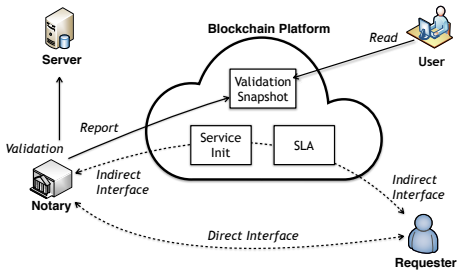

Fig. 1: A high-level overview of PADVA.

```
def request(self, req):  # Schedule request
  assert sender == self.owner
  self.add_request(req); self.deposit(req.fee)

def accept(self, req):  # Accept request
  assert req in self.reqs and sender == self.owner
  self.deposit(self.SLA_DEPOSIT)
  self.add_service(req); self.remove_request(req)

def timeout(self, req):  # Timeout pending request
  assert req in self.reqs and sender == req.sender
  self.transfer(req.sender, req.fee)
  self.remove_request(req)
```

Fig. 2: Pseudocode of the service setup operations.

smart contracts built upon the blockchain platform allow to automate notaries by *a*) introducing automated payments, and *b*) implementing SLA mechanisms securing requesters (in the case of a misbehaving or unavailable notary).

We *decouple the interface between notaries and requesters* into a *direct* and *indirect* interface. Requesters and notaries follow the client-server model for the direct interface, which is used mainly for auditing notaries (i.e., querying validation views). For critical operations, this model is enriched by the blockchain-based indirect interface, where a smart contract acts as a proxy for queries and as an oracle resolving potential disputes. Operations over the indirect interface are implemented as blockchain transactions thus are transparent, accountable, and difficult to censor.

High-level Overview A high-level overview of PADVA is presented in Fig. 1. The central point of our design is a blockchain platform storing validation views and implementing the indirect interface used to initialize a PADVA service and to enforce SLA. We assume that a requester knows (e.g., through a website) a notary, the location (address) of its smart contracts, the service fee, and the guaranteed SLA. Then, the protocol is executed as follows. First, the requester requests the notary service by sending a transaction (to the notary's contract) that contains a fee for the service and specifies request details like a domain name to be validated and its legitimate public keys. The notary notices the request and configures the service accordingly. Then, the notary sends a deposit used to ensure its SLA.

Every *validation period* T, the notary contacts the server and obtains authenticated and fresh information about the server's public key. If needed, the notary publishes in the blockchain a *validation snapshot* indicating a change of the previous validation view. Anyone can access these snapshots.

At any time the requester can audit the notary through the direct interface. If the notary cannot provide the requested result (e.g., it is unavailable or refuses the query), then the requester can use the indirect interface to send the same query over the blockchain. If the notary stays unresponsive, the SLA is executed sending the deposit to the requester.

IV. PADVA Details

A. Service Setup

To order a notary service, a requester sends a transaction to the notary's contract with the following parameters: *Domain name:* whose public key(s) will be validated. *Whitelist of public keys:* that allows the requester to specify which public keys are legitimate. If the notary will observe a listed key, then it is treated as a normal event and no warning notification is issued. Keys in the list are identified by their hashes. If the whitelist is empty, then any new public key observed will be reported. *Fee:* that is paid to the notary for the service. Fees are predefined by notaries and determine the service duration and the SLA provided. *Reference time source:* specifies a time source server that will be used as a reference. This parameter is optional and used if the monitored server does not return the correct time. If the time source is not provided, then the monitored server act as the reference time source. Beside parameters given by requesters, each service has associated conditions of the SLA it offers. These conditions specify *a*) how frequent the notary validates the public key of the server (the validation interval T), and *b*) the availability level provided and a penalty when the level is not met.

The notary observes all transactions sent to its contract and processes incoming requests. The notary checks whether the requester's input is correct (checking the domain name, the whitelist, the fee, and the time source if specified) and accepts the request (`accept()` from Fig. 2) by creating a new service and transferring deposit that will be used for ensuring the SLA.

The notary may refuse the request if the given parameters are incorrect, the server is unresponsive or misconfigured, or for any other reason. In such a case, the requester gets the fee back by calling the `timeout()` method (see Fig. 2).

B. Public-Key Validation and Reporting

PADVA is designed to be compatible with the current TLS infrastructure. Since we want to keep notaries auditable and accountable, a notary to prove that it is validating a domain's public key has to periodically obtain authentic and fresh information that the corresponding private key is being used by the domain. But the TLS protocol was not designed to provide such information.

To overcome this issue, we exploit a message of the TLS handshake protocol. Namely, the `ServerKeyExchange` message (see § II) is signed with a server's private key, and is sent by a server when the key exchange protocol negotiated is a variant of the `ECDHE_ECDSA` or `ECDHE_RSA` methods. These methods are widely deployed and used [3]. The `ServerKeyExchange` message is introduced to provide authentication for client-server negotiated parameters. It contains a server's signature over the `signed_params`

```
def snapshot(self, srv, new):  # Change snapshot
  assert srv in self.srvs and srv.snapshot != STOP
  assert sender == self.owner
  if (new.vid > srv.snapshot.vid and
      new.status != srv.snapshot.status):
    srv.snapshot = new
```

Fig. 3: Pseudocode of the snapshot update procedure.

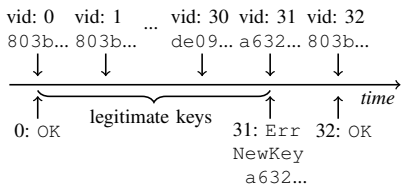

Fig. 4: Timeline of the validation views and the corresponding snapshot changes published in the blockchain. In the example, the public-key whitelist is set to {803b..., de09...}.

structure, which consists of client and server random values and server DH parameters. This is the only message signed by a server during the TLS handshake, hence only that can be obtained by notaries as a non-repudiable proof that a given key was used. However, the authenticity of the message is not enough, as it only proves that the key was used, but does not specify when it happened. A message has to be fresh, as otherwise, a misbehaving notary could conduct multiple handshakes at once, and then use them to claim a fake event timeline. There is no explicit timestamp in the structure, but fortunately, the client and server random inputs should contain the current GMT Unix timestamp.

Using these messages, the notary can periodically obtain a signed statement from the server which implies that a given key was used at a given point in time. To do so, the notary at least every T conducts a new TLS handshake with the server and saves a *validation view* including the server's certificate, signed values (with the server's timestamp), and the signature. These messages are sent during a TLS handshake in plaintext, and they allow to prove to anyone that the corresponding private key was used at the time specified by in the timestamp field sent by the server. A notary assigns a unique *validation id* (vid) to every validation. Validation views with their corresponding vids are stored by notaries for audits.

A notary has to report the results of the validation process. A naive approach would be to just to publish a validation view every T in the blockchain, but that would cause significant overhead. To minimize the interactions between notaries and the blockchain (i.e., to not flood the blockchain with the protocol messages), notaries keep all validation views but report in the blockchain only *validation snapshots* which are changes of the view. A validation snapshot is encoded using the following messages:

OK: a successful validation (the observed key is whitelisted).

Error: an unsuccessful validation. The following error types are specified. **NewKey:** a new public key, outside the whitelist, was observed. It can denote a misconfiguration or an impersonation attack. The notary changing the snapshot to `NewKey` publishes also the hash of the new observed key. **Time:** timestamp signed by the monitored server (or the time source, if specified — see the next section) in the `ServerKeyExchange` message is incorrect (i.e., deviating from the notary time). **Connect:** the notary cannot conduct a successful TLS connection with the server. It can be caused by a network outage, a server-side error, or an adversary blocking the connection. **Stop:** the service was stopped prematurely. For instance, it can be caused by a key revocation (i.e., when the requester contacts the notary with information that the key is revoked and the service can be stopped). No other message can follow the first `Stop` message. **Other:** other error, not specified above.

The notary submits each snapshot update by calling the `snapshot()` method of the notary's contract (see Fig. 3). A snapshot update is associated with the corresponding vid (i.e., when the view change happened). If there is no snapshot change, then no message is sent. Using this simple approach, PADVA optimizes for the common case (i.e., a public key of a server changes infrequently).

Reported snapshot changes allow drawing an event timeline. An example is presented in Fig. 4. In the first validation, the notary obtains a whitelisted public key, thus it reports the `OK` message with vid equals 0. The subsequent 30 validations are also successful, thus no snapshot change is published. With validation 31, the notary detects a public key outside the whitelist, therefore a `NewKey` error specifying the hash of the observed key is reported. The last validation presented is again successful, hence the notary changes the state to `OK`.

To increase the security of the validation process, one strategy is to use multiple vantage points from different networks and locations [16], [9]. PADVA notaries should follow this strategy, as otherwise, a MitM adversary could easily enumerate notary servers and pass their request to the victim, making the attack undetectable.

C. Handshake Timestamping

So far, we present PADVA in the setting where a timestamp returned by the monitored server acts as the reference time. However, the TLS protocol does not require clocks to be set correctly [12], and moreover, some implementations violate the specification and do not set correct timestamps [23]. As we report in § VI, about 63% of tested TLS servers do not put correct timestamps into their `ServerHello` messages.

That could limit the deployability since TLS servers that do not return correct timestamps cannot be reliably monitored. To overcome this issue, PADVA allows notaries and requesters to agree on a reference time source (see § IV-A) which will be used for timestamping TLS handshakes. For simplicity, we assume that a time source server for any client's input returns this input timestamped and signed (like in TSP [2]).

To prove that the public-key validation happened at the given time (according to an external time source) we introduce the following protocol (see also Fig. 5) that the notary executes at every validation: 1) the notary sends a random value r to the time source server. 2) The time source, responds with a signed message (r, t_1, σ_1) which contains the random input, a timestamp t_1 (set by the time source), and the signature σ_1. 3) Next, the notary prepares the `ClientHello` message CH to be

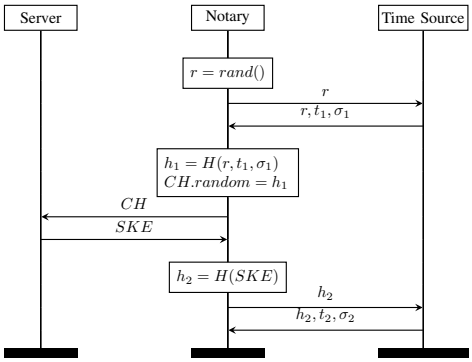

Fig. 5: The extended public-key validation protocol, with the time source server involved. $H()$ stands for a cryptographic hash function.

```
def slaQuery(self, q):  # Publish requester's query
  assert q.srv in self.srvs and sender==q.srv.sender
  q.time = time(); self.addQuery(q)

def slaResponse(self, q, resp):  # Notary's response
  assert q in self.queries and q.srv in self.srvs
  assert sender == self.owner
  self.addResponse(resp)
  if time() - q.time <= self.SLA_TOUT:
    self.removeQuery(q)

def slaClaim(self, q):  # Execute SLA
  assert q in self.queries and q.srv in self.srvs
  assert sender == q.srv.sender
  if time() - q.time > self.SLA_TOUT:
    self.transfer(q.srv.sender, self.SLA_DEPOSIT)
    self.removeQuery(q); self.removeService(q.srv)
```

Fig. 6: Pseudocode of the SLA operations.

sent to the monitored server. The notary sets the `random` field of this message to the hash of (r, t_1, σ_1), and subsequently initiates a TLS handshake with the monitored server by sending CH. 4) The server sends the signed `ServerKeyExchange` message SKE. The timestamp of this message is incorrect, since we assume unreliable server timestamps, however, the client's random field equals $H(r, t_1, \sigma_1)$, encoding trusted timestamp t_1, implying that SKE was created after t_1. 5) Then, the notary sends $h_2 = H(SKE)$ to be timestamped by the time server. 6) Finally, the time source returns a signed (h_2, t_2, σ_2) message, which contains the (trusted) timestamp t_2 and $H(SKE)$ in the client's `random` field. This implies, that SKE is older than (h_2, t_2, σ_2), thus is older than t_2.

By executing the protocol, the notary gets evidence that the message SKE signed by the monitored server is older than (r, t_1, σ_1) and newer than (h_2, t_2, σ_2). The notary keeps all the messages of the protocol as a validation view, that can prove that the server's public key was validated between t_1 and t_2.

Interestingly, any TLS server that can be monitored by PADVA, can also act as a time source. In such a case, the notary simply uses the random field of the `ClientHello` message to submit data to be timestamped and receives a `ServerKeyExchange` message that signs this data. As we show in § VI, in such a setting an average time difference between receiving t_2 and t_1 was measured as 0.34s, thus such an estimation is precise. Moreover, the protocol can be also easily combined PADVA with other protocols and infrastructures (as discussed in § VII).

D. Auditing and SLAs

To facilitate the audit process, a notary is obligated to: a) publish all changes of the validation view in the blockchain (as described in § IV-B), b) store validation views for all monitored domains throughout their service lifetimes.

At any time the requester can ask a notary (via the direct interface) about any validation view. The notary has to return the relevant validation view, such that the requester can compare it with the snapshot encoded in the blockchain. In the case of inconsistency detected, the requester can publicly show the view contradicting the snapshot encoded in the blockchain, as it is provable evidence that the notary misbehaved. For instance, it can be a validation view for a non-whitelisted key, while at the same time the snapshot was not updated (e.g., when any validation between vid 1 and 30 was for a non-whitelisted key – Fig. 4).

One challenge with the direct interface (and any client-server service) is that the notary can be unavailable or censor queries while the requester cannot prevent or prove it. In our context, a notary can have some reasons to not return some validation views (e.g., it was unavailability, misbehaved, or made a mistake). PADVA provides a framework to keep notaries accountable and responsible for such cases. If the requester notices that the notary is unavailable (or was unavailable at a given time interval) or censors queries, the requester can query the notary over the blockchain-based indirect interface. The requester calls the `slaQuery()` method of the notary's contract (see Fig. 6), which registers the query in the smart contract and obligates the notary to respond. If the notary does not return any view until the deadline specified in the SLA then the smart contract will execute the SLA, sending the deposit to the requester. To trigger it, the requester calls the contract's `slaClaim()` method with the unanswered query specified. It is also visible to anyone that the notary did not respond to the query. Alternatively, when the notary returns a response (calling the contract's `slaResponse()` method), the SLA is not executed, as the notary *proved* its availability, and the response can be processed by the requester (see details in § V). Also, in that case, the response is visible to everyone. Thanks to dividing interactions into the two interfaces, the regular audit operations are conducted efficiently using the direct client-server interface, while the blockchain (indirect) interface is used only when necessary.

V. Security Analysis

MitM attacks The main goal of notary systems is to detect and prevent MitM attacks through multipath probing. PADVA follows this strategy, except the public key validation is positioned as a service, where notaries constantly validate public keys and draw event timelines in the blockchain. These timelines are essential for measuring the key continuity and detecting anomalies. We do not mandate PADVA to be deployed in a specific way. We assume that a domain operator would usually act as a requester and we also emphasize that PADVA is orthogonal to the TLS PKI. A requester can whitelist any public keys believed or known to be legitimate. However, in

all deployment cases considered, PADVA can provide effective protection against MiTM attacks as long as an attack is short-lived or is limited only to a fragment of the network topology. In the former case, the attack is seen as an anomaly (i.e., a new observed public key outside the whitelisted set), thus it is suspicious. In the latter case, the attack is identified as an inconsistency between the notary and the TLS client views. In this context, the previous systems provide similar properties, however, the advantage of PADVA is that it provides more reliable and auditable event timelines.

Misbehaving Notary PADVA keeps notaries accountable, and *if an adversary impersonates the domain throughout the validation interval, then the notary cannot make a false statement undetected that it has not happened*. More specifically, if the notary contacts the monitored server and notices a public key that is not whitelisted by the requester, then either *a)* the notary announces the view change making it visible, or *b)* it does not report the view change, hiding this fact.

In the latter case, the notary takes a risk, as the requester conducting an audit will notice that either the snapshot is incorrect or the validation view(s) are missing. (We emphasize that the notary cannot produce a validation view on behalf of the monitored server, as the private key corresponding to a whitelisted key is required.) If any validation view contradicts a published snapshot, it proves the notary's misbehavior. If validation views are missing/censored, the requester can query the notary via the indirect interface. The notary is obligated to response, so either it responds with incorrect data, or it refuses to respond, losing the SLA deposit and showing its unavailability to everyone.

(D)DoS Resilience PADVA provides a more flexible architecture that the previous systems, by placing a blockchain as a publishing medium, while still exposing a client-server interface for heavyweight operations. An adversary can still (D)DoS notary front-end servers blocking their direct interfaces, however, it is much more difficult to stop the notary's operation. The notary can use any back-end server (unknown to the adversary) to conduct public-key validations and report the corresponding view changes (if any) in the blockchain. Namely, PADVA changes availability requirements from *"a notary needs to be up all the time"* to ***"a notary has to have at least one back-end up once per the validation period*** T***"***. Moreover, to avoid enumeration, back-ends can be in different locations, with dynamic IPs, using network tunnels or anonymity infrastructures. To completely block notary operations, the adversary has to either block the underlying peer-to-peer blockchain network or censor blockchain transactions.

Privacy PADVA provides privacy benefits over the previous schemes, as the validation view history can be read from the blockchain, without contacting any third party. In particular, no notary infrastructure is contacted by clients.

VI. Implementation and Evaluation

Implementation The notary's internal architecture consists of the following elements. The monitoring module periodically conducts TLS handshakes with monitored domains, collects cryptographic proofs, and supports the timestamping protocol (as described in § IV-C). Results gathered by this are reported to the database module, which stores all necessary state and data to run a notary service (i.e., active and inactive services, results of all validations, pending requests, ...). The reporting module observes the state of validations and reports validation snapshots into the blockchain via the blockchain API. The direct interface is an HTTP API serving validation results to requesters. It can be publicly accessible to allow any party to audit the notary, and it uses the blockchain API to handle service initialization and SLA requests.

Tab. I: The cost of interacting with the notary contract.

Oper.	`addSrv`	`snapshot`	`slaQuery`	`slaResp`	`slaClaim`
Gas	100 353	35 988	72 921	21 698	32 933
USD	\$0.052	\$0.018	\$0.038	\$0.011	\$0.017

We implemented most of the modules in Python. The monitoring module was implemented with the Scapy and Scapy-SSL-TLS libraries. We make our validations server-friendly by not finishing TLS handshakes, but instead, sending a `TCP RST` packet whenever a `ServerHelloDone` message is received (after receiving the packet the server removes the associated connection state). We chose Ethereum as the blockchain platform, and the smart contract part of PADVA was implemented using the Solidity language. The blockchain API was implemented in Python with the Web3 package.

Smart Contract We deployed a notary smart contract on an Ethereum testnet and measured how expensive is to deploy a PADVA service. Deploying a smart contract on Ethereum is associated with bearing a pre-paid cost for every method execution. The cost is determined by the resource demands of invoked operations and is expressed in the unit called *gas*. Gas is purchased in the Ethereum's native currency – Ether. Although the gas cost is deterministic, the monetary cost in a fiat currency may be highly volatile. For our estimation, we used the data from https://ethgasstation.info/ which estimates the "standard gas price"as 3.5 Gwei per gas (10^9 Gwei = Ether) and one Ether worth around \$150.

The obtained results, expressed both in gas and USD, are presented in Tab. I. The most expensive operation is adding a new PADVA service. It is a one-time operation that costs around 100k gas and \$0.052, respectively. The gas consumption is mainly caused by storing data on the blockchain (what is considered as an expensive operation), thus `slaQuery` has a higher cost than other methods (it has to create a query over the blockchain). Interestingly, the `slaResp` method is cheaper, consuming around 22k gas, as this call removes the responded query (Ethereum treats it as a memory saving and returns some gas).

The snapshot update operation (`snapshot`) is relatively cheap, consuming 35k gas per call. To check how expensive it would be to deploy PADVA in practice, we selected 100 random domains supporting TLS from the Alexa top list, and we kept checking their public keys with the interval T equal six hours. We found that on average the `snapshot` method would have to be called 2.09 times per website over a month (there was no view change for 79 servers, between 1-10 changes for 15 servers, and more than 10 changes for 6 servers). Therefore, maintaining a notary contract would require \$0.04 per month on average.

Tab. II: The distribution of timestamps returned by the TLS servers. Δ is defined as $abs(t_l - t_s)$, for the local time t_l and the server's time t_s.

Δ (s)	0-1	2-5	6-60	61-300	>300
# of servers	3 061	420	130	82	5 823
	32.17%	4.41%	1.37%	0.86%	61.19%

Supported TLS Servers Next, we investigated what is the fraction of TLS servers can be monitored by PADVA. To this end, we scanned 15 000 domains from the Alexa top list and checked how many of them deploy TLS on the HTTPS port (i.e., 443). If a domain does not deploy TLS we also try to prepend it with the "`www.`" prefix. Then, for the deploying domains, we checked how many of them return correct timestamps in the `ClientHello` message (before this test we synchronized our clock with time.google.com). The results are presented in Tab. II. As presented, about 37% of the TLS servers investigated return precise timestamps (i.e., deviating from the correct time up to one second). These servers can be monitored by PADVA notaries directly (without involving a time source) or can act as time sources for servers that return incorrect timestamps.

Overheads Next, we conducted a series of experiments to evaluate our implementation of PADVA in a realistic scenario. We deployed a notary server using Linux Ubuntu 16.04 (64 bit) equipped with Intel i7-6600U CPU @ 2.60GHz and 8GB of RAM. From an academic network, we conducted 100 public-key validations selecting a random supported TLS server (the *single-server* case) each time, and another 100 public-key validations where for each validation we selected a random supported TLS server and another supported TLS server acting as a time source (the *timestamp* case). We reported the obtained results in Tab. III, specifying minimum, maximum, average, and median value for every measurement.

For both, the single-server and timestamp case, we report time required to conduct a validation (t_{single} and t_{ts} respectively), as well as transmission overhead incurred in the validation (s_{single} and s_{ts} respectively). The validation time is mainly determined by network latency. (We emphasize, that in our implementation only *half* TLS handshakes are conducted, minimizing this latency.) For the single-server case, it ranges between 0.04-0.31s with an average value of 0.16s. For the same case, the number of bytes transmitted is between 1.3-6.9KB, with an average around 4.5KB. The timestamp case, which requires three TLS handshakes, needs between 0.19s to 0.73s with an average value of half a second, and to transmit between 6-17KB with an average of about 13.5KB.

Next, we investigated how precise is the variant of the public-key validation with time source server involved. To this end, we measured the time difference between receiving the timestamps t_1 and t_2 from a time server (see Fig. 5). This measurement illustrates how much time it takes to timestamp a TLS handshake (i.e., what overhead is introduced by running the protocol from Fig. 5). The results obtained by our measurements are presented in the table. As we can see, the overhead introduced by deploying the timestamping protocol is around 0.20s on average, what should be marginal when compared with a realistic validation period T (see § VII).

PADVA notaries have to locally store exchanged TLS messages as validation views and when stored naively this storage overhead may be significant. On average, a single validation takes 4 483B, thus storage required for yearly validations of one server, with T equals one hour, would be around 39.27MB. However, validation views are usually highly redundant, as their size is dominated by certificates sent from servers to notaries. Since certificates are usually the same, there is no need to store them all. As an average certificate chain's size s_{cert} was measured as 3 976B, the same yearly validation views without storing redundant certificates would require 4.44MB (almost 9 times less). We emphasize that this storage is local (at notaries) and is not part of the blockchain.

Tab. III: The obtained performance results.

	Min	Max	Avg.	Med.
t_{single} (s)	0.04	0.31	0.16	0.15
t_{ts} (s)	0.19	0.73	0.50	0.50
s_{single} (B)	1 313	6 865	4 369	4 483
s_{ts} (B)	6 031	17 973	13 190	13 445
$t_2 - t_1$ (s)	0.16	0.48	0.34	0.34
s_{cert} (B)	806	6 071	3 815	3 976

VII. Discussion

Time Sources PADVA can deploy TLS servers as external time sources (see § IV-C), as well as other protocols and infrastructure that provide authentic and reliable timestamps. For instance, TSP [2], a document timestamping protocol that relies on the X.509 PKI can be deployed. In TSP, a client submits a hash of data to a timestamp authority which in turn timestamps and signs this data. The signed message is returned to the client, so the client can prove to everyone when the data was timestamped by the authority. This model is almost the same as the model presented in § IV-C (see Fig. 5). Thus, a TSP server can be used as an external time source in PADVA. Another example is secure time synchronization protocols which usually use random client inputs to prevent replay attacks. Such input is timestamped and signed by a time server, thus it can be used analogically as presented in Fig. 5. One instantiation of this approach could be a novel time synchronization protocol Roughtime proposed by Google.

TLS 1.3 The new version (1.3) of the TLS protocol was recently standardized [27]. Learned from other server-side protocol upgrades [26], we should not expect a quick upgrade to TLS 1.3 (especially, when the TLS 1.3 failure rate is currently high [29]), but the new specification removes the `ServerKeyExchange` message type and the GMT timestamp fields from the client and server random fields.

`CertificateVerify` has similar semantics as `ServerKeyExchange`, thus this change is not disruptive. The latter change was introduced to prevent fingerprinting (most of the handshake messages are exchanged in the plaintext) [23]. Due to this change, it is impossible to prove when a given key was used (as there is no timestamp); thus, the simple version of the protocol (see § IV-B) will become unusable as notaries would not be able to prove when a given key was used. Fortunately, PADVA can still be deployed in the scenario with an external time source (see § IV-C). Moreover, if the monitored server deploys TLS 1.3, then any server with a lower TLS version (e.g., TLS 1.2) can act as a time source. Furthermore, as described in § VII other services can be used for providing reference timestamps.

VIII. Related work

Perspectives [33] was the first comprehensive notary system presented. In Perspectives, notaries continuously observe domain certificates. Clients can contact a notary server and compare their view of the domain's key with the view of the notary. Convergence [22] is a similar system where the main improvements over Perspectives are related to privacy and performance. Deployment of Convergence was analyzed by Bates et al. [5], where the increased latency of TLS connection establishment (about 108 ms) is reported. Besides that, these systems are critiqued as they need to have highly available front ends, they require a significant trust in notaries, and introduce privacy violations [25].

Domain Validation++ (DV++) [9] is a recent system where domain validation is conducted through multiple vantage points to mitigate various on-path and off-path adversaries. PADVA notaries could benefit from DV++ by deploying a similar network architecture as proposed.

Certificate Transparency (CT) [19] is a log-based scheme aiming to introduce transparency to the CA ecosystem, by making each certificate visible. CT introduces publicly verifiable certificate logs that maintain append-only databases of certificates. To accept a TLS connection, the client has to obtain a certificate accompanied by a signed statement from a log. Logs are required for certificate issuance, thus their front-end servers have to be highly available, what in practice is challenging [17]. Unfortunately, logs can easily equivocate by creating alternative log snapshots and to detect such equivocations, they need to be monitored by external protocols run by TLS clients [10].

CT has inspired other log-based approaches, that improve its efficiency, enhance security, and add new features [28], [4]. Unfortunately, some of these systems increase the latency of TLS connection establishment or require major changes to the TLS PKI. As for today, no other log-based system than CT is widely deployed.

CoSi [30] is a witness framework proposed to keep CA accountable. In CoSi, a large number of witnesses co-signs assertions about certificates they have seen. As a witness scheme, CoSi is focused on detection rather than prevention, and its design objective (similarly to CT) is to make attacks visible. It is assumed that an adversary would not conduct an attack if it would be eventually visible. To achieve efficient co-signatures, CoSi requires coordination among witnesses.

Recently, blockchains were proposed to approach other problems existing in PKI. For instance, Namecoin [21] (the first Bitcoin fork) provides a namespace where names can be associated with public keys. Certcoin [14] improves Namecoin's inefficiency and adds features like key revocation and recovery. Multiple systems try to improve the drawbacks of log-based systems [32], enhance their capabilities [13], or provide tools to audit trusted entities [8], [24].

IX. Conclusions

In this paper, we presented PADVA, the next-generation TLS notary system. By leveraging the TLS specification and redesigning the validation process, notaries in PADVA do not have to be trusted as much as they were in the past. They become auditable and accountable, able to monitor desynchronized servers, and ready for TLS 1.3. By placing a blockchain platform as one of its central elements, PADVA provides a flexible payment framework and mechanisms for defining and enforcing service-level agreements and relaxes availability requirements making the overall system more resilient. With the increasing capabilities of blockchain platforms, PADVA could be improved by more sophisticated smart contracts. For instance, a contract validating notary responses.

References

[1] DigiNotar removal follow up, 2011. Mozilla Security Blog.
[2] C. Adams et al. Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP). RFC 3161, 2001.
[3] D. Adrian et al. Imperfect forward secrecy: How diffie-hellman fails in practice. In *ACM CCS*, 2015.
[4] D. Basin et al. ARPKI: Attack Resilient Public-Key Infrastructure. In *ACM CCS*, 2014.
[5] A. Bates et al. Forced perspectives: Evaluating an ssl trust enhancement at scale. In *ACM IMC*, 2014.
[6] K. Bhargavan et al. Formal modeling and verification for domain validation and acme. In *Financial Crypto*, 2017.
[7] S. Blake-Wilson and A. Menezes. Authenticated diffie-hellman key agreement protocols. In *Selected Areas in Cryptography*, 1998.
[8] J. Bonneau. Ethiks: Using ethereum to audit a coniks key transparency log. In *Financial Crypto*, 2016.
[9] M. Brandt et al. Domain validation++ for mitm-resilient pki. In *ACM CCS*, 2018.
[10] L. Chuat et al. Efficient gossip protocols for verifying the consistency of certificate logs. In *IEEE CNS*, 2015.
[11] D. Cooper et al. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, 2008.
[12] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, 2008.
[13] L. Dykcik et al. Blockpki: An automated, resilient, and transparent public-key infrastructure. In *BlockSEA*, 2018.
[14] C. Fromknecht et al. Certcoin: A namecoin based decentralized authentication system. *MIT Tech. Rep*, 2014.
[15] P. Gutmann. Key Management through Key Continuity (KCM), 2008.
[16] R. Holz et al. X. 509 forensics: Detecting and localising the ssl/tls men-in-the-middle. *ESORICS*, 2012.
[17] K. Joyce. https://sabre.ct.comodo.com below 99% uptime, 2017.
[18] P. Kotzias et al. Coming of age: A longitudinal study of tls deployment. In *ACM IMC*, 2018.
[19] B. Laurie et al. Certificate Transparency. RFC 6962, 2013.
[20] Y. Liu et al. An End-to-End Measurement of Certificate Revocation in the Web's PKI. In *ACM IMC*, 2015.
[21] A. Loibl. Namecoin. *namecoin.info*, 2014.
[22] M. Marlinspike. Convergence. 2011. http://convergence.io.
[23] N. Mathewson. Let's remove gmt_unix_time from TLS, 2013.
[24] S. Matsumoto and R. M. Reischuk. Ikp: Turning a pki around with decentralized automated incentives. *IEEE SP*, 2017.
[25] G. Merzdovnik et al. Whom you gonna trust? a longitudinal study on tls notary services. In *IFIP DBSec*, 2016.
[26] C. Nykvist et al. Server-side adoption of certificate transparency. 2018.
[27] E. Rescorla. The transport layer security (tls) protocol version 1.3. 2017.
[28] M. D. Ryan. Enhanced Certificate Transparency and End-to-End Encrypted Mail. In *NDSS*, 2014.
[29] N. Sullivan. Why TLS 1.3 isn't in browsers yet, 2017.
[30] E. Syta et al. Keeping authorities "honest or bust" with decentralized witness cosigning. In *IEEE S&P*, 2016.
[31] P. Szalachowski. Blockchain-based tls notary service. *arXiv preprint arXiv:1804.00875*, 2018.
[32] Z. Wang et al. Blockchain-based certificate transparency and revocation transparency. *Financial Crypto*, 2018.
[33] D. Wendlandt et al. Perspectives: Improving ssh-style host authentication with multi-path probing. In *USENIX ATC*, 2008.